B827126   Computer Science and Mathematics   COC255

TIMETABLE CREATION USING ARTIFICIAL INTELLIGENCE

by

Aiden Nico Tempest

Supervisor: Dr. S. Fatima

Department of Computer Science

Loughborough University

August 2023

Abstract

lol placeholder text for abstract lorem ipsum etc test

Acknowledgements

I would like to thank Dr. Shaheen Fatima, and Thomas Venhuizen.

Contents

Chapter 1

Introduction

Chapter 2

Literature Review

2.1 The Timetabling Problem

The timetabling problem is a constraint satisfying problem and it is known to be NP-Complete. The aim is to create a university timetable where several constraints are met. These constraints are either hard constraints or soft constraints. For a solution to be valid, all the hard constraints must be met. Not all (or in fact, any) of the soft constraints need to be met for the solution to be valid, however it is preferable for as many soft constraints to be met as possible. Examples of possible hard constraints include:

- At most only one session (i.e., a lecture or lab) is happening in a specific room in a specific period
- A student can only be attending at most one session in a specific period
- A teacher (e.g., a lecturer or lab helper) can only be attending at most one session in a specific period
- The size of a student group cannot exceed the capacity of the room
- The room must be appropriate for the type of session, e.g., a lecture must be in a lecture theatre, and a lab session must be in a computer lab
- Part time teachers can only be assigned certain time slots, e.g., they may not work on Tuesdays, so sessions they teach cannot be scheduled for time slots on Tuesday.

Possible soft constraints include:

- Students do not have more than two consecutive hours scheduled
- The capacity of a room is well suited to the size of the student group, to make an efficient use of space e.g., a group of twenty students are not going to be in a room with a capacity of two hundred
- If a student or teacher has one session immediately after another, then the respective rooms are relatively close to each other

A solution is invalid if at least of one the hard constraints are met. For example, if a student is scheduled to be in two different sessions at the same time — this is known as a clash.

2.2 Methods

2.2.1 Genetic Algorithms

Genetic algorithms (GAs) make up a group of search metaheuristics, inspired by Darwin's theory of evolution [7]. Here, the fittest members of a population survive and produce offspring, which inherit the characteristics of the parents. It is also possible for the offspring to have small mutations within their genetic code, which may or may not be beneficial towards the population's survival. This theory can be applied to search problems. In this case, the population represents the search space, which is a collection of candidate solutions to a problem, and the population of solutions evolves as the algorithm searches for a desired solution.

There does not exist a rigorous definition of GAs, but most methods use these five phases:

1. Initial population — populations of chromosomes
2. Fitness function
3. Selection — according to fitness
4. Crossover — to produce new offspring
5. Mutation — random mutation of new offspring

The initial **population** is a set of **individuals**, where each individual represents a candidate solution to the problem. These solutions will almost definitely not satisfy the problem, as they are randomly generated, but that is not an issue.

An individual (and hence that candidate solution), is defined by its **chromosome**. The chromosome is often encoded as a string of binary characters; however, any alphabet can be used. These characters are called **alleles**, and a single character or group of adjacent characters that encode a particular element of the candidate solution are known as a **gene**.

Using the example of the university timetabling problem, several alleles may be used to encode a room, but together they are one gene.

The **fitness** function measures how well a candidate solution solves the problem. For example, in the instance of the university timetabling problem, the fitness of the solution could be measured by the number of times in the solution that a hard constraint is not met. This means that a solution with a score of 0 is a valid solution, as there are instances of a hard constraint not being met.

Once the fitness function has been used to calculate the fitness of each individual, the fittest individuals are chosen for reproduction in the **selection** phase. There are several ways to select individuals to use for producing offspring. One way is to simply choose the two individuals with the best fitness. Another way is to use roulette-wheel sampling, where any individual could be chosen for producing offspring, but fitter individuals are more likely to be chosen. This second method introduces more variation into the offspring, to reduce the chance of convergence onto a local maximum.

The **crossover** phase, or reproduction phase, is where genetic material is exchanged between two parents. The crossover operator randomly chooses a locus (position) in a chromosome, in between alleles. The subsequent before and after parts of the chromosome are swapped between the two parents to produce two offspring. This is repeated multiple times to produce a population of the same size as the initial population. Whether this is by two parents reproducing multiple times using different loci or not, depends on the method used in the selection phase.

After reproduction, **mutations** can be introduced into the offspring. For example, if the chromosomes are encoded by binary strings, then some bits are randomly flipped. However, there is a very small chance of this occurring at each bit, a suggested probability is 0.1%. By introducing mutations into the offspring, the likelihood of reaching a local maximum is reduced.

Once there is a new population made up of the offspring, the process repeats until a solution is found. Each repetition is called a **generation**, and the entire set of generations is known as a **run**.

A version of a genetic algorithm was implemented by Perzina (2006) [8] and was applied to a timetabling problem with real world data. Timetables were generated for 1807 students, but for how courses are organised at this university, there are no groups of students with the same timetable, and it is unlikely that two students would even have the same timetable. Because of this, the author anticipated that it would be too difficult to generate timetables without any clashes at all, and instead aimed to minimise the number of clashes. The best solution found had 83 students with at least one clash on their timetable, or 4.59% of the students.

2.2.2 Binary Integer Programming

Binary integer programming is used to solve constraint satisfiability problems. Variables must either take a value of 1 or 0 (hence binary integer), as they are used to represent decisions, i.e., in the case of the university timetabling problem, a teaching session is happening in a specific room, at a specific time, on a specific day, with a specific teacher, with a specific teacher, with a specific student group, about a specific module, or not [2]. Then constraints are applied and used with the objective function to find what value each variable takes.

First, a mathematical model of the problem must be constructed. Abdellahi and Eledum (2006) [2] modelled the features of the university timetabling problem as a group of sets:

- $I = \{1, \ldots, n_i\}$: set of days in the week where courses are offered
- $J = \{1, \ldots, n_j\}$: set of time slots in a day
- $K = \{1, \ldots, n_k\}$: set of courses
- $L = \{1, \ldots, n_l\}$: set of student groups

- $M = \{1, \dots, n_m\}$: set of teachers
- $N = \{1, \dots, n_n\}$: set of classrooms

Next, the decision variables are defined. The basic variables $x_{i,j,k,l,m,n}$ are defined as $\forall i \in I, \forall j \in J, \forall k \in K, \forall l \in L, \forall m \in M, \forall n \in N$

$$x_{i,j,k,l,m,n} = \begin{cases} 1, & \text{if a course } k \text{ taught by teacher } m \text{ for the group of students } l \text{ is assigned to the } j^{th} \text{ time slot of day } i \text{ in classroom } m \\ 0, & \text{otherwise} \end{cases}$$

In other words, the basic variables represent the decision of whether or a not a course is being taught by a specific teacher for a specific group of students is happening in a specific time slot of a specific day in a specific classroom. Then the authors defined their auxiliary variables as

$\forall i \in I, \forall j \in J, \forall l \in L$

$$y_{i,j,k,l} = \begin{cases} 1, & \text{if a course } k+s \text{ for group of students overlap with its prerequite } k \text{ for the same group } l \text{ in the } j^{th} \text{ time slot of day } i \\ 0, & \text{otherwise} \end{cases}$$

for $s \in 1, \dots, n_k - 1, \text{where } k + s \leq n_k$

Finally, (people) defined two further sets of variables:

- z_{im}, which represents the existence of lectures for teacher m on day i
- z_{il}, which represents the existence of lectures for student l on day i

$\forall m \in M$

$$z_{im} = \begin{cases} 1, & \text{if } \sum_{j \in J} x_{i,j,k,l,m,n} \neq 0 \\ 0, & \text{otherwise} \end{cases}$$

$\forall l \in L$

$$z_{il} = \begin{cases} 1, & \text{if } \sum_{j \in J} x_{i,j,k,l,m,n} \neq 0 \\ 0, & \text{otherwise} \end{cases}$$

z_{im} and z_{il} are for use with the object function, later. The next thing to be modelled is the restraints. For the university modelled by the authors, these were:

1. There is no overlapping for courses
2. Each teacher cannot be assigned to more than one course for any given period
3. Each classroom cannot hold more than one course for any given period
4. A student has some courses amount to 18 hours per week. Each course consists of 3 hours and taught in 2 periods (each period is 90 minutes long), which is expressed as the number of slots worked per day.
5. For a student, each course occupies only one slot per day
6. The lectures of each course must be distributed in such a way that there is one day off between them
7. All lectures of a given course in a week must be held in the same classroom
8. The overlap of course with prerequisites for the same group of undergrad students l is permitted (note: the authors refer to undergrad students as pre-graduated students)

As an example, the first restraint is modelled as:

$$\sum_{k\in K}\sum_{m\in M}\sum_{n\in N} x_{i,j,k,l,m,n} \leq 1 \quad \forall i \in I, \forall j \in J, \forall l \in L$$

The objective function needs to be either minimised or maximised (dependent on how it is modelled), by changing the values assigned to the variables, whilst ensuring that the constraints are met [5]. In this instance, the total dissatisfaction of teachers and students needs to be minimised, which is equivalent to maximising the number of lectures per day, which implies decreasing the waiting time between lectures.

$$\text{(Total dissatisfaction)} = \text{Teacher}\{\text{number of lectures per day}\} + \text{Regular student}\{\text{number of lectures per day}\} + \text{Predicted graduate student}\{\text{number of lectures per day}\}$$

The model produced may not be tractable, meaning that the problem may not be able to be solved in a reasonable period. To make the problem easier to be solved, the model needs to be reduced. Abdellahi and Eledum achieved this by removing the index representing classrooms ($n \in N$) and by adding another constraint, that the number of courses cannot exceed the number of classrooms in a timeslot j in a given day i (constraint 9). To further simplify the model, the term corresponding to the overlapping of courses and their prerequisites has been removed from the objective function, with the related constraint 8.

