CARDIFF UNIVERSITY

SCHOOL OF COMPUTER SCIENCE & INFORMATICS

University Student Timetabling as an Optimization Problem

BSc Computer Science

Final Report

CM3203 - One Semester Individual Project, 40 Credits

Author: James Davis (1422447)

Supervisor: Frank C Langbein

Moderator: Alun D Preece

https://www.linkedin.com/in/james-davis-601406b5

Acknowledgements

Firstly, I would like to thank my supervisor, Frank Langbein, for his invaluable guidance, patience and support. His time, input and advice has been integral to the success of this project.

Secondly, I would like to thank the timetabling officer for the school of computer science at Cardiff university, Helen Williams, for taking the time to compile the resources vital to the testing and evaluation of the final algorithm.

Lastly, I would like to thank all the staff that I have come into contact with throughout my degree. You have instilled a passion in me that will last a lifetime.

Abstract

Timetabling is an essential task undertaken by every educational institution around the world. Most of these institutions still create their timetables for each semester by hand, draining staff time and therefore the institutions money. In this project, I present a genetic algorithm to automatically generate timetabling solutions for university course timetabling problems, thus allowing universities to free up staff time by automating their timetable generation.

The final algorithm presented can efficiently and quickly solve a real-world scenario that occurred for a previous semester in the school of computer science at Cardiff university. It also shows the potential to solve very complex scenarios, given more development time.

TABLE OF CONTENTS

List of Figures

1 – Introduction

1.1 – Introducing the problem and project motivation

Every educational institution all over the world faces a very similar problem regarding how they organize when and where their students are taught over a set period of time (normally on a week by week basis). This is named the Timetabling problem and was defined by *Wren (1996)*[1] in the following way: *"Timetabling is the allocation, subject to constraints, of given resources to objects being placed in space time, in such a way as to satisfy as nearly as possible a set of desirable objectives."* The Timetabling problem encompasses the assignment of many resources (including teaching spaces, teaching staff and student availability) over a set period, in such a way that no clashes occur. A clash is considered to occur when one resource has been assigned more than once in the same time slot, for example when one room has been assigned to have two separate classes being taught in it, at the exact same time.

Most small institutions can solve this problem relatively simply by hand through a trial and error type approach. Their ability to do this is due to the lack of complexity of their timetabling problem. The issue starts to arise when the institutions increase in size and inherently also increase in complexity. There comes a point where the increase of complexity causes trial and error approaches to become very costly in terms of manpower whilst often causing the solution to be far from optimal. Despite this being the case, many educational institutions around the world still must set aside a large amount of manpower annually to produce the year's timetables, when it could be avoided by tasking the problem to a computer.

With the previous in mind, I have created an algorithm to solve the timetabling problem that universities face regarding their weekly scheduling (not examinations), with the goal of allowing implementation in these educational institutions. The benefits of implementing this kind of automated solution are numerous, with the main ones being a much more optimal timetabling solution and the ability to free up a large chunk of staff manpower annually (which in turn causes economic benefits).

One important note is my use of the word 'commitments' in this report. A commitment refers to a teaching session, be it either a lecture, a tutorial or a lab etc. Another important note to make is my use of the terms: 'hard constraint' and 'soft constraint' in this report. A hard constraint is a constraint, which if not satisfied, would render the timetable invalid, and impossible to implement. Examples of such violations could be: one lecturer being assigned to instruct two modules in two different rooms in one timeslot i.e. a lecturer clash. A soft constraint is used to test how optimal a timetable solution is. They represent features that would be preferable to have, but which are not necessary for a valid timetable. An example of a soft constraint could be not assigning any commitment to start before 10am, allowing students and lecturers a later start to the day. The validity of a timetable is judged on its fitness score which is assigned by a fitness

function. In this report, I have designed a fitness function that tests constraint violations and assigns a weighted score, with hard constraint violations being given a more impactful penalty to their fitness score than soft constraint violations.

1.2 – Project goals

Taking the aforementioned into account, the overall purpose of this project is to create an algorithm which provides a way to automate the generation of university timetables. I have decided to implement the main algorithm as a genetic approach and will discuss my reasoning later in the report. I also have created alternative algorithms which have different genetic operators and one that implements a hybrid of genetic and local search methods. The testing sections shows the optimal algorithm parameter and feature combination that is used for the final algorithm.

The final application will allow users to input details of their specific timetabling problem such as available rooms, lecturers and the modules they instruct, modules, students enrolled on each module and amount of commitments necessary for each module. It then generates and outputs a human readable timetable. This timetable will aim to violate zero hard constraints (or else it will not be valid), and as few soft constraints as possible.

The main goal of this project is to create an algorithm that efficiently solves real world university course timetabling problems, creating a solution that is as optimal as possible given the time frame to run the algorithm. Furthermore, the following provides a more detailed breakdown of my aims/goals for this project.

- The application must aim to output a timetable violating zero hard constraints and as few soft constraints as possible
- Output timetable is human readable
- Timetable should be generated in a practical time frame (under two hours)
- The application should be easy to use. A minimal amount of training should be needed to teach a user the input configuration file syntax, allowing them to provide input into the application. This ensures that it could be used in real world scenarios.
- The application should be able to solve problems of a realistic complexity.

1.3 – Intended stakeholders and project scope

The stakeholders in this project mostly exist in university settings. These are the stakeholders that I have identified: Myself, university administration staff, people affected by the result of the algorithm (lecturers, students etc.) and other computer scientists that can use aspects of this report to influence their own Genetic algorithm/Timetabler. The audience of this report is very similar to the stakeholders, although could include other people generally interested in the concepts used in this paper (genetic algorithms, local search techniques).

The main scope of the project extends to all higher education institutions that require a timetable to be developed for their provided courses. This could even be extended to different types of institutions that require timetabling for other, yet similar types of situations e.g. for schools or colleges.

1.4 – Approach

After comparison of the main methods used to solve timetabling problems, I decided on using an evolutionary approach, and more specifically a genetic algorithm. I provide a more detailed explanation of genetic algorithms in section 2 of this report, but in general terms, a genetic approach uses the model of evolution found in nature to create generations of timetable solutions that merge towards local and global optima. Individuals of each generation act as a possible solution to a problem and are represented using genes (a set of data defining the details of the solution). These individuals are then modified through a combination of genetic operations, creating a new generation that should contain more optimal solutions than the previous. This incremental improvement of the average quality of individuals in each generation leads to the convergence on an optimal solution.

Initially, I am creating a basic algorithm which implements basic genetic operations, more specifically a basic form of genetic crossover and a random mutation. I am then incrementally changing operator parameters and adding new operators to this basic algorithm, testing the observed change in performance after each modification so that I can create the optimal final algorithm. Such additional operators and parameter values include a form of local search, a novel feature I am calling mutation ramping, different mutation and crossover rates and different implementations of the crossover and mutation operators. I am then testing the final algorithm against a previous scenario taken from the school of computer science at Cardiff university to gauge the algorithms effectiveness at solving it. Finally, I am testing how the algorithms resource consumption (memory, CPU usage) scales with problem complexity.

1.5 – Assumptions

I am making assumptions based on the nature of the problem I am solving. I am specifically solving university course timetabling problems, and therefore I have a set of constraints that apply to this problem that does not include other constraints that apply to different timetabling problems. These constraints include: no room clashes, lecturers aren't assigned to multiple commitments at the same time, students aren't assigned to multiple commitments at the same time etc. I am also assuming that the algorithm will be run on a standard power computer. I am not specifying any exact software/hardware requirements but assuming that it will not be run on an extremely powerful computer such as a large cluster of computers. I am also assuming that a correct input is given to the algorithm in order for it to work correctly. In the interest of saving

time, and to allow further development of the algorithm, I have not implemented many user error mitigation elements to the application.

1.6 – Project Overview and Achievements

This report documents the theory, implementation methods and testing results of a predominantly genetic based approach to the University course timetabling problem. Different optimisation methods are documented and tested, with the final and best performing algorithm consisting of a genetic algorithm with specialized genetic operators and a potentially original feature that, to my knowledge, is not mentioned in any of the scientific papers I could find on the subject.

The final algorithm can solve a timetabling problem taken from a previous academic term for the school of computer science at Cardiff university. Not only is it able to solve this problem, it does so in under two minutes, which makes it extremely more efficient than performing timetabling by hand (human generated) in terms of time and money (due to saved staff time). The final algorithms ability to scale with complexity is also explored, resulting in memory and CPU efficient results and proving that it has the potential to scale up to solving high complexity scenarios posed by large higher education institutions.

2 – Background

2.1 – Approaches to the Timetabling problem

The Timetabling problem which I am referring to in this report is more specifically called an educational timetabling problem, distinguishing it from other slight variants of the timetabling problem. Even more specifically, it can be further narrowed down to a course timetabling problem, as defined by *Schaerf (1999)*[2] who classified educational timetabling into three main groups i.e. course timetabling, school timetabling and examination timetabling. Although these three groups can be separated by some minor differences in the constraints placed upon them, they are similar enough as to allow a well-made timetabling application, that allows user inputted constraints, to be able to solve a scenario that falls into any of the three categories.

Timetabling problems define a class of hard-to-solve constrained optimisation problems of combinatorial nature. A constrained optimisation problem is defined as[3] "*the process of optimising an objective function with respect to some variables in the presence of constraints on those variables".* Such problems are mainly classified as constraint satisfaction problems [4], where the goal is to satisfy as many of the constraints as possible, as opposed to optimising a set of objectives. At present, an exhaustive search is the only method to guarantee the optimal solution for every timetabling problem, but due to the complexity of most problems, an exhaustive search approach is impractical for most real-world problems. Many different approaches to the timetabling problem have been suggested and researched to date and can be split into four categories:

1. Constraint-based methods [4]
2. Cluster methods [5]
3. Sequential methods [6]
4. Meta-heuristic methods[7]

All these different approaches have scenarios in which they perform best. With this in mind I will describe a select few who are best suited to solving course timetabling problems below.

2.1.1 – Genetic Algorithms

Genetic algorithms are first referenced in academic literature by John H.Holland in a book[8] called "Adaptation in Natural and Artificial Systems" and have since been more accurately defined as metaheuristic optimisation algorithms. They are part of a set of algorithms that fall under the umbrella term of an "Evolutionary algorithm". Evolutionary algorithms use mechanisms taken from the process of evolution found in nature, the main one being the idea of survival of the fittest, where a "fitter" member of a population has an increased chance of

reproducing and passing on its genes. The fitness of the individuals in a generation is represented by a fitness score, which is generated by a fitness function. A fitness function is a type of objective function that allocates a score to a solution in order to represent, with a single number, how optimal the solution is. For this project, the fitness function will allocate a score based on the number of hard and soft constraints violated.

Evolutionary algorithms work in an iterative manner, in which a population of candidate solutions (which I will refer to as individuals) is created on each iteration. Each individual in a given population can be viewed as a possible solution to the given timetable problem and is represented by its "genes", which is normally a binary string. Although a binary string is normally used to represents genes, for many situations (including this project), a different representation may be needed.

Individuals of a new generation are created by applying genetic operations to members of the previous generation. The most common genetic operators are Selection, Crossover and Mutation (normally applied in that order).

- **Selection**

  In order for individuals to be created to populate a new generation, there has to be a way of selecting "parent" individuals to provide the new individuals genes. There are different methods to select these parent individuals, but most of them work through a fitness-based selection process, where fitter solutions have a higher probability of being selected. The type of selection and the way in which a selection method is implemented is important to the effectiveness of the whole algorithm. If a selection method has too high a bias on selecting fitter individuals then we see a faster convergence which can lead to portions of the search space being lost, and as a result, a local optimum being reached instead of the global optimum. On the other hand, if lower fitness individuals are given too high a probability of being selected then we may see a very slow convergence, or none at all. This is called selection pressure, with a high fitness individual bias being said to have a high selection pressure and when lower fitness individuals have a higher chance of being selected it is said to have a lower selection pressure. Selection pressure is one main characteristics that separates selection methods. Below are some of the most relevant selection methods to our timetabling problem[9].

  - Fitness proportionate (Roulette wheel) selection

    Fitness proportionate selection selects individuals by associating every individual fitness score in a generation with a probability of being selected, e.g. an individual with a better fitness score than another has higher chance of being selected. This method of selection is the most commonly used, due to its bias towards better solutions whilst still allowing a chance for worse fitness individuals to be

selected, giving it a medium selection pressure. Fitness proportionate selection requires fitness score to be normalised (modified to be within the range of 0 and 1) and will only work for minimising fitness functions (function scores that are better the closer to 0 they are).

Figure 1[19]

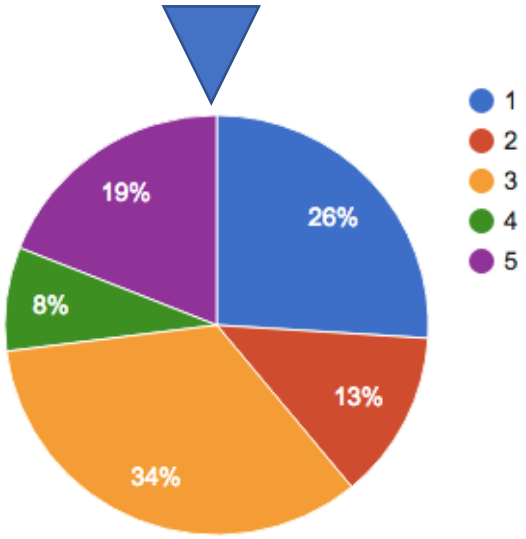


Figure 1 shows an example of how individuals are selected. Individual three has the best fitness score and therefore has the highest chance of being selected by the "spinner", due to its largest percentage of the spinner.

- Tournament selection

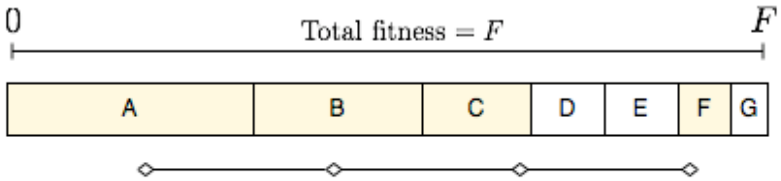
  Tournament selection is a method of selection that involves running several "tournaments" between several randomly chosen individuals from a generation. The winner of each tournament is the individual with the best fitness score, who then goes on to be selected for breeding. The number of individuals (n) randomly selected for each tournament has a large impact on how this selection process works e.g. if n = 1 then tournament selection becomes a simple random selection. On the other hand, if n becomes too large a percentage of the total population size, then its selection pressure is probably too high, leading to fast convergence. Tournament selection is more efficient than the other selection methods as only a few individual's fitness values need to be processed, as opposed to every individual in a generation. It also allows for the selection pressure to be easily adjusted; a useful and rare characteristic.

- Stochastic Universal sampling

  Stochastic universal sampling (SUS) was a sampling technique introduced by[10] James Baker in 1987 aimed at reducing the inefficiency and bias in selection methods. SUS is a development of the aforementioned fitness proportionate

selection, but instead of using repeated random sampling, SUS uses just one random value in order to sample all the solutions by choosing them at evenly spaced intervals.

