Solving a University Timetabling Problem Using a Genetic Algorithm

Rafaël van Nieuwkerk
001030749

A project presented for the degree of
Software Engineering, BEng Hons
Supervisor: Markus Wolf
Word count: 11,326

Computing and Mathematical Sciences
University of Greenwich
May-2021

Solving a University Timetabling Problem Using a Genetic Algorithm

Rafaël van Nieuwkerk

Abstract

This paper looks at solving a university course timetabling problem using a genetic algorithm and testing various genetic operators including a custom crossover method. An algorithm is programmed specifically for this purpose. The dataset that is used is timetable data from a real university and is imported into the program. Each class in this dataset is represented by a gene in the DNA of each individual. This means each gene has a fixed position and each of them a set amount of possible states based on the data from the dataset. Multiple experiments are done to find if there are any improvements that can be made to the default implementation that was made. The result of this is that some minor adjustments should be made to the elitism and mutation rate. More research can also be done in making crossover and selection methods that are specific to this problem. The final result is an algorithm that is promising but would still need human tweaking before releasing a timetable. It is a promising area of research and there is room for improvement once enough research is done.

Acknowledgements

I would like to thank Markus Wolf for being a very attentive supervisor, making sure to check up on my progress and giving relevant feedback. I would also like to thank YouTube user Kryzarel[1] for his videos on writing a genetic algorithm. Understanding the theory is one thing, but finding real code to learn from was more difficult, so his videos were a nice start.

[1]https://www.youtube.com/c/Kryzarel

Contents

List of Figures

1. Introduction

Timetabling problems occur all the time in every day life. Things such as work schedules, public transport schedules and of course university timetables all need to be generated. The latter is what this project will focus on. To solve a timetabling problem there is a list of constraints that need to be satisfied. Generally there will be soft constraints, which are similar to preferences, and hard constraints, which should not be violated. The overall goal is to minimise the number of violations of both these types of constraints. What is especially important is to not violate the most basic constraint when making a timetable, room conflicts. Classes cannot be taught at the same time in the same room, so this should always have a priority. Timetabling problems are therefore a type of optimisation problem, and algorithms used for optimisation are often used to solve timetabling problems, including Genetic Algorithms (GAs).

GAs are a way of finding the most optimal solution to a problem using fitness and building blocks of previous solutions. The way this works is that a GA generates many solutions to a problem. Solution in this context doesn't mean it actually satisfies all the constraints, in fact the first population of solutions is likely very unsatisfactory. Out of this population of solutions some method is used to select which solution should be used for offspring, this is usually based on fitness. In order to do this a formula for fitness needs to be established. The offspring will then form a new population and the cycle continues (Budhi, Gunadi, and Wibowo 2015).

The aim of this project is to find out if using a genetic algorithm to solve university course timetabling problems is a viable solution and if any improvements can be made to already existing examples. To achieve this a dataset of real data will be used from UniTime.org.[1] This dataset is provided in XML format which makes it easy to process and load into the program. Also a custom crossover method will be suggested and tested to see if it can improve results. The program will also be designed in such a way that it is easy to reuse for different tests regarding this dataset and GAs. This is why it outputs CSV files for easy analysis.

The report will consist of 6 chapters. Chapter 2 will discuss other research in this area. This includes different algorithms that solve scheduling problems, but most importantly the many genetic operators that can be used in genetic algorithms. Chapter 3 will show the dataset used for this problem and how it works. This includes an explanation of all the constraints and code demonstrating the structure of the dataset. Chapter 4 discusses the implementation and the development process of the program. Using this program experiments are done which are discussed in Chapter 5 including the results of these experiments. Then finally Chapter 6 will evaluate what this project has done and suggest further research on this topic.

[1]https://www.unitime.org/uct_datasets.php

2. Literature Review

2.1 Timetabling Problem

There are a couple different types of timetabling problems within universities, the two main ones are exam timetabling and term timetabling. For exam timetabling, as the name suggests, the problem of scheduling exams within a certain amount of weeks needs to be solved. For term timetabling lectures, tutorials, labs and any other types of classes during normal term times need to be scheduled.

Within term timetabling there are also two different types of problems. There is curriculum based course timetabling and post enrollment based timetabling (Rudová, Müller, and Murray 2010). For post enrollment timetabling students enroll into courses by themselves whereas with curriculum based timetabling the University generally creates a number of curricula. Each of these consist of a set of courses and a number of students enrolled in them (Achá and Nieuwenhuis 2012).

To solve a timetabling problem, a list of constraints must be formulated. These constraints can heavily depend on the University and what they find acceptable. Some hard constraints are generally universal like a teacher cannot be scheduled to teach two different classes at once, rooms can (generally) not be scheduled to run two classes at once and staff availability has to be taken into account (Achá and Nieuwenhuis 2012). One example of a constraint that depends on the type of timetable is a student having two courses at the same time. For curriculum based timetabling this is generally unacceptable and therefore a hard constraint. For post enrollment timetabling however this is more likely to be a soft constraint as it is more difficult to avoid.

In most cases soft constraints will have some kind of weight associated with them, since not all soft constraints are equally important. One example could be the distance between classrooms. Ideally you wouldn't want two classes back to back that are a significant distance away from each other. If this distance means that students are say 10 or more minutes late every time, then this is a soft constraint with a high weight. An example of a soft constraint with a low weight is the amount of free time between classes. A gap of say 3 hours would not be ideal but it is relatively unimportant and therefore has a lower weight.

2.2 Analysis of Algorithms

2.2.1 Local Search

Local search is one of the major methods of solving constraint satisfaction problems such as timetabling. Initially a local search algorithm starts from a random point within its search space. The search space consists of all values that don't break certain hard constraints. In a paper about a hybrid local search algorithm Bellio, Di Gaspero, and Schaerf (2011) mentions not considering all hard constraints when defining the search space but instead only certain essential ones. One hard constraint that they do consider is that all lectures of a course must be scheduled. The other constraint they consider is that a teacher must be available to teach their course. The former is important because if this constraint is violated it means that the solution is incomplete in some way. The latter is considered because it is not effected by any other variables, and is therefore easy to prevent from the start. The other hard constraints are treated similarly to soft constraints, but with a higher weight associated with them.

Another important part of local search is the neighbourhood relations. This is what defines the way a solution can change. Bellio, Di Gaspero, and Schaerf (2011) describes their algorithm having two distinct neighbourhood relations. The first is MoveLecture, in this case one lecture changes to a different period and/or room. The other relation is SwapLectures, here two lectures from different courses are swapped. This includes both the room and time period.

2.2.2 Iterative Forward Search

In Iterative Forward Search an assigned or unassigned variable is selected for each step. Generally an unassigned variable is used but an assigned variable can also be used. This occurs usually when all variables are already assigned but the solution still violates too many soft constraints. The selected variable then gets assigned a value from its domain. This can cause a hard conflict to occur between this variable and any of the already assigned variables. The algorithm checks for this and the conflicting variable becomes unassigned and the selected variable is assigned the selected value. This process is repeated until some condition is met. This will usually be a limit on the number of iterations or the amount of time that is consumed running the algorithm (Müller 2004).

An important part of this algorithm is comparing the solutions. The way this happens is by comparing the current solution to the best known solution. To do this some kind of scoring system should be made. Generally the score will be based on the amount of unassigned variables and violated soft constraints (Müller 2004).

Selecting which variable to assign next is another important step for this algorithm. When there are still unassigned variables, the most complicated one is assigned first. This means the variable that is most likely to violate soft constraints is selected first. When all variables are already assigned, we might still need to continue to get a more satisfactory solution. In this case it would be best to select a variable that is most likely to improve the solution when it is changed. This is likely a variable that violates the most soft constraints (Müller 2004).

Perhaps the most important part of the process is actually assigning a value to a variable. The best way of doing this is by trying to find the best fitting value. This means having to find a value with the least potential for future issues. Most important is to first look at values that don't cause any hard conflicts violations. After that check which values are least likely to cause such violations in the future, and then look at which value has the least potential future soft constraint violations (Müller 2004).

To make sure the algorithm does not iterate indefinitely, a stopping condition has to be defined. As mentioned before this is generally based on the amount of iterations or time spent running the algorithm. It would also be possible to base a stopping condition on the success of the algorithm. For example the algorithm could keep iterating until a solution that only violates a certain amount of soft constraints (Müller 2004).

2.2.3 Genetic Algorithm

One of the most important factors of genetic algorithms is the way that the algorithm produces new populations. The main way of doing this is using crossover. With single point crossover you essentially chop the chromosome of the two parents into two parts at a randomly chosen position. Then two offspring are created, each getting one part of each parent. This single point crossover method has its problems though, a good example of this is given by Mitchell (1996) "it cannot in general combine instances of 11*****1 and ****11** to form an instance of 11**11*1". Another issue that single point crossover can cause is certain bits in a chromosome that are not desirable being carried over because they are close to more beneficial bits. A potential improvement on this is two point crossover, where 2 points are chosen and the bits are exchanged accordingly. Another alternative is giving every single bit a probability to crossover Mitchell (1996).

Another important factor in genetic algorithms is the use of mutation. Mutation is an important way of increasing the diversity of a population. This helps prevent the genetic algorithm from stagnating (Delima, Sison, and Medina 2019). While mutation often acts as more of a background step in genetic algorithms, it can also be used more aggressively. For example it would be possible to keep the amount of crossover low, but have a high amount of mutation. This would make the algorithm act more similarly to local search, in the sense that the main way the solution progresses is by swapping out certain values.

While crossover and mutation are the two most common methods of evolving the population of a GA, some other methods have been developed too. Quirino, Kubat, and Bryan (2010) proposes an interesting approach where instead of using probability, mating partners are chosen based on complimenting features. This is similar to real animals choosing mates based on features that compliment their own, they also call this "opposites-attract". Another interesting strategy is the "crowding" approach mentioned in (Mitchell 1996). With this approach offspring replace existing chromosomes that are most similar to themselves. This prevents too many similar chromosomes existing in the same population.

Genetic algorithms will be the main focus of this project. There are many inspiring papers on GAs and their application in timetabling. One such paper is (Raghavjee and Pillay 2010) where a GA is used to solve a high school timetabling problem. Another example is (Wang, Liu, and Yu 2009) where the author proposes a self-fertilising genetic algorithm for university timetabling. The main aim of this project will be to find new ways to change and hopefully improve on existing genetic algorithms and apply this to the university timetabling problem. To do so, selection, crossover and other methods will be discussed in more detail.

2.3 Selection Methods

2.3.1 Roulette Wheel Selection

Roulette wheel selection, as the name implies, is based on a roulette wheel except with different sized segments. Each member of the population is assigned a segment, where the size of the slice is based on the fitness of the individual. Higher fitness means bigger segment which means a higher chance of the 'pin' landing on this segment (Razali and Geraghty 2011a). The circumference of the wheel is equal to the sum of all fitness values in the population. The advantage of this is that it does not rule out any of the population, each individual has a chance of being selected which is good for diversity. Because there is no real limit on how big a segment can be it is possible for a comparatively fit individual to overshadow the others early on. This would cause the algorithm to converge too early on a solution that is not ideal and can cause the algorithm to get stuck. The opposite can also be an issue, where all members have a fitness that is very similar to each other. This makes them have almost identical odds of being selected even though it would be more beneficial to pick the higher fitness individuals. This is only the case in proportional roulette wheel selection, where segment size is directly related to fitness.

Another method of roulette wheel selection is rank-based. The simplest way to imagine this is to have an array with all the individuals in it and sort it by fitness highest to lowest. Then the rank will be equal to the position of the individual in the population array. Then each individual is assigned a fixed probability based on its rank rather than its fitness (Razali and Geraghty 2011b). This way there is a limit on the probability assigned to each individual which resolves the earlier overshadowing problem. It also solves the problem of individuals having almost identical chances of being selected since each rank has a fixed probability assigned.

Jadaan, Rajamani, and Rao (2008) has done tests comparing proportional and rank based roulette wheel selection. These tests used several fitness functions and rank based roulette wheel selection was found to outperform proportional in number of generations needed to achieve an optimal solution.

2.3.2 Sexual Selection

Sexual selection was proposed by Goh, Lim, and Rodrigues (2003). The main principle of this method is to split the population into 2 groups (sexes). Assigning a sex to each member of the population can be done either randomly or it can be specifies to the problem the GA is trying to solve. The selection method is based on the principle that all females will mate with exactly 1 male, while the males mate with 0 or more females. Goh, Lim, and Rodrigues (2003) say they were inspired by Darwin et al. (1859), who suggests some species mate based on female choice. This method consists of 2 main stages. The first stage is assigning a sex to each member of the population. The second stage is selecting pairs of males and females that will create offspring.