The model is now:

$$\max\{1.5 \sum_{j\in J}\sum_{k\in K}\sum_{l\in L}\sum_{m\in M} x_{i,j,k,l,m} + \sum_{m\in M} z_{im} + \sum_{l\in L} z_{il}\}$$

(objective function)

subject to

$$\sum_{k\in K}\sum_{m\in M} x_{i,j,k,l,m} \leq 1 \quad \forall i \in I \forall j \in J, \forall l \in L$$

(Constraint 1)

$$\sum_{l\in L}\sum_{k\in K} x_{i,j,k,l,m} \leq 1 \quad \forall i \in I, \forall j \in J, \forall m \in M$$

(Constraint 2)

$$\sum_{j\in J}\sum_{k\in K}\sum_{m\in M} x_{i,j,k,l,m} \leq n_j \quad \forall i \in I, \forall l \in L$$

(Constraint 3)

$$\sum_{j\in J} x_{i,j,k,l,m} \leq 1 \quad \forall i \in I, \forall k \in K, \forall l \in L, \forall m \in M$$

(Constraint 4)

$$\sum_{j\in J}(x_{i,j,k,l,m} + x_{i+2,j,k,l,m} = 2), i + 2 \leq 5 \quad \forall i \in I, \forall k \in K, \forall l \in L, \forall m \in M$$

(Constraint 5)

$$\sum_{k\in K}\sum_{l\in L}\sum_{m\in M} x_{i,j,k,l,m} \leq n_n \quad \forall i \in I, \forall j \in J$$

(Constraint 9)

$$s \in \{1, \ldots, n_k - 1\}, k + s \leq n_k$$

$$\sum_{m \in M} z_{im} \leq 1 \quad \forall i \in I$$

$$\sum_{l \in L} z_{il} \leq n_l \quad \forall i \in I$$

$$x_{i,j,k,l,m}, y_{i,j,k,l,m}, z_{im}, z_{il} \leq 0 \quad \text{(so all variables are non-negative)}$$

The model can be reduced further into two models, one for students and one for teachers. The problem is solved with using an external penalty function method. Penalty functions convert constrained problems to those without constraints by introducing a penalty for violating the constraints ([9]). By using a penalty function, the authors found a real solution to the unrestrained problem, and then approximated it to a binary solution with an algorithm.

When demonstrating their model, the authors assigned very small numbers to their parameters: 5 workable days per week, 2 slots per day, 3 courses to be assigned, 2 student groups and 2 classrooms. A timetable was generated.

2.2.3 Tabu Search

Tabu search is a metaheuristic that guides a local search to explore the solution space outside of the local optimum, by using a **Tabu list**. A Tabu list is flexible memory structure that stores solutions that are not to be used. Tabu search is used to solve combinatorial optimisation problems, which have a finite solution set. The university timetabling problem has a finite solution set, as there are a finite number of teachers, student groups, time slots, rooms, courses, etc., so there is a finite number of different ways that the timetable can be created. Tabu search makes uses of three main strategies:

- Forbidding strategy: this controls what enters the Tabu list
- Freeing strategy: this controls what exits the Tabu list
- Short-term strategy: this is for managing interplay between the forbidding strategy and the freeing strategy to select trial solutions

Tabu search examines neighbouring solutions, which are solutions that are one "step" away from the current solution. Using the travelling salesman problem as an example, suppose the current solution is B C D A F E, where each letter represents a city. An example neighbouring solution is E C D A F B, where only one swap has occurred, between the positions of B and E. However, the solution E C D F A B is not in the neighbourhood of the current solution, as two swaps have occurred. If a solution has been used, then it is put into the Tabu list, until it meets an **aspiration criterion**. Aspiration criteria provide reasons for a solution to be freed from the Tabu table (freeing strategy). For example, if a using a Tabu move results in a solution better than any other so far, then it can come out of the Tabu table. Much like with genetic algorithms, a fitness function is used to measure how optimal a solution is [1].

A basic algorithm is:

1. Choose an initial solution i in the solution space S. Set $i' = i$ and $k = 0$, where k is the number of iterations.
2. Set $k = k + 1$ and generate a set of possible solutions $N(i, k)$, where $N(i, k)$ is the set of neighbouring solutions.
3. Choose a best $j \in N(i, k)$, where j is not Tabu (in the Tabu list). If j is Tabu, but meets an aspiration criterion, then choose j. Set $i = j$.
4. If $f(i) > f(i')$ (where f is the fitness function), set $i' = i$. Note this is for the case when a higher fitness function is better. For the case when a lower fitness function is better, set $i' = i$ when $f(i) < f(i')$.
5. Put i into the Tabu table. Update aspiration criteria.
6. If a stopping condition is met, stop. Else, go to step 2.

There are several possible stopping criteria, such as:

- There are no more feasible solutions in the neighbourhood of the current solution
- k is larger than the maximum number of allowed iterations
- The number of iterations since the last improvement of i' is greater than a specified number —- meaning that convergence has been reached, and that there are diminishing returns on finding a more optimal solution
- An optimal solution has been obtained. There would need to be a method to find what the vicinity of the optimal solution, so that upper lower bounds can be set.

Tabu search has been used to solve the university timetabling problem [3]. Awad et al. defined four different neighbourhoods:

- Nb1: Randomly choose a particular course and progress to a feasible timeslot, which can produce the smallest cost
- Nb2: Randomly select a room. Also, randomly select two courses for that room. Next, swap timeslots.
- Nb3: Randomly select two times. Next, swap timeslots.
- Nb4: Randomly select a particular time and swap it with another time, in the range between 0 and 44, which can produce the smallest penalty cost.

In order to produce an initial solution which met all the hard constraints, a least saturation degree algorithm is used, where events that are more different to schedule are scheduled first. If that did not produce a feasible solution, then Nb1 is used for a specific number of repetitions, then Nb2 is used for a specific number of repetitions if Nb1 did not reach a feasible solution. Next an improvement algorithm was used with Nb3 and Nb4, specifically adaptive Tabu search. The penalty cost is checked every 1,000 iterations and if the penalty cost has not changed, then two solutions are removed from the Tabu List.

The authors compared their algorithm to the work of others, by running 11 different datasets through them, and totalling the number of violations of their hard and soft constraints. The authors' algorithm performed best against other implementations of Tabu search, and was also compared to other algorithms that did not use Tabu search at all. The authors' implementation performed better than all the other algorithms, except variable neighbourhood search on the largest dataset, and simulated annealing which performed better across all dataset sizes.

2.2.4 Answer Set Programming

Answer set programming (ASP) reduces problems to logic programs, and then answer set solvers are used to do the search [4]. The logic programs are sets of rules that take the form:

$$a_0 \texttt{ :- } a_1, \ldots, a_m \texttt{ not } a_{m+1}, \ldots, \texttt{ not } a_n$$

where every a_i is a propositional atom and `not` is default negation. If $n = 0$ then a rule is a fact. A rule is an integrity constraint if a_0 is omitted. A logic program induces a collection of answer sets, which are recognized models of the program determined by answer set semantics.

To make ASP better for real-world use, some extensions were developed. Rules with first-order variables are viewed as shorthand for the set of their ground instances. Additionally, there are conditional literals that take the form:

$$\texttt{a:} \quad b_1, \ldots, b_m$$

where `a` and b_i are possibly default-negated variables. There are also cardinality constraints which take the form:

$$\texttt{s } \{c_1, \ldots, c_n\} \texttt{ t}$$

where each c_j is a conditional literal. `s` and `t` provide lower and upper bounds on the number of satisfied literals in the cardinality constraint. For example, `2{a(X):b(X)}4` is true when 2, 3 or 4 instances of `(X)` (subject to `b(X)`) are true. Also $N = c_1, \ldots c_n$ binds N to the number of satisfied conditional literals c_j. And objective functions that minimise the sum of weights w_j of conditional literals c_j are expressed as `#minimize{` $w_1 : c_1, \ldots, w_n : c_n$ `}.`

The logic programs can be written in a language called AnsProlog (Answer Set Programming in Logic), which can be used with answer set solvers such as smodels.

Banbara et al. (2019)[4] used Answer Set Programming to solve the timetabling problem. They split their constraints into hard and soft constraints, where the soft constraints have either constant cost or calculated cost. If a constraint has constant cost, then there is one penalty point per violation, whereas constraints that have a calculated cost attached to them have their penalty points calculated dynamically with each violation. The authors defined their constraints as:

- H_1 **Lectures**: All lectures of each course must be scheduled, and they must be assigned to distinct timeslots.
- H_2 **Conflicts**: Lectures of courses in the same curriculum or taught by the same teacher must be all scheduled in different timeslots.
- H_3 **RoomOccupancy**: Two lectures cannot take place in in the same room in the same timeslot.
- H_4 **Availability**: If the teacher of the course is unavailable to teach that course at a given timeslot, then no lecture of the course can be scheduled at that timeslot.
- S_1 **RoomCapacity**: For each lecture, the number of students that attend the course must be less than or equal the number of seats of all the room that host its lectures. The penalty points, reflecting the number of students above the capacity, are imposed on each violation.
- S_2 **MinWorkingDays**: The lectures of each course must be spread into a given minimum number of days. The penalty points, reflecting the number of days below the minimum, are imposed on each violation.
- S_3 **IsolatedLectures**: Lectures belonging to a curriculum should be adjacent to each other in consecutive timeslots. For a given curriculum we account for a violation every time there is one lecture not adjacent to any other lecture within the same day. Each isolated lecture in a curriculum counts as one violation.
- S_4 **Windows**: Lectures belonging to a curriculum should not have time windows (periods without teaching) between them. For a given curriculum we account for a violation every time there is one window between two lectures within the same day. The penalty points, reflecting the length in periods of time window, are imposed on each violation.
- S_5 **RoomStability**: All lectures of a course should be given in the same room. The penalty points, reflecting the number of distinct rooms but the first, are imposed on each violation.