Figure 2[20]

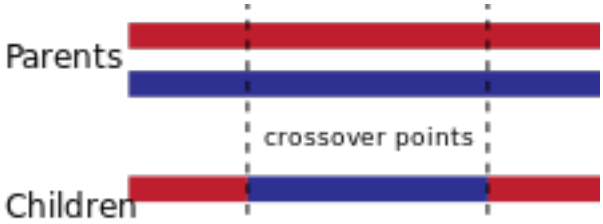


Above, figure 2 shows an example of how the one random value is used to sample at evenly spaced intervals (diamond icons at the bottom denote when the random value occurs and, by looking above the icon, the individual selected).

- **Crossover**

The crossover genetic operation is used to combine the genes of selected parents in some way. This operation is modelled on the split of genes that a child receives from its mother and father, as observed from reproduction in nature. Differences in Crossover methods stem from the number of parents used to make up the child's genes, and the point in the genes at which the split occurs. Here a few examples of different crossover methods:

Two-point crossover (figure 3)[21]



Uniform crossover (figure 4)[22]

With a probability of 0.5, the child has 50% of the genes from the first parent and 50% from the second parent, even with randomly chosen crossover points.

- **Mutation**

  Mutation is normally the last genetic operation performed. Its purpose is to add the element of random mutation that is found in nature, and in turn to maintain genetic diversity from one generation to another. This genetic diversity helps prevent a saturation of one high fitness individual in separate generations.

  The mutation operation is very simple and involves randomly selecting individuals and changing some parts of their genes. Different mutation methods vary in the way the individuals are selected (e.g. randomly and uniformly) and in the way that the genes of the individual are modified. Here are some of the most common gene changes carried out by mutation operations:

  - Boundary: Replaces the genome with either the lower or upper bound randomly. Only works for genomes with integer values (binary/integer string genes).
  - Flip bit: Inverts a genomes binary value. Only works for genes encoded as binary strings.
  - Uniform: Replaces the chosen genome with a random value between a given upper and lower bound.

Figure 5 – Advantages/Disadvantages of genetic algorithms

Advantages	Disadvantages
Genetic algorithms always produce some kind of answer, and the answer that's produced only improves over time.	Repeated fitness function evaluation for individuals (for selection purposes) can be very expensive and lead to slow algorithm speeds if not designed efficiently.
They are very easy to distribute, making implementation in parallel computing very easy.	Genetic algorithms can converge to local optimum without any way of identifying whether it is a local or global optimum.
The mutation operation causes the solutions in each generation to constantly change slightly, making them more diverse and giving genetic algorithms the ability to prevent becoming stuck in a plateau.	Genetic algorithms can't solve problems where the fitness value is either a true or false value because differentiation between two individuals with true values to find the optimum is impossible.
Genetic algorithms are very flexible and allow for a high level of heuristic customization and tweaking, leading to a very specific algorithm for a distinct type of problem.	Genetic algorithms don't scale in terms of time well with complexity, making them unsuitable for solving very complex problems.

2.1.2 – Tabu Search

The idea of Tabu search was first proposed in a journal by Fred Glover in 1986[11], where he defined Tabu search as: *"A meta-heuristic that guides a local heuristic search procedure to explore the solution space beyond local optimality"*. Tabu search is an improvement to a previously existing search method called Local search. Local search is an iterative process which starts with a potential problem solution and then checks other solutions that have minor differences (its neighbours). It then iteratively selects the best neighbour and checks all of its neighbours. Local search has a strong tendency to plateau in areas of the search space where all neighbours have very similar/equal fitness scores due to falling into a loop were the same neighbours keep being selected.

Tabu search is designed to overcome this tendency to plateau by relaxing the rule of selecting only the best neighbour and by "tracking" previously explored areas of the search space. Firstly, Tabu search allows for worse neighbours to be explored if there are no better alternatives, thus preventing the issue of plateauing at similar fitness areas of a search space. Secondly, and the reason for the name "Tabu", Tabu search implements a feature called the "Tabu list" which consists of a set of rules present in three types of memory structures: short-term, intermediate-term and long-term. Short term memory structures are designed to store rules to ban the inverse of a recent, previous move. The idea behind this being that storing the inverse of a previous move as a banned move prevents the immediate inverse of a move and encourages the search to move away from that area of the search space. Short-term memory structures only store a minimal number of moves, preventing it becoming too large and because, after a few moves, a banned move becomes redundant. Intermediate-term memory structures are used to store rules that are designed to "intensify" the search and direct it to other, more promising areas of the search space. Long-term memory structures are used to store diversification rules, which have the purpose of forcefully driving the search to new regions of the search space. These diversification rules are normally used when a search becomes stuck in a plateau. In practice, all three types of rules overlap and are used in conjunction.

Figure 6

Advantages	Disadvantages
Is effective when used with other methods such as Genetic algorithms. Can be used to speed up convergence to local optima.	A large of solutions fitness scores must be calculated, due to each 10isualiz having to be calculated on each iteration. This can become very costly if the fitness function is complex.
Improves massively on aspects of basic local search approaches.	Can be timely to implement.
Can be very efficient if fitness scores are fast to calculate.	When used in conjunction with another method, there is no guaranteed speed increase.

2.1.3 - Backtracking

Backtracking algorithms, as explained by Gurari and Eitan[12], are mainly used to solve constraint satisfaction problems. They are depth-first search algorithms; a method used to traverse problems represented in a tree structure. The search starts from the root of a tree graph, and traverses down each leaf node, starting from the very left and working its way right as the search progresses. Backtracking only works for problems which allow for the concept of a "partially correct" solution, and that allow for a test to determine that fact, making it a viable method for solving timetabling problems.

Partial solutions are represented in a tree, where parent nodes are partial candidates that differ from the child nodes by a single step, and leaf nodes are partial solutions that cannot be altered in any way that would make it different to all its parents. The backtracking algorithm begins a depth first search and checks each node to see if it can be "completed" (changed in a way to make the solution valid). If it cannot be completed then the node's entire sub-tree (all of its children nodes) are pruned. If, however, a node is found that can be completed, then every child node in the sub-tree is tested and the best solution is returned as the located solution. The "backtrack" element of the algorithm comes from the ability to jump back to a previous parent node if a dead end is reached (a candidate is found that can't be completed).

Although possible to use for timetabling problems, it is not an ideal method when trying to find the most optimal timetable. Due to the nature of the algorithm, as soon as one valid solution is discovered, it is returned. This makes it very good for problems where a valid solution is the only requirement e.g. for solving a Sudoku puzzle. However, for our particular timetabling problem with the presence of soft constraints, it is a very ineffective method as the minimization of soft constraint violations is not considered.

Figure 7

Advantages	Disadvantages
Can be extremely fast if only a valid solution is required.	Search can take a large amount of time searching sections of the tree containing no possible solutions, with no mechanism to detect such situations.
Can be used in combination with other algorithms, such as Tabu search, in order to further streamline/prune the search tree.	It is extremely unlikely to return the best possible solution if a problem factors in soft constraints.
It can be found to discover all possible solutions if allowed to search the entirety of the tree, however, this is normally very costly.	The size of the tree does not scale well with the complexity of the problem, causing extremely costly search times.

2.1.4 - Ant-colony Optimisation

The ant colony optimisation algorithm is inspired by the ant's ability to locate food. In nature, ant colonies require their worker ants to leave the nest and locate food to sustain the queen and the rest of the colony. Ants move randomly to find food and always leave a track of pheromones behind them which can be detected by other worker ants. When an ant finds a pheromone track it follows it and if it leads to food, then the ant returns to the nest, leaving its own pheromone track alongside the initial one. Over time less favourable pheromone tracks evaporate and become less travelled, causing tracks that lead to food to be more potent and the most attractive ones for other ants to follow.

Marco Dorigo first proposed the use of this natural phenomena in 1992[13], categorising it with other swarm intelligence methods. Originally the idea was limited to finding an optimal path in a graph but has since been developed allowing its application to other problems, with one such problem being timetabling. The algorithm works using traversal through a tree structure. An ant is given a probability of moving from a node I to a node j with a certain probability. The probability is calculated for each node using an equation which relates the "pheromone level" of the edge between the nodes, some constants and the "desirability" of the edge. After each move, the pheromone levels of each edge are recalculated accordingly. This process continues iteratively until the best path through the tree is found.

The Max-min extension[14] to the ant-colony optimisation algorithm was developed by Hoos and Stüzle in 1996 and is the most effective extension when addressing a timetabling problem, due to its ability to distinguish paths according to a maximum and minimum. This allows a for soft constraint optimisation, making it a good method for solving our timetabling problem.

Figure 8

Advantages	Disadvantages
Can be run very effectively in parallel.	Can become quite complex to develop and debug.
Can be used for dynamic problems (adapts to new changes to the problem).	Aspects based on randomness (although each random decision is not independent)
Positive feedback accounts for fast discovery of viable solutions.	Time to converge on a solution is completely unpredictable and can be very long.

2.1.5 – Genetic algorithm choice justification

After comparing the aforementioned optimisation algorithms, I made the decision to use a genetic approach as the basis of my main algorithm. This is mostly due to its nature of reaching the global optima far more often than other optimisation methods (given enough time) due to its large coverage of the search space, but also because there are many ways to improve and optimise a genetic algorithm. Genetic algorithms work very effectively in conjunction with aspects of other methods e.g. Local search methods. Being able to combine the best of other solutions in order to streamline my genetic algorithm will allow for a large portion of the search space to be explored whilst also guaranteeing an optimal solution (or even the global optimum).

One of the main advantages that the other methods have over a genetic method is the speed in which they return a viable solution. In most areas of computer science this would indeed be a huge advantage, but due to the nature of the course timetabling problem, and its real-world applications, is not so much of an advantage. University timetables are only generated a small number of times per year. This means that having a slower algorithm becomes much less of a problem as time is not a limiting factor, and in fact, time is not a factor at all because the algorithm can be started a long time before its resultant solution is required. Genetic algorithms are also relatively easy to implement, which gives me time to make my algorithm more advanced with the additions of other methods, and more time to improve aspects of the genetic algorithm such as the fitness function, and the efficiency of the code.

2.2 – Previous solutions to course timetabling solutions

Solutions to the timetabling problem, and more specifically the course timetabling problem have been discussed for over 20 years, with a vast amount of different approaches being attempted. I will only mention variants that have used a genetic algorithm as the base of the approach, although most high performing algorithms are a hybrid between different search techniques. Although I only mention a few examples in this section, I am aware of many other papers addressing this topic. Some of which are present in my bibliography, and others including examples given by the Cardiff University mathematics department[26][27]. I don't go into more detail on the Cardiff papers due to their highly theoretical nature, which does not aid this project much. I have chosen the following examples because they either demonstrate a novel approach to the problem, or they are specifically aimed at a genetic approach. They also are very detailed, covering most fundamental material found in other papers that I have read, making the reference to other papers redundant unless seeking very specialized information.

2.2.1 – Spyros Kazarlis, Pavlina Fragkou and Vassilios Petridis

This paper[15] presented a method for solving university timetabling problems using a genetic algorithm that had a heuristic local search element. In the testing section of the paper, a large number of different hill climb operators are tested to determine which resulted in the most

efficient algorithm. The final version of the algorithm which is presented at the end of this paper used a MicroGA Combinatorial Hill Climb operator[16] set with a probability equal to one.

This paper is interesting and presents a wide array of different operators that can be applied to genetic algorithms. It also gives me a benchmark as to the difficulty that I set my test cases at.

2.2.2 – Enzhe Yu and Ki-Seok Sung

This paper[17] presents a rare approach, using a sector-based genetic algorithm to solve university course timetabling. Again, this testing element of this paper is very useful and gives insight about the kind of benchmark I should be setting my test cases at. The paper also presents an interesting method of decreasing the search space to a feasible size by implementing a "check-and-repair" routine to ensure that each solution in a population does not violate any hard constraints. Although the paper is short, and the tests presented in conclusion were only preliminary, it shows the use of a technique (check-and-repair) that I had not considered until reading the paper.

2.2.3 – E. K. Burke, J. P. Newall and R. F. Weare

In this paper[18] it is explained that genetic algorithms are one of the best methods to approach course timetabling problems due to the algorithm's ability for optimisation. It goes on however to state that algorithms which combine genetic and local search techniques normally produce better results than just a genetic algorithm, concreting my belief that a combination of techniques will work better than just a genetic approach on its own.

2.2.4 – Problems with previous solutions

There have been many different algorithms created to solve university timetabling problems, with some of the earlier attempts being made over thirty years ago. Most of these problems have been created for a Master's, PHD thesis or for a scientific journal. Taking that into account, I believe that the time limit placed on this one-term project alone rules out the goal of trying to improve on these past algorithms, especially considering the amount of time that can be spent writing a PHD thesis. With that in mind, I will attempt to create an efficient algorithm that can solve real world problems.

Due to ageing nature of many previous solutions to this problem, I will be able to explore the use of new techniques that had not be developed at the time of these older solutions. I will also be able to use more complex test cases, pushing the algorithm to perform at a new scale due to the power increase of computing hardware/software.

2.3 – Cardiff Universities current timetabling method

To understand the current approaches to timetabling currently practiced by universities, I requested a brief overview of the current process used by the school of Computer science and Informatics in Cardiff University. The timetabling officer (Helen Williams) sent me an overview of the current process that they currently use.

Firstly, it should be noted that everything listed here is done by hand, taking up staff time to the extent where a job role has been created to manage the task (although timetabling isn't her only role). Below is a brief overview of their current approach:

1. Information is collected for the next academic year
    a. Module list, staff delivering each module, whether the module is core or optional, each modules lecture, lab (hardware), tutorial requirements and expected number of students.
    b. Collect specific constraints (e.g. staff availability, university imposed free periods etc.).
    c. Lecture, lab (hardware) and tutorial requirements for each module.
    d. Room list with capacities.
2. Start timetabling with the most constrained programs first.
3. For year 1 allocate lectures and support sessions for Computational thinking weeks (1-4)
4. Continue with second and final year students, checking for clashes.
5. Timetables then circulated to all teaching staff for checking, and then adjustments made as required.
6. Work with the Universities central timetabling team to upload and check timetable data.
7. Adjustments made throughout semester as required.

This method, whilst effective, is very resource inefficient. All teaching staff are required to individually check the timetable to check for issues personal to themselves. This, alongside the need for a central timetabling team and a specific timetabling officer, demands a huge amount of staff time. Implementation of my final algorithm would allow Cardiff university to task stages 2 through 5 to my automated algorithm, saving huge amounts of staff time. I also notice that there is no optimisation stage, where different variations are compared for soft constraints to maximize staff/student convenience. This would be something that automatically occurs in the generation of timetables if my final algorithm was to be implemented.

3 – Problem Statement and Basic Algorithm

3.1 – Problem statement

As already stated, the algorithm I am creating is designed to solve university course timetabling problems. As such, there are a set of constraints and problem variables that define the type of solvable problems, and how these problems are "solved".

Due to the time constraint of this one-term project, I have decided to focus solely on the main algorithm, and pay minimal attention to developing a user-friendly application. Due to this, I will "hard-code" in the constraints that are used to evaluate the target function of a given timetable, as opposed to allowing users to input their own hard and soft constraints. I selected these constraints to reflect the basic constraints normally implemented into university course timetables. Below are all the hard and soft constraints that I will be using this algorithm.