An example of assigning sex in a way that is specific to the problem would be assigning the individuals with the lowest amount of soft constraint violations as female. Soft constraints generally have a lower weight than hard constraint because they are less important to the overall solution, however they should not be neglected either. Because every female is guaranteed to pass on part of its genes this will increase the diversity in the population.

Selecting the female when making the couple is easy as the list of females can just be iterated through. Selecting a male will still require some other type of selection method. Goh, Lim, and Rodrigues (2003) discuss using tournament selection with a tournament size of 2 to select the males. However they also discuss the possibility of using some compatibility function that selects males based on compatibility with the female.

This method also does not have the problem that proportional roulette wheel selection can have where it can get stuck on an individual that has relatively high fitness early on. This can cause the algorithm to converge too early on a solution that is not ideal (Razali and Geraghty 2011a). Sexual selection avoids this due to females being chosen regardless of fitness but still focuses on fitness when it comes to the males, to avoid too much diversity.

2.3.3 Tournament Selection

Tournament selection is a popular selection method, especially in GAs (Xie and Zhang 2013). In its most basic form, tournament selection randomly adds a number of individuals to a tournament. The individual with the highest fitness wins the tournament and is used as a parent. This process is repeated every time a parent needs to be selected (Goldberg and Deb 1991). Important in this type of tournament is selecting the best tournament size. When the tournament size is too large it can lead to a higher loss in diversity (Blickle and Thiele 1996). This is because the fittest individuals will always win and the larger the size the more likely they are to be in the tournament and therefore win. On the other hand, when the tournament size is too small it can lead to too much diversity because then selection becomes more chance based.

Goldberg and Deb (1991) suggests a method where a random value is added to tournament selection. Here the person with the highest fitness is not guaranteed to win. Instead, there is a probability p of the fittest individual to win, with the other individual having a probability of $1 - p$. With p being a value between 0 and 1 this can be used to tune the randomness of the selection method. Goldberg and Deb (1991) applies this to binary tournament selection where the tournament size is 2, but it could be made to work with larger tournament sizes too. Julstrom and Robinson (2000) achieves this in k-tournaments with $k > 2$ by associating a probability to each rank. This makes it so that the higher fitness has a higher chance of being chosen, but it is never guaranteed to win, like in traditional tournament selection.

Harik (1995) proposes another interesting variation of tournament selection where only individuals with similar genes compete with each other. This is especially helpful when a GA is solving a problem that consists of multiple smaller problems. This way the it is less likely that some of the smaller problems are ignored due to bias. Matsui (1999) proposes a similar method where similarity plays a role. In this method the first parent is selected using traditional tournament selection. Then a second tournament is formed for the second parent but this tournament won't be won by just fitness. A value g is calculated which is a combination of the fitness and similarity to parent one, of each member in the second tournament. The individual with the highest value g wins the tournament and will then become the second parent.

Another variation on tournament selection is also proposed by Matsui (1999) which is family based. First a family is formed which consists of 2 parents and their 2 offspring, obtained through crossover. First the fittest individual in the family is chosen and assigned x. The Hamming distance of all other family members compared to x is then calculated. The individual with the highest Hamming distance is assigned y. Individuals x and y are added to the next generation, while the other two family members are not. Similar to the previous approach this intends to cause high diversity while still focusing on fitness.

2.4 Crossover Methods

2.4.1 k-point Crossover

One of the simplest forms of crossover is one-point crossover. This is where a single point in the DNA of the parents is selected. All genes to the right of that point are swapped. So parent 1 now has parent 2's chromosomes beyond this point and vice versa (Shafiee et al. 2016). After the swap these then become the offspring that will form the next generation. A slightly more complex version of this is using 2 points. In this case the genes in between the two points get swapped between the parent. The amount of points used in this method can in theory be as large as the amount of genes. However, once the amount of points used gets too large, it essentially becomes uniform crossover.

2.4.2 Uniform Crossover

Uniform crossover also uses swapping, similar to k-point crossover with a high value for k. In uniform crossover genes are randomly selected for swapping. Generally each gene will have a 50% chance to be swapped, creating a uniform mix of the genes of both parents. The result of this then forms the offspring and will move on to the next generation.

2.4.3 Discrete Crossover

Discrete crossover is similar to uniform crossover. In discrete crossover a child is created by looping through every position in its gene array. For each position a random number is generated, generally between 0 to 1. If the random number is lower than 0.5, then the gene in that position will be given by parent 1. If the random number is higher or equal to 0.5, then the gene in that position will be given by parent 2(Gwiazda 2006). Unlike uniform crossover, this only results in one child instead of two. Although, it could be possible to generate a child that is the complete opposite of the first child, but this would essentially just be uniform crossover as the offspring would be the same as if the genes would have been swapped instead of passed down.

2.5 Other Genetic Operators

2.5.1 Mutation

The purpose of mutation is to prevent early convergence in genetic algorithms by introducing some (random) variation to the genes in the population. Like in biology, mutation is often necessary for new genes to be introduced into the gene pool. If this gene is beneficial, it will likely become a larger part in the gene pool. If not, it is more likely to vanish over time.

Mutation is very dependent on the problem that a GA is trying to solve. A basic example that works only when dealing with binary data is flipping mutation, here a 1 is turned into a 0 and vice versa. Another example is interchanging mutation where randomly selected genes within an individual are swapped in position. Another similar approach is where genes are swapped specifically with their neighbour (Khair et al. 2018). However, these methods cannot always be applied, since there are problems that are not binary or where the position of a gene within a chromosome does not matter. The most basic method, that works in almost every situation, is to select a random gene, and replace it with a randomly generated gene. The method used to generate a new random gene is likely the same one used when creating the first generation. This way new genes are added to the population that might not have been there yet.

2.5.2 Elitism

Sometimes when creating a new generation it is possible that the fittest individual is less fit than the fittest individual in the previous generation. To prevent the genes of the fittest individuals from being lost, elitism is introduced. In simple elitism the top n individuals are added directly to the next generation (Sridharan 2010). They are still used in the current generation's selection and crossover, but now also have the benefit of being able to be used in the selection and crossover of the next generation.

A different version of elitism is global elitism. Here an offspring can only replace their parents if and only if they have higher fitness. This method makes sure that the overall fitness cannot decrease. However, this could cause issues in diversity as it is a greedy approach. The problem with low diversity is that the algorithm gets stuck before finding the optimal solution.

Sridharan (2010) discusses another type of elitism, dynamic elitism. The idea is that the amount of elitism is dynamic across generations. This approach combines the members of the current generation with their offspring. This group will therefore be twice as big as the normal population size. Out of this group, the only the fittest individuals are used for the next generation, cutting down the group back to the normal population size. This then forms the next generation.

Another interesting point raised by the same paper is the concept of chromosome age. The main reason to add ageing to genes is to prevent loss of diversity. Each time a chromosome is passed directly to the next generation due to elitism, its age is increased. This age will then be used to either directly decrease its fitness or will be taken into account in a different way during the parent selection process. Either way, the end result is that an older chromosome will have a lower chance of being selected, sometimes even if it has a slightly higher fitness than a younger chromosome.

2.6 Similar Research

Only some research has been done on using Genetic Algorithms specifically for University course timetabling. An example of this is work done by Alnowaini and Aljomai (2021). This paper discusses a GA that is focused on timetabling faculty problems. This paper gets its data from a user-interface that lets the user insert 'block' data which consists of a course ID, lecture ID, department ID, and level ID. This block is then assigned a time and room to form a lecture, which serves as a gene in their implementation. They describe their chromosome length as dynamic because it can vary in length. The reason for this is because the number of lectures varies depending on the semester or the faculty and so this flexibility allows it to be used for all of those. They also mention in their paper that going beyond 82 courses increases the number of conflicts drastically. However, their timetable only takes 5 minutes to generate.

A paper written by Khonggamnerd and Innet (2009) uses an interesting approach to their genetic algorithm. They don't seem to use the normal structure of having a population, selecting parents out of that, applying crossover and/or mutation, and then repeat. Instead, they use only 2 chromosomes in the initial generation and never have more than 4 at once. They also have an interesting approach to crossover and mutation, where only one or the other is used to create offspring instead of a combination of the two. The way their algorithm works is that first a random number between 0 and 1 is generated and if this number is lower than the crossover rate, then crossover is used. If it is higher, mutation is used. After either crossover or mutation is used to generate the children, the fitness is calculated of the children and parents. The two fittest individual out of this group will serve as the parents for the next generation. The best results of their research show a 0.3 chance of mutation being most beneficial. In a standard GA a chance of 0.3 would be incredibly high, but in this application it makes sense. Since the population size is so small this mutation rate has to be so high to maintain diversity. Their dataset however was relatively simplistic, only having 5 class rooms and 55 subjects.

Ghaemi, Taghi Vakili, and Aghagolzadeh (2007) has another interesting approach to applying GA to a university timetabling problem. Their implementation uses 2 chromosomes that together form 1 solution, unlike other each single chromosome is a possible solution. These chromosomes are generated in separate population. The reason for this is that their timetable has lectures that occur either once or twice a week. The first occurrence will be in the first chromosome, and the second in the other chromosome. If the class only has one lecture per week, then the second solution is not used. The first GA described in this paper takes three chromosomes from each population, and puts them into the other population. These three chromosomes will combine with each population to form full solutions. The other method seems to keep the two species separate and only mixes them for calculating the fitness.

3. Dataset

3.1 The Dataset

The dataset used for this project is created by UniTime.org (2007). It is an XML file containing timetabling information from Purdue university. All private information is removed while retaining complexity. Their website[1] contains various types of datasets for different types of timetabling problems.

The problem chosen for this project is the Large Lecture Room (llr) problem. In many universities rooms are divided among faculties with little overlap but there are usually also large rooms that are shared among multiple faculties. This overlap in faculties is what gives this problem its complexity. A solution for this dataset generated by UniTime is also available. Because the main focus of this project is to solve a timetabling problem and not student sectioning problems, the student sectioning solution was taken from the solution and combined with the original dataset. This then formed the file used in the program. Some information about the dataset is seen in Table 3.1

Data Type	**Amount**
Hard Constraints	153
Soft Constraints	130
Rooms	63
Classes	896

Table 3.1: Dataset information

The dataset contains nodes for rooms, classes, groupConstraints and students. Inside each class each possible time and room that this class can be taught at is listed. The room and time chosen for the solution will have a "solution=true" attribute added to it. The rooms inside each class only have an ID and a preference assigned to it. The lower the preference the better. This ID can be used to find the corresponding room in the rooms list. Here more information about the room is available like the location and capacity of the room.

Group constraints are used to assign constraints to a group of classes. An example of this is the BTB constraint, which requires the classes in the group to be taught back-to-back and in the same room. This constraint, like many others, can be either a hard constraint or a soft constraint. Hard constraints can be either required or prohibited, while soft constraints will have a value assigned to them that can be positive or negative. The most used constraint in this dataset is the SPREAD constraint. It requires the overlapping of classes in its group to be minimised. This constraint can only be required meaning its always a hard constraint.

[1]https://www.unitime.org/uct_datasets.php

3.2 The dataset structure

```xml
<timetable version="2.4" nrDays="7" slotsPerDay="288">
<rooms>
    <room id="1" constraint="true" capacity="118" location="451,435"/>
    ...
</rooms>
<classes>
    <class id="5" offering="5" config="5" committed="false" subpart="5"
     ↪  classLimit="105">
        <room id="1" pref="0"/>
        <room id="2" pref="0"/>
        <time days="1010100" start="90" length="12" breakTime="10"
         ↪  pref="2.0"/>
        <time days="1010100" start="102" length="12" breakTime="10"
         ↪  pref="-10.0"/>
    </class>
    ...
</classes>
<groupConstraints>
    <constraint id="1" type="BTB" pref="-2">
        <class id="829"/>
        <class id="830"/>
    </constraint>
    ...
</groupConstraints>
</timetable>
```

A small example of the structure of the dataset can be seen above. There are groups for rooms, classes, groupConstraints and students. The students group is left out as it is not used in this project. Each class in the classes list has one or more possible rooms and times that this class can use. The time nodes in each class are self-contained, they store all the necessary information in each tag. The room nodes only have an ID and preference. This is because there is only a certain set of rooms so they are all stored in their own room list. The room ID in the class then refers to the ID in the rooms list where important information such as location is accessible. Each of these nodes has a preference and minimisation of this value is one of the optimisation criteria of the algorithm. The classes are referred to as Klas in the code to prevent confusion between programming classes and classes found in the dataset.

Each constraint in the group constraints list is fairly straightforward as well. Each constraint node contains the classes that must obey this constraint. The importance of each constraint is given by either an integer or by a letter stating whether it is a prohibited or required constraint.

3.3 Constraints

Below is a list of all the constraints in this dataset and a description of what they mean. All descriptions are based on the definitions given by UniTime.org[2] and are mostly paraphrased.