- S_6 **StudentMinMaxLoad**: For each curriculum, the number of daily lectures should be within a given range. The penalty points, reflecting the number of lectures below the minimum or above the maximum, are imposed on each violation.
- S_7 **TravelDistance**: Students should have the time to move from one building to another one between two lectures. For a given curriculum we account for a violation every time there is an instantaneous move: two lectures in rooms located in different buildings in two adjacent periods within the same day. Each instantaneous move in a curriculum counts as 1 violation.
- S_8 **RoomSuitability**: Some rooms may not be suitable for a given course because of the absence of necessary equipment. Each lecture of a course in an unsuitable room counts as 1 violation.
- S_9 **DoubleLectures**: Some courses require that lectures in the same day are grouped together (double lectures). For a course that requires grouped lectures, every time there is more than one lecture in one day, a lecture non-grouped to another is not allowed. Two lectures are grouped if they are adjacent and in the same room. Each non-grouped lecture counts as 1 violation.

A solution is feasible when all the hard constraints are satisfied, but the objective is to find a solution with the minimal penalty cost. The authors formulated the problem in different ways, where a formulation is defined as specific set of soft constraints together with weights associated with each of them. Different formulations allow for timetables to be generated for different scenarios — for instance, double lectures may not be required, so the constraint S_9 would not be used. Formally, the timetabling problem is formulated as a combinatorial optimisation problem with the objective function to minimise the weighted sum of the penalty points.

Next, the authors encoded the constraints and facts into the correct format for the specific ASP system, which then returned an assignment representing a solution.

When testing, the authors used multiple sets of input data, some of which were very large, for instance, one of them consisted of 850 courses, 132 rooms, 850 curricula, 7,780 unavailability constraints, and 45,603 room constraints. The system was run on a cluster of machines with server-grade hardware, which was probably necessary given the size of the datasets. They also used different configurations for their ASP system, all of which managed to produce optimal solutions.

2.2.5 Bat Inspired Algorithm

The Bat Inspired Algorithm (BA) is a heuristic method originally proposed by Yang (2010) [10]. It is inspired by echolocation, which is a method used by bats and other animals to navigate, using sound. To simplify echolocation, Yang suggested these rules:

1. Bats use echolocation to determine distance, and they can differentiate between food and barriers.
2. Bats fly randomly with velocity v_i at position x_i having a fixed frequency $f_{\min}$, varying wavelength λ and loudness A_i to search for prey. In this rule, it is assumed that bats can adjust automatically the frequency (or wavelength) of their emitted pulses as wall the rate of pulse emission $r \in [0, 1]$. The automatic adjustment relies on the proximity of the target.
3. The loudness of the bats can vary in many ways, however, it is assumed that the loudness can vary from between positive values A_0 and $A_{\min}$.

Pseudocode for the Bat Algorithm can be seen in listing 1.

In the first step, the bat population initialised with parameters position x_i, velocity v_i and frequency f_i. With each generation, every bat moves by changing velocity v_i^t and position x_i^t at time t with the equations:

$$f_i = f_{\min} + (f_{\max} - f_{\min})\beta \tag{2.1}$$

$$v_i^t = v_i^{t-1} + (x_i^{t-1} - x_*)f_i \tag{2.2}$$

$$x_i^t = x_i^{t-1} + v_i^t \tag{2.3}$$

where β is a random number from the interval $[0, 1]$ and x_* represents the current best global solution among all the bats in the population.

In the local search part, a new solution for each bat is generated using a random walk:

$$x_{new} = x_{old} + \epsilon < A^t >$$

```
Define the objective function f(x)
Generate initial population of the bat X = {x1, ..., xn}
For each bat xi in X
    Initialise the pulse rate ri, velocity vi and loudness Ai
    Define the pulse frequency fi
While termination criterion not reached
    For each bat xi in X
        Generate new solutions using equations (1.1), (1.2) and (1.3)
        Generate a random number rand
        If rand > ri
            Select one solution among the best one
            Generate a a local solution around one of the best
        If (rand < Ai) and (f(xi) < f(x*))
            Accept the new solution
            Increase ri and reduce f(xi)
Ranks the bats and obtain the current best bat
```

Listing 1: Pseudocode for the bat algorithm [6]

where ϵ is a random number from the interval $[0, 1]$ and $< A^t >$ is the average loudness of the bats in the population at a specific time step t. Also the rate of sound pulse emission r_i and the loudness A_i of each bat needs to updated with the equation:

$$A_i^t = \alpha A_i^{t-1}$$

$$r_i^t = r_i^0[1 - e^{-\gamma t}]$$

where α and γ are constants.

The above BA was designed for continuous optimisation problems, but as the timetabling problem is a discrete combinatorial optimisation problem, Limota et al. (2021) [6] modified the algorithm. In the context of the timetabling problem, i represents a candidate solution and each position x_i is initialised by randomly choosing a lecture and then an algorithm tries to find timeslots without clashes and a large enough room. This is to start with a good initial timetable.

The parameter frequency f_i is fixed to 1 to reduce the complexity of the algorithm, meaning that velocity is now calculated as:

$$v_i^t = f(x_i) - f(x_*)$$

v_i can be interpreted as the number of operations that bat i will perform in updating the current position. The new position x_i^t is calculated with Equation 1.3, which implies that the current position is dependent on velocity and the previous position. But now, this can be interpreted to mean that the current position is obtained by performing v_i^t moves.

To update the current solution of an instance of the timetabling problem, a lecture is moved from one timeslot to another randomly selected one.

When testing their algorithm, the authors used input data consisting of 124 courses, 273 lectures, 10 rooms and 5551 students. They ran their algorithm 4 times, and each time generated timetables where all the hard constraints were satisfied.

2.3 Comparison of Methods

Table 2.1 summarises how the different methods for solving the timetabling problem performed.

Method	Dataset size (number of courses)	Solution generated?	Errors?
Genetic Algorithm[8]	340	Yes	84 students had at least one clash
Binary Integer Programming[2]	3	Yes	None
Tabu Search[3]	400*	Yes	347* hard/soft constraint violations
Answer Set Programming[4]	850	Yes	None
Bat Inspired Algorithm[6]	124	Yes	None

Table 2.1: Summative table of the results of the different methods

*largest dataset

It is also worth recalling a few other details about the different methods used:

- At the university used for the genetic algorithm method [8], courses are arranged in such a way that essentially every student has a unique timetable, therefore making the problem more complex compared to if students were put into groups.
- Multiple different configurations were used for the answer set programming system [4], some of which were able to produce a greater number of optimal solutions than others.

2.4 Chosen Method

I will not be using binary integer programming as my chosen method, as it was only tested on such a small dataset it is unknown how performance will scale for larger datasets. Likewise, I will not be using answer set programming as the dataset was so large that the algorithm had to be run on server-grade hardware, which I do not have access to, and I do not know how the performance scales with a decrease in the dataset size. I am not going to use Tabu search, as there were a large number of constraint violations in relation to the size of the dataset. I have hence decided to use a genetic algorithm as I think it would be easier to implement than a bat inspired algorithm.

Chapter 3

Requirements

3.1 Necessary Requirements

These are requirements that are specified by the project briefing, and it is ideal that they are all achieved.

1. The required information for the timetable can be entered by the user.
2. A correct timetable is generated, where:
   (a) Every module has all its hours timetabled.
   (b) A room is being used for at most one learning session in a time slot.
   (c) A student has at most one learning session in a time slot.
   (d) A teacher has at most one learning session in a time slot.
   (e) A session is in the correct type of room, for example, a lecture is in a lecture theatre, and a lab session is in a lab.
   (f) The number of students in the room must not exceed the capacity of the room.
   (g) The size of the room is optimal for the number of students, meaning that the size of the room as small as possible.
   (h) Teachers can make preferences for when they are teaching.

3. The generated timetable is returned to the user in a way that they can understand.

3.2 Aspirational Requirements

These are requirements that could be achieved, but are not essential for completion.

- Use a graphical user interface.
- Allow the user to change the parameters of the genetic algorithm.
- Return the final timetable as multiple timetables individualised to each teacher, student group, room, etc. The timetables should be in a very legible format, such as in a traditional tabular timetable format.

Chapter 4

Design

4.1 Modelling the Input Data

Before the whole algorithm can be modelled mathematically, we must first model the input data as set of variables.