Hard

- There can be no student clashes, e.g. one student cannot be assigned to two commitments in the same time slot.
- There can be no lecturer clashes, e.g. one lecturer cannot be assigned to teach two commitments in the same time slot.
- The same room cannot be allocated to two different teaching commitments in the same time slot.
- Teaching commitments should not be assigned to time slots not specified to be in the working day.
- Every teaching commitment for every module should be allocated in its own timeslot/room combination.

Soft

- There should be no lectures assigned to 9 am slots.
- Lectures should be grouped into blocks and not spread out throughout the day. The specific constraint is that no two adjacent slots for a student should be more than four-time slots apart.

Problem variables

There are four types of problem variables that I have designed the program to work with:

- Commitments
- Timeslots
- Modules
- Rooms

Each of these variables allows the input timetable problem to be expressed. The commitments are a set of variables that express each individual teaching commitment and the lecturer required

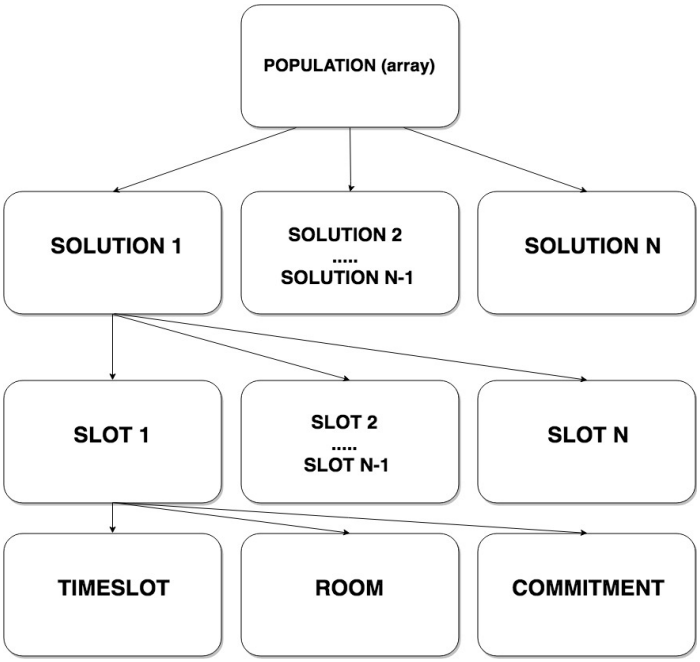
to instruct it that must be present in the final timetable, e.g. one module may have a set of ten commitments that are required per week to deliver the module to the students. Rooms are a set of variables that express how many rooms are available for use. The timeslots are a set of variables that express each slot in which a commitment can be placed. Therefore, the total amount of possible allocations for the commitments is equal to (rooms x timeslots), because for every timeslot a commitment can be allocated to each room. Finally, the modules are a set of variables that express all the modules, and all the lecturers that assigned to each module.

I have designed the variables in this way as I believe it is the most intuitive way. It allows for the user input configuration file to be relatively easy to create whilst modelling the problem in a way which is easy to understand, regardless of increasing complexity.

3.2 – Representing the problem

The basis of any genetic algorithm is having a population of individual solutions, each having a set of genes that defines it. I have decided to stray away from the normally used binary string gene representation, and instead, use a class-based representation. The main reason I have made this design choice is for simplicity. I wanted to be able to easily visualize the problem, and avoid having code that worked on a very long binary string which was difficult to understand, especially with increasing problem complexity. Each individual solution in a population is an object of the solution class. This solution class contains different getter and setter methods, but most importantly contains a list of slot objects. Each slot object contains a mapping from a given commitment, to a room and a timeslot. So, the genes of an individual are made up of a group of slot objects, where there is one slot object for every given commitment.

Figure 9



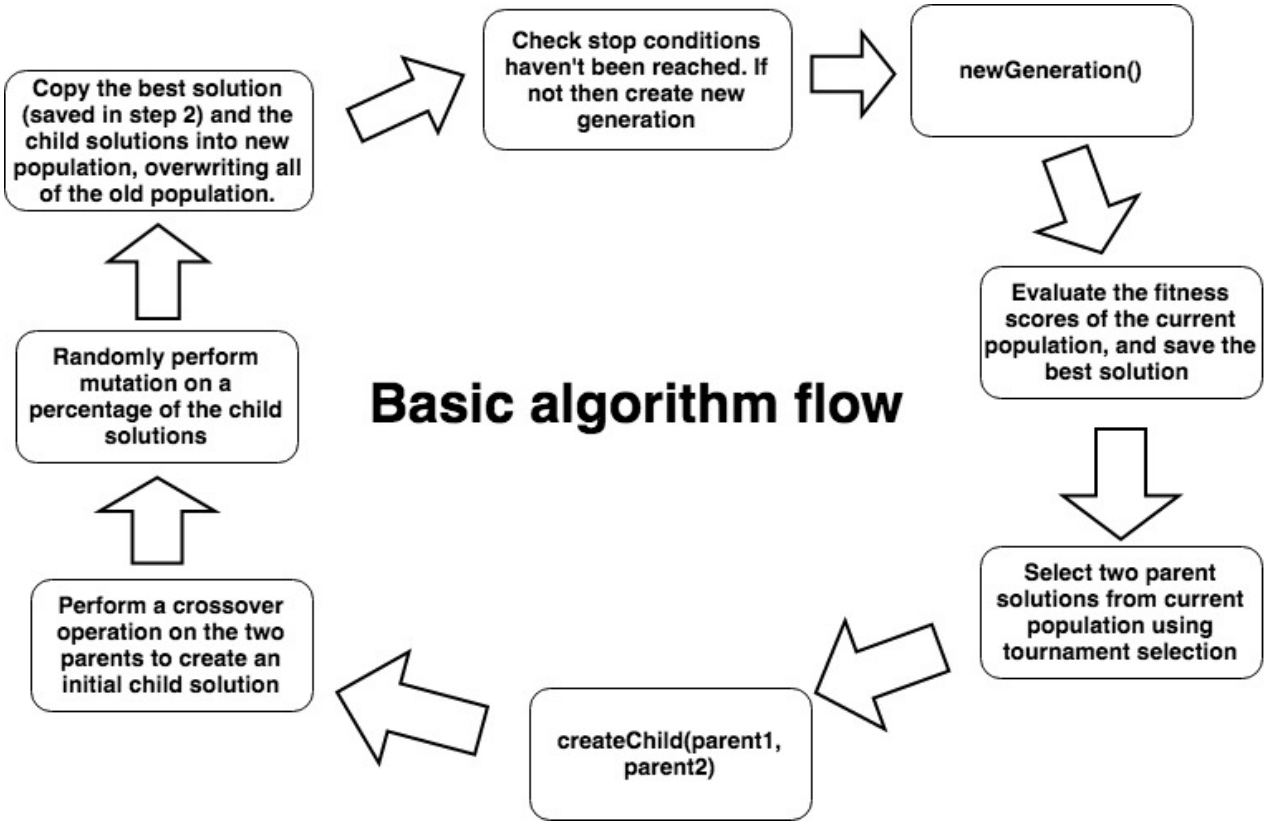
3.3 – Basic algorithm

The basic algorithm that I have designed is a genetic algorithm with tournament selection for the parent selection, and crossover and mutation operators for the child population generation. This basic algorithm provides the base genetic operations of most genetic algorithms and provides a baseline to gauge the value of additional, more advanced features. There are five main methods in the timetable class that are used in the basic genetic algorithm, these are:

1. **newGeneration()** – Used every iteration to create the new generation.
2. **createChild()** – Takes two parent solutions as arguments, and returns a child solution.
3. **crossover()** – Takes two solutions as arguments, and returns a child solution that is a result of performing a crossover operation on the two parents.
4. **mutateChildOld()** – Takes a child solution as an argument, and mutates it.
5. **tournamentSelection()** – Randomly selects a number of individuals from the population, and returns the index of the individual with the best (lowest) fitness score.

The newGeneration() method is called at the start of every iteration of the algorithm, and it's responsible for enacting each part of the main genetic algorithm. Below is the program flow for each iteration of the basic algorithm, all of which occurs from inside the newGeneration() method.

Figure 10



When designing each section of the algorithm I have aimed to keep the time complexity as small as possible, with only two methods: one constraint check and the hill climbing operator added later in this report having a time complexity of $O(n^2)$. Every other method has a time complexity of O(log(n)n) or better. I have done this to allow the algorithm to scale with problem size and not becoming impractical to run.

Below I will describe in more detail the three main aspects of the basic genetic algorithm: tournament selection, genetic crossover and mutation.

Tournament selection

I use the selection method of tournament selection in this basic algorithm. The reasoning behind this is due to the use of a minimising fitness function. The main alternative to tournament selection is roulette wheel selection, but this can be difficult to implement for a minimising target function. Implementation would involve normalisation and further modification to each solutions fitness score, which would add more complexity to the algorithm design. I do however implement roulette wheel selection later in the testing section of this report, and provide a comparison of the two methods. Additionally, tournament selection allows me to control the selection pressure; a feature that roulette wheel selection doesn't allow. I have decided to use three contestants per round for preliminary testing of the basic algorithm. This is because I believe 2 contestants would cause too weak of a selection pressure, resulting in a very slow convergence, and 4 or more contestants would result in too high of a selection pressure resulting in too fast of a convergence and a saturation of similar solutions, cutting out large sections of the search space. Below is pseudo code for the tournament selection function.

Tournament selection

```
1: function TOURNAMENTSELECTION()
2:     for i = 0; where i < number of contestants do
3:         Select random solution from population
4:         if  random solutions fitness score < best saved contestants score then
5:             Save new score as the best
6:         end if
7:     end for
8:     return index of the solution wih the best score
9: end function
```

Crossover

Genetic crossover is a fundamental genetic operation used in every genetic algorithm. For the basic algorithm, I will use random midpoint selection; a method which selects a random mid-point and selects the first parent's genes from in front of the mid-point, and the second parent's genes from after the mid-point. I am using this because it is the standard implementation and the

most often used; however, I test different implementations in the testing section of this report. Below is the pseudo code for the random point crossover operation.

```
Random one point crossover
1: input: solution parent1, solution parent2
2: output: solution child

3:     Create a new solution object called child
4:     Generate a random midpoint m in the range of 0 < m < number of slots in a solution
5:     for each slot i before the midpoint m do
6:         Give child solution the slot i from parent1
7:     end for
8:     for each slot j equal to and after the midpoint m do
9:         Give child solution the slot j from parent2
10:    end for
11:    return child solution
```

Mutation

Mutation is another essential operator in genetic algorithms. Its purpose is the add the element of random mutation that is found in nature, in turn maintaing genetic diversity from one generation to another. This genetic diversity helps prevent a saturation of one high fitness individual in separate generations. When testing the basic algorithm, I will mutate 20% of the child population (although each individual has a 20% chance of being mutated, so the 20% of the whole population is an estimate and subject to random chance). I will then mutate 5% of the selected solutions slots. The selection of these values has no basis in testing, but purely on what I believe to be reasonable values, however, the optimal values for the final algorithm are tested for in the testing section. It is impossible to have a set mutation rate that works for every possible genetic algorithm, therefore the only real way to find the optimal value is through trial and error testing, which is something performed later in this report. Below is pseudo code for the mutation operator used in the basic algorithm.

```
Mutate operation
1: function MUTATECHILDOLD(solution child)
   input: solution object
   output: mutated solution object

2:     for i = 0; where i < population size do
3:         population[i] = new solution object
4:     end for
5: end function
```

3.4 – Algorithm support code

Alongside the main algorithm, there are elements of the program which the algorithm requires to function. These elements are all located in the timetable class and will never change for the rest of this project, even when the main algorithm changes later in the testing section. These are:

- Reading in the input file
- Outputting the final timetable
- First population initialization
- Checking stop condition every iteration
- Calculating the fitness score for an individual in a population

Reading in the input file

This section of code reads the problem variables from the input configuration file and saves it in memory for use by the algorithm. It scans the input file line by line and looks for tags (much like HTML syntax), with a tag indicating what the following lines of input are. This part of the program is very simple and doesn't require any in-depth explanation. The code itself is contained in a method called *readInput(),* which takes the input filename as an argument. The specific usage and syntax of the input file are explained later in this report.

Outputting the final timetable

After the algorithm has finished running, and a suitable timetable has been generated, it is outputted in a human readable format. This is handled by a method called *outputTimetable().* It scans each commitment allocation in the final solution timetable, and outputs it to a text file called "outputTimetable.txt".

First population initialization

For the genetic algorithm to have a starting point, an initial population must be created. It is important for this population to be random so that there is as much of the search space covered as possible. If there are large gaps in the search space not covered by the initial population, then there will be lost solutions, one of which may be the global optimum. I have therefore created the individuals of the initial population using a pseudo-random number generator. A pseudo-random number generator will ensure that there is a good coverage of the search space due to way the way in which pseudo random generators behave. Instead of a true random number sequence, pseudo random sequences tend to be more evenly spread. The final coverage will be determined by the population size (a larger population will result in a better coverage). This is handled by a method called *initialPop().* InitialPop() generates each individual in the initial population with random timeslot and room value for every commitment. Not only does this ensure a good coverage of the search space, but it also ensures that every commitment is being assigned, and being assigned only once. Below is pseudo code for the function initialPop().

Generate initial population

```
1:  function INITIALPOP()
2:      for i = 0; where i < population size do
3:          population[i] = new solution object
4:      end for
5:      for Each solution object i in population[ ],  do
6:          for Each commitment j,  do
7:              com = j
8:              ts = random timeSlot
9:              rm = random room
10:             Add slot to solution i, with genes (ts, rm, com)
11:         end for
12:     end for
13: end function
```

Stop condition check

At the end of each iteration, when the new population has been created and the previous best solution has been updated, a stop condition is checked before the start of the next iteration. This stop condition is when the previous generations best solution has a fitness score of 0 (violates no constraints), or if a maximum number of generations has been reached. The maximum number of generations is set by the instance variable "maxGenerations". This check is performed in a for loop, inside the constructor method of the timetable class.

Fitness score calculation

Calculating the fitness score for a timetable solution is arguably the most important element of every genetic algorithm. The fitness score must accurately portray how optimal a solution is, whilst allowing for a large range of values for solutions to be compared easily (e.g. a binary value of 0 or 1 is a very bad range for a fitness function as it is impossible to differentiate between all the solutions with a 0 value, or all the solutions with a 1 value).

The fitness function I have created for the algorithm is a minimising target function (meaning that the best possible fitness score is 0). For every hard constraint violated, a score of 10000 is added to the total fitness score. For every soft constraint violated a score of 10 is added to the total fitness score. This clearly biases hard constraints to have far more of a negative impact on the fitness score if violated. The purpose of doing this is to ensure that an invalid solution is never picked over a valid solution (breaks 0 hard constraints), despite however many soft constraints each solution violates. The purpose of the drastically smaller value of 10 being assigned to soft constraint violations is to allow valid solutions to be optimised (a valid solution

with less soft constraint violations will be picked over another valid solution with more violations).

Calculating a fitness score for a solution is handled by the calcFitnessScore() method. This method contains checks for all the constraints and awards an appropriate score based on the violations.