[2]https://www.unitime.org/uct_grconstraints_v24.php

The constraints are listed in order of how often they occur in the dataset. The code created for each of these constraints can be found in the appendix.

SPREAD

Time overlap of given classes must be minimised, regardless of room. This constraint can only be required, meaning it is a hard constraint. However, since it is about minimisation it will be counted as a soft constraint in this project.

SAME_STUDENTS

Classes are treated as if attended by the same students. Thus student conflicts must be taken into account. This means they cannot overlap in time, and distance limits are set when they are taught back-to-back. These limits are as follows; the distance between the two rooms cannot exceed 670 meters unless the first class is 18 time slots in length (90 minutes), in which case the allowed maximum distance is 1000 meters. The distance between rooms is calculated as follows (UniTime.org 2007):

$$d = 10 \cdot \sqrt{((x_2 - x_1)^2 + (y_2 - y_1)^2)}$$

Here (x_1, y_1) are the room coordinates of one room and (x_2, y_2) are the coordinates of the other room.

MEET_WITH

Essentially this constraint combines the SAME_ROOM, SAME_TIME and SAME_DAYS constraints. The original description also mentions CAN_SHARE_ROOM however, this is not really a constraint it is more like an attribute. All these constraints combined means that classes with MEET_WITH must be taught at the same time, same room, and same day as each other. This constraint can only be required.

SAME_ROOM

This is a fairly straightforward constraint, as the name suggests all classes in this constraint must be taught in the same room, regardless of time or day. This constraint can be preferred or required but also prohibited or discouraged, in which case none of the classes are allowed to be taught in the same room.

SAME_START

Classes must start within the same half-hour time period. This is independent of day or room. This can also be discouraged or prohibited, in which case none of the classes can share the same start time.

BTB

Classes must be taught in the same room at back-to-back time slots on the same days. Meaning a student that has both classes in this constraint should be able to remain in the same room for their next class. If this is prohibited or discouraged, there must be at least a half-hour gap between these classes, but they must still be taught on the same days and in the same room.

DIFF_TIME

This constraint is similar to the SPREAD constraint, however here the original description says classes "cannot overlap", rather than minimising overlap. This is why when required or prohibited, it will actually count as a hard constraint. When it is preferred or discouraged it is still a soft constraint. When prohibited or discouraged classes must overlap in time.

SAME_INSTR

Similar to SAME_STUDENTS except here the classes are treated as if they are all taught by the same instructor. This means instructor conflicts must be prevented. Instructors cannot teach classes that overlap in time. Similar to SAME_STUDENTS, when the classes are back-to-back the distance is limited. The table on the right shows the weights associated with each distance. Here d is the distance. A weight of 1 means discouraged, 4 is strongly discouraged, and P is prohibited.

Distance	**Weight**
$0 < d <= 50$	1
$50 < d <= 200$	4
$d > 200$	P

SAME_DAYS

All classes in this constraint must be taught on the same days or a subset of the same days. This means not all classes need to have the same amount of meetings. As long as the class with fewer meetings has all its meetings on the same days as the class with more meetings, the constraint is satisfied. For example, if class one has the day pattern 1010100 and class two has 0010100, then the constraint is satisfied. This constraint can be prohibited or discouraged, in which case classes cannot have any overlapping days.

SAME_TIME

This constraint is similar to SAME_START but only for classes of the same length. When one class is shorter than the other, the shorter class must be taught within the begin and end time of the longer class. This is demonstrated by the following formula:

$$(start_1 \leq start_2 \wedge end_1 \geq end_2)\ for\ length_1 \geq length_2$$

NHB_GTE(1)

Classes must have 1 hour or more between them. This can be prohibited or discouraged, in which case classes must have less than 1 hour between them.

BTB_TIME

Same as BTB, classes must be taught in adjacent time slots except here the room does not matter but days still do. This means a class must be taught on the same day, taught at back-to-back time segments, but can be taught in different rooms. This can be discouraged or prohibited in which case classes must still be taught on the same day, but can no longer be back-to-back. They cannot overlap in time and must have at least a half-hour in between classes.

NHB(1.5)

Similar to NHB_GTE except here the time between classes must be exactly one and a half hour. Other time values for this constraint are possible but do not occur in this dataset. Classes must be taught on the same days. This can be prohibited or discouraged in which case classes must still be taught on the same days, cannot overlap, and must have any other amount of time between them that is not 1.5 hours.

4. Implementation

4.1 Overview

A genetic algorithm consists of certain key parts.

- DNA/Chromosome
- Genes
- Population
- Fitness function
- Genetic Operators:
  - Selection
  - Crossover
  - Mutation
  - Elitism

A very simple example of a GA is a GA that tries to generate some target string, like "Lorem ipsum dolor sit amet". The DNA in this scenario is a string, with each character being a gene. To start, an initial population of random DNA needs to be made. In this example that means assigning a random character to each position in the string. Once this is done the fitness of each member of the population needs to be calculated. For this example that means comparing the DNA (a string) of each individual in the population, to the target string. The fitness will then be the number of characters (genes) that are the same as the characters in the target string. In this example the order of the genes is important.

Based on these fitness values a new generation can be created. The first step in doing so is selection. The selection method is what is used to select which members of the population are used in crossover. A simple example is roulette wheel selection where each individual has a probability of being selected based on its fitness value. Once two parents are selected, crossover takes place. Crossover takes the genes of both parents and applies some function to mix these genes to form offspring. Along with crossover, mutation is often used. Mutation is when, based on some probability, a random gene is mutated into something else. The simplest form of this is where a gene is replaced by a randomly generated new gene. This helps maintain diversity in the population. Another method that can be included in making a new generation is elitism. Here a certain number of the most elite (highest fitness) members of the population are directly transferred to the new generation. This process of creating new generations and calculating their fitness is then repeated until a satisfactory result is reached. In this example a perfect fitness of 26, the number of characters in the target string, should be easy to reach.

This project will apply some similar methods as similar papers discussed in Section 2.6. The classes will form the DNA, each class having certain attributes assigned to it such as room and time. Another similarity is calculating the fitness based on all the constraints set for the timetable. This implementation aims to improve on those existing methods by using a real life dataset, introducing a novel crossover method, and testing various parameters and their effects on performance.

4.2 Development Process

To start developing this project a very basic genetic algorithm was first created as a foundation for the rest of the code. The code for this is based on a video created by Kryzarel.[1] In this video he shows an example similar to the example mentioned in Section 4.1 where he creates a

[1]https://www.youtube.com/c/Kryzarel

simple GA that generates text. His code is written in such a way that the DNA class could be easily adapted to other types instead of the default string used in the example.

The first step in building on this foundation is to have some way of reading the dataset used in this project. To process all the data from the dataset, an XML parser is made. This can read all the nodes, their attributes and any child nodes it has. This information is then stored in the corresponding object. The class object for example will contain information such as, the ID of the class, the possible rooms, possible times and the preferences for the rooms. All these genes are then added to the genes list inside the DNA.

The next step in developing this code was to determine what the DNA will consist of. As mentioned in Section 3.1 the dataset stores all possible times and rooms inside each class node and uses this to store the solution as well. For this reason each gene in the DNA was initially a class object. Each one of these objects will contain all the information found within the XML file. However, this turned out to not be a good solution since it would require making a clone of each of the originally created class objects. Considering there are 896 class objects in one DNA object, and 500 DNA objects in each generation, this would require a large amount of unnecessary object cloning. To solve this a special solution gene class is created that only contains information essential to creating a solution. It contains a solution time, solution room and class ID.

Now that the structure for DNA and genes is in place, a function needs to be created that can generate the initial population. This function takes a class object as a parameter, picks a random room and time based on this class and then uses this to create a new solution gene. When creating the initial population the master list of classes can be looped through, passed to this function, and then the resulting genes will be added to the gene list of the DNA object. All 500 DNA objects in the population will do this to form the first generation.

Creating new generations will require a new generation function to be made. This will include crossover, mutation and elitism. The easiest part of this is the elitism, this will pass a number of DNA directly to the next generation. This way the fittest members of the population will stay in the gene pool until they are out-performed by new generations.

For the crossover function some sort of parent selection function needs to be made first. For this project tournament selection is used. Tournament selection selects a number of random members of the population. Out of these, the individual with the highest fitness is chosen as a parent (Goldberg and Deb 1991). This process is repeated until enough parents are selected to create a new population. This means individuals with a higher fitness will have a higher chance of being a parent and can occur multiple times in the parent pool. The tournament size is something that can be experimented with to find the most optimal number.

To create a child, a crossover function is used. This project uses discrete recombination where two parents create a single child. This method goes over each gene and decides, based on a 50% chance for each parent, which parent passes its gene on to the child (Mühlenbein and Schlierkamp-Voosen 1993). In a code example this would look as follows

$$c[\,i\,] = (r < 0.5)\ \ ?\ \ p1[\,i\,] : p2[\,i\,];$$

Here the r is a random number between 0 and 1, p1 and p2 are arrays with the parents' genes and c is an array with the new child's genes.

Mühlenbein and Schlierkamp-Voosen (1993) explains this best using the following formulas. If $(x_1, ..., x_n)$ and $(y_1, ..., y_n)$ are the genes of the parents, then the child genes are generated using $z_i = \{x_i\} or \{y_i\}$

The next step is the fitness function. This is one of the more time consuming steps since it requires a fitness function to be written for every one of the constraints used in the dataset. The constraints are split into two lists, the hard constraint list and the soft constraint list. The llr dataset only contains eight soft constraints. However, the SPREAD constraint, even though it is a hard constraint, acts more like a soft constraint. This is because it requires minimisation of overlap between the classes in its group. Because of the use of the word minimisation in the constraint description, the function for this returns a fraction of its weight dependent on the number of violations within its group. The main difference between soft constraints and hard constraints is that soft constraints will return its preference when the constraint is adhered to, while a hard constraint returns 0. When a hard constraint is violated it will return a penalty, while a soft constraint returns 0. In other words, soft constraints use positive reinforcement while hard constraints use negative reinforcement.

The fitness function has to be called for each member of the population. This function then loops through all the soft and hard constraints and passes the genes to the constraint to calculate whether it passes or violates each of the constraints. After that, other factors such as the time preference and room preference of each class is added to the calculation. One extra constraint is added to the list of constraints manually which is the ROOM_CONFLICTS constraint. This will have a list of all the classes and checks if any of them are in the same room at the same time. This constraint has the highest weight because it is one of the most fundamental aspects of timetabling. There are some exceptions to this as some classes share a CAN_SHARE_ROOM constraint which means they are allowed to share a room at the same time.

With the main GA now working, multi-threading was added to the fitness calculations to increase speed. Because each member of the population needs to have its fitness calculated separately, each of these calculations could run simultaneously. The rest of the program waits until all the threads have finished calculating the fitness. This has about halved the time it takes per generation and has thereby made testing much easier.

4.3 Creating a Custom Crossover Function

For this project a custom crossover function is made. This function is based on the idea that instead of genes being passed down mostly randomly, they will be passed down based on performance. This is similar to how selection occurs in nature. Animals do not always select their mates based solely on the overall performance of the mate, but rather on specific beneficial traits. To attempt to find which genes within an individual are beneficial, and which ones are not, a Violations property is assigned. Every time a gene is involved in violating a constraint, this gene has its violations property increased by one.

From here, it would be possible during crossover to always pick the most beneficial gene. However, this is likely a too greedy approach and would lead to a severe decrease in diversity. Instead, a probability of selection is assigned to each gene based on the number of violations. For example, the i^{th} gene from parent 1 has x violations and the i^{th} gene from parent 2 has y violations. The gene from parent 1 will then have a probability of $\frac{y}{x+y}$ and the gene from parent 2 will have a probability of $\frac{x}{x+y}$.

Further research would need to be done to determine if there are better approaches to calculating the probability for each gene to pass on. This could include looking into using more than 2 parents and applying different selection methods for the genes in each position. Later on this method will be tested and compared to the well established discrete crossover method.