- Let $T = \{t_1, \dots, t_n\}$ be the set of teachers where:
  - a teacher $t_i = \{M_{t_i}, H_{t_i}\}$ where:
    - $H_{t_i} = \{h_{t_{i_1}}, \dots, h_{t_{i_n}}\}$ is the set of the teacher's preferred timeslots, and
    - $M_{t_i} = \{m_{t_{i_1}}, \dots, m_{t_{i_n}}\}$ is the set of modules that the teacher teaches.
- Let $R = \{r_1, \dots, t_n\}$ be the set of rooms where:
  - a room $r_i = \{y_{r_i}, c_{r_i}\}$ where:
    - y_{r_i} is the type of room, which takes the value 1 if the room is a lecture theatre, or 2 if the room is a lab, and
    - c_{r_i} is the maximum capacity of the room.
- Let $S = \{s_1, \dots, s_n\}$ be the set of student groups where:
  - a student group $s_i = \{M_{s_i}, z_{s_i}\}$ where:

    * $M_{s_i} = \{m_{s_{i_1}}, \ldots m_{s_{i_n}}\}$ is the set of modules taken by the student group s_i, and
    * z_{s_i} is the number of students in s_i.

- Let $M = \{m_1, \ldots, m_n\}$ be the set of modules where:
  - a module $m_i = \{p_{m_i}, q_{m_i}\}$ where:
    * p_{m_i} is the number of lecture hours of the module m_i, and
    * q_{m_i} is the number of lab hours of the module m_i.
- Let $H = \{h_1, \ldots, h_n\}$ be the set of time slots.

Further to this, we can also describe a set of teaching sessions X where each session $x_{t_\alpha, r_\beta, s_\gamma, m_\delta, h_\epsilon}$ is a session taught by teacher t_α, in room r_β, with student group s_γ, about module m_δ, during time slot h_ϵ.

4.2 Constraints

We next need to represent the constraints. We take the requirements from §2.1, but add two more constraints (2 and 3).

1. Every module has all its hours timetabled.
2. Each session is taught by the correct teacher for the module.
3. Each session has the correct student group attending it.
4. A room is being used for at most one learning session in a time slot.
5. A student has at most one learning session in a time slot.
6. A teacher has at most one learning session in a time slot.
7. A session is in the correct type of room, for example, a lecture is in a lecture theatre, and a lab session is in a lab.
8. The number of students in the room must not exceed the capacity of the room.
9. The size of the room is optimal for the number of students, meaning that the size of the room is as small as possible.

10. Teachers can make preferences for when they are teaching.

We will now formulate these constraints below.

4.2.1 Every module has all its hours timetabled

We can say that the number of sessions scheduled is equal to the sum of the number of lecture sessions and the number of lab sessions for all the modules, or:

$$|X| = \sum_{i=1}^{|M|}(p_{m_i} + q_{m_i})$$

However, this is not sufficient to prove that the correct number of sessions per module has been scheduled, for instance one module might be missing a session and another module might have an extra session scheduled, and the above statement would still hold. To further prove this, we also have constraint 7.

4.2.2 Each session is taught by the correct teacher for the module

This can be reworded to: for every session, the module taught in that session is in the set of modules taught by the teacher, or:

$$\forall x_{t_\alpha,r_\beta,s_\gamma,m_\delta,h_\epsilon} \in X,\ m_\delta \in M_{t_\alpha}$$

4.2.3 Each session has the correct student group attending it

Very similarly to the previous constraint, this constraint can be reworded to: for every session, the module taught in that session is in the set of modules taken by the student group, or:

$$\forall x_{t_\alpha,r_\beta,s_\gamma,m_\delta,h_\epsilon} \in X,\ m_\delta \in M_{s_\gamma}$$

4.2.4 A room is being used for at most one learning session in a time slot

This can be rephrased as: for each room, there is at most one of each time slot scheduled for that room.

$$\forall r_i \in R,\ \exists X_{r_i} \subseteq X \text{ where } X_{r_i} = \{x_{t_\alpha, r_\beta, s_\gamma, m_\delta, h_\epsilon} \mid r_\beta = r_i\}$$

This creates subsets X_{r_i} which are all the learning sessions scheduled to the room r_i. Then:

$$\forall h_j \in H\ \exists X_{r_i,h_j} \subseteq X_{r_i} \text{ where } X_{r_i,h_j} = \{x_{t_\alpha, r_i, s_\gamma, m_\delta, h_\epsilon} \mid h_\epsilon = h_j\}$$

This creates subsets X_{r_i,h_j} which are all the learning session scheduled in time slot h_j for the room r_i. We then require the condition

$$|X_{r_i,h_j}| = 1 \text{ or } |X_{r_i,h_j}| = 0$$

to hold $\forall X_{r_i,h_j}$.

4.2.5 A student has at most one learning session in a time slot

This very similar to constraint 4.

$$\forall s_i \in S,\ \exists X_{s_i} \subseteq X \text{ where } X_{s_i} = \{x_{t_\alpha, r_\beta, s_\gamma, m_\delta, h_\epsilon} \mid s_\gamma = s_i\}$$

$$\forall h_j \in H\ \exists X_{s_i,h_j} \subseteq X_{s_i} \text{ where } X_{s_i,h_j} = \{x_{t_\alpha, r_\beta, s_i, m_\delta, h_\epsilon} \mid h_\epsilon = h_j\}$$

It is then required that

$$|X_{s_i,h_j}| = 1 \text{ or } |X_{s_i,h_j}| = 0,\ \forall X_{s_i,h_j}.$$

4.2.6 A teacher has at most one learning session in a time slot

This very similar to constraints 4 and 5.

$$\forall t_i \in T,\ \exists X_{t_i} \subseteq X \text{ where } X_{t_i} = \{x_{t_\alpha, r_\beta, s_\gamma, m_\delta, h_\epsilon} \mid t_\alpha = t_i\}$$

$$\forall h_j \in H\ \exists X_{t_i,h_j} \subseteq X_{t_i} \text{ where } X_{t_i,h_j} = \{x_{t_i, r_\beta, s_\gamma, m_\delta, h_\epsilon} \mid h_\epsilon = h_j\}$$

It is then required that

$$|X_{t_i,h_j}| = 1 \text{ or } |X_{t_i,h_j}| = 0,\ \forall X_{t_i,h_j}.$$

4.2.7 A session is in the correct type of room

This can be thought of as: for each module, the number of sessions in a lecture room is equal to the number of lecture sessions for that module. The same applies for lab sessions.

To begin with (recall that $y_{r_i} = 1$ if r_i is a lecture room):

$$\exists X_{lecture} \subseteq X \text{ where } X_{lecture} = \{x_{t_\alpha, r_\beta, s_\gamma, m_\delta, h_\epsilon} \mid \forall r_\beta,\ y_{r_\beta} = 1\}$$

This gives us $X_{lecture}$, a set that contains all the sessions in a lecture room. Then:

$$\forall m_i \in M,\ \exists X_{lecture,m_i} \subseteq X_{lecture} \text{ where } X_{lecture,m_i} = \{t_\alpha, r_\beta, s_\gamma, m_\delta, h_\epsilon \mid m_\delta = m_i\}$$

Finally, it is required that

$$\forall X_{lecture,m_i},\ |X_{lecture,m_i}| = p_{m_i}.$$

Similarly, for labs:

$$\exists X_{lab} \subseteq X \text{ where } X_{lab} = \{x_{t_\alpha, r_\beta, s_\gamma, m_\delta, h_\epsilon} \mid \forall r_\beta,\ y_{r_\beta} = 2\}$$

$$\forall m_i \in M,\ \exists X_{lab,m_i} \subseteq X_{lab} \text{ where } X_{lab,m_i} = \{t_\alpha, r_\beta, s_\gamma, m_\delta, h_\epsilon \mid m_\delta = m_i\}$$

and it is required that

$$\forall X_{lab,m_i} |X_{lab,m_i}| = p_{m_i}.$$

As teaching sessions are either labs or lectures, it can be implied that $|X_{lecture}| + |X_{lab}| = |X|$.

Because of the above, there are the correct number of lectures and labs for each module, and the conditions are now sufficient for constraint 1 to hold.

4.2.8 The number of students in the room must not exceed the capacity of the room

This can be reworded to: for every session, the size of the student group must be less than or equal to the capacity of the room, or:

$$\forall x_{t_\alpha, r_\beta, s_\gamma, m_\delta, h_\epsilon} \in X,\ z_{s_\gamma} \leq c_{r_\beta}$$

4.2.9 The size of the room is optimal for the number of students

This means, for every session $x_i \in X$ in room r_i, if there exists a room $r_j \in R$ where $c_{r_j} < c_{r_i}$, change r_i to r_j, only if no other constraints are violated. Or, $r_i = \min\{r_j \,|\, c_{r_j} \geq c_{r_i}\} \in R$.

4.2.10 Teachers can make preferences for when they are teaching

We first define a set of sessions for each teacher:

$$\forall t_i \in T,\ \exists X_{t_i} \subseteq X \text{ where } X_{t_i} = \{x_{t_\alpha, r_\beta, s_\gamma, m_\delta, h_\epsilon} \mid t_\alpha = t_i\}$$

We then define a set containing just the time slots for each X_{t_i}:

$$\forall X_{t_i},\ \exists H_{xt_i} \text{ where } H_{xt_i} = \{h_j \mid h_j = h_\epsilon \forall x_{t_i, r_\beta, s_\gamma, m_\delta, h_\epsilon} \in X_{t_i}\}$$

Then, for the constraint to be satisfied, $H_{xt_i} \subseteq H_{t_i}$ must hold $\forall t_i \in T$, or alternatively, the constraint is violated if $\exists H_{xt_i} \ni h_j \notin H_{t_j}$.

4.3 Timetable Encoding

A solution will be encoded as a set of 5-tuples, where each tuple is {time slot, room, student group, module, teacher}, where a solution represents an entire timetable.

For the genetic algorithm, the population is a set of solutions.