Instance variables

There are many instance variables inside the timetable class, most of which are used to control different aspects of the main algorithm. I won't go into detail here because they are mainly used for testing purposes, however, I have made comments in the code to explain them.

3.5 – Input file syntax

I have allowed my algorithm to take an input configuration file for users to input the details of their specific problem. The file is a plain text .txt file and its file path should be included as the first command line argument. The file consists of a set of tags which denote the beginning and end of a group of variables, much like the tag system used in hypertext languages such as HTML. In-between these tags are each individual variable input, as either a string or a tuple of integers and strings. Below is a list of each possible tag:

1. ::start:: - (This must be in the first line of every input file)
2. ::ts:: - (This denotes the section defining each timeslot)
3. ::c:: - (This section defines each academic commitment)
4. ::m:: - (This section defines each module)
5. ::r:: - (This section defines each room available room)
6. ::end:: - (This denotes the end of the file)

1. The start and end tag must be present in a valid input file.
2. Entries in the ::ts:: section allow each timeslot to be specified in the form of a tuple where index 0 is an integer indicating the day a slot is in, and index 1 is a string indicating the start time of each slot.
   For example, [1, 09:00] is an input specifying the timeslot is on the first day and starts at 09:00.
3. Entries in the ::c:: section allows for each educational commitment to be specified in the form of a tuple. Index 0 of the tuple is a string indicating the name of the module being taught in that commitment, and index 1 is another string indicating the name of the lecturer assigned to teach that specific commitment. Each individual input indicates one commitment for that module. Therefore, there may be many duplicate entries in this section if there are a lot of commitments for one module, all taught by the same lecturer.
   For example, [cm0000, ACJ] is an input specifying the commitment is for a module cm0000 and the lecturer assigned to it has the intials ACJ.

4. Entries in the ::m:: section allows for each module to be specified in the form of a tuple. Index 0 of the tuple is a string indicating the name of the module, and index 1 is another string indicating the name of the lecturer assigned to teach that module. Multiple entries for one module is necessary if more than one lecturer is allocated to teach it.
   For example, [cm0000, ACJ] is an input specifying the module is named cm0000 and the lecturer assigned to it has the initials ACJ.
5. Entries in the ::r:: section allows for each room to be stated in the form of a single string denoting the name of the room.
   For example, room1 would denote that there is a room named "room1".

Simple example file

```
::start::
::ts::
[1,09:00]
::ts::
::c::
[cm0000,1422447]
::c::
::m::
[cm0000,1422447]
::m::
::r::
room1
::r::
::end::
```

I intend for my algorithm to only solve on a weekly basis, however, it is possible to design the inputted configuration file in a way that can create timetables for longer periods i.e. for a whole academic term. To do this, just add extra weeks by incrementing the days for each consecutive week e.g. week 2, day 1 would be day 6 if we are working with a 5-day week.

3.6 – Output timetable format

After the algorithm has generated a suitable solution timetable, a text file is generated in the directory where the run.java file is located. This text file contains the solution timetable in a human readable form and a breakdown of the number of hard and soft constraint violations for the timetable. For the purposes of testing it also includes the best fitness score and the generation on which it was achieved, the maximum allocated heap memory and the total time taken to complete. Below is a very simple example of the file format.

Figure 11

```
---------- Fitness Score / Generation ----------
Fitness score of: 0, achieved on generation: 67
---------- Maximum JVM memory usage ----------
Maximum memory = 1864192KB
```

---------- Time taken to run ----------
Time taken = 50878ms
---------- Violated Constraints ----------
Fitness Score = 0
Hard constraints: 0
Soft constraints: 0

---------- Timetable ----------
-------- Day 1 --------
Time: 10:00, Room: room1, Module: cm0000, Lecturer: l1
Time: 11:00, Room: room1, Module: cm9000, Lecturer: l3
Time: 12:00, Room: room1, Module: cm0000, Lecturer: l1
Time: 13:00, Room: room1, Module: cm0000, Lecturer: l1
Time: 14:00, Room: room1, Module: cm0300, Lecturer: l1
Time: 15:00, Room: room1, Module: cm9000, Lecturer: l3
Time: 16:00, Room: room1, Module: cm0300, Lecturer: l1
Time: 17:00, Room: room1, Module: cm0040, Lecturer: l2
-------- Day 2 --------
Time: 10:00, Room: room1, Module: cm9000, Lecturer: l3
Time: 11:00, Room: room1, Module: cm2000, Lecturer: l2
Time: 12:00, Room: room1, Module: cm0040, Lecturer: l2

3.7 – Using the application

I have decided to develop the application in Java. I had the choice between four languages that I have experience with: Java, Python, C++ or Matlab. Python has by far the slowest performance of the four so I didn't want to use it. C++ or Matlab would have been the best in terms of speed, however, I am unfamiliar in Matlab and have been told that it is easy to write very inefficient Matlab code, and I am also not confident enough in C++ to avoid memory leaks and inefficient code. Therefore, I have decided to use Java as it has better performance than scripting languages like Python, and Java is my most comfortable language to develop in. As I'm writing the main program for this project in Java, it makes sense to design it in an object-oriented manner. Therefore, the final program consists of four java class files: run.java, timetable.java, slot.java and solution.java.

Run.java

This class file contains the *main* method which is required by java to run programs, and this is its only function. The main method creates a new instance of a timetable object, which in turns runs the constructor method of a new timetable object, causing the main algorithm to run.

Timetable.java

This class file contains all the methods and instance variables for running the main program. Creating an instance of the timetable class causes its constructor to run, which in turn runs the main program.

Slot.java

This class file is used to create slot objects. A slot object represents the genes for a specific timeslot/room/commitment combination. A list of slot objects contained in a solution object defines that solution "genes". Each slot object contains three instance variables used to save the

timeslot, room and commitment. It also contains a setGenes() and copy() method, which are used to set the genes and copy itself in the form of a new slot object respectively.

Solution.java

This class file is used to make solution objects. Each separate individual in a population is a unique timetable solution, which is represented as a solution object. It contains a list of slot objects, the solutions last calculated fitness score and several setter and getter methods for these instance variables.

Once all the above .java files have been compiled, the program can be run using the command:

java run config.txt

The run element refers to the run class file, and the config.txt element refers to the name of the input text file. It is important that all the class files and the configuration file are in the same directory.

3.8 – Implementation Issues

During the development process, two major issues arose. The first issue was a coding mistake that caused the genes for every individual in a population to change at seemingly random times, even when not being explicitly modified in the code. This issue was caused by the way I was copying objects between different generations. There were two points in my code (a getter in the solution class and an operation in my mutation class) that were passing references when they should have been copying the objects contents and returning a new object. This was causing multiple individuals in a population to all contain a reference to the same group of slot objects, effectively making them the same individual. This mistake set me back over two weeks, but also forced me to learn more in depth debugging techniques, such as the use of the debugger module in the IntelliJIDEA IDE. I discovered the bug after stepping through a call to the solution object and noticing multiple identical instances stored in memory, this made me think about what could possibly cause that scenario, with the obvious answer being that references were being passed somewhere that they shouldn't.

The second major implementation issue was a bug in my room feature code. I had initially planned to include room features (room size, room type etc.) into my algorithm, however after fixing the aforementioned reference bug, another bug appeared that was causing the room constraint checks to either not work (didn't register a room feature clash), or it did work but almost doubled the run time. After attempting to debug it with no success, I decided to just remove the functionality due to the lack of time to complete the project. Given more time, a fix of this bug would be the first addition I would make to the algorithm.

4 – Algorithm improvements

4.1 – Hill climb operators

Hill climbing[23] is an optimisation technique which belongs to the local search family. I will use it to locally optimise a specific individual solution or set of solutions. Instead of the normal local search method of exploring neighbouring solutions and selecting the best, a hill climb search incrementally changes a single element of the solution and evaluates the solutions fitness score after each change. If the fitness score has improved (got lower), then the change is saved. Normally, this process continues until a better solution cannot be found, making it a full hill climb search.

I will be implementing two separate hill climb operators, one for each of the two changeable parts of the solutions genes: the room and the time slot. I will implement the hill climbing operators in four different ways: on the best solution in the population, on the best N solutions in a population, on every solution in the population and on a random selection from the population. This section of the algorithm has one of the highest costs to run, with a time complexity of $O(n^2)$. This will make it time consuming to run on a high number of solutions, meaning that I will be restricted to the percentage of the population I can run it on. I have tried my best to find a more efficient method of designing this method, but due to time constraints I have had to settle with this time complexity.

I aim for the implementation of a hill climb operator to give the genetic algorithm a "push" towards a more valid population. Small improvements to individuals in each generation should have the effect of narrowing down the search space search to areas with more promising solutions by allowing a slight increase in convergence speed. This should have the effect of increasing the fitness score of the final solution.

I am testing each implementation to conclude on the best method. Instead of completing a full hill climb search, I will be stopping after the first improvement has been made. I have decided to do this for two reasons: firstly, completing a full hill climb search will be extremely expensive, especially when there is a high complexity timetable problem that has many possible time slots/room combinations. Secondly, completing a full hill climb search would prematurely converge the whole population (when hill climbing the best solutions in a population) as a small number of individuals would coverage on a local optimum too rapidly. This would result in a population saturation of non-global optima solutions and cause large amounts of gene loss.

Below is pseudo code for my implementation of both operators.

Timeslot hill climb operator

```
1:  function HILLCLIMBTS(solution child)
    input: solution object
    output: solution object

2:      for each slot i in child do
3:          for each possible timeSlot j do
4:              Change the timeSlot gene in slot i to timeSlot j
5:              Evaluate the fitness of the child solution with the changed slot i
6:              if new fitness score < old fitness score then
7:                  Save the changes to slot i
8:                  Break inner for loop
9:              else
10:                 Revert gene changes to slot i
11:             end if
12:         end for
13:     end for
14:     return child
15: end function
```

Room hill climb operator

```
1:  function HILLCLIMBROOM(solution child)
    input: solution object
    output: solution object

2:      for each slot i in child do
3:          for each possible room j do
4:              Change the room gene in slot i to room j
5:              Evaluate the fitness of the child solution with the changed room i
6:              if new fitness score < old fitness score then
7:                  Save the changes to slot i
8:                  Break inner for loop
9:              else
10:                 Revert gene changes to slot i
11:             end if
12:         end for
13:     end for
14:     return child
15: end function
```

4.2 – Mutation rate ramping

This is a feature that I have designed and implemented myself, and after searching have subsequently not been able to find mention of a similar technique being used before in any of the scientific papers that I have found. This feature waits until a fitness score plateau is detected (a better fitness score has not been found in many consecutive generations), and begins to steadily increase the rate of mutation, hence why I have called it "Mutation ramping". After running for many generations (plateau has existed for a very large amount of generations), mutation rates reach 100% and has the effect of transforming the last generations into a random search. The

idea behind implementing this kind of feature came when I noticed that after a long plateau, a very large percentage of the population had the exact same genes. As a result, any further generations of individuals would take too long to improve as the default mutation rate clearly wasn't causing an increase in fitness scores. Therefore, by incrementally increasing the mutation rates, there is a higher chance that a solution is found which allows the plateau to be broken. After a better solution is found, the mutation rates revert to their default values and the algorithm continues as normal. I hope that mutation ramping will provide a way of speeding up the escape from plateaus observed in test cases of large complexity.

I am testing the effectiveness of this technique in section 5 of this report. I have no basis regarding what to expect as I couldn't find a previous approach which has included a similar technique. Therefore, it is possible that this functionality has no effect or decreases the chance of breaking out of a plateau. I will be testing different settings for the mutation ramping, including the number of consecutive generations of similar fitness scores required to begin ramping the mutation rates and the amount by which the mutation rate is increasing each time. Below is pseudo code for my implementation of "Mutation ramping".

Mutation rate ramping

```
1:  function MUTATIONRAMPING()
2:      if this generations best fitness score = last generations best fitness score then
3:          if ramping counter has all ready been started then
4:              if ramping counter = ramping interval then
5:                  Increase mutation rates by specified amount
6:                  Reset ramping counter
7:              else
8:                  increment ramping counter
9:              end if
10:         else
11:             start ramping counter
12:         end if
13:     else
14:         if was previously ramping last generation then
15:             Stop ramping counter
16:             Return mutation rates to default settings
17:         end if
18:     end if
19: end function
```

4.3 – Roulette wheel selection

As described in section 2.1.1, roulette wheel selection is an alternative to tournament selection. I have implemented it in a way that enables it to be used with a minimising target function. I have compared the two methods of selection in section 5. Below is pseudo code for how I have implemented roulette wheel selection.

Roulette wheel selection

```
1:  function ROULETTESELECTION()
2:      Iterate through population, and find the largest fitness score
3:      for each individiual i in population do
4:          Inverted score = (Largest fitness score - individual i fitness score) + 10
5:          Save inverted score for individual i
6:      end for
7:      invertedSum = Sum all of the populations inverted fitness scores
8:      for each individiual i in population do
9:          Selection weighting = individual i inverted score / invertedSum
10:         Save selection weighting for individual i
11:     end for
12:     Randomly generate a random number m where 0 <= m < 1
13:     for each individiual i in population do
14:         Subtract individual i selection weighting from random number m
15:         if m <= 0 then
16:             Return index of individual i
17:         end if
18:     end for
19: end function
```

As seen above in line 4, the inverted score is calculated by subtracting the individual's fitness score from the largest present in the population. To prevent individuals who have the largest score from getting a score of 0 and becoming impossible to select, I add 10 to every solution so that individuals no longer have inverted scores of 0, but of 10.

4.4 – Types of crossover

There are many different methods of implementing a crossover operation into a genetic algorithm, many of which were mentioned in section 2.1.1. To improve the basic algorithm, I will be testing four different types of crossover implementation:

- Fixed mid-point
- Random point
- Random two point
- Uniform

The first three are common implementations, where I expect the fixed mid-point to perform the worst. This is because having a fixed mid-point increases the speed of which individuals in a population become the same due to the crossover operator mixing the exact same halves of genes

every time. Random and Random two-point should perform better than the fixed mid-point because the same halves of genes aren't being crossed over every generation, leading to the likelihood of two individuals becoming identical being reduced. Uniform crossover has interesting advantages such as helping to prevent premature convergence by providing the most mix of genes. I believe that these advantages, coupled with the most random split of genes possible, will mean that this operator performs the best out of the four. Below is pseudo code for my implementation of Uniform crossover.

```
Uniform crossover
1: input: solution parent1, solution parent2
2: output: solution child

3:     Create a new solution object called child
4:     for each slot i in the child object do
5:         50% chance of giving child solution the slot i from parent1
6:         50% chance of giving child solution the slot i from parent2
7:     end for
8:     return child solution
```

4.5 – Variations in operator/genetic parameter values

In addition to all the new functionality, it is possible to improve the basic algorithm by changing the values for which all the operators and other evolutionary variables use. I am modifying the values of the following when testing in section 5:

- Population size
- Number of contestants used in tournament selection
- Percentage of each generation mutated
- Percentage of slots mutated for each selected individual
- Number and type of individuals subjected to the hill climb operators
- Mutation ramping variables
    - Number of consecutive unchanging generations to trigger mutation ramping
    - Amount of increase in the percentage of population mutation
    - Amount of increase in the percentage of slots for a selected individual

5 – Results and Evaluation

With genetic algorithms, it is difficult to calculate the best parameter values theoretically. As a result, we must find the best combination of values and additional functionality through a trial and error form of testing. In this section I display how the final algorithm parameters and operators were settled on, and demonstrate the capabilities of my final algorithm in terms of its ability to solve real world timetabling problems and its performance on more complex test cases.