4.4 Practical Code Examples

4.4.1 Reading the XML data

```
XmlNode node = classes.ChildNodes.Item(i);
List<XmlNode> TimeNodeList = new List<XmlNode>();
List<XmlNode> RoomNodeList = new List<XmlNode>();
foreach (XmlNode n in node.ChildNodes) {
    if (n.Name == "time") {
        TimeNodeList.Add(n);
    }
    if(n.Name == "room") {
        RoomNodeList.Add(n);
    }
}
KlasTime[] TimeArr = ReadTimes(TimeNodeList);
```

```
private KlasTime[] ReadTimes(List<XmlNode> TimeList) {
    KlasTime[] result = new KlasTime[TimeList.Count];
    for(int i  = 0; i<TimeList.Count; i++) {
        result[i] = new KlasTime(GetStringAttr(TimeList[i], "days"),
        ↪  GetIntAttr(TimeList[i], "start"), GetIntAttr(TimeList[i],
        ↪  "length"), GetIntAttr(TimeList[i], "breakTime"),
        ↪  GetDoubleAttr(TimeList[i], "pref"));
    }
    return result;
}
private string GetStringAttr(XmlNode node, string name) {
    return node.Attributes.GetNamedItem(name).Value;
}
```

As an example of how the XML dataset is loaded into the algorithm, the above code shows how class times are read. The first code block gets executed for each class node that is found in the class list. Each class has both room and time nodes which are added to their respective lists. This time list is then passed to the read times function seen in the second code block. This function will turn the XML class time nodes into KlasTime objects. The node list is looped through and the attributes of each node are used to create a new KlasTime object. To shorten the code and make it more readable, functions like GetStringAttr are made. GetDoubleAttr and GetIntAttr work in the same way, they all return the value found in the attribute of the given node. A similar, slightly more complicated, approach is used to read all the rooms. Once all the data for each class is read it is used to create a new Klas object and add it to the Klas list.

4.4.2 Creating Genes

```
private SolutionGene GetRandomSolutionGene(Klas k) {
    Room a = (k.Rooms.Length>0) ? k.Rooms[LockedRandomInt(0, k.Rooms.Length)]
    ↪ : null;
    KlasTime b = (k.Times.Length>0) ? k.Times[LockedRandomInt(0,
    ↪ k.Times.Length)] : null;
    SolutionGene output = new SolutionGene(k.ID, a, b);
    return output;
}
```

The SolutionGene object is created to store a possible solution for a class without effecting the original Klas object. Each gene stores a reference to one of the rooms and times available to the corresponding Klas object along with the ID. When creating the initial population the GetRandomSolutionGene function is used to generate random possible solutions for each class. If a class does not have any rooms or any times assigned to it, then null will be assigned to the corresponding variable in the gene.

4.4.3 Selection function

```
private DNA ChooseParent() {
    int tournamentSize =  (Int32)Math.Floor(Population.Count * 0.1);
    DNA[] tournamentMembers = new DNA[tournamentSize];
    for (int i = 0; i < tournamentSize; i++) {
        DNA x;
        do {
            int id = LockedRandomInt(0, Population.Count);
            x = Population[id];
        } while (tournamentMembers.Contains(x));
        tournamentMembers[i] = x;
    }
    Array.Sort(tournamentMembers, CompareDNA);
    return tournamentMembers[0];
}
```

The choose parent function uses tournament selection. First it calculates the tournament size as a percentage of the population. A too high tournament size would mean the parent pool would have little diversity, while a too low tournament size would mean more low fitness individuals would be in the parent pool. After initialising the tournament members array using the tournament size, a for loop loops through each position of said array. This position is then assigned a random member of the population. To prevent duplicates, a do-while loop is used to check if the randomly picked individual is already in the tournament or not. After the tournament members array is filled, the array is sorted based on the fitness of each member. This means the member with the highest fitness will be in the 0th position in the array. This member is then returned.

4.4.4 Crossover

```
for (int i = 0; i < Population.Count; i++) {
    if (i < Elitism) {
        NewGenerationArr[i] = Population[i];
    } else {
        DNA parent1 = ChooseParent();
        DNA parent2 = ChooseParent();
        DNA child = parent1.Crossover(parent2, i);
        child.Mutate(MutationRate);
        NewGenerationArr[i] = child;
    }
}
```

```
public DNA Crossover(DNA otherParent, int id) {
    DNA child = new DNA(id, Genes.Length, random, getRandomGene,
     ↪  fitnessFunction, shouldInitGenes: false);
    for (int i = 0; i < Genes.Length; i++) {
        child.Genes[i] = LockedRandomDouble() < 0.5 ? Genes[i] :
         ↪  otherParent.Genes[i];
    }
    return child;
}
```

To generate a new population the for-loop in the first code block loops through the old population. The first number of members get added to the next generation directly based on the value of Elitism. This is because they have the highest fitness of the old population. Next, two parents are selected using the selection method mentioned before. The crossover method in the second code block uses Discrete crossover. First it creates a new DNA object which will be the child. Then a for-loop goes through all the genes of the child and randomly decides whether to use the i-th gene from parent 1 or parent 2. Once this is done it returns the child. The mutate method used in the first code block goes over all the genes and generates a random number. If the random number is lower than the mutation rate then that gene will be replace by a random gene using the GetRandomSolutionGene function.

5. Experiments

5.1 Experimental settings

The most important thing to solve in this dataset is room conflicts. This is when multiple lessons are scheduled to be taught on the same day at the same time in the same room. This is obviously not possible and should be avoided at all costs. For this reason, the weight of violating this constraint is higher than that of any other. The weight is $100,000$ for each violation. This makes it so that these violations will always have the priority of all others. This is why, along with the minimisation of the fitness value, the minimisation of room conflicts is one of the things used to compare the results of experiments.

5.1.1 Selection methods

For the selection method a lot of different tests can be done. The methods that will be compared here are 3 variations of rank based selection, sex based selection, tournament selection and completely random selection. Whether a GA using random selection is really a GA is debatable, since it removes the key part of a GA which is evolution based on fitness. However, it can still function as somewhat of a baseline to compare the other methods to.

All of these methods, except for random, have in common that they focus on the higher fitness individuals, without ignoring the lower fitness individuals. This is an important part in any selection method. This can be especially fine tuned in rank based selection. Rank based selection alone could have a large amount of tests done all using different distributions of probabilities for the ranks. Three variations will be tested here.

The Rank 1.5 and Rank 2.5 methods both use an equations for assigning probability which is based on a paper by Whitley (2000). The formula is as follows:

$$(int)(Population.Count \cdot (bias - \sqrt{bias^2 - 4.0 \cdot (bias - 1) \cdot r})/2.0/(bias - 1))$$

The bias used in this formula refers to the bias the algorithm will have towards more fit individuals. The higher the bias, the more the algorithm favours high fitness individuals. The (int) at the start of the formula means that any decimals are removed, without any rounding taking place The other method is a more simplistic approach which is essentially like throwing two dice with 500 sides each, then subtracting 501 from the total and ignoring any negative values. If a value is negative, throw until it is not. The reason it subtracts 1 extra, is because it retrieves an individual from an array, and the last position in an array is always $(count - 1)$.

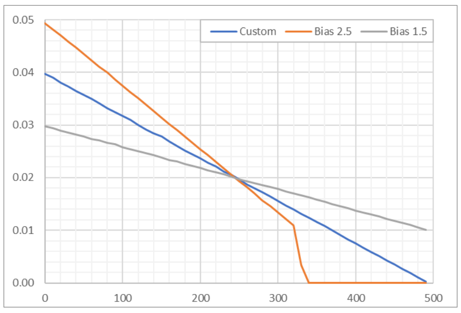

Figure 5.1: Probability for each rank to be selected, per group of 10

$$RandomInt(0, Population.Count) + RandomInt(0, Population.Count) - Population.Count - 1;$$

Both formulas were used to generate $10,000,000$ random numbers and the results were each stored in its own array. The range of 0 to 500 was split into sections of 10 to make it easier to calculate how often each number occurs and thus what its probability is compared to the rest. The result of this can be seen in Figure 5.1. The probabilities all decrease linearly the higher the rank, except for the one with a bias of 2.5. That one seems to cut off around 350, and then the probability becomes 0. It will be interesting to see how this will effect the performance of the Rank Based selection method.

5.1.2 Crossover methods

There are many different methods of crossover used in GAs. Some examples of these will be used in this section to compare their impact on performance. These are the crossover methods that have been tested:

- Discrete crossover
- Violation based crossover

Discrete crossover uses a random number to decide, for each gene position of the child, from which parent it will receive the gene in that same position. For example, starting at the first gene, a random number between 0 and 1 is generated. If the number is lower than 0.5 then the child will receive the first gene from the second parent, otherwise it will receive the first gene from the first parent. This process is repeated for each position in the DNA.

Violation based crossover is an adaptation on Discrete crossover created specifically to use for this project. Each gene will be given a 'Violations' property which is increased every time that gene is involved in violating a constraint. During crossover, instead of each parent having an equal chance of passing on each gene, the chance will be dependent on the violations property. For example, the first gene on the first parent has 2 violations, while the first gene on the second parent has 5. This adds up to 7, giving the first parent a 5 in 7 chance to pass on its gene and the second parent a 2 in 7 chance. Notice that the numbers 5 and 2 are swapped around here because the gene with the least violations will have the highest chance of being passed on.

5.1.3 Mutation Rate

The mutation rate is an important part in genetic algorithms. It introduces a certain amount of randomness that helps increase the diversity in genetic algorithms and decreases early converging. However, a good balance needs to be found. When there is too much mutation the algorithm will never reach the optimal solution as there is too little good solutions left in the population.

To test what a good amount of mutation is and to demonstrate what too much mutation does, the following values were tested: 0.01, 0.001, 0.1 and 0. Out of these values 0.01 is what is used by the algorithm by default as this was estimated to be a good balance.

5.1.4 Tournament Ratio

The tournament ratio is the amount of individuals used in each tournament. Instead of using a fixed value, this program uses a certain ratio. With a default population of 500, the default ratio of 0.02 results in tournament sizes of 10. Other values for the tournament ratio that will be tested are 0.01, 0.1 and 0.2.

What is important about the tournament size is that the higher it is, the less diversity the next generation will have, and the lower it is, the more diversity it will have. The reason for this is that a high tournament size will likely pick a lot of the same individuals for crossover, as these individuals have the highest fitness. While a low tournament size will make the selection more random and thus will keep more diversity. Technically speaking tournaments can never increase diversity, only maintain it.

5.1.5 Elitism

Elitism is when the top n number of individuals are added directly to the next generation. To test the effects of this the following values of elitism are tested: 5, 0, 10, 15. The default value used by this program is 5. The idea is that elitism makes sure that the genes of the highest fitness individuals does not get lost. This makes it so that the highest fitness in each next generation can never be lower than that of the previous generation. However, a too high value of elitism could have adverse effects on the diversity of the population.

5.1.6 Population Size

The size of each population determines the amount of individuals that need to be generated for each generation but also the amount of time the fitness function needs to be called. This has a big impact on the time it takes between each generation, as the fitness function is the most time consuming part of the algorithm already. It does however come with the advantage that a larger population increases the diversity and has a higher chance of finding beneficial genes for the gene pool.

The current population size is set to 500 for the default algorithm. This is purely an estimate vaguely based on the time it took to generate new populations. With 500 individuals per generation it takes around 5 seconds per generation which seems like a manageable amount. A generation size of 250 will be tested to see if this trade-off in per generation performance will be compensated with higher efficiency. To test the opposite, a population size of 1000 will also be tested. The goal is to find out what the best balance is in performance and speed.

5.2 Experimental results

5.2.1 Selection results

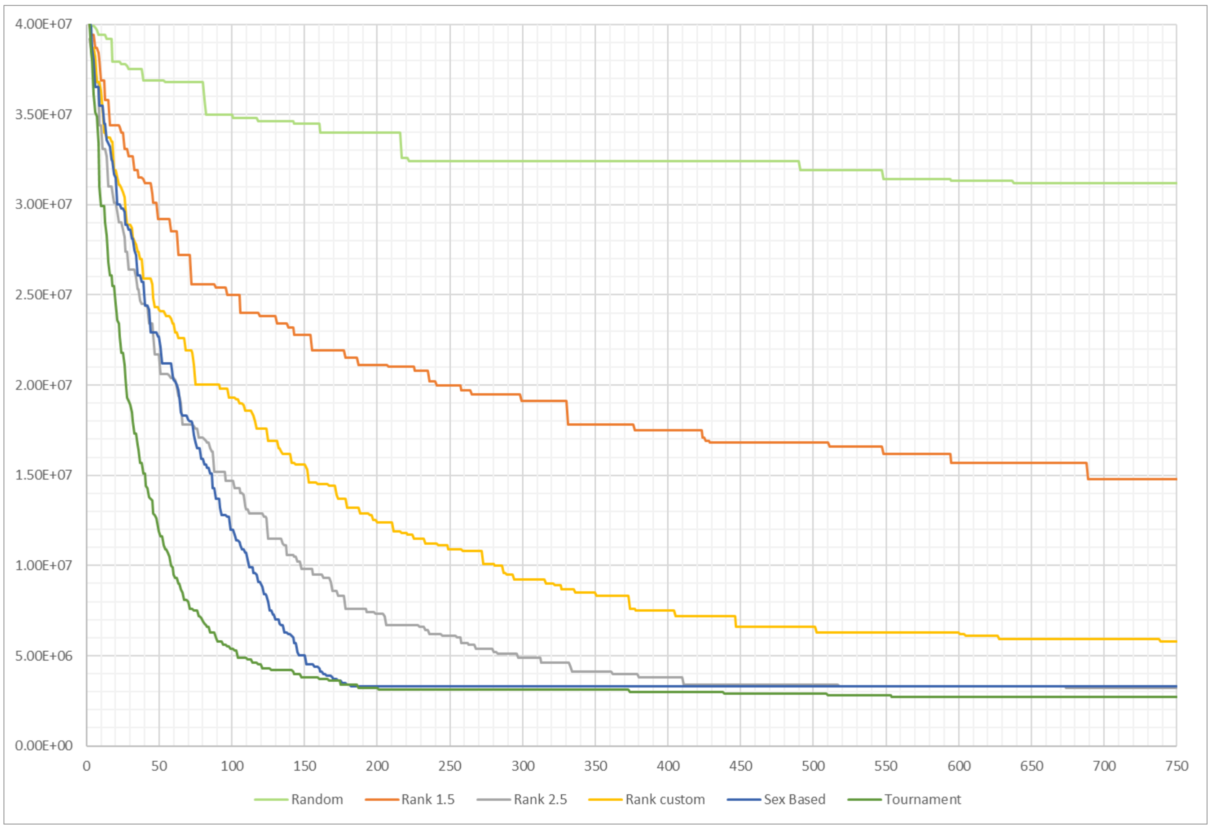

Figure 5.2: Comparing various Selection methods

As can be seen in Figure 5.2, Tournament selection, Sex Based selection and Rank selection(2.5) all achieved close results given enough time. These tests were done until a later generation than all the other tests because of how big the differences in convergence speed is. The custom rank based method and the rank based method with bias of 1.5 both only started to stabilise after 500 generations.