4.4 Genetic Algorithm Pseudocode

```
function GeneticAlgorithm

GetSettingsData (return: populationSize, mutationChance)
GetConfigData (return: sessions, rooms, timeSlots, teacherTimes)
CreatePopulation (return: population)
CheckFitness of population (return: populationFitness,
                            validSolution, validSolutionBool)
while not validSolutionBool:
    SelectParents (return: parentA, parentB)
    CreateOffspring (return: offspring)
    MutateOffspring (return: mutatedOffspring)
    append parents to mutatedOffspring
    CheckFitness of mutatedOffspring (return: offspringFitness,
                                      validSolution, validSolutionBool)
    if not validSolutionBool:
        population := offspring
        repeat(2):
            remove individual from population with worst fitness
output timetable from validSolution
```

Listing 2: Pseudocode for the genetic algorithm

The settings data contains `populationSize` and `mutationChance`, which are parameters for the genetic algorithm the user may wish to change. `populationSize` is quite self-explanatory, simply the number of individuals in the population. `mutationChance` is the reciprocal of the actual chance of a mutation occurring, for example, if the chance of a mutation occurring is $\frac{1}{1000}$, then `mutationChance` needs to be 1000.

Note that the `CheckFitness` function is called both inside the while loop and before it. This is so that the initial population generated can be checked for a valid solution, removing the need to proceed with the rest of the algorithm. Hence, the population's fitness needs to be calculated again inside the while loop, after the mutation phase.

4.4.1 Initial Population

```
function CreatePopulation
input: sessions, rooms, timeSlots, populationSize

population := empty list
repeat(populationSize):
    solution := empty list
    for session in sessions:
        timeSlot := random(timeSlots)
        room := random(rooms)
        studentGroup := session[1]
        module := session[0]
        teacher := session[2]
        newSession := (timeSlot, room, studentGroup, module, teacher)
        append newSession to solution
    append solution to population
return population
```

Listing 3: Pseudocode for generating the initial population

The initial population is generated within two for loops, as the population consists of many solutions, each of which consist of many sessions. For each session, the time slot and room are chosen randomly, but a session's module, student group and teacher is predetermined. This is because when the configuration data is imported, a module is imported with its student group and teacher, as a triple. Then each triple is stored in the list `sessions`. This means that a session automatically has the correct student group and teacher for the module, therefore reducing the difficulty of the problem. The session is stored n times in `sessions`, where n is the number of lab and lecture sessions of that module, so that the correct number of teaching sessions are created.

4.4.2 Fitness Function

```
function CheckFitness
input: population, teacherTimes

populationFitness := empty list
validSolutionBool := False
validSolution := None
for each solution in population:
    sort solution by timeSlot
    violations := 0
    for i in range(size(solution) - 1):
        if solution[i][0] == solution[i + 1][0]:
            if solution[i][1] == solution[i + 1][1]:
                violations := violations + 1
            if solution[i][2] == solution[i + 1][2]:
                violations := violations + 1
            if solution[i][4] == solution[i + 1][4]:
                violations := violations + 1
    for in range(size(solution)):
        timeSlot = solution[i][0]
        teacherNumber = solution[i][4]
        teacherPrefs = teacherTimes.get(teacherNumber)
        if timeSlot not in teacherPrefs:
            violations := violations + 1
    if violations == 0:
        solutionFitness := 0
    else:
                solutionFitness := 1 / violations
    append solutionFitness to populationFitness
    if solutionFitness == 0:
        valid solution bool := True
        valid solution := solution
        break for loop
return populationFitness, validSolution, validSolutionBool
```

Listing 4: Pseudocode for the fitness function

The fitness function is calculated by totalling the number of constraint violations, then raising it to the power of -1, so a higher fitness function is better. However, solutions with no constraint violations have a fitness function of 0.

Violations are worked out in different ways depending on which constraint is violated. To check for clashes for student groups, rooms, and teachers, the sessions in a solution are sorted by time slot, so it can be checked if two consecutive sessions in the solution list share the same time slot. If the two sessions do share the same time slot, then it is checked if the rooms are the same, and then if they are, constraint 4 has been violated, and the count of violations is increased — the same applies to student groups and teachers.

To check a session against the teacher's time preferences, the session's time slot is checked against a list of that teacher's preferred time slots. If the session's time slot does not appear on the list, then constraint 10 has been violated and the number of and the count of violation is increased.

4.4.3 Selection

```
function SelectParents
input: populationFitness

NormaliseValues(return: normalisedLimits)
ChooseParent(return: parentA)
remove the fitness of parentA from populationFitness
NormaliseValues(return: normalisedLimits)
ChooseParent(return: parentB)
return parentA, parentB
```

Listing 5: Pseudocode for the selection phase

```
function ChooseParent
input: normalisedLimits

choice = random[0, 1]
lowerValue = 0
upperValue = size(normalisedLimits) - 1
foundParent = False
parent = None
while not found_parent:
    perform binary search to find interval in which the parent lies
return parent
```

Listing 6: Pseudocode for a subroutine of the selection phase

Firstly, the fitness values of the population are normalised so that they add up to one. The selection of a parent can be modelled like a spinner wheel, where the fitter the solution, the larger its ‘slice’ on the spinner wheel, so it is more likely to be chosen. The function `NormaliseValues` returns a list of limits, which are the intermediate points of the cumulative sum of the normalised fitness values. As an example, if the population has normalised fitness values of $0.25, 0.5, 0.25$, then the returned `normalisedLimits` will be $[0, 0.25, 0.75, 1]$. Referring back to the spinner wheel analogy, these refer to the ‘edges’ where the ‘slices’ meet.

To select a parent, a random float between 0 and 1 is selected, which represents where the spinner lands. To continue the previous example, if the random float `choice` is 0.52, then the second solution will be chosen, as $0.25 < 0.52 < 0.75$. Once this is selected, a binary search is performed to find the index of the chosen solution (`parent`). After the first parent has been selected, its fitness value is removed from the list and the `normalisedLimits` are recalculated and `ChooseParent` is run again, to determine the second parent.

4.4.4 Crossover

```
function CreateOffspring
input: populationSize, numOfSessions, parentA, parentB

offspring := empty list
halfPop := ceiling(populationSize / 2)
for i in range(halfPop):
    outerLocus := RandomInteger(0, numOfSessions - 1)
    if outerLocus == 0:
        innerLocus := RandomInteger(1, 2)
    else:
        innerLocus := RandomInteger(0, 2)
    end if
    leftA := parentA[:outerLocus]
    centreA = parentA[outerLocus][:innerLocus] + parentB[outerLocus][innerLocus:]
    rightA := parentB[outerLocus + 1:]
    append centreA to leftA
    childA := leftA + rightA
    append childA to offspring
    leftB := parentB[:outerLocus]
    centreB := parentB[outerLocus][innerLocus] + parentA[outerLocus][innerLocus:]
    rightB := parentA[outerLocus + 1:]
    append centreB to leftB
    childB := leftB + rightB
end for
if halfPop =/= floor(populationSize / 2):
    delete the last element in offspring
end if
return offspring
```

Listing 7: Pseudocode for the crossover phase

Recall that to generate offspring, a locus in the chromosome is randomly chosen, and the before and after parts of the chromosome are swapped between the two parents to produce two offspring. So the first task of the crossover phase is determining where the locus will be in the chromosome. As the chromosome in this instance is a solution, which is modelled as set of sets, we first need to determine which session contains the split — which is indexed with `outerLocus`. The `innerLocus` determines whether the locus is before the time slot (`innerLocus` = 0), before the room (`innerLocus` = 1), or before the student group (`innerLocus` = 2). There is no split between the student group and the module, or the module and the teacher, because that part of the solution is already known to be correct, so there is no point in changing it.

The way that the crossover phase typically works is that the locus is in between alleles, i.e. between characters in a string. However, due to how the data has been encoded, splitting between characters would create e.g. rooms that do not exist, which is counterproductive.

4.4.5 Mutation

```
function MutateOffspring
input: offspring, mutationChance, timeSlots, rooms, sessions

mutatedOffspring = offspring
for solution in mutatedOffspring:
    for session in solution:
        for i in range(3):
            mutate = RandomInteger(1, mutationChance)
            if mutate == 0:
                match i:
                    case 0:
                        newTimeSlot = random(timeSlots)
                        offspring[solution][session][0] := newTimeSlot
                    case 1:
                        newRoom = random(rooms)
                        offspring[solution][session][1] := newRoom
                    case _:
                        newSession = random.choice(sessions)
                        offspring[solution][session][2] := newSession[0]
                        offspring[solution][session][3] := newSession[1]
                        offspring[solution][session][4] := newSession[2]
return mutatedOffspring
```

Listing 8: Pseudocode for the mutation phase

There is a chance of mutation for individual genes, rather than for the individual as a whole, meaning an individual could have multiple mutations. Like with the crossover phase, the student group, module and teacher are not mutated separately, for the same reason.

Chapter 5

Implementation

The program is written in Python, using a procedural programming approach. The program is split across multiple files, for ease of reading.