I have performed all the testing on the same machine, with the only non-necessary process being the algorithm. Below is the Hardware and software specifications for the machine used.

Hardware

- **Model:** iMac (21.5-inch, Late 2013)
- **Processor**: 2.9 GHz Intel Core i5
    - *Speed:* 2.9GHz
    - *Number of Processors:* 1
    - *Number of cores*: 4
- **Memory (RAM)**: 2 x 4GB 1600 MHz DDR3 (8GB total)
- **Graphics:** NVIDIA GeForce GT 750M
- **L3 Cache:** 6MB

Software

- **Operating System:** OS X El Capitan (Version 10.12.4)
- **Kernel Version:** Darwin 16.6.0
- **JVM:** Java HotSpotI 64-Bit Server VM (25.101-b13, mixed mode)
- **Java:** version 1.8.0_101, vendor Oracle Corporation

5.1 – Testing goals and methodology

My test experiments will be split into two sections. Firstly, I will perform experiments to determine the best parameter values / additional functionality for the following:

- Mutation rates
- Crossover methods
- Selection methods
- Mutation ramping
- Hill climb operator

Secondly, I will "stress test" the final algorithm by running it with a large, high complexity test case. I will additionally determine its ability for solving real-world problems by running it on a previous year's scenario that occurred in the computer science department of Cardiff university.

5.1.1 – Test cases

A vital part of my testing is the test cases I will be using, and how they are created. Different features of the algorithm can use test cases of different size and complexity, and this is necessary due to the time constraints of this project. Ideally, I would like to test each feature on a large/complex test case and run it for a long-time period. However, if I can determine the difference in feature values by using a simpler test case and running it for a shorter time, than this is clearly beneficial when under working under the time constraint of this project. Shorter test cases will also allow me to run each experiment multiple time and calculate an average score, something which I consider more of an advantage then testing on larger test cases, especially with the element of randomness present in my algorithm (mutation, initial population generation etc.). For that reason, I will use three different types of test cases: small, medium and large, each with the following parameter values.

Figure 12

Test cases	Number of students	Number of modules	Year groups	Total commitments for all modules	Number of rooms	Number of lecturers
Small	80	20	4	100	5	20
Medium	300	20	5	200	6	30
Large	600	60	6	10	15	40

The difference between the size of the test cases is due to their complexity. I have increased the complexity in the following ways.

- Increasing the number of students and modules
- Increasing the number of year groups, causing the potential for more student/lecturer clashes
- Increasing the total commitments more than the increase in the number of rooms, causing there to be less potential ways to formulate a valid solution. I have however increased the number of rooms in a bid to maintain valid solutions.
- Increasing the number of lecturers

I aimed to design the values of each test case so that they would be like real world solutions. For example, the medium and large test cases rely on an increase in rooms to manage the complexity when increasing the amount of commitments. Fewer rooms is a much more likely scenario than each module having a large amount of commitments per week. Ultimately, I settled on the above values after some calculations and trial and error. I did research benchmarks and test cases used in previous university course timetabling solution attempts documented in other scientific papers, however many of the previous solutions are very outdated, causing their benchmarks to also be outdated. They were based on the limitations of the hardware of their time and don't translate well to the specifications of modern hardware. Therefore, I concluded to design my own test cases.

Due to nature of my input file syntax, and the size of the medium and large test cases, it would be impractical to create each test cases by hand. Therefore, I wrote a python script to automatically generate a test case based on each parameter given in the above table. This script will be included in the appendix. I require multiple of the same sized test cases, so therefore require each test case to be generated in a random way, allowing for variety in the same sized test case whilst still aiming for the test case to be solvable. This is a challenge that I have struggled with as it is impossible to determine if the algorithm is returning a poor score because of its design or because of a poorly constructed test case. I manage to overcome this uncertainty to an extent by mathematically approximating the difficulty of the test cases based on its values, and by controlling the randomness to values within a range which should result in more viable test cases. For example:

Small test case

- Possible allocations = time slots * rooms
- Allocation and commitment overlap = possible allocations – commitments
- If the overlap is positive then there are enough ways to allocate the commitments. If it is negative then the test case is impossible.

There are other factors that also need to be considered, such as lecturer to module ratio. This will not impact as much on the small test cases, but on the medium and large it is a major determiner of complexity. The medium and large test cases rely on more rooms to decrease the complexity to realistic levels. However, this becomes a problem if the module to lecturer ratio is too low. It leads to more lecturer clashes as there are many different modules occurring in the same time slots due to many rooms being available.

Some test cases may still be unsolvable, and there is no way of knowing whether this is the case. However, this is not an issue for the first stage of testing as a difference in fitness score is still possible to be observed, and the validity of results is unchanged. The python script aims to introduce randomness into the test case generation whilst still aiming for solvable solutions using the following method:

Test case script: genTest.py

```
1:  input: Time slots per day, Days per week, Number of students, Number of modules,
    Groups, Total commitments, Number of rooms, Number of lecturers
2:  output: newConfig.txt text file

3:      Generate all lecturers, modules, students, rooms
4:      Assign lecturers to modules
5:          for every lecturer do
6:              if There is an empty module then
7:                  Assign lecturer to the empty module
8:              else
9:                  Assign lecturer to a random module
10:             end if
11:         end for
12:     Generate commitments y for each module within a range of:
13:         x = Total commitments / modules
14:         x-2 <= y <= x+2
15:     Students/modules are randomly, but equally distributed to each year group
16:     Students from each year group are assigned to modules in the same group
17:     Output to config file in required syntax
```

5.2 – First testing stage: finding the optimum algorithm

The overall goal of the first stage of testing is to optimize the algorithm to generate time tables with the lowest fitness score possible. The algorithm will be optimized by identifying the best parameter values and combination of additional functionalities. One way of constructing experiments to achieve this goal is by doing an exhaustive test which compares every possible combination of algorithm. This is impractical due to the amount of variation that my algorithm could have. Therefore, I am testing the algorithm in an iterative manner, one feature at a time. For example, I start with the mutation feature; testing all its parameters (percentage of population mutated each generation, percentage of slots mutated for each selected individual) and determining the best parameter values based on the results. I then take the best value found for mutation, and test the next feature: crossover method. Using this method, by the end of testing the final feature (hill climbing operator), I will be left with the optimal algorithm and can go on to test it as the final version.

When testing the features: Mutation, Crossover and selection, I test each dataset five times and record the best achieved fitness score (lowest score) and the average of the five achieved fitness scores. I do this because, when using small test sets in combination with a relatively low maximum generation, only taking one result per test set would be unreliable. The low number of max generations (necessary because of the projects time constraints) causes a fluctuating score for the same algorithm because not enough time is given to allow it to properly converge on an optimum. I am only testing the final fitness score for these three features because of the low complexity of the test case. Recording time for use in comparisons is unreliable due to randomness in the convergence rate, and other non-related factors such as other background processes running on the testing computer. In addition, time taken is not a major factor when determining how optimal a university course timetable solver is, due to the lack of importance of the time taken to run the algorithm (as aforementioned in the report).

When testing the final two features: mutation ramping and hill climbing, I changed the method of testing slightly. I decided to test these features against the medium sized test cases because at this stage of testing the small size test cases became too easy to solve, and didn't show enough fitness score variation for comparative analysis. As I required more complex test cases, it became infeasible to test each one five times and take the average score from the five results. Therefore, I increased the max generations to account for any randomness in the convergence rate and recorded the final fitness score and the generation at which this score was found (where the algorithm reached a plateau). This was useful for comparing the convergence rate of different variants of the algorithm, and helped me decide on which variant was the best when the scores achieved were similar.

5.2.1– Mutation

I initially tested mutation rates. I began with mutation rates because it is (alongside crossover and selection) the most fundamental aspect of any genetic algorithm. To test mutation rates, I created five different variations of the basic algorithm, each with a differing value of a mutation rate of some kind.

Variant	Percentage of population mutated (%)	Percentage of slots mutated (%)
M1	5	5
M2	10	5
M3	20	5
M4	10	10
M5	10	20

I then tested each variation on five different small size test cases, with each test case being tested five times and the average fitness score being recorded. Below are the details of the testing, and the results achieved:

Algorithm parameter values

	Crossover	Selection	Max generations	Population Size	Mutation ramping	Hill climbing
Feature values	Random 1 point	Tournament: 3 contestants	100	1000	OFF	OFF

These percentage mutation rates have been chosen to cover a sensible range of values, whilst allowing a large enough gap in between the different variants to demonstrate a difference in their results. I believe that any mutation rate higher than 20% of the population is too high, leading to a potential loss of optima and a significant slow in convergence. Therefore, 20% will be the maximum value tested.

Results – Figure 13

Data in bold identifies it as the best of its type for that test case

Test case	M1		M2		M3	
	Best	**Average**	**Best**	**Average**	**Best**	**Average**
Small1	*80*	**14066**	**50**	*22068*	*10070*	*18064*
Small2	*20090*	*34084*	**10110**	**24076**	*20100*	*28086*

Small3	*30060*	*38082*	*10050*	**20084**	**70**	*30076*
Small4	*80*	*12072*	*10060*	*24062*	**40**	**10072**
Small5	*10050*	*32052*	**100**	**26084**	*10090*	*28074*

Test case	**M2**		**M4**		**M5**	
	Best	**Average**	**Best**	**Average**	**Best**	**Average**
Small1	**50**	**22068**	*70*	*28072*	*90*	*22088*
Small2	*10110*	**24076**	**10030**	*30070*	*10100*	*26082*
Small3	**10050**	**20084**	*10070*	**20084**	*20030*	*32048*
Small4	*10060*	*24062*	*10090*	*26076*	**70**	**14080**
Small5	**100**	*26084*	*10080*	**26080**	*10060*	*30070*

Analysis and conclusions

Variant M2 achieves the most best and average scores over the five test cases. M2 performs the best in the first half of testing (M1, M2, M3), and performs the best in the second half of testing(M2,M4,M5). Therefore, I will be using the parameter values of M2 in the algorithm that I progress with to further stages of testing.

5.2.2 – Crossover

The crossover operation is another essential operation of most genetic algorithms. For this reason, I tested it before any of the more niche algorithm features. To test the crossover operation, I created four different algorithms variants, each with a different method of performing the crossover operation.

Variant	**Type of crossover**
C1	Random one point
C2	Random two point
C3	Fixed mid-point
C4	Uniform point

I tested each variant against the five small test cases used to test mutation rates, with each test case being tested five times, and the best and average score being recorded for these five runs.

Algorithm parameter values

	Mutation	Selection	Max generations	Population Size	Mutation ramping	Hill climbing
Feature values	10% of pop, 5% of slots	Tournament: 3 contestants	100	1000	OFF	OFF

Results – Figure 14

Data in bold identifies it as the best of its type for that test case

Test case	C1		C2		C3		C4	
	Best	**Average**	**Best**	**Average**	**Best**	**Average**	**Best**	**Average**
Small1	*50*	22068	*30*	*46*	*170130*	*198106*	**0**	**24**
Small2	10110	24076	*40*	*52*	*190120*	*202120*	**0**	**18**
Small3	10050	20084	*30*	*4048*	*190110*	*208096*	**0**	**32**
Small4	10060	24062	*40*	*4052*	*200110*	*210116*	**10**	**20**
Small5	100	26084	*30*	*2056*	*180120*	*212106*	**20**	**30**

Analysis and conclusions

Firstly, variant C4 (Uniform point crossover) performed the best for all the test cases. I expected Uniform crossover to perform the best out of all the methods, but was surprised as to the extent of performance increases found. Uniform crossover is known for helping to control premature convergence, so I assumed that it would perform better, but would also take a lot longer to convergence to an answer. This hypothesis proved true, but not to the extent that I expected. Variant C1, C2 and C3 converged to their best scores around the 50th generation, whereas C4 (uniform crossover) converges on average at generation 90. This shows its capability in preventing premature convergence. However, this is a significant increase in time taken to produce a solution, but I believe that the improvement in fitness score, and the control over premature convergence that uniform crossover provides for more complex test cases outweighs the time increase. Therefore, I am selecting uniform crossover to take further as my crossover method of choice.

5.2.3 – Selection

Selection is another fundamental part of any genetic algorithm. Although I am only testing two different methods (roulette and tournament), there are many different possible values for the number of contestants used in tournament selection. Therefore, I am testing four values in the

range of two and eight contestants. Any more than eight contestants will create too high of a selection pressure, and will lead to premature convergence and a loss of gene variation.

Variant	Type of selection
S1	Roulette
S2	Tournament – 2
S3	Tournament – 3
S4	Tournament – 5
S5	Tournament – 8

I am testing each algorithm variant on the same five small test cases as used in mutation and crossover testing. I am testing each variant on a test case five times, and will record the best and average fitness scores. The first results didn't provide enough of a difference, so I extended the testing to a further medium sized test case, where there was only one run with the fitness score and plateau generation being recorded.

Algorithm parameter values

	Mutation	Crossover	Max generations	Population Size	Mutation ramping	Hill climbing
Initial test	10% of pop, 5% of slots	Uniform point	100	1000	OFF	OFF
Second test	10% of pop, 5% of slots	Uniform point	400	1000	OFF	OFF

Results – Figure 15

Data in bold identifies it as the best of its type for that test case

Initial

Test case	S1		S2		S3		S4		S5	
	Best	Average	Best	Average	Best	Average	Best	Average	Best	Average
Small1	*180110*	*186100*	*70100*	*84084*	*10*	*20*	***0***	***6***	***0***	*14*
Small2	*160100*	*172120*	*40080*	*86092*	*30*	*40*	***0***	*18*	***10***	*16*
Small3	*180130*	*192108*	*80110*	*90112*	*20*	*34*	***10***	***20***	***10***	***20***
Small4	*150140*	*170122*	*70080*	*84096*	*20*	*26*	***0***	***6***	*10*	*20*
Small5	*180100*	*190110*	*90110*	*110100*	*20*	*30*	***0***	***14***	*10*	*22*

Second test

Test case	S3		S4		S5	
	Best	**Plateau generation**	**Best**	**Plateau generation**	**Best**	**Plateau generation**
Medium1	*180170*	*140*	*190160*	*95*	*190180*	*57*

Analysis and conclusions

After the results obtained from the first part of the testing, it was clear that tournament selection performed much better than roulette selection. However, a distinction couldn't be made between the tournament values due to the similarity in results, as well as the higher contestant variants having an advantage as they could converge faster due to a higher selection pressure. Therefore, I tested S3, S4 and S5 on a medium sized test case to further differentiate between the different values. I also increased the maximum generation to 400 to remove any convergence speed advantage. The results from the second test showed that there was not much of a difference in the fitness score achieved, but a clear correlation between tournament contestants and convergence speed emerged. Therefore, I have decided to select variant S3 (three contestants) as the selection method to take forward into my final algorithm. This is due to a marginally better fitness score being achieved, but also because it reached a plateau much later than the other variants. This will be an advantage when used with more complex test cases as any control over premature convergence will be advantageous in exploring more of the search space, which will become very large.