It is interesting to see how large the difference is in performance between a bias of 1.5 and a bias of 2.5. The fact that bias 2.5 performed so much better than 1.5 could suggest that the timetabling problem favours a slightly greedy approach as opposed to a more diversity focused approach. It was unexpected to see the random selection performing as well as it did, it in no way produced a valid solutions but it did make some progress.

Out of the three methods that performed almost equally well, Tournament selection is arguably the best one. The reason for this is because it not only reached the lowest fitness, it also reached this low fitness significantly faster than the other methods. It did however stabilise at almost the same stage as sex based selection did.

5.2.2 Crossover results

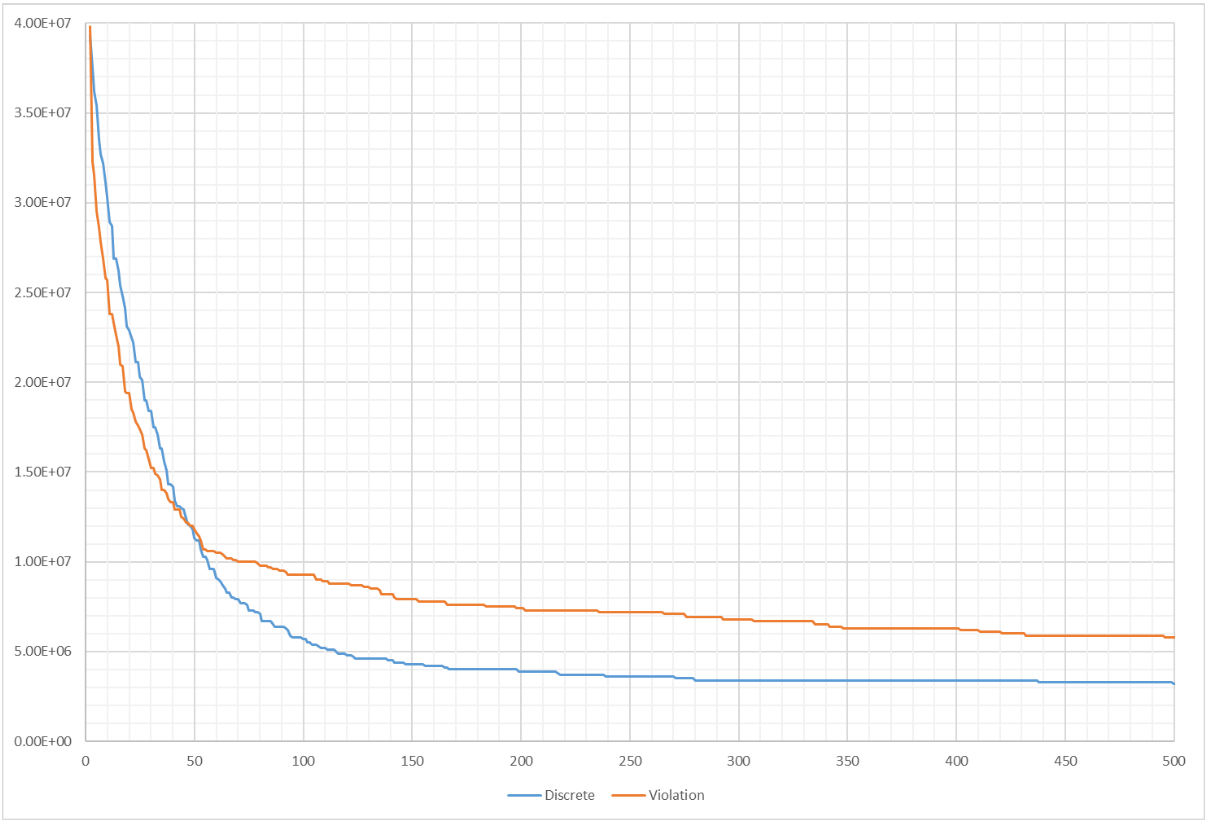

Figure 5.3: Comparing Discrete crossover to violation based crossover

The results of comparing violation based crossover to discrete crossover can be seen in Figure 5.3. The graph shows how violation based crossover seems to converge much earlier than discrete crossover does. In violation based crossover convergence starts between generation 50-100, while it seems to start between 100-150 for discrete crossover. This has caused the discrete crossover to have 32 room conflicts while violation based has a higher 58 room conflicts. This can also be seen in their respective fitness values of $3,202,380$ and $5,802,529$.

The likely explanation for this early converging is that the violation based system is too greedy. This system is likely to favour the same genes over and over again which causes a reduction in diversity. Without enough diversity it is unlikely the algorithm will ever, or at least within a reasonable time, find the optimal solution.

Another observation is that the average time it took between each generation was 4872 for discrete and 5030 for violation based. This is likely due to the extra calculations necessary to assign the violation variables during each constraint calculation, and comparing fitness for each gene.

5.2.3 Mutation Rate Results

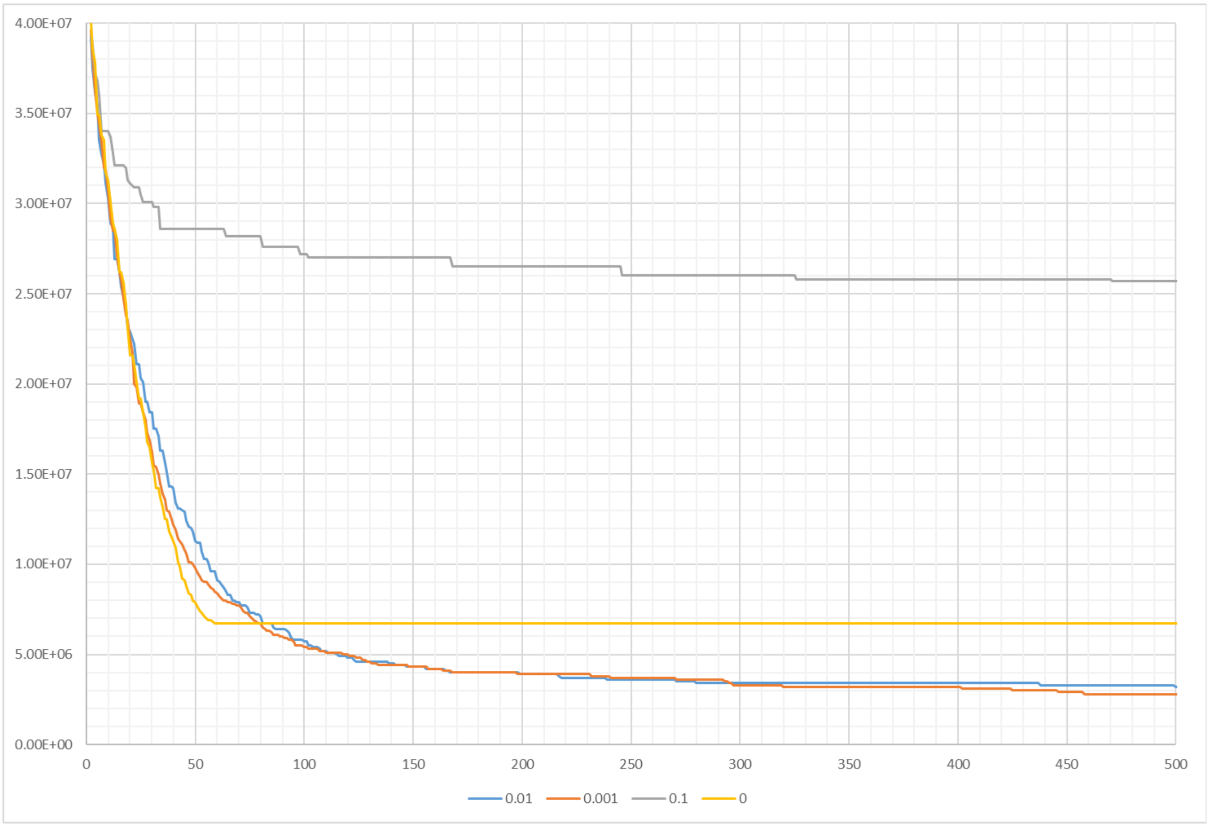

Figure 5.4: Comparing various mutation rates

As can be seen in Figure 5.4 using a mutation rate of 0.001 performed the best. The final number of room conflicts for each mutation rate is 32, 28, 257 and 67. This is for 0.01, 0.001, 0.1 and 0 respectively.

When looking at the mutation rate of 0.1 it is clear to see that too much mutation dilutes the amount of fit individuals in the population to a point that no more real progress is made. With every 1 in 10 genes being mutated, fit individuals barely have a chance to pass on their genes properly.

The results for 0.01 and 0.001 are rather similar. They have a difference of 0.01 having 4 more room conflicts than 0.001. These results are less interesting as they are just a matter of fine tuning the algorithm.

The tests for 0.1 and 0 were expected to perform poorly. These tests are more as a demonstration of the importance of a good mutation rate. When the mutation rate is set to 0, the algorithm will eventually fixate itself on 1 solution and make no further progress. This is clearly demonstrated in Figure 5.4 in the line for mutation rate 0. Once it reaches only 67 room conflicts in generation 59, it seizes to make any progress. It did however, reach this fitness faster than any of the other mutation rates.

This raises the question whether a variable mutation rate might be beneficial. For example if the mutation rate is initially 0, but once progress slows down the mutation rate could be automatically increased. This could be an interesting topic for further research.