5.1 Data Storage

The settings data and the config data are stored in two json files, `settings.json` and `data.json` respectively. Listing 9 and listing 10 are examples of these files, each containing a small amount of data.

```json
{
    "population-size": "10",
    "mutation-chance": "1000"
}
```

Listing 9: An example of settings.json

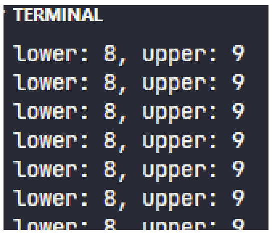
In `data.json` object names cannot be changed, but the values can take any format, e.g. a `teacher`'s `id` could be a string instead. The same does not apply to `settings.json`, the format of the values must stay the same.

```json
{
  "teachers": [ {
      "id": "123",
      "name": "A Name",
      "available_times": [ {
          "day": "Tuesday",
          "start_times": ["09", "10", "11", "12", "13", "14", "15", "16", "17"] }, {
          "day": "Wednesday",
          "start_times": ["09", "10", "11", "12"] }, {
          "day": "Thursday",
          "start_times": ["09", "10", "11", "12", "13", "14", "15", "16", "17"] } ] } ],
  "rooms": [ {
      "id": "B101", "building": "Building Name", "type": "Lecture", "capacity": "200"} ],
  "student_groups": [ {
      "id": "4567", "modules": ["A123"], "num_of_students": "200"} ],
  "modules": [ {
      "id": "A123", "title": "Module Title", "teachers": "123",
      "lecture_hours": "1", "lab_hours": "2", "student_group": "4567"} ],
  "time_slots": [ {
      "day": "Monday",
      "start_times": ["09", "10", "11", "12", "13", "14", "15", "16", "17"] }, {
      "day": "Tuesday",
      "start_times": ["09", "10", "11", "12", "13", "14", "15", "16", "17"] }, {
      "day": "Wednesday",
      "start_times": ["09", "10", "11", "12", "13", "14", "15", "16", "17"] }, {
      "day": "Thursday",
      "start_times": ["09", "10", "11", "12", "13", "14", "15", "16", "17"] }, {
      "day": "Friday",
      "start_times": ["09", "10", "11", "12", "13", "14", "15", "16", "17"] } ]
}
```

Listing 10: An example of data.json

5.2 Debugging

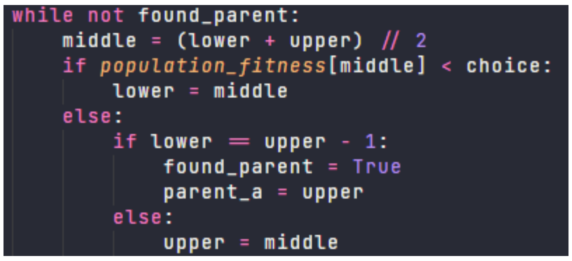
One bug that I encountered was that the lower and upper bounds of the binary search in the `choose_parent` function in `selection.py` were not updating, as evidenced in figure 5.1.



```
TERMINAL
lower: 8, upper: 9
lower: 8, upper: 9
lower: 8, upper: 9
lower: 8, upper: 9
lower: 8, upper: 9
lower: 8, upper: 9
lower: 8, upper: 9
```

Figure 5.1: Terminal output showing repeating values of “lower” and “upper”

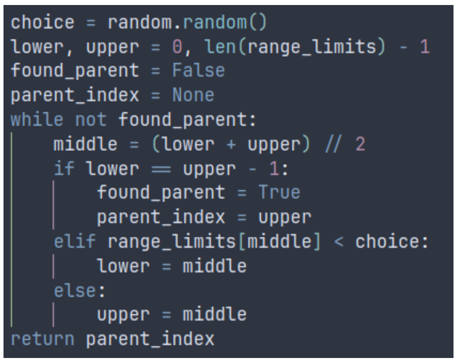
The relevant code is below in figure 5.2



```python
while not found_parent:
    middle = (lower + upper) // 2
    if population_fitness[middle] < choice:
        lower = middle
    else:
        if lower == upper - 1:
            found_parent = True
            parent_a = upper
        else:
            upper = middle
```

Figure 5.2: The problem code

Changing the order of the `if else` statements solved the problem.



```python
choice = random.random()
lower, upper = 0, len(range_limits) - 1
found_parent = False
parent_index = None
while not found_parent:
    middle = (lower + upper) // 2
    if lower == upper - 1:
        found_parent = True
        parent_index = upper
    elif range_limits[middle] < choice:
        lower = middle
    else:
        upper = middle
return parent_index
```

Figure 5.3: The fixed code

Another issue I encountered was that after thousands of generations, no valid solution was being found, and after further investigation, I discovered that the population was converging to the same solution.

The first thing I tried doing to fix this was to the increase the chance of a mutation occurring, as this does help to reduce convergence. This was simply done by decreasing the value of `mutation-chance` in `settings.json`. This was not sufficient, and I also found that in several cases, improvements to the fitness of the overall population (by means of fitter offspring) were diminished when they were mutated, as a mutation would often worsen the fitness of an individual. I fixed this by calculating the fitness of the offspring immediately after the crossover phase. If a member of the offspring has fitness that is the same as or worse than the fitness of the worse parent, then it is mutated, otherwise it is not. Some pseudocode for this is shown in listing 11.

```
CheckFitness of offspring(return: offspringFitness)
mutateList := empty list
notMutateList := empty list
for solution in offspringFitness
    indexOffspring := index(offspringFitness(solution))
    if solution <= worstParentFitness:
        append offspring[indexOffspring] to mutateList
    else:
        append offspring[indexOffspring] to notMutateList
```

Listing 11: Only mutating offspring if it does not have improved fitness

For this, I also created another function that only returns the fitness values, and not a boolean for if there exists a valid solution, and that solution if it does exist, because they are not needed. The offspring that are not mutated are added to their own list so that they can be recombined with the mutated offspring later. Also note that `SelectParents` also now returns `worstParentFitness` so it can be used.

However, this did still not fix the problem of convergence. I then decided to change the number of parents involved in the crossover phase, as it would help to diversify the “gene pool”. Additionally, only using two parents to generate very large populations (over 100 individuals) would counteractive the benefit of the greater population size. Therefore, it makes more sense for the number of parents to scale with the size of the population.

So instead of selecting two parents using the previous “spinner wheel” method, the half of the population with the better fitness will be chosen, and four offspring will be produced from a pair of parents.

Firstly, I decided to change how the fitness is calculated. Because I do not need better fitness to be a larger number (as probability is no longer being used choose the parents), the number of constraint violations is not inverted, and instead that number is used to denote the fitness, meaning that now, lower fitness is better, instead of before where higher fitness was better. The pseudocode for this is now shown in listing 12.

The selection process for the parents is now greatly simplified, as they just need to be sorted by their fitness, and then the best ones are selected. However, it is not as simple as selecting half of population, if the size of the population is an odd number, or if half of the population is an odd number (as there be one parent without a partner). Therefore the size of the population needs to be a multiple of 4. The pseudocode is now:

Next, the crossover phase had to be changed so that two parents are chosen randomly, and they crossover twice, with a random locus each time, to create a total of four offspring between the pair of them.

But, I found that if there were multiple parents with the same fitness value, only one of their chromosomes was being used. This was due to how the fitness values were stored. The fitness values were being stored as a list of integers, and the only thing that was linking them to which solution they belonged to was the order, as they were being stored in the same order as the solutions were within the population list. This meant that if the index of a fitness value was used to refer to a solution, the same solution would be referred to multiple times if the same value was in the fitness value list multiple times. (Note: this is not illustrated in previous pseudocode.)

To solve this, I tried to store the fitness values as a dictionary, which stores data in a list of key:value pairs. I was going to store each solution as the key, with the corresponding fitness value as the value. However, this would not work as the key has to be a string, not a list, which is the data type each solution is stored as.

Instead, I stored them as a list of pairs, in the format

```
[
function CheckFitness
input: population, teacherTimes

populationFitness := empty list
validSolutionBool := False
validSolution := None
for each solution in population:
        sort solution by timeSlot
        violations := 0
        for i in range(size(solution) - 1):
                if solution[i][0] == solution[i + 1][0]:
                        if solution[i][1] == solution[i + 1][1]:
                                violations := violations + 1
                        if solution[i][2] == solution[i + 1][2]:
                                violations := violations + 1
                        if solution[i][4] == solution[i + 1][4]:
                                violations := violations + 1
        for in range(size(solution)):
        timeSlot = solution[i][0]
        teacherNumber = solution[i][4]
        teacherPrefs = teacherTimes.get(teacherNumber)
        if timeSlot not in teacherPrefs:
            violations := violations + 1
        append violations to populationFitness
        if violations == 0:
                valid solution bool := True
                valid solution := solution
                break for loop
return populationFitness, validSolution, validSolutionBool
```

Listing 12: Pseudocode for modified fitness function

```
    [solution1, fitnessValue1],
    [solution2, fitnessValue2],
    ...,
    [solutionN, fitnessValueN]
]
```

so referencing a fitness value by its index would always return the right solution, if needed.

```
function SelectParents
input: populationFitness, populationSize

parents := empty list
sortedFitness := populationFitness
sort sortedFitness  // sorts from low to high
for i in range(2 * ceiling(population_size / 4)):
    parent = sorted_fitness[0]
    append parent to parents
    if i == (2 * ceiling(population_size/4)) - 1:
        worst_parent_fitness = sorted_fitness[0]
    delete sorted_fitness[0]
return parents, worst_parent_fitness
```

Listing 13: Pseudocode for modified selection phase

Finally, I noticed that the chromosomes of the chosen parents were changing at some point between after the selection phase and after the mutation phase. Through trial and error, I worked out that they were changing when a mutation occurred. This was happening because variables in Python work by pointing at values, rather than storing their own copy of a value. Because of this, when mutating a child's chromosome (which points to the parent), the parent's chromosome also mutates. I fixed this by creating a deep copy (from the `copy` module) of the parents, which creates a completely separate variable with an ID which is not referenced by its offspring, and hence does not change when the offspring does.