5.2.4 – Mutation ramping

Mutation ramping is the first feature being tested that is not a typical element in genetic algorithms. As aforementioned, this was a concept that I have implemented myself and which I could not find evidence of in any existing literature that I have found. Therefore, I am unsure as to what to expect. I predict that this feature will be useful for complex cases to speed up the escape from plateaus. To test mutation ramping I will be testing five different variants, each with a differing mutation rate increases interval.

Variant	Rate interval
R1	0 (No ramping)
R2	5
R3	10
R4	15
R5	20

The rate values were chosen to show improvement between a more frequent ramping, and a less frequent. I did not test a rate higher than 20 due to the max generation cap of 300. Any higher

than 20 would make the mutation ramping to occur too slowly and any escape from a plateau may not be the result of the mutation ramping. There are three changeable parameters for mutation ramping: Rate interval (amount of generations of identical fitness scores in a row), Population mutation percentage increase and Slots percentage increase. I am, however, only changing the rate interval parameter. This is because changing this parameter has the same effect as changing the other two (e.g. decreasing the rate interval has the same effect as increasing the mutation rate increases because the increases occur more often). I am testing each variant on three different medium sized test cases, and recording the best score achieved and the generation at which the algorithm plateaued.

Algorithm parameter values

	Mutation	**Crossover**	**Selection**	**Max generations**	**Population Size**	**Hill climbing**
Feature values	10% of pop, 5% of slots	Uniform point	Tournament: 3 contestants	300	1000	OFF

Results – Figure 16

Data in bold identifies it as the best of its type for that test case

Test case	**R1**		**R2**		**R3**		**R4**		**R5**	
	Best	**Plateau gen**	**Best**	**Plateau gen**	**Best**	**Plateau gen**	**Best**	**Plateau gen**	**Best**	**Plateau gen**
Medium1	*190170*	*150*	*190180*	*142*	***180180***	*142*	*200180*	*143*	*190170*	*125*
Medium 2	*90200*	*162*	*80180*	*152*	*100160*	*153*	*80180*	*152*	***80160***	*144*
Medium 3	*150150*	*149*	*150190*	*146*	*170170*	*136*	*160160*	*155*	***150120***	*151*

Analysis and conclusions

The results are similar and don't show much on the surface. However, the purpose of mutation ramping was to speed up the escape from plateaus, and this seems to have been the case. The plateau generation for R1 (no ramping) was higher than that of the variants with mutation ramping on. This shows that mutation ramping had a positive effect, by decreasing the amount of generations needed to achieve the same score, which in turn shows that plateaus were escaped faster. However, I observed that as plateaus were hit, and mutation rates began to ramp, the generations were taking progressively longer to complete. This is most likely due to the increase in individuals being mutated, causing the mutation algorithm having to be run on more individuals for each successive ramp. Although this increase in time was observed, it was not large enough to record and give evidence on, and was nowhere near the time saved by reaching the final fitness score in fewer generations.

I have concluded that variant R5 will be carried on as my mutation ramping variant. This is because, on average, it reached its final fitness score in the fewest generations, whilst still attaining some of the highest final scores.

5.2.5 – Hill climb operator

The hill climb operator is the last feature that I will be testing. I expect that adding hill climb in any variant will have a positive impact on the final fitness score due to the guaranteed improvement made each generation. However, I also expect to see a much faster convergence (especially in methods that work on the best individuals) due to the improvement of individuals, causing lower fitness score solutions to emerge a lot faster than if the genetic algorithm was running without the help of the hill climb search operator. I am testing five different algorithm variations:

Variant	Hill climb type
H1	No hill climbing
H2	Best solution
H3	Best 5 solutions
H4	Random 10 solutions
H5	Whole population

The value of five for the H3 variant was decided on as to show a difference between the other best solution variant H2, but without increasing the time taken to run the algorithm by too much. The value of ten for H4 was decided on to be high enough so that a difference emerges with the small maximum generation cap, but not so high that it takes too long to complete each test.

I am splitting this section into three individual tests. I will initially test each variant on a small test case, measuring the final fitness score achieved and the generation this score was achieved. This will show a brief comparison to determine whether adding the hill climb operator improves on the algorithm with no hill climb element. Secondly, I will test each variant on two medium sized test cases. Thirdly, after analysing the result data, I decided to alter the algorithm to only have a small, random amount of an individual's slots subjected to hill climbing (in contrast to performing the hill climb search on every slot of an individual). This change in algorithm will be denoted as V2 (with the original being V1). I will explain the reasoning for this in the conclusion section of this test section. Additionally, I decided to leave mutation ramping turned off to test the hill climb operators. I did this because I wanted any improvement to be solely a result of the change in hill climb operator, and not by the potential random search scenario caused by mutation ramping.

Algorithm parameter values

	Mutation	Crossover	Selection	Population Size	Mutation ramping
Feature values	10% of pop, 5% of slots	Uniform point	Tournament: 3 contestants	1000	OFF

Test case	Max generations	Hill climb V1/V2
Small1	100	V1
Medium1	200	V1
Medium2 V1	200	V1
Medium2 V2	500	V2

Results – Figure 17

Data in bold identifies it as the best of its type for that test case

Test case	H1		H2		H3		H4		H5	
	Best	Plateau gen	Best	Plateau gen	Best	Plateau gen	Best	Plateau gen	Best	Plateau gen
Small1	*10*	*96*	*0*	*3*	*0*	*3*	*0*	*7*	*0*	*2*
Medium1	*180180*	*136*	*180100*	*4*	*180040*	*3*	*210120*	*52*	*170140*	*3*
Medium2 V1	*90200*	*170*	*80160*	*6*	*70280*	*5*	*110230*	*23*	*70240*	*7*
Medium2 V2	*80180*	*161*	*70240*	*98*	*70150*	*108*	*20310*	*215*	*70100*	*68*

Analysis and conclusions

Firstly, it is clear that adding any form of hill climb operator results in a large improvement in fitness score. I expected to see an improvement, but the speed (small amount of generations required) in which H2-H5 found a perfect solution to the small1 test case was extremely significant. It is important to note that the results for the small test case we produced by the V1 version of the hill climb algorithm. The V1 algorithm version performed the hill climb operation on every slot of an individual, and this would explain the rapid convergence noticed. Although this rapid convergence is desirable, we can observe that it doesn't yield a better fitness score for the test cases: Medium1 and Medium2 v1. This rapid convergence is the result of the hill climbing operator taking over and effectively making the genetic algorithm redundant. The reason that no improvement to fitness score can be seen in the medium1 and medium2 v1 test cases if because the hill climb operator converged too fast, and converged on a local optimum and not a global one.

The V2 algorithm was created to test this hypothesis. When tested again on the medium2 test case, we see an overall improvement in final fitness score, and a much slower convergence (took longer to reach its plateau generation). This hypothesis is confirmed true when we observe the results for the algorithm H4 when tested using the V2 algorithm. Previously no result in any test section for the medium2 test case has broken the 70000 threshold, meaning that this is probably a local optimum with no other better local optimum near it in the search space. However, the fitness score of 20310 achieved shows that when the convergence caused by the hill climb is limited, and random solutions are selected to be hill climbed (not the best solutions), new optimum can be discovered in previously ignored areas of the search space. When hill climbing the whole population, it would be expected that a very good population score should be observed. However, it is likely this wasn't the case in the test results because the same 70000 local optima was converged on too quickly (because the best solutions were being hill climbed every generation), not allowing the worse solutions to find the same (likely global) optimum found by H4.

Finally, I have concluded that I keep the V2 algorithm changes, and implement the hill climb method in H4 (random individuals). This is because they both, when used in conjunction, achieved by far the best fitness score. After testing I realized that a static number of random individuals to select each generation does not scale with population size. Therefore, for the final algorithm, I have changed the algorithm to select a random percentage of the population. In the testing, I selected 10 individuals out of a population size of 1000. This means that I was selecting 1% of the population. So, this parameter value of 1% will be used in the final algorithm.

5.2.6 – First stage of testing conclusion

Each stage of my first testing section has accumulatively resulted in my final algorithm having the most optimal combination of features and parameter values. Below are the details of my final algorithm:

Figure 18

	Mutation	**Crossover**	**Selection**	**Mutation ramping**	**Hill climbing**
Feature values	10% of pop, 5% of slots	Uniform point	Tournament: 3 contestants	- Rate interval: 20 - Slot % increase: 5 - Population % increase: 10	Random 1% of population, with the V2 algorithm changes

The improvement when comparing the first basic algorithm to the final algorithm is large. When testing the small1 test case in the first test (mutation testing), the best score observed was 50 (out of five runs) and the best average score observed was 14066. When comparing this to the final

stage of testing (hill climb operator testing), where a perfect score of zero was observed after only seven generations, the improvement is obvious.

5.3 – Second testing stage – analysing the final algorithm

Because of the first stage of testing, I have concluded on the variation that will be used for my final algorithm. In this stage of testing I have multiple aims, the most interesting being an analysis into how the final algorithm performs when solving a real-world test case, which has been taken from previous year at Cardiff university. I also aim to push my algorithm to see how it performs on very large/complex test cases, while testing the maximum heap memory used and the time taken to reach a solution. I will only be testing the heap memory because this is the section that the JVM (java virtual machine) uses to store objects and any increase in memory will be due to an increase in the number of objects being created and stored (e.g. solution and slot objects). These test results will then be used to indicate how the algorithm will scale, in terms of speed and memory usage, to much larger problems of a scale that would be impractical to implement given the time frame of this project. I will also analyse various aspects of the final algorithm e.g. convergence rate, time taken to complete one generation with respect to population size and the effect population size has on the final fitness score.

5.3.1 – Testing against real-world example

The main goal of this project was to design and build an algorithm that could solve university course timetabling problems on a scale that is present in the average university school. Therefore, I will be largely gauging the success of the project on the next element of testing.

To test the above, I requested a previous year's timetables for years: 1, 2 and 3 for BSC computer science and variants, along with available rooms and the number of students enrolled on each module from the school of computer science at Cardiff University. I received timetables for years 1, 2 and 3 for week 1 of the Autumn Semester 2016/2017. This was the most detailed information that I had access to, as any further details e.g. which students were enrolled in what module etc. not only presented ethical and confidentiality issues, but also would require the timetabling officer for computer science having to spend a large amount of time finding and consolidating all the extra information not present in the already compiled previous years' timetables. The timetables that I received (and have included in the appendix) had to be converted into a test case in the syntax of my input configuration file. Due to issues that I experienced whilst adding functionality to define room features (size and type), I have had to make the test case harder than it actually is by only using rooms with a large enough capacity to hold all of the modules. Additionally, I am only timetabling lectures (as opposed to lectures, labs and tutorials), however, this will not decrease the difficulty of the test case because labs and tutorials are held in different types of rooms to the ones I'm using. This effectively means that my algorithm could still be used in its current state to create a timetable for lectures, labs and tutorials, they would just have to be done as three separate timetables and then combined. It

should also be noted that because I wasn't given the details of which students were enrolled on each module, I had to assign them myself, with the estimate that each student was enrolled in each year group was enrolled in a maximum of five modules (because this was the most modules that I have been enrolled in for one term). Below are the parameter values of the test case that I created from the timetable documents sent by Cardiff University:

Figure 19

Years	Total number of students	Total number of modules	Total number of commitments for all modules
1	166	1	12
2	134	8	19
3	102	10	29

As I am creating a timetable for a specific university school, it is important that I copy all their hard constraints as to avoid a timetable which would be unusable for that school. In the case of Cardiff university, all schools are required to incorporate the following two additional hard constraints: there must be no commitments after 13:00 on a Wednesday and the 13:00 – 14:00 timeslot on a Friday should be left free to accommodate students requiring quiet time. Therefore, I have added these two temporary hard constraints for this section of testing to ensure realistic test conditions and so any outputted timetable would be valid for real world use.

The test was performed on the same machine as in the first section of testing, and with the only running process being the algorithm. This allowed the algorithm to use up to 100% of the CPU and have maximum possible use of RAM for its heap memory. The population size was set at 1000.

Results and analysis – Figure 20

	Fitness score	Generations taken	Time taken (ms)	Maximum heap memory used (kb)
Results	0	62	89814	932352

As can be seen from the results, a perfect timetable was generated which violated no soft or hard constraints. The timetable was generated in 89.8s, a time that is extremely practical and much faster than the stated goal of completing in under two hours. The maximum RAM used to store the heap data of the java virtual machine was 932.4mb. This value shows that it can be executed on any average machine, as most machines built in the last decade have at least 1GB of RAM. Given more time for the project to fix the bugs I was having with room features, this algorithm could easily be implemented to generate the timetables for the school of computer science at Cardiff University.

Measuring the quality of a timetable based on its constraint violations is effective, but it relies on the quality of the constraint design. Comparing the timetable generated by my algorithm and the original which was generated by a human will show the weaknesses and strengths of my constraint design. The original timetable and my generated timetable are both included in the appendix. Comparing the two shows that my timetable contained no 09:00am lectures, and ensured that students and lecturers had their commitments for the day grouped together, and not spread over the whole day. As discussed earlier in the report, no such soft constraints are taken into consideration when Cardiff university designs their timetables and this shows when the two timetables are compared in respect to the convenience of students and lectures.

In conclusion, the final algorithm handled the Cardiff test case easily by taking under two minutes to produce a perfect solution, even with the added complexity of having to use a small number of the actual available rooms (due to no room features functionality). If given more time to implement functionality allowing user inputted constraints, and room feature functionality, it is realistic to state that my algorithm could be used to increase timetable quality for Cardiff university, whilst additionally freeing up staff man power.

5.3.2 – Resource demands and scalability

The only parameter that has remained unchanged is the population size. The population size has a direct impact on the way in which the algorithm runs, not only in terms of the fitness score of the outputted solution but also in terms of the hardware resource usage and time taken to arrive at a final solution. In this section I test three different population sizes on the real-world Cardiff test case, and measure the CPU usage over time, the Heap memory usage over time and the number of different instances requiring the biggest percentage of that heap memory. I aim to show a correlation between resource usage increases and population size increase, ultimately allowing me to determine how the algorithms performance differs with changes in population size.