5.2.4 Tournament Ratio Results

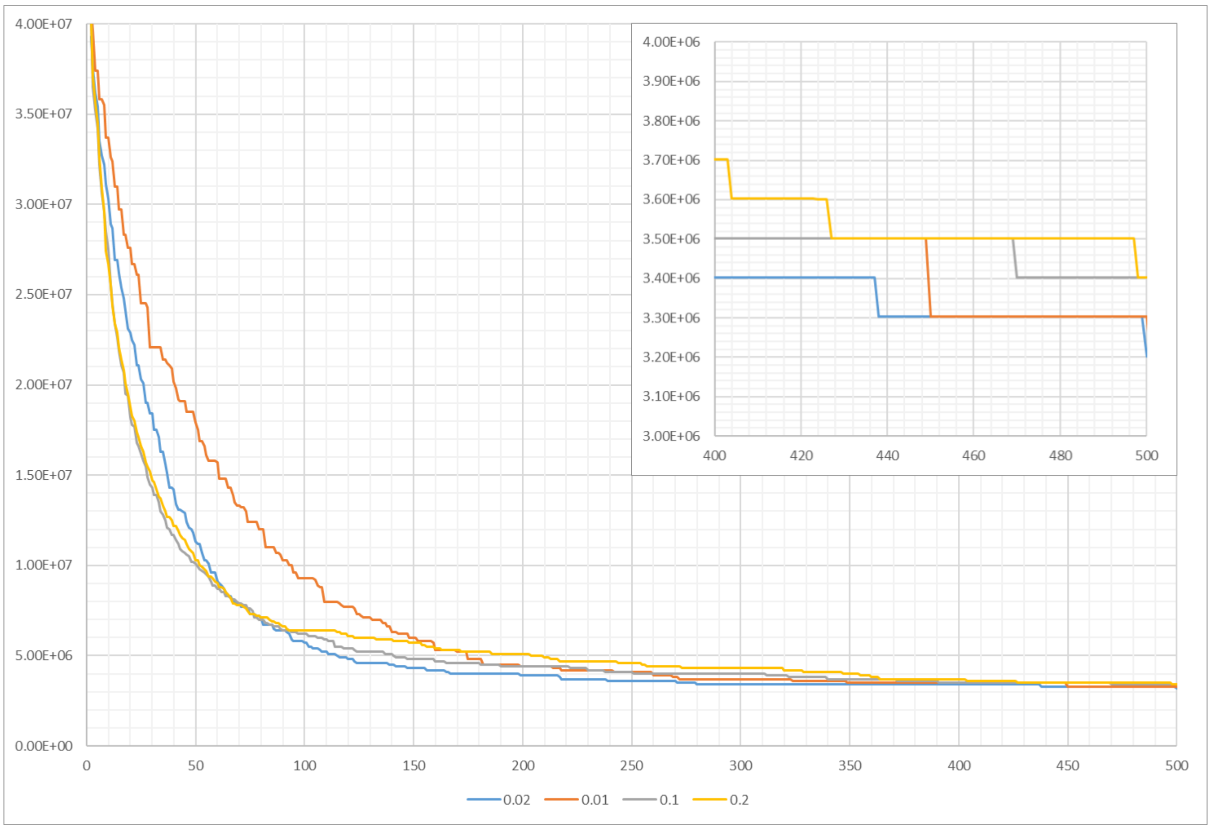

Figure 5.5: Comparing various tournament ratios

The graph in Figure 5.5 consists of two parts. The main graph shows the fitness of each tournament ratio through 500 generations. The smaller graph is a more zoomed in image of the last 100 generations.

It was unexpected to see how large the difference in convergence speed is between a tournament ratio of 0.01 and 0.02. Each leading to a tournament size of 5 and 10 respectively. The ratio of 0.01 only started converging after around 200 generations whereas all the others did between 100 to 150. However, in the end they all ended up with a very similar fitness. In fact the fitness is so close that the difference is negligible.

5.2.5 Elitism Results

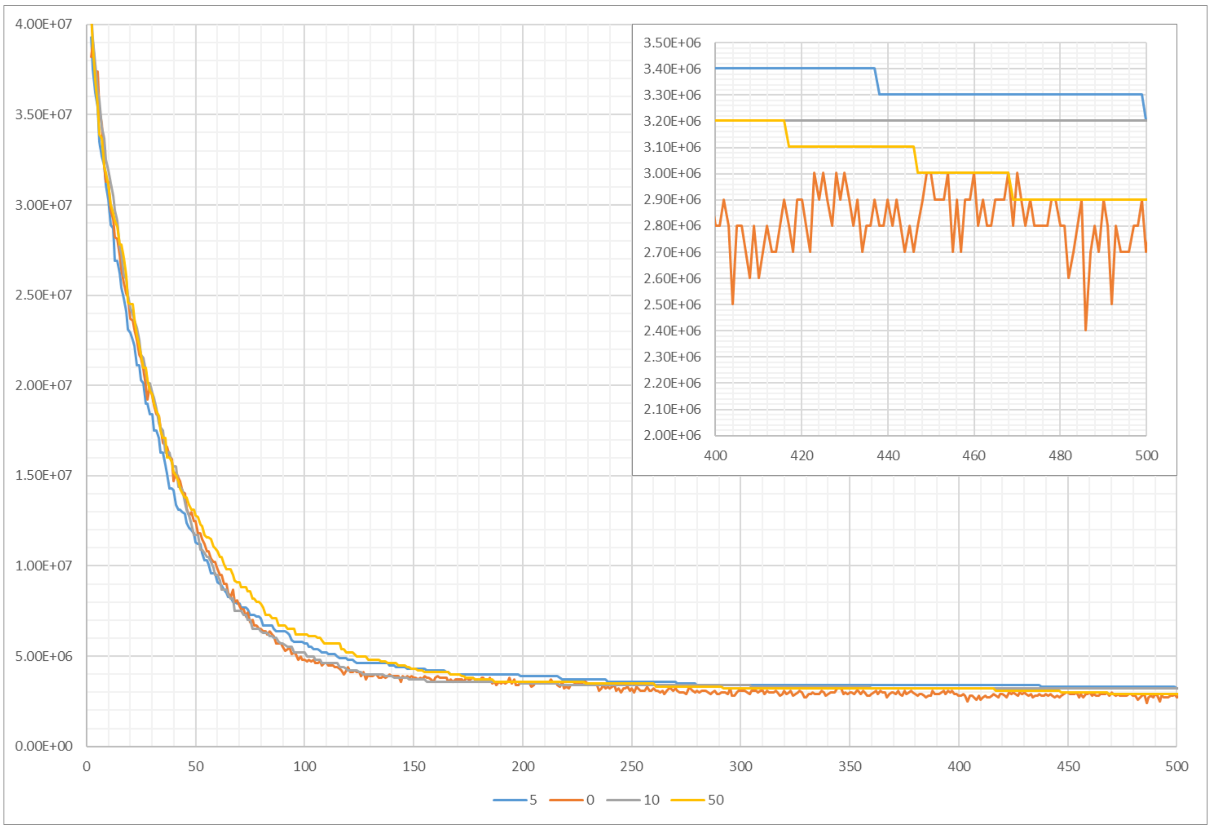

Figure 5.6: Comparing various amounts of elitism

This experiment turned out to be less interesting than expected. The fitness for all the elitism values is almost the same in every generation. It is clear to see that the bottom line in Figure 5.6 is the one without any elitism as it is the most shaky. What was somewhat surprising is that even though the line without any elitism goes up and down, it rarely goes higher than the other lines especially near the end. This shows that, at least for this problem, elitism is not necessarily beneficial.

5.2.6 Population Size Results

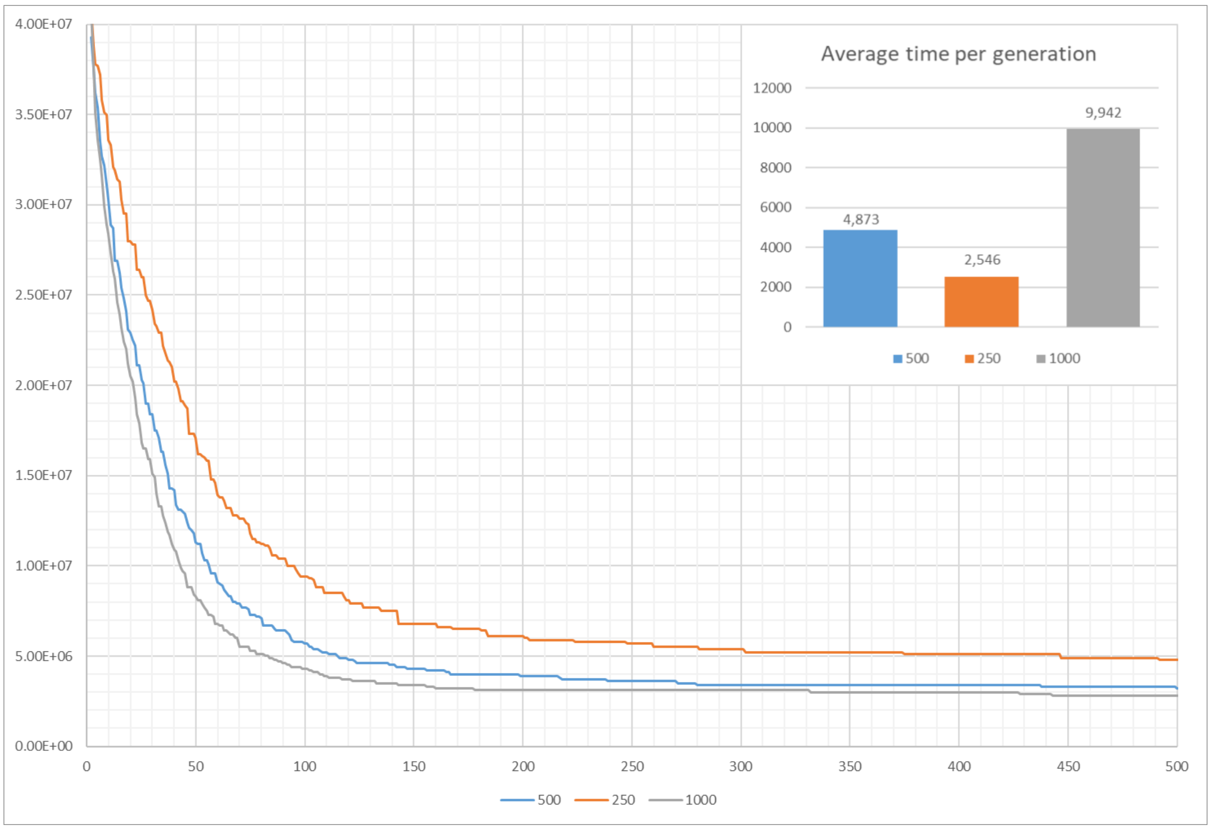

Figure 5.7: Comparing various population sizes

Figure 5.7 shows that the best fitness was achieved by a population size of 1000. This was to be expected as a higher population size is generally beneficial. What is more important however is to compare the time increase. The smaller graph shows that doubling the population size almost exactly doubles the time between generations as well. For this to be worth it the performance of the larger population should also close to doubled. however, the graph shows this is not the case. Going from 250 to 500 does make a significant difference in fitness. So using a population size of 500 seems to be worth it. A population size of 1000 however only barely performs better than 500 but does take twice as long. This shows that overall, using a population size of around 500 is the best option.

5.2.7 Discussion of all results

Looking at all the results in this section, the following should achieve the fittest final result:

- Discrete crossover
- Mutation rate of 0.001
- Tournament ratio of 0.02
- No elitism
- Population size of 1000

However, for the algorithm to be most efficient the population size should be turned down to around 500.

6. Conclusion

6.1 Evaluation

The overall performance of the algorithm was reasonably good. Unfortunately there was no clear data available to compare this project to, like initially planned. So it is difficult to say whether this specific approach is successful compared to others. However, the main goal of this project was to show the potential of genetic algorithms in solving timetabling problems, and to conduct tests. The program created successfully accomplished both those goals. Out of the 896 classes in the dataset, having only 28 room conflicts is reasonably good. However, if this were to be used in a real life situation, scheduling staff would still need to tweak the result manually as with most scheduling solutions.

As for testing, this same program could be reused and adapted easily to perform many different types of tests on solving this dataset. More selection or crossover methods could be added to the program and it would still output useful CSV files for comparing results. The results of the tests conducted in this project conclude that most of the estimates were already optimal, except the mutation rate could be reduced to 0.001 and elitism could be removed for slightly better performance.

6.2 Further research

What is probably most interesting about genetic algorithms is that you can be creative in designing new genetic operators. Sex based selection for example, something that is based on biology and it performed almost as well as tournament selection. There is still a lot of potential in finding new ways of selection, crossover, mutation, elitism or maybe even something completely new. As for this specific implementation, more research can be done to see if a different way of calculating the probability of genes in the violation method could improve its performance. Another idea is testing the effects of adding an individual to the population with completely random genes and making it guaranteed to be selected at least once. This could increase diversity, and if used only after a large amount of generations, could perhaps lead the algorithm to increase fitness again.