As seen in lines 17 and 18 of listing 2, only the two members of the population with the worst fitness are removed. However, now the difference between the parameter population size and the current number of individuals varies depending on the population size. So instead, the individual with the worst fitness is removed from the population until the size of the population is equal to the parameter population size.

Now, my program works as expected.

Chapter 6

Testing

All the testing was run on computer using Arch Linux x86_64, with a dual core processor clocked @ 2.7 GHz, and 8 GB of RAM. I used 3 different sizes of data sets, and used different sizes of population, but kept the chance of mutation constant at $\frac{1}{1000}$. I used the number of generations to generate a correct solution to measure the speed of the program with the different inputs. Table 6.1 shows the mean number of generations for each number of modules, for each population size. The full results are in table 6.2.

	Number of Modules		
Population Size	10	50	100
10	155	605	1205
50	17	111	216
100	11	73	162
200	9	57	110

Table 6.1: Mean average of number of generations for each number of modules and population size

As would be expected, increasing the size of the population reduces the number of generations needed to produce a valid timetable. However, there are diminishing returns with increasing the population size, as illustrated in the graph 6.1.

Modules		10				50				100			
Population		10	50	100	200	10	50	100	200	10	50	100	200
Result	1	68	14	9	7	391	103	80	55	584	239	161	110
	2	86	17	9	9	445	85	67	60	777	204	145	115
	3	91	7	12	8	535	92	71	61	671	231	139	104
	4	117	10	11	8	1082	112	75	49	682	226	211	101
	5	265	9	10	10	421	122	79	62	4572	209	154	109
	6	157	24	13	12	552	108	68	49	864	200	148	107
	7	172	32	17	10	580	126	66	55	1258	228	153	96
	8	186	14	8	7	802	120	71	54	895	202	167	129
	9	188	177	10	9	498	141	76	62	928	191	177	101
	10	224	23	11	8	746	100	73	59	819	232	169	125
Mean		155	17	11	9	605	111	73	57	1205	216	162	110
Std Dev		64.2	7.7	2.6	1.6	214	16.9	4.9	5.0	1197	16.8	20.7	10.6

Table 6.2: Full testing results

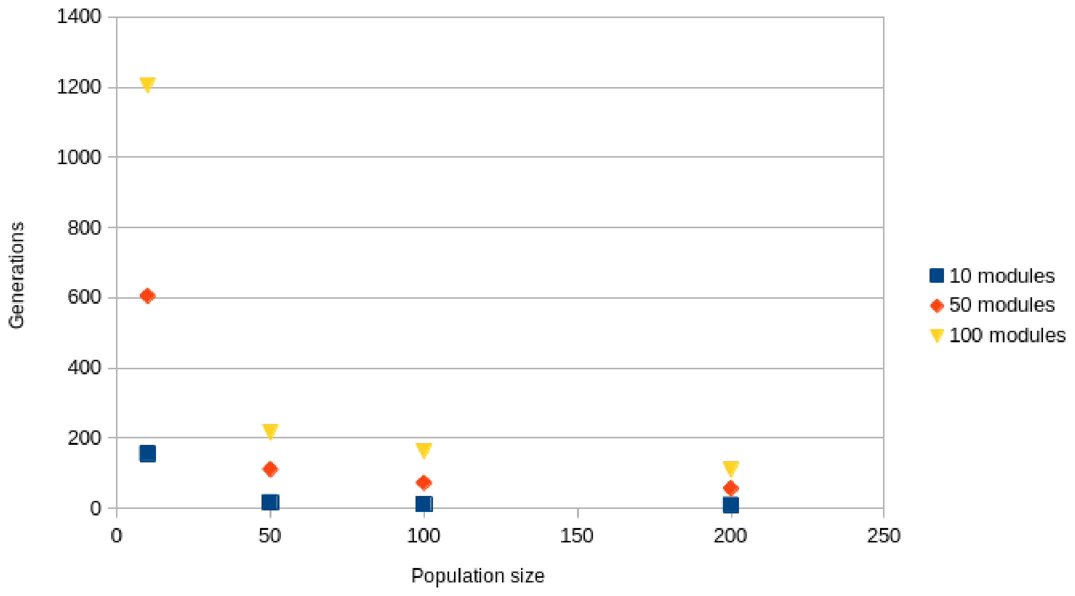


Figure 6.1: Graph showing generations against population size for different numbers of modules

Chapter 7

Evaluation

Chapter 8

Conclusion

References

[1] IIT Kharagpur July 2018. Lecture 37: Tabu search. `https://www.youtube.com/watch?v=A7cTp1Fhg9o`, Sept. 2018.

[2] Moustapha Abdellahi and Hussein Eledum. The university timetabling problem: modeling and solution using binary integer programming with penalty functions. *International Journal of Applied Mathematics and Statistics*, 56(6):164–178, 2017.

[3] Fouad H Awad, Ali Al-Kubaisi, and Maha Mahmood. Large-scale timetabling problems with adaptive tabu search. *Journal of Intelligent Systems*, 31(1):168–176, 2022.

[4] Mutsunori Banbara, Katsumi Inoue, Benjamin Kaufmann, Tenda Okimoto, Torsten Schaub, Takehide Soh, Naoyuki Tamura, and Philipp Wanko. teaspoon: solving the curriculum-based course timetabling problems with answer set programming. *Annals of Operations Research*, 275:3–37, 2019.

[5] Cuemath. Objective function. `https://www.cuemath.com/algebra/objective-function/`. Accessed: 2022-12-22.

[6] Ushindi Limota, Egbert Mujuni, and Allen Mushi. Solving the university course timetabling problem using bat inspired algorithm. *Tanzania Journal of Science*, 47(2):674–685, 2021.

[7] Melanie Mitchell. *An introduction to genetic algorithms.* MIT press, 1998.

[8] Radomír Perzina. Solving the university timetabling problem with optimized enrollment of students by a self-adaptive genetic algorithm. In *International Conference on the Practice and Theory of Automated Timetabling*, pages 248–263. Springer, 2006.

[9] Stanford University. Penalty functions. `https://web.stanford.edu/group/sisl/k12/optimization/MO-unit5-pdfs/5.6penaltyfunctions.pdf`. Accessed: 2022-12-22.

[10] Xin-She Yang. A new metaheuristic bat-inspired algorithm. In *Nature inspired cooperative strategies for optimization (NICSO 2010)*, pages 65–74. Springer, 2010.

List of Figures

List of Tables

List of Listings

Appendix A

Testing Output Timetables

Appendix B

Source Code

B.1 main.py

```python
"""Main menu for the timetable maker.

Functions:
    menu()
    clear()
    get_user_input()
"""
from os import system, name
import modules.ga as ga


def menu():
    """Print a welcome message to the user, with instructions."""
    clear()
    print("Timetable Maker\nPlease ensure that settings.json and data.json \
          are filled in correctly. \nType start to create a timetable with \
          output to a text file, or type help for more commands")
    get_user_input()


def clear():
    """Clear the terminal display."""
    if name == "nt":  # Windows
        _ = system("cls")
    else:  # *nix based systems (Linux, MacOS, etc)
        _ = system("clear")


def get_user_input():
    """Get text input from the user."""
    user_input = input()
    match user_input:
        case "start":
            ga.generate_timetable()
        case "help":
            print("Please read the documentation.")
            get_user_input()
        case _:
            print("'" + user_input + "' is not a recognised command. Type \
                help for the help guide")
            get_user_input()


if __name__ == "__main__":
    menu()
```

B.2 ga.py

```python
"""Genetic algorithm base.

Functions:
    generate_timetable()
    get_settings_data() -> [int, int]
    generate_output_text(solution: list, person_type: str)
"""
import modules.initial_population as p1
import modules.fitness_function as p2
import modules.selection as p3
import modules.crossover as p4
import modules.mutation as p5
import json
import os


def generate_timetable():
    """Generate the timetable.

    Basis of the timetable generation, goes through the whole of the
    genetic algorithm.
    """
    population_size, mutation_chance = get_settings_data()
    sessions, rooms, time_slots, teacher_times = p1.get_config_data()
    print("Generation: 1")
    generation_count = 1
    population = p1.generate_initial_population(sessions, rooms, time_slots,
                                                population_size)
    population_fitness, valid_solution_bool, valid_solution = \
        p2.check_population_fitness(population, teacher_times)

    while not valid_solution_bool:

        parents, worst_parent_fitness = p3.select_parents(population_fitness,
                                                          population_size)

        offspring, parents_copy = p4.crossover(parents)

        offspring_fitness = p2.check_fitness_only(offspring, teacher_times)
        mutate_list = []
        not_mutate_list = []
        for solution in offspring_fitness:
            if solution[1] >= worst_parent_fitness:
                mutate_list.append(solution[0])
            else:
                not_mutate_list.append(solution[0])

        mutated = p5.mutate(mutate_list, time_slots, rooms, sessions,
                            mutation_chance)

        new_pop = []
        for solution in mutated:
            new_pop.append(solution)
        for solution in parents_copy:
            new_pop.append(solution)
        for solution in not_mutate_list:
            new_pop.append(solution)

        # check if any of the offspring is a valid solution
        new_pop_fitness, valid_solution_bool, valid_solution \
            = p2.check_population_fitness(new_pop, teacher_times)

        if not valid_solution_bool:
            new_pop_fitness.sort(key=lambda x: x[1])
            while len(new_pop) > population_size:
                worst_sol = new_pop_fitness[-1][0]
                new_pop.remove(worst_sol)
                del new_pop_fitness[-1]
            generation_count += 1
            print("Generation: " + str(generation_count))
            population = new_pop
            population_fitness = new_pop_fitness

    print("Timetable solution found. Writing output to text files...")
    print("Creating directory...")
    try:
        os.mkdir("./timetables")
        print("Directory created.")
    except FileExistsError:
        print("Directory already exists.")
    file = open("./timetables/solution.txt", "w", encoding="utf-8")
    for session in valid_solution:
        line = str(session) + "\n"
        file.write(line)
    file.close()
    print("Output written to file in ./timetables")


def get_settings_data() -> [int, int]:
    """Get settings for timetable generation.

    Returns:
        int: Population size set by the user.
        int: Chance of mutation set by the user.
    """
    file = open("settings.json", "r", encoding="utf-8")
    settings = json.load(file)
    file.close()
    population_size = int(settings["population-size"])
    mutation_chance = int(settings["mutation-chance"])
    return population_size, mutation_chance
```