Figure 21

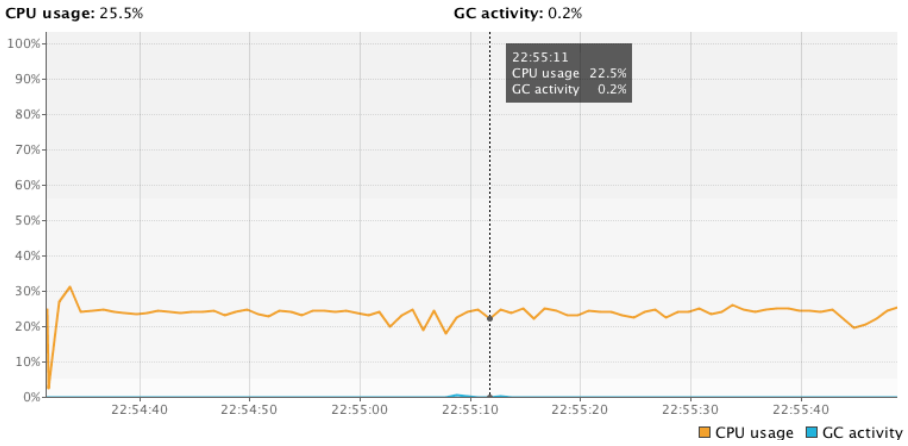
Population	Fitness score	Generations taken	Time taken (s)
100	20	80	22
1000	0	60	79
10000	0	134	180

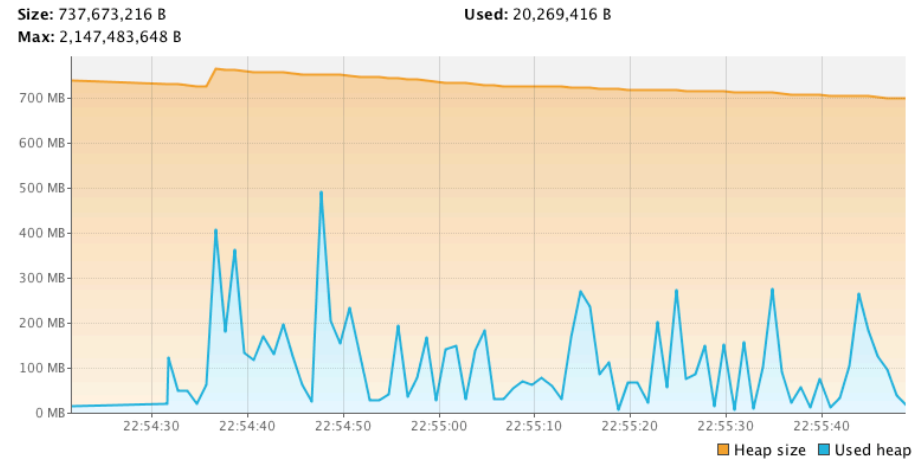
These results show two things: firstly, that an increase in population size results in a slow in the algorithm and secondly that a decrease in population size can result in a worse outputted solution. The slow caused by an increase in population size can be attributed to the increase in the number of individuals to process every generation. Additionally, because we are now hill climbing a percentage of the given population, a tenfold increase in population leads to a tenfold increase in the number of individuals being subjected to a hill climb in every generation. Therefore, because the hill climb algorithm implemented in my algorithm has a time complexity

of $O(n^2)$, an increase in the amount of times its run will have a significant impact on the time increase observed in the testing. The worse score of 20 observed in the 100-population test is a result of a much smaller search space sampling. The 100-population algorithm will cover an area of only 10% the size of the search space covered by the 1000 population algorithm, and only 1% of the search space covered by the 10000-population algorithm. This reduced search area explains a loss of possible solutions, which in the case of the 100-population algorithm included a solution that had a better fitness score than 20.

To analyse how the population size effects memory usage, I need more detailed information than just the maximum heap memory that was allocated to the Java virtual machine. This is because on startup, the Java virtual machine is allocated a set maximum heap size which doesn't vary much with the amount of the heap memory actually used by the individual java process. This heap size can increase when needed up to a default of ¼ of the maximum RAM for that machine. Therefore, I have used an application called "VisualVM" [24] to record how the used heap memory and CPU usage changes over time for each of the three population sizes.

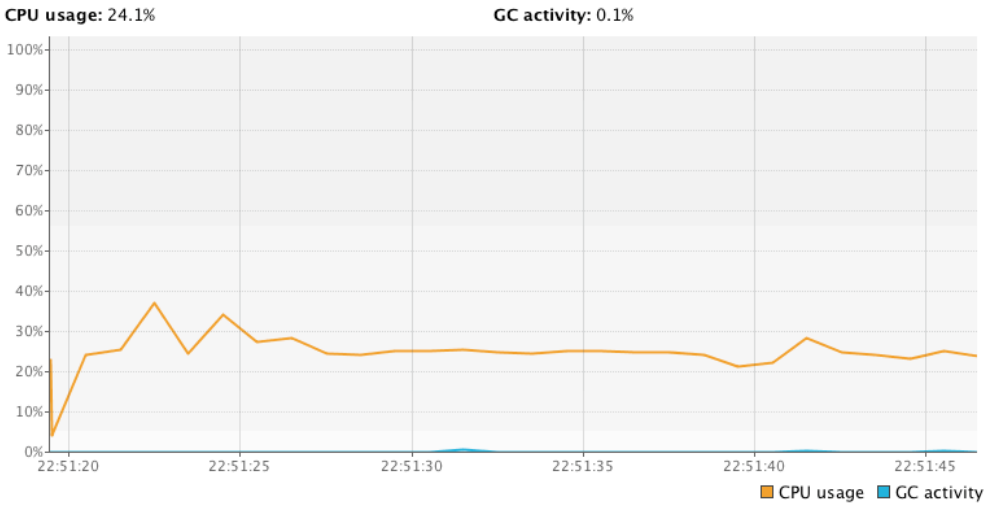
Population of size 100 – Figures 22

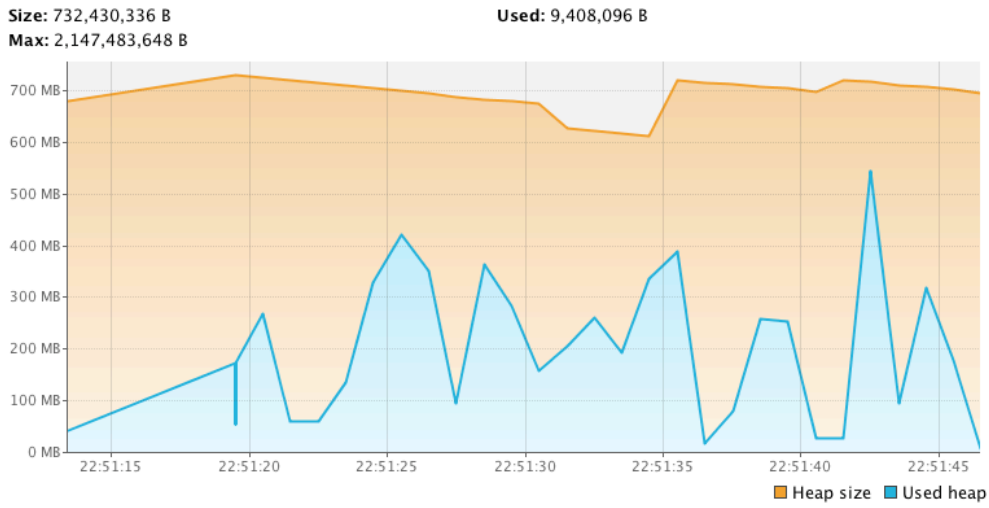




Class Name	Instances [%]	Instances	Size
char[]		8,849 (19.7%)	560,002 (16.7%)
java.lang.**String**		8,789 (19.5%)	246,092 (7.3%)
slot		5,858 (13%)	164,024 (4.9%)
java.lang.**Object[]**		1,904 (4.2%)	233,944 (7%)
java.lang.**Class[]**		1,617 (3.6%)	55,904 (1.7%)

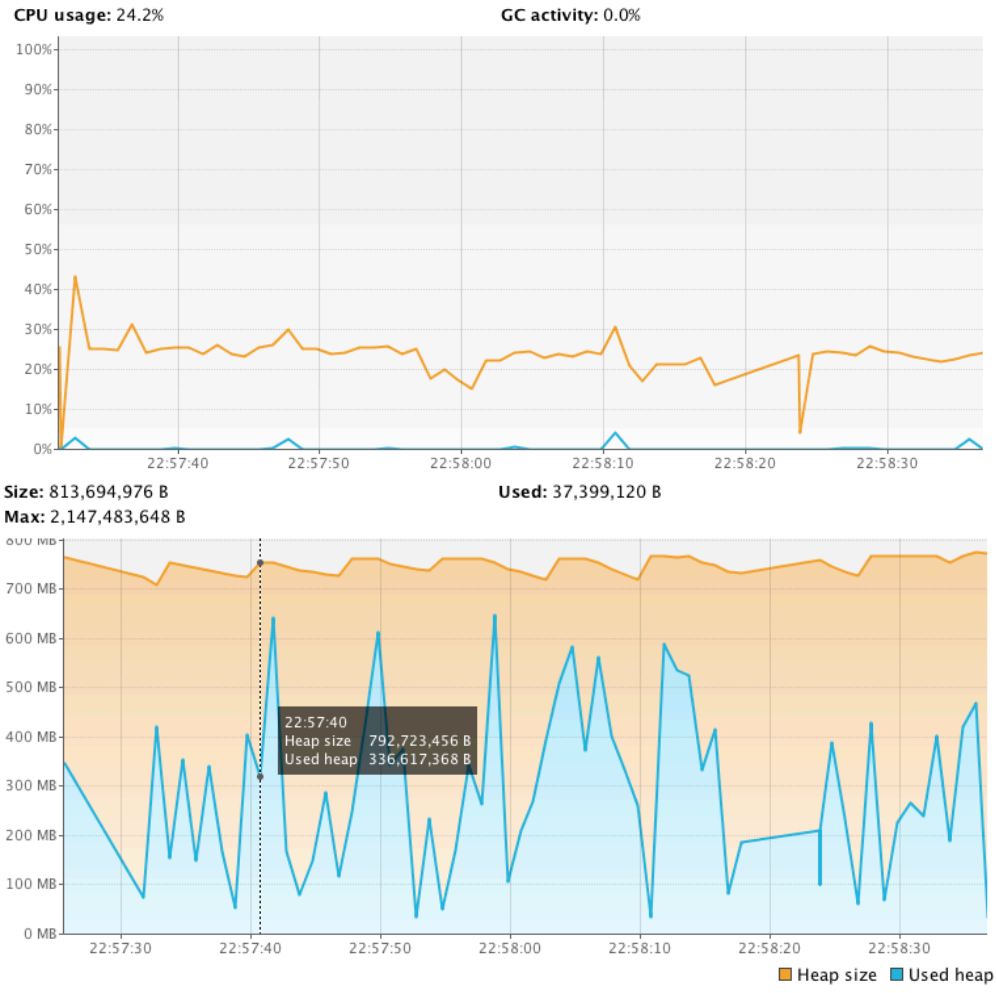
Population of size 1000 – Figures 23





Class Name	Instances [%]	Instances	Size
slot		116,067 (72.6%)	3,249,876 (46.3%)
char[]		8,807 (5.5%)	556,840 (7.9%)
java.lang.**String**		8,750 (5.5%)	245,000 (3.5%)
java.lang.**Object[]**		3,728 (2.3%)	1,382,840 (19.7%)
java.util.**ArrayList**		2,137 (1.3%)	68,384 (1%)
solution		2,000 (1.3%)	56,000 (0.8%)

Population of size 10000 – Figures 24



Class Name	Instances [%]	Instances	Size
slot		1,160,058 (92.2%)	32,481,624 (67.2%)
java.lang.**Object[]**		21,814 (1.7%)	12,333,928 (25.5%)
java.util.**ArrayList**		20,138 (1.6%)	644,416 (1.3%)
solution		20,000 (1.6%)	560,000 (1.2%)
char[]		8,882 (0.7%)	562,922 (1.2%)

Results conclusion

As could be predicted, an increase in population size results in an increase of memory usage. However, even though there was an increase in memory, the memory used for the largest population size of 10000 never exceeded 800mb, a value which is practical for most modern computers. The spikes that can be seen in the heap memory graphs are caused by the java garbage collector "cleaning" up old objects (mainly slots objects). The CPU usage also never peaked beyond 26%, indicating that a speed increase will be observed if more CPU is allocated to the process. By far the largest use of the heap memory is the slot class, with the solution class averaging (2 * population size) number of instances (child and parent population). Although the slot class had a large number of instances (especially for the 10000 population algorithm), they still only used 32.5mb of Heap memory which is a fraction of the available 800mb.

These test results have shown that the algorithm scales very well regarding memory usage with an increase in population size, so therefore will also scale well with an increase in test case size. The increase in population size causes an increase in the number of slot instances, which is the same effect that an increase in a test cases total number of commitments would have. Therefore, we can assume that any average modern computer would have the hardware capabilities to deal with any practical test case, even with a larger population size being implemented to achieve a more optimal final timetable solution.

In conclusion, an increase in test case complexity or population size doesn't cause the algorithms hardware usage to scale very much at all, meaning that a realistic sized input case would not cause excessive RAM or CPU usage. However, the increase in complexity or population size would cause the algorithm to run more slowly. This is not a major issue due to the nature of the university course timetabling problem not normally being under any form of time constraint. However, even if time became an issue to the user, they could simply just allocate more CPU usage to the JVM, allowing the algorithm to run a lot faster.

5.3.3 – Large test case performance

For the final tests, I aim to see how my algorithm performs on an exceedingly large/complex test case. The goal of this experiment is to establish how well my final algorithm performs when having to traverse a very large search space, and to determine, given more time, if it would be

possible to solve this type of complex test case. To cater for the large increase in complexity, I will not be implementing the Cardiff specific hard constraint used in the previous experiment. I generated this large test case using my python script using the following parameters:

	Number of students	Number of modules	Year groups	Total commitments for all modules	Number of rooms	Number of lecturers
Test case	1200	60	6	600	675	50

I decided on these numbers to make the test case still possible in theory, but incredibly constraining in terms of student and lecturer clashes. To make the test case more feasible, I would decrease the number of rooms and the number of commitments so there is less opportunity for student and lecturer clashes to occur. I couldn't make the test case any more complex than this due to time constraints for the project deadline, however this would be something that I would like to do if given more time. I ran this experiment with the following algorithm parameters (same as the final algorithm variation):

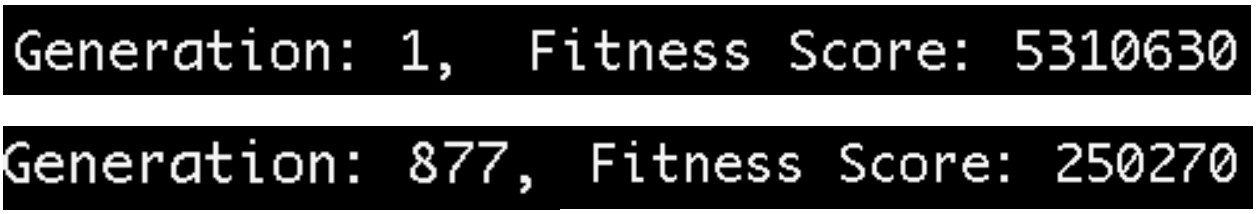
	Mutation	Crossover	Selection	Mutation ramping	Hill climbing
Feature values	10% of pop, 5% of slots	Uniform point	Tournament: 3 contestants	- Rate interval: 20 - Slot % increase: 5 - Population % increase: 10	Random 1% of population, with the V2 algorithm changes

Ideally for a test case this complex a very large population size would be used so that an adequate portion of the search space is covered, however, due to the time constraints of this project, I was only able to use a population size of 10000. Using this smaller value of population size allowed me to run the experiment for more generations, allowing the fitness score to converge a lot faster than if I was to use a population size of 100000. I didn't cap this experiment at a maximum number of generations, but at a time limit of ten hours. It turns out that by this time limit the algorithm had plateaued at a fitness score and hadn't changed in over 100 generations, so I don't believe letting it run longer would have increased the final fitness score. Although, letting it run longer, and giving it a larger population size would have the potential to increase the final fitness score.