A. Appendix

```
public float DIFF_TIME(List<SolutionGene> cGenes) {
    float result = Pref;
    int adhered = cGenes.Count;
    for (int i = cGenes.Count-1; i >= 0; i--) {
        bool violation = false;
        SolutionGene c1 = cGenes[i];
        cGenes.RemoveAt(i);
        int minTime = c1.SolutionTime.Start;
        int maxTime = minTime + c1.SolutionTime.Length;
        for(int j = 0; j < cGenes.Count; j++) {
            SolutionGene c2 = cGenes[j];
            if(i != j) { //make sure you're not comparing the same Klas to itself
                BitArray andResult = (BitArray)c1.SolutionTime.Days.Clone();
                andResult.And(c2.SolutionTime.Days);
                if(andResult != AllFalse) { //this means that at least one day overlaps, so we must check if times overlap
                    if(c2.SolutionTime.Start >=minTime && c2.SolutionTime.Start <= maxTime && !violation) {
                        //This checks if its whithin the times and if there's already been a violation
                        if (IsHardConstraint) {
                            return Pref;
                        }
                        violation = true;
                        adhered--;
                        c1.Violations++; //mark that this gene caused a violation
                    }
                }
            }
        }
    }
    if (IsHardConstraint) {
        return 0;
    }
    float temp = (float)(adhered) / (float)(ClassIDs.Length);
    result = temp * Pref;
    return result;
}

public float SPREAD(List<SolutionGene> cGenes) {
    int adhered = cGenes.Count;
    for (int i = cGenes.Count - 1; i >= 0; i--) {
        bool violation = false;
        SolutionGene c1 = cGenes[i];
        cGenes.RemoveAt(i);
        int minTime = c1.SolutionTime.Start;
        int maxTime = minTime + c1.SolutionTime.Length;
        for (int j = 0; j < cGenes.Count; j++) {
            SolutionGene c2 = cGenes[j];
            if (i != j) { //make sure you're not comparing the same Klas to itself
                BitArray andResult = (BitArray)c1.SolutionTime.Days.Clone();
                andResult.And(c2.SolutionTime.Days);
                if (andResult != AllFalse) { //this means that at least one day overlaps, so we must check if times overlap
                    if (c2.SolutionTime.Start >= minTime && c2.SolutionTime.Start <= maxTime && !violation) {
                        //This checks if its whithin the times and if there's already been a violation
                        violation = true;
                        adhered--;
                        c1.Violations++; //mark that this gene caused a violation
                    }
                }
            }
        }
    }
    float temp = (float)(adhered) / (float)(ClassIDs.Length);
    var result = temp * Pref;
    return result;
}

public float SAME_STUDENTS(List<SolutionGene> cGenes) {
    int geneCount = cGenes.Count;
    int studentConflicts = 0;
    for(int i = geneCount-1; i >= 0; i--) {
        var c1 = cGenes[i];
        cGenes.RemoveAt(i);
        for (int j = 0; j < cGenes.Count; j++) {
            var c2 = cGenes[j];
            if (c1.ID != c2.ID) {
                var c1Start = c1.SolutionTime.Start;
                var c2Start = c2.SolutionTime.Start;
                var c1End = c1Start + c1.SolutionTime.Length;
                var c2End = c2Start + c2.SolutionTime.Length;
                var sameDay = false;
                foreach (bool b1 in c1.SolutionTime.Days) {
                    foreach (bool b2 in c2.SolutionTime.Days) {
                        if (b1 && b2) {
                            //Check if they have any days in common, if yes break both loops
                            sameDay = true;
                            break;
                        }
                    }
                    if (sameDay) { break; }
                }
                if (sameDay) {
                    if ((c1Start <= c2Start && c1End >= c2End) || (c2Start <= c1Start && c2End >= c1End)) {
                        studentConflicts++;
                        c1.Violations++;
                    } else if (c1Start == c2End || c2Start == c1End) { //back-to-back
```

```
                        int distance = c1.SolutionRoom.CalculateRoomDistance(c2.SolutionRoom);
                        if (c1.SolutionTime.Length >= 18 && distance > 1000) {
                            //if its a long lesson, the limit is 1000
                            studentConflicts++;
                            c1.Violations++;
                        } else if (distance > 670) {
                            //otherwise, the limit is 670
                            studentConflicts++;
                            c1.Violations++;
                        }
                    }

                }

            }
        }
    }
    return (studentConflicts > 0) ? Pref : 0; //if there was any conflicts, return pref (cz its always R), otherwise return 0
}

private float SAME_ROOM(List<SolutionGene> cGenes) {
    int numberOfRoomsShared = 0;
    int nrOfGenes = cGenes.Count;
    for(int i = cGenes.Count-1; i>=0; i--) {
        var c1 = cGenes[i];
        cGenes.RemoveAt(i);
        for (int j = 0;j<cGenes.Count;j++) {
            var c2 = cGenes[j];
            if (c1.ID != c2.ID) {
                if (c1.SolutionRoom.ID == c2.SolutionRoom.ID) {
                    numberOfRoomsShared++;//this is a good thing
                } else {
                    c1.Violations++;
                }
            }
        }
    }
    if (IsHardConstraint) {
        if (numberOfRoomsShared == nrOfGenes - 1) {
            //if all rooms are shared, that's good
            return 0;
        } else {
            //if not all rooms are shared, return pref. (for R thats 40, for P its -40)
            return Pref;
        }
    } else { //since I know this dataset only has this as a hard constraint, this shouldn't happen.
        return 0;
    }
}

private float BTB(List<SolutionGene> cGenes) {
    int numberOfBTB = 0;
    List<SolutionGene> goodGenes = new List<SolutionGene>();
    foreach(var g1 in cGenes) {
        foreach(var g2 in cGenes) {
            if (g1.SolutionTime.Days.Equals(g2.SolutionTime.Days)) {
                if (g1.SolutionTime.Start + g1.SolutionTime.Length == g2.SolutionTime.Start || g1.SolutionTime.Start + g1.SolutionTime.Length == g2.SolutionTime.Start + 1) {
                    //checks if the end of g1 is equal to the start of g2 (or 1 slot later)
                    if (g1.SolutionRoom.ID == g2.SolutionRoom.ID) { //check if its the same room
                        numberOfBTB++;
                        if (!goodGenes.Contains(g1)) {
                            goodGenes.Add(g1);
                        }
                    }
                }
            }
        }
    }
    foreach(var gene in cGenes) {
        if (!goodGenes.Contains(gene)) {
            gene.Violations++;
        }
    }
    if(numberOfBTB == cGenes.Count - 1) {
        return IsHardConstraint ? 0 : Pref;
        //satisfied therefore no penalty
    } else {
        return IsHardConstraint ? Pref : 0;
        //violated therefore penalty (unless p then this is a good thing)
    }

}

private float BTB_TIME(List<SolutionGene> cGenes) {
    int numberOfBTB = 0;
    List<SolutionGene> goodGenes = new List<SolutionGene>();
    foreach (var g1 in cGenes) {
        foreach (var g2 in cGenes) {
            if (g1.SolutionTime.Start + g1.SolutionTime.Length == g2.SolutionTime.Start || g1.SolutionTime.Start + g1.SolutionTime.Length == g2.SolutionTime.Start + 1) {
                //checks if the end of g1 is equal to the start of g2 (or 1 slot later)
                numberOfBTB++;
                if (!goodGenes.Contains(g1)) {
                    goodGenes.Add(g1);
                }
            }
        }
    }
    foreach (var gene in cGenes) {
        if (!goodGenes.Contains(gene)) {
            gene.Violations++;
        }
    }
    if (numberOfBTB == cGenes.Count - 1) {
        return IsHardConstraint ? 0 : Pref;
        //satisfies therefore no penalty
    } else {
        return IsHardConstraint ? Pref : 0;
        //violated therefore penalty (unless p then this is a good thing)
    }

}

private float SAME_TIME(List<SolutionGene> cGenes) {
    int numberOfSameTime = 0;
    List<SolutionGene> goodGenes = new List<SolutionGene>();
    cGenes.Sort((a, b) => {
        if (a.SolutionTime.Length > b.SolutionTime.Length) {
            return -1;
        } else if (a.SolutionTime.Length < b.SolutionTime.Length) {
            return 1;
        } else {
            return 0;
        }
    });
    for(int i = 0; i < cGenes.Count; i++) {
        if (i != cGenes.Count - 1) {
            var t1 = cGenes[i].SolutionTime;
            var t2 = cGenes[i + 1].SolutionTime;
            if(t1.Start <= t2.Start && t1.Start+t1.Length >= t2.Start + t2.Length) {
                numberOfSameTime++;
                if (!goodGenes.Contains(cGenes[i])) {
                    goodGenes.Add(cGenes[i]);
                }
            }
        }
    }
    foreach (var gene in cGenes) {
        if (!goodGenes.Contains(gene)) {
            gene.Violations++;
        }
    }
    if (numberOfSameTime == cGenes.Count - 1) {
        return IsHardConstraint ? 0 : Pref;
    } else {
        return IsHardConstraint ? Pref : 0;
        //violated therefore penalty (unless p then this is a good thing)
    }
}

private float SAME_START(List<SolutionGene> cGenes) {
    int numberSameStart = 0;
    List<SolutionGene> goodGenes = new List<SolutionGene>();
    SolutionGene[] genesArr = cGenes.ToArray();
    for (int i = cGenes.Count-1; i > 0; i--) {
        var g1 = cGenes[i];
        var t1 = g1.SolutionTime;
        cGenes.RemoveAt(i);
        for (int j = 0; j < cGenes.Count; j++) {
            var t2 = cGenes[j].SolutionTime;
            if (t1.Start >= t2.Start && t1.Start < t2.Start+6) {
                //if g1 and g2 are within the same 30min timeslot
                numberSameStart++;
                if (!goodGenes.Contains(g1)) {
                    goodGenes.Add(g1);
                }
            }
        }
    }
    foreach (var gene in genesArr) {
        if (!goodGenes.Contains(gene)) {
            gene.Violations++;
        }
    }
    if (numberSameStart == genesArr.Length - 1) {
        return IsHardConstraint ? 0 : Pref;
    } else {
        return IsHardConstraint ? Pref : 0;
        //violated therefore penalty (unless p then this is a good thing)
    }
}

private float SAME_DAYS(List<SolutionGene> cGenes) {
    int numberOfSameDays = 0;
    BitArray[] daysArray = new BitArray[cGenes.Count];
    SolutionGene[] genesArr = cGenes.ToArray();
    cGenes.Sort((a, b) => {
        if (a.SolutionTime.NrOfDays > b.SolutionTime.NrOfDays) {
            return -1;
        } else if (a.SolutionTime.NrOfDays < b.SolutionTime.NrOfDays) {
            return 1;
        } else {
            return 0;
        }
    });

    for (int i = cGenes.Count - 1; i > 0; i--) {
        var g1 = cGenes[i];
        var d1 = cGenes[i].SolutionTime.Days;
        daysArray[i] = d1;
        cGenes.RemoveAt(i);
        for (int j = 0; j < cGenes.Count; j++) {
            var d2 = cGenes[j].SolutionTime.Days;
            bool bad = false;
            for(int k = 0; k < d1.Length; k++) {
                if (d2[k] && !d1[k]) {
                    //bad, not same day
                    bad = true;
                }
            }
            if (!bad) {
                numberOfSameDays++;
            } else {
                g1.Violations++;
            }
        }
    }
    if(numberOfSameDays == genesArr.Length - 1) {
        return IsHardConstraint ? 0 : Pref;
    } else {
        return IsHardConstraint ? Pref : 0;
        //violated therefore penalty (unless p then this is a good thing)
    }
}

private float MEET_WITH(List<SolutionGene> cGenes) {
    float score = SAME_DAYS(new List<SolutionGene>(cGenes)) + SAME_ROOM(new List<SolutionGene>(cGenes)) + SAME_TIME(new
    ↪  List<SolutionGene>(cGenes));
    if(score == 0) {
        return 0;
    } else {
    //this constraint is always R so no need to check
        return Pref;
    }
}

private float SAME_INSTR(List<SolutionGene> cGenes) {
    List<SolutionGene> listClone = new List<SolutionGene>(cGenes);
    float score = DIFF_TIME(listClone);
    if(score != 0) {
        return Pref; //this means classes overlap, so bad
    }
    //score is guaranteed to be 0 after this point
    foreach (var g1 in cGenes) {
        foreach (var g2 in cGenes) {
            if (g1.SolutionTime.Start + g1.SolutionTime.Length == g2.SolutionTime.Start || g1.SolutionTime.Start + g1.SolutionTime.Length
            ↪  == g2.SolutionTime.Start + 1) {
                //checks if the end of g1 is equal to the start of g2 (or 1 slot later)
                int distance = g1.SolutionRoom.CalculateRoomDistance(g2.SolutionRoom);
                if(distance >0 && distance <= 50 && score < 1) {
                    score = 1; //set score to 1 unless its already higher
                } else if (distance > 50 && distance <= 100) {
                    score = 4;
                } else if(distance > 100){
                    g1.Violations++;
                    return Pref; //discance > 100 is prohibited
                }
            }
        }
    }
    return score;
}

private float NHB1_5(List<SolutionGene> cGenes) {
    int numberOfNHB = 0;
    List<SolutionGene> goodGenes = new List<SolutionGene>();
    foreach (var g1 in cGenes) {
        foreach (var g2 in cGenes) {
            if (g1.SolutionTime.Days.Equals(g2.SolutionTime.Days)) {
                if (g1.SolutionTime.Start + g1.SolutionTime.Length == g2.SolutionTime.Start+18) {
                    //checks if the end of g1 is 1 and a half h ours later than g2 start (18 units)
                    numberOfNHB++;
                    if (!goodGenes.Contains(g1)) {
                        goodGenes.Add(g1);
                    }
                }
            }
        }
    }
    foreach (var gene in cGenes) {
        if (!goodGenes.Contains(gene)) {
            gene.Violations++;
        }
    }
    if (numberOfNHB == cGenes.Count - 1) {
        return IsHardConstraint ? 0 : Pref;
        //satisfied therefore no penalty
    } else {
        return IsHardConstraint ? Pref : 0;
        //violated therefore penalty (unless p then this is a good thing)
    }
}

private float NHB_GTE1(List<SolutionGene> cGenes) {
    int numberOfNHB_GTE = 0;
    List<SolutionGene> goodGenes = new List<SolutionGene>();
    foreach (var g1 in cGenes) {
        foreach (var g2 in cGenes) {
            if (g1.SolutionTime.Days.Equals(g2.SolutionTime.Days)) {
                if (g1.SolutionTime.Start + g1.SolutionTime.Length >= g2.SolutionTime.Start + 12) {
                    //checks if the end of g1 is 1 and a half h ours later than g2 start (18 units)
                    numberOfNHB_GTE++;
                    if (!goodGenes.Contains(g1)) {
                        goodGenes.Add(g1);
                    }
                }
            }
        }
    }
    foreach (var gene in cGenes) {
        if (!goodGenes.Contains(gene)) {
            gene.Violations++;
        }
    }
    if (numberOfNHB_GTE == cGenes.Count - 1) {
        return IsHardConstraint ? 0 : Pref; //satisfied therefore no penalty
    } else {
        return IsHardConstraint ? Pref : 0; //violated therefore penalty (unless p then this is a good thing)
    }
}
private float ROOM_CONFLICTS(List<SolutionGene> cGenes) {
    int roomConflicts = 0;
    SolutionGene[] genesArr = cGenes.ToArray();
    //this is mostly for debugging to save the original array
    Dictionary<SolutionGene, List<SolutionGene>> conflictingGenes = new Dictionary<SolutionGene, List<SolutionGene>>();
    var KlasList = XMLParser.GetKlasList();

    for (int i = cGenes.Count - 1; i > 0; i--) {
        var c1 = cGenes[i];
        cGenes.RemoveAt(i);
        for (int j = 0; j < cGenes.Count; j++) {
            var c2 = cGenes[j];
            if (KlasList[c1.ID - 1].Can_Share_Room != null) {
                if (Array.Exists(KlasList[c1.ID - 1].Can_Share_Room, element => element == c2.ID)) {
                    continue;
                    //if the two classes are allowed to share a room, skip the rest of this itteration
                }
            }
            if (c1.ID != c2.ID) {
                var c1Start = c1.SolutionTime.Start;
                var c2Start = c2.SolutionTime.Start;
                var c1End = c1Start + c1.SolutionTime.Length;
                var c2End = c2Start + c2.SolutionTime.Length;
                if((c1Start<=c2Start && c1End>=c2End) || (c2Start<=c1Start && c2End >= c1End)) {
                    //The above checks if they are taught at or within the same time
                    if (c1.SolutionRoom.ID == c2.SolutionRoom.ID) {
                        //Check if they're in the same room
                        var sameDay = false;
                        for(int k = 0; k<c1.SolutionTime.Days.Count;k++) {
                            var b1 = c1.SolutionTime.Days[k];
                            var b2 = c2.SolutionTime.Days[k];
                            if (b1 && b2) {
                                //Check if they have any days in common, if yes break both loops
                                sameDay = true;
                                break;
                            }
                        }
                        if (sameDay) {
                            //if all if statements have passed true, and they share a day,
                            //increase the conflict counter and add to the conflict list for debugging
                            roomConflicts++;
                            c1.Violations++;
                            if (conflictingGenes.ContainsKey(c1)) {
                                conflictingGenes[c1].Add(c2);
                            } else {
                                conflictingGenes.Add(c1, new List<SolutionGene>() { c2 });
                            }
                        }
                    }
                }
            }
        }
    }
    float score = RoomConflictWeight*roomConflicts;
    return score;
}
```

Bibliography

Budhi, Gregorius Satia, Kartika Gunadi, and Denny Alexander Wibowo (2015). “Genetic Algorithm for Scheduling Courses”. In: *Communications in Computer and Information Science*. Springer Berlin Heidelberg, pp. 51–63. DOI: 10.1007/978-3-662-46742-8_5. URL: https://doi.org/10.1007/978-3-662-46742-8%5C_5.

Rudová, Hana, Tomáš Müller, and Keith Murray (May 2010). “Complex university course timetabling”. In: *Journal of Scheduling* 14.2, pp. 187–207. DOI: 10.1007/s10951-010-0171-3. URL: https://doi.org/10.1007/s10951-010-0171-3.

Achá, Roberto Asín and Robert Nieuwenhuis (Feb. 2012). “Curriculum-based course timetabling with SAT and MaxSAT”. In: *Annals of Operations Research* 218.1, pp. 71–91. DOI: 10.1007/s10479-012-1081-x. URL: https://doi.org/10.1007/s10479-012-1081-x.

Bellio, Ruggero, Luca Di Gaspero, and Andrea Schaerf (Feb. 2011). “Design and statistical analysis of a hybrid local search algorithm for course timetabling”. In: *Journal of Scheduling* 15.1, pp. 49–61. DOI: 10.1007/s10951-011-0224-2. URL: https://doi.org/10.1007/s10951-011-0224-2.

Müller, Tomáš (2004). “Iterative Forward Search Algorithm: Combining Local Search with Maintaining Arc Consistency and a Conflict-Based Statistics”. In: *Principles and Practice of Constraint Programming – CP 2004*. Springer Berlin Heidelberg, pp. 802–802. DOI: 10.1007/978-3-540-30201-8_81. URL: https://doi.org/10.1007/978-3-540-30201-8%5C_81.

Mitchell, Melanie (1996). *An introduction to genetic algorithms.* Cambridge, Mass: MIT Press. ISBN: 9780262631853.

Delima, Allemar Jhone, Ariel Sison, and Ruji Medina (June 2019). “A modified genetic algorithm with a new crossover mating scheme”. In: 7, pp. 165–181. DOI: 10.11591/ijeei.v7i2.1047.

Quirino, T., M. Kubat, and N. J. Bryan (2010). “Instinct-Based Mating in Genetic Algorithms Applied to the Tuning of 1-NN Classifiers”. In: *IEEE Transactions on Knowledge and Data Engineering* 22.12, pp. 1724–1737. DOI: 10.1109/TKDE.2009.211.

Raghavjee, Rushil and Nelishia Pillay (2010). “An informed genetic algorithm for the high school timetabling problem”. In: *Proceedings of the 2010 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on - SAICSIT 10.* ACM Press. DOI: 10.1145/1899503.1899555. URL: https://doi.org/10.1145/1899503.1899555.

Wang, Zan, Jin-lan Liu, and Xue Yu (2009). “Self-fertilization based genetic algorithm for university timetabling problem”. In: *Proceedings of the first ACM/SIGEVO Summit on Genetic and Evolutionary Computation - GEC 09.* ACM Press. DOI: 10.1145/1543834.1543993. URL: https://doi.org/10.1145/1543834.1543993.

Razali, N. M. and J. Geraghty (2011a). “Genetic Algorithm Performance with Different Selection Strategies in Solving TSP”. In:

— (2011b). “Genetic Algorithm Performance with Different Selection Strategies in Solving TSP”. In:

Jadaan, O. A., Lakishmi Rajamani, and C. Raghavendra Rao (2008). “IMPROVED SELECTION OPERATOR FOR GA 1”. In:

Goh, Kai Song, Andrew Lim, and Brian Rodrigues (2003). “Sexual Selection for Genetic Algorithms”. In: *Artificial Intelligence Review* 19.2, pp. 123–152. DOI: 10.1023/a:1022692631328. URL: https://doi.org/10.1023/a:1022692631328.

Darwin, Charles et al. (1859). *On the origin of species by means of natural selection, or, The preservation of favoured races in the struggle for life /.* John Murray, Albemarle Street, DOI: 10.5962/bhl.title.82303. URL: 10.5962/bhl.title.82303.

Xie, Huayang and Mengjie Zhang (2013). "Parent Selection Pressure Auto-Tuning for Tournament Selection in Genetic Programming". In: *IEEE Transactions on Evolutionary Computation* 17.1, pp. 1–19. DOI: 10.1109/TEVC.2011.2182652.

Goldberg, David E. and Kalyanmoy Deb (1991). "A Comparative Analysis of Selection Schemes Used in Genetic Algorithms". In: *Foundations of Genetic Algorithms*. Elsevier, pp. 69–93. DOI: 10.1016/b978-0-08-050684-5.50008-2. URL: https://doi.org/10.1016/b978-0-08-050684-5.50008-2.

Blickle, Tobias and Lothar Thiele (Dec. 1996). "A Comparison of Selection Schemes Used in Evolutionary Algorithms". In: *Evolutionary Computation* 4.4, pp. 361–394. DOI: 10.1162/evco.1996.4.4.361. URL: https://doi.org/10.1162/evco.1996.4.4.361.

Julstrom, B.A. and D.H. Robinson (2000). "Simulating exponential normalization with weighted k-tournaments". In: *Proceedings of the 2000 Congress on Evolutionary Computation. CEC00 (Cat. No.00TH8512)*. Vol. 1, 227–231 vol.1. DOI: 10.1109/CEC.2000.870299.

Harik, Georges R. (1995). "Finding Multimodal Solutions Using Restricted Tournament Selection". In: *Proceedings of the 6th International Conference on Genetic Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., pp. 24–31. ISBN: 1558603700.

Matsui, K. (1999). "New selection method to improve the population diversity in genetic algorithms". In: *IEEE SMC'99 Conference Proceedings. 1999 IEEE International Conference on Systems, Man, and Cybernetics (Cat. No.99CH37028)*. Vol. 1, 625–630 vol.1. DOI: 10.1109/ICSMC.1999.814164.

Shafiee, Alireza et al. (2016). "Automated process flowsheet synthesis for membrane processes using genetic algorithm: role of crossover operators". In: *26th European Symposium on Computer Aided Process Engineering*. Ed. by Zdravko Kravanja and Miloš Bogataj. Vol. 38. Computer Aided Chemical Engineering. Elsevier, pp. 1201–1206. DOI: https://doi.org/10.1016/B978-0-444-63428-3.50205-8. URL: https://www.sciencedirect.com/science/article/pii/B9780444634283502058.

Gwiazda, Tomasz (2006). *Crossover for single-objective numerical optimization problems*. omianki: TOMASZGWIAZDA E-BOOKS. ISBN: 83-923958-1-6.

Khair, Ummul et al. (2018). "Genetic Algorithm Modification Analysis Of Mutation Operators In Max One Problem". In: *2018 Third International Conference on Informatics and Computing (ICIC)*, pp. 1–6. DOI: 10.1109/IAC.2018.8780463.

Sridharan, Balaji (2010). "Modifications in Genetic Algorithm using additional parameters to make them computationally efficient". In: *2010 IEEE 2nd International Advance Computing Conference (IACC)*, pp. 55–59. DOI: 10.1109/IADCC.2010.5423037.

Alnowaini, Ghazi and Amjad Abdullah Aljomai (Mar. 2021). "Genetic Algorithm For Solving University Course Timetabling Problem Using Dynamic Chromosomes". In: *2021 International Conference of Technology, Science and Administration (ICTSA)*. IEEE. DOI: 10.1109/ictsa52017.2021.9406539. URL: https://doi.org/10.1109/ictsa52017.2021.9406539.

Khonggamnerd, Pariwat and Supachate Innet (2009). "On Improvement of Effectiveness in Automatic University Timetabling Arrangement with Applied Genetic Algorithm". In: *2009 Fourth International Conference on Computer Sciences and Convergence Information Technology*, pp. 1266–1270. DOI: 10.1109/ICCIT.2009.202.

Ghaemi, Sehraneh, Mohammad Taghi Vakili, and Ali Aghagolzadeh (2007). "Using a genetic algorithm optimizer tool to solve University timetable scheduling problem". In: *2007 9th International Symposium on Signal Processing and Its Applications*, pp. 1–4. DOI: 10.1109/ISSPA.2007.4555397.

UniTime.org (2007). *University Course Timetabling Benchmark Datasets*. https://www.unitime.org/uct_datasets.php. Online; accessed 12 April 2021.

Mühlenbein, Heinz and Dirk Schlierkamp-Voosen (Mar. 1993). "Predictive Models for the Breeder Genetic Algorithm I. Continuous Parameter Optimization". In: *Evolutionary Computation*

1.1, pp. 25–49. DOI: `10.1162/evco.1993.1.1.25`. URL: `https://doi.org/10.1162/evco.1993.1.1.25`.

Whitley, Darrell (June 2000). “The GENITOR Algorithm and Selection Pressure: Why Rank-Based Allocation of Reproductive Trials is Best”. In: 89.