For this experiment, I am recording the fitness score of the best solution present in the initial population and the fitness score of the final solution. This will give me an indication as to how well my algorithm performs, and the improvements made to from the first to last populations.

Results – Figure 25

	Initial Generation		Final generation	
	Generation	Best score	Generation	Best score
Results	*1*	*5310630*	*877*	*250270*



Analysis and conclusion

The initial score of 5310630 observed was very large, translating to 531 hard constraint violations and 63 soft constraint violations. This score was improved significantly to the final score of 250270 which translates to 25 hard constraint violations and 27 soft constraint violations. Although a valid solution could not be found, I attribute this to a small population size and a restricted run time. If a user had multiple days to run the algorithm, a better and potentially valid solution could have been found. If given more time I would have liked to run further experiments with increasingly complex test cases to establish at one point the algorithm fails to be practical (i.e. takes weeks to complete).

5.4 – Testing conclusion

Through testing I have been able establish the combination of algorithm features and parameters to produce the lowest fitness score solutions. This combination was then taken as my final algorithm and improved upon the initial basic algorithm considerably. My final algorithm then easily solved a real-world timetabling scenario taken from Cardiff university, showing that is was easily capable of performing timetable task for universities. Next, my algorithm showed that it scales in terms of hardware resource usage very well with population size and test case complexity. However, the speed of the algorithm doesn't scale as well, but this is not a significant issue due to the lack of time constraints placed upon real world timetabling scenarios. Finally, my algorithm was tested with an abnormally large test case and, although not reaching a perfect solution, still managed to reach a comparatively low score. Therefore, I believe that I have achieved the goal of solving real world problems set out at the beginning of this project, and have additionally created an algorithm that has the potential to solve extremely hard real world test cases, if given enough time.

6 – Future Work

As the process of this project has moved on, I have reassessed and reworked sections of the project, as-well as focusing on the algorithm performance instead of producing a user-friendly GUI application. I have learnt a huge amount about genetic algorithms and now have even more ideas for improvements and extra functionality. There are large areas for improvement that I have identified, and other smaller changes that I would have liked to implement.

Firstly, I believe that the final algorithm performed at a level for use in most university settings. However, there are aspects of the final application that make it not user-friendly. The first major work that I would do is the addition of the room feature functionality that I was forced to remove due to experiencing bugs. For my algorithm to be used for practical applications, it is vital that it has the functionality for attaching features such as room size and room type to potential rooms. It is also important for the application to have a GUI not only to run the main algorithm, but also to create the input text files. This would allow staff use with minimal training, and would make it a practical alternative to creating timetables by hand. Additionally, I would change the format of the input text file from the syntax that I created, to XML. Doing this would create a more structured configuration file, and allow for universities to create functionality allowing them to automatically generate the configuration files from their internal systems. Additionally, the functionality for user inputted constraints, and for custom constraint weightings would also be necessary for real world use. This could be easily achieved by defining a constraint input format, allowing user submitted constraint and using a parser to create constraint tests from the users input which effect the fitness scores assigned to the individuals by the fitness function.

There are also improvements I would like to make regarding the algorithms performance and design. Firstly, it would be more efficient to re-code the algorithm in a language such as C++. Doing this would allow me to take more control of the codes optimization, instead of tasking this to the Java compiler. I would also put more time into making the inefficient sections of the code (e.g. the hill climb operator) run with a smaller time complexity, ideally O(n) or better. These speed increases would reduce the time impact caused by larger population sizes, resulting in more optimal timetable solutions. Genetic algorithms have the scope for numerous improvements, of which I've only included a few in my final algorithm. New functionality such as a Tabu search feature for improving the quality of the initial population, or more specialized data structures designed specifically for genetic algorithms like the MEM data structure [25] would improve the speed and quality of the output solutions.

Lastly, given more time, I would conduct more thorough experiments concerning how well the final algorithm performs for very large test cases. I would increase population size to 100000 and run each experiment for 3 days. This would give me more of an insight into the maximum potential of my final algorithm. More time would also allow for additional expansions of

functionality, e.g. having the ability to solve university exam timetables problems, a feature which would not take a large amount of effort to implement. These functionality expansions, with all the further changes mentioned above, give the application commercial potential due to the real opportunities for time and economic saving for universities.

7 – Conclusions

The aim of this project was to create an algorithm to solve real world university course timetabling problems. The definition of a timetable that would be deemed to "solve" this real-world problem is one that didn't violate any of the hard constraints placed upon it, and ideally violate as few of soft constraints as possible. I also aimed for the algorithm to produce an output in a practical time frame of under two hours, and for the output to be in a human readable format.

I researched and compared various optimisation techniques including Tabu search, Backtracking algorithms and Ant colony optimisation algorithms. However, I decided to implement a form of evolutionary algorithms called a genetic algorithm due its potential for problem specific customizations, as well as its large search space coverage. It was also noted that genetic algorithms normally take more time to converge on an answer, but due to the nature of the timetabling problem, time taken to converge was not an important factor. Although it was decided that the main algorithms method should be a genetic approach, additional methods such as a form of local search called hill climbing were added to improve the quality of the algorithms population. The initial stage of testing proved these additions to greatly increase the quality of the output solution, making the final algorithm solve given timetable problems faster and to a higher quality than just a normal genetic approach. The final algorithm was then tested against a real-world scenario taken from a previous academic year at the school of computer science at Cardiff university. This scenario was for the lectures of years one, two and three of student on BSC Computer science and variants. The algorithm could generate a perfect timetable (no hard or soft constraint violations) in under two minutes, a time that was quicker than the two hours set out in the initial aims. These experiments showed that the final algorithm can solve real-world problems, therefore achieving the main aim of this project. I then tested the final algorithm against a very complex test case, and although a valid solution was not found, a promising effort was made, with the final solution being a drastic improvement on the best solution of the original population. However, for the final algorithm to be able to solve test cases of a similar complexity the algorithm must be allocated a larger population size and longer to run. Additionally, improvements need to be made to slower parts of the algorithm, and further measures need to be taken to control premature convergence.

In conclusion, I have achieved what I had initially aimed to achieve at the start of this project. However, as I have progressed through the project, it is clear that there are many things left to improve upon to allow the algorithm to become an application which could actually be used by universities for real world applications.

8 – Reflection on Learning

Beginning this project, I did not know much about optimisation techniques, and had never built an algorithm this complex or wrote a report of this detail. Throughout the whole process, I have learnt many new concepts and now have a grasp on numerous different methods for solving optimisation problems. Having to specifically redesign sections of my code to increase efficiency is also a problem that I haven't ever needed to tackle before. Having such a focus on speed has taught me new ways of designing code to solve problems, whilst not comprising on efficiency or correct practices.

Apart from the developments mentioned above, I will mainly focus on attempting to identify areas of success and areas of improvements surrounding development methods and planning, an approach to self-development called double-loop learning. In this section I will not only aim to identify what aspects of my approach worked well so that I can apply them again to future projects, but will also analyse my approach for aspects that could be improved upon.

Firstly, there are many techniques for time and task management that I have been forced to learn to complete this project for the deadline. One of the most useful techniques was the use of a "To-Do" list to keep track of what tasks needed completing and in what order. Creating this list of tasks helped me break the initially daunting tasks into smaller chunks, making it easier to see what needed to be done and kept me motivated by having smaller milestones. Another useful technique that helped with problem solving was the use of mind-maps for solving the larger theoretical challenges (e.g. the logic behind each stage of the genetic algorithm). Having a mind map allowed me to view the problem with more clarity, whilst allowing me to have reference to my current method, allowing me to implement changes without causing future confusion.

Due to the bugs that I experienced in this project, and the complexity of the algorithm, I have had to improve my approach to debugging. Initially I have debugged very simply by writing system output code into the code and using this output to narrow down to what is the root of the bug. For this project, I was forced to use debugging tools, such as the one in my IDE (IntellijIDEA). This tool allowed me to step through the problem areas of code using breakpoints, and view what was happening to the variables causing the bug. This allowed me to notice that they were changing when not being explicitly referenced in the code, leaving the only explanation being that references were being passed instead of the variables contents. This new understanding of debugging techniques will certainly help me in every future coding project, and will save me huge amounts of time.

One major skill that I have improved upon through this project is my ability to conduct proper research. Before this project, I have never had the need to research a subject in depth by consulting numerous scientific resources. The value of knowledge that can be obtained through

the study of a scientific paper on a subject is huge, and is something that I will do much more when an interest in a specific topic arises.

With hindsight, I would change the initial stages of the project development. Due to my keenest to begin coding, I began development without conducting thorough back ground research. This led to me having to redesign sections of code, with the most drastic case being a complete redesign of the method used to represent an individual. This cost me a lot of time, which I could have used in much more productive ways. Therefore, in the future I will ensure that I have a very strong theoretical basis before I begin any form of actual development, and will make sure my development plan is in line with the background research.

Again, with hindsight, I could have managed my time more efficiently. I believe that I spent too much time on unnecessary diversions and non-critical areas of the application. For example, a disproportionate amount of time was spent on formatting the output timetable. In future projects I will clearly lay-out which are the crucial areas of development that warrant the most development time, and only focus on less important areas once these crucial parts have been completed.

9. Bibliography

[1] A. Wren. (1996). Scheduling, timetabling and rostering – A special relationship? The Practice and Theory of Automated Timetabling I: Selected Papers from 1st International Conference on the Practice and Theory of Automated Timetabling (PATAT I), Edinburgh, UK, Lecture Notes in Computer Science 1153, Springer-Verlag. (Editors: E.K. Burke and P. Ross), pp 46-75.

[2] A. Schaerf. (1999). A survey of automated timetabling. Artificial Intelligence Review, 13(2), pp 87-127.

[3] Michalewicz, Z. and Schoenauer, M., 1996. Evolutionary algorithms for constrained parameter optimization problems. Evolutionary computation, 4(1), pp.1-32.

[4] Kumar, V., 1992. Algorithms for constraint-satisfaction problems: A survey. AI magazine, 13(1), p.32.

[5] G.M. White and P.W. Chan, "Towards the Construction of Optimal Examination Timetables," INFOR 17, 1979, p.p. 219-229.

[6] M.W. Carter, "A Survey of Practical Applications of Examination Timetabling Algorithms," Operations Research vol. 34, 1986, pp. 193-202.

[7] Landa Silva, J. D., Burke, E. K., & Petrovic, S. (2004). An introduction to multi-objective meta-heuristics for scheduling and timetabling. In X. Gandibleux, M. Sevaux, K. Sorensen, & V. Tkindt (Eds.), Lecture notes in economics and mathematical systems : Vol. 535. Multiple objective meta-heuristics (pp. 91–129). Berlin: Springer.

[8] John H.Holland (1975) Adaptation in Natural and Artificial Systems, Cambridge: MIT Press Cambridge.

[9] Thomas Baeck, D.B Fogel, Z Michalewicz (2000) Evolutionary Computation 1: Basic Algorithms and Operators, : CRC Press.

[10] Baker, James E. (1987). "Reducing Bias and Inefficiency in the Selection Algorithm". Proceedings of the Second International Conference on Genetic Algorithms and their Application. Hillsdale, New Jersey: L. Erlbaum Associates: 14–21.

[11] Fred Glover (1986) Future paths for integer programming and links to artificial intelligence, Computer and Operations Research

[12] Gurari, Eitan (1999) "CIS 680: DATA STRUCTURES: Chapter 19: Backtracking Algorithms", Available at: https://web.archive.org/web/20070317015632/http://www.cse.ohio-state.edu/~gurari/course/cis680/cis680Ch19.html#QQ1-51-128 (Accessed: 9/04/2017).

[13] Marco Dorigo (1992) Optimization, Learning and Natural Algorithms, Italy: Politecnico di Milano.

[14] Amin Hadidi (2015) 'A survey of approaches for university course timetabling problem', Computers & Industrial Engineering, (), pp. 47 [Online]. Available at: https://www.researchgate.net/publication/274705544_A_survey_of_approaches_for_university_course_timetabling_problem (Accessed: 9/09/17).

[15] Spyros Kazarlis, Vassilios Petridis, Pavlina Fragkou (2005), "Solving University Timetabling Problems
Using Advanced Genetic Algorithms," In proceeding of the 5th international conference on technology and automation, Thessaloniki, Greece, pp.131-136.

[16] S.Kazarlis, S.Papadakis, J.Theocharis and V.Petridis (2001), "Micro-Genetic Algorithms as Generalized Hill Climbing Operators for GA
Optimization," IEEE Transactions on Evolutionary Computation, Vol.
5, No. 3, pp. 204-217.

[17] Yu, E. and Sung, K (2002) A genetic algorithm for a university weekly courses timetabling problem. : International Transactions in Operational Research.

[18] Burke, E. K., J. P. Newall, and R. F. Weare. "A Memetic Algorithm For University Exam Timetabling". Practice and Theory of Automated Timetabling (1996): 241-250. Web. 10 Apr. 2017.

[19] Alicia Y.C. Tang, 2013, Digital image, viewed 13/04/17, <https://www.researchgate.net/figure/269840981_fig1_Figure-4-Roulette-wheel-selection-example>

[20] Digital image, viewed 13/04/17, <https://upload.wikimedia.org/wikipedia/commons/b/b7/Statistically_Uniform.png>

[21] Digital image, viewed 13/04/17, <https://upload.wikimedia.org/wikipedia/commons/thumb/5/56/OnePointCrossover.svg/1200px-OnePointCrossover.svg.png>

[22] Digital image, viewed 13/04/17, https://upload.wikimedia.org/wikipedia/commons/8/8f/UniformCrossover.png

[23] Renders, J.M. and Bersini, H., 1994, June. Hybridizing genetic algorithms with hill-climbing methods for global optimization: two possible ways. In Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on (pp. 312-317). IEEE.

[24] https://visualvm.github.io/

[25] Sadaf Naseem Jat, Shengxiang Yang (2009) 'A Guided Search Genetic Algorithm for the University Course Timetabling Problem', *Multidisciplinary International Conference on Scheduling*

[26] Kheiri, A. and Keedwell, E. 2016. A hidden Markov model approach to the problem of heuristic selection in hyper-heuristics with a case study in high school timetabling problems. Evolutionary Computation

[27] Ahmed, L. N. , Özcan, E. and Kheiri, A. 2015. Solving high school timetabling problems worldwide using selection hyper-heuristics. Expert Systems with Applications 42 (13), pp.5463-5471.

10. Appendix

For convenience and ease of access, I have included all elements of interest in .zip files submitted with this report. Below is a list of the .zip files names and an explanation of their contents.

code.zip

- run.java : The run java class used to run the main algorithm
- timetable.java : The timetable class containing the main algorithm methods
- solution.java : The solution class used to create solution objects
- slot.java : The slot class used to create slot instances

testFiles.zip

- gentest.py : The python script used to generate the test cases in section 5
- info.txt : Text file explaining how to use the python script
- testResults1 : Folder containing all the timetables from the first stage of testing
- testRaw1 : Folder containing spreadsheets of the first stage raw test data
- realInput.txt : Input configuration file for the real-world Cardiff test case
- realOutput.txt : Output timetable file from the real-world Cardiff test
- large.txt : Input configuration file for section 5.3.3 (large test case scaling)
- realOriginal : Folder containing the original real world timetables