B.3 initial_population.py

```python
"""Phase 1 (initial population) of genetic algorithm.

Functions:
    get_config_data() -> [list, list, list, list]
    generate_initial_population(sessions: list, rooms: list, time_slots: list,
        population_size: int) -> list
    create_session_solution(session: list, rooms: list, time_slots: list)
        -> list
    create_complete_solution(sessions: list, rooms: list, time_slots: list)
        -> list
"""
import json
import random


def get_config_data() -> [list, list, list, list]:
    """Get the timetable data from data.json.

    Returns:
        list: Session consisting of module ID, student group and teacher.
        list: List of rooms.
        list: List of time slots.
        list: List of teachers' preferred time slots.
    """
    file = open("data.json", "r", encoding="utf-8")
    data = json.load(file)
    file.close()

    sessions, rooms_id, time_slots_id = [], [], []
    for module in data["modules"]:
        session = [module["id"], module["student_group"], module["teachers"]]
        for i in range(int(module["lecture_hours"])):
            sessions.append(session)
        for i in range(int(module["lab_hours"])):
            sessions.append(session)

    for room in data["rooms"]:
            rooms_id.append(room["id"])

    day_number = 1
    for day in data["time_slots"]:
        for start_time in range(len(day["start_times"])):
            time_number = day["start_times"][start_time]
            time_slots_id.append(str(day_number) + str(time_number))
        day_number += 1

    teacher_times = {}
    for teacher in data["teachers"]:
        teacher_time = []
        teacher_id = (teacher["id"])
        for day in teacher["available_times"]:
            day_name = day["day"]
            match day_name:
                case "Monday":
                    day_number = 1
                case "Tuesday":
                    day_number = 2
                case "Wednesday":
                    day_number = 3
                case "Thursday":
                    day_number = 4
                case "Friday":
                    day_number = 5
                case "Saturday":
                    day_number = 6
                case "Sunday":
                    day_number = 7
                case _:
                    day_number = 0
            for i in range(len(day["start_times"])):
                time_number = day["start_times"][i]
                teacher_time.append(str(day_number) + str(time_number))
        teacher_times.update({teacher_id: teacher_time})

    return sessions, rooms_id, time_slots_id, teacher_times

```

```python

def generate_initial_population(sessions: list, rooms: list,
                                time_slots: list, population_size: int) \
                                -> list:
    """Generate the first population randomly from the imported data.

    Args:
        sessions (list): List of sessions consisting of module ID, student
            group and teacher.
        rooms (list): List of rooms.
        time_slots (list): List of time slots.
        population_size (int): Size of the population.

    Returns:
        list: The generated population.
    """
    population = []
    for i in range(population_size):
        solution = create_complete_solution(sessions, rooms, time_slots)
        population.append(solution)
    return population


def create_session_solution(session: list, rooms: list, time_slots: list) \
     -> list:
    """Create a session, to become part of a solution.

    Args:
        session (list): A single session consisting of module ID, student
            group and teacher.
        rooms (list): List of rooms.
        time_slots (list): List of time slots.

    Returns:
        list: The complete session with time slot and room.
    """
    time_slot = random.choice(time_slots)
    room = random.choice(rooms)
    student_group = session[1]
    module = session[0]
    teacher = session[2]
    solution = [time_slot, room, student_group, module, teacher]
    return solution


def create_complete_solution(sessions: list, rooms: list, time_slots: list) \
         -> list:
    """Create a complete solution from a list of session solutions.

    Args:
        sessions (list): List of sessions consisting of module ID, student
            group and teacher.
        rooms (list): List of rooms.
        time_slots (list): List of time slots.

    Returns:
        list: A solution, which is a member of the population.
    """
    solution = []
    for session in sessions:
        session_solution = create_session_solution(session, rooms, time_slots)
        solution.append(session_solution)
    return solution
```

B.4 fitness_function.py

```python
"""Phase 2 (fitness function) of genetic algorithm.

Functions:
    check_population_fitness(population: list, teacher_times: list) -> [list,
        bool, list]
    check_fitness_only(population: list, teacher_times: list) -> list
    calculate_fitness(solution: list, teacher_times: list) -> int
"""


def check_population_fitness(population: list, teacher_times: list) \
        -> [list, bool, list]:
    """Calculate the fitness of the whole population.

    Args:
        population (list): The population of solutions.
        teacher_times (list): The time slot preferences of the teachers.

    Returns:
        list: The fitness values of the population.
        bool: Whether or not there is a correct solution.
        list: The correct solution (if it exists).
    """
    population_fitness = []
    valid_solution_bool = False
    valid_solution = None

    for solution in population:
        sol_fitness = calculate_fitness(solution, teacher_times)
        population_fitness.append([solution, sol_fitness])
        if sol_fitness == 0:
            valid_solution_bool = True
            valid_solution = solution
            break

    return population_fitness, valid_solution_bool, valid_solution

```

```python

def check_fitness_only(population: list, teacher_times: list) -> list:
    """Check the fitness of the population.

    Does not return if there is a valid solution, and what it is if it exists.

    Args:
        population (list): The population of solutions.
        teacher_times (list): The time slot preferences of the teachers.

    Returns:
        list: The fitness values of the population.
    """
    population_fitness = []
    for solution in population:
        sol_fitness = calculate_fitness(solution, teacher_times)
        population_fitness.append([solution, sol_fitness])
    return population_fitness


def calculate_fitness(solution: list, teacher_times: list) -> float:
    """Calculate fitness of a solution.

    Args:
        solution (list): A possible solution to the timetabling problem.
        teacher_times (list): The time slot preferences of the teachers.

    Returns:
        int: Fitness of the solution.
    """
    # Sort by time slot
    solution_sorted = solution
    solution_sorted.sort(key=lambda x: x[0])
    problem_count = 0

    for i in range(len(solution_sorted) - 1):
        # Check if sessions are happening at the same time
        if solution_sorted[i][0] == solution_sorted[i + 1][0]:
            # Room clash:
```

```
            if solution_sorted[i][1] == solution_sorted[i + 1][1]:
                problem_count += 1
            # Student group clash:
            if solution_sorted[i][2] == solution_sorted[i + 1][2]:
                problem_count += 1
            # Teacher clash:
            if solution_sorted[i][4] == solution_sorted[i + 1][4]:
                problem_count += 1
    # Check teacher preferences
    for i in range(len(solution_sorted)):
        time_slot = solution_sorted[i][0]
        teacher_num = str(solution_sorted[i][4])
        teacher_prefs = teacher_times.get(teacher_num)
        if time_slot not in teacher_prefs:
            problem_count += 1
    return problem_count
```

B.5 selection.py

```python
"""Phase 3 (selection) of genetic algorithm.

Functions:
    select_parents(pop_fitness: list, population_size: int)
        -> [list, int]
"""
import math


def select_parents(pop_fitness: list, population_size: int) \
        -> [list, int]:
    """Select the parents for crossover.

    Args:
        pop_fitness (list): The fitness values of the population.
        population_size (int): The size of the population.

    Returns:
        list: The chosen parents
        int: Fitness of the chosen parent with the worst fitness
    """
    parents = []
    sorted_fitness = pop_fitness
    sorted_fitness.sort(key=lambda x: x[1])
    for i in range(2 * math.ceil(population_size / 4)):
        parent = sorted_fitness[0][0]
        parents.append(parent)
        if i == (2 * math.ceil(population_size/4)) - 1:
            worst_parent_fitness = sorted_fitness[0][1]
        del sorted_fitness[0]
    return parents, worst_parent_fitness
```

B.6 crossover.py

```python
"""Phase 4 (crossover) of genetic algorithm.

Functions:
    crossover(chosen_parents: list) -> [list, list]
"""
import copy
import random
import math


def crossover(parent_list: list) -> [list, list]:
    """Crossover parents to produce offspring.

    Args:
        chosen_parents (list): The parents used to create the offspring.

    Returns:
        list: The offspring.
        list: Unmodified copy of the parents.
    """
    offspring = []
    parent_list_copy = copy.deepcopy(parent_list)
    num_of_sessions = len(parent_list[0])

    for i in range(int(len(parent_list_copy) / 2)):

        # Select two parents to produce two offspring together
        parent_a = random.choice(parent_list)
        parent_list.remove(parent_a)
        parent_b = random.choice(parent_list)
        parent_list.remove(parent_b)

        for i in range(2):

            # locus_outer: session that contains :while condition:
            locus_outer = random.randint(0, num_of_sessions - 1)

            # locus_inner: split after session[locus_inner]
            if locus_outer == 0:
                locus_inner = random.randint(1, 2)
            else:
                locus_inner = random.randint(0, 2)

            # Crossover of child a
            left_a = parent_a[:locus_outer]
            centre_a = parent_a[locus_outer][:locus_inner] + \
                parent_b[locus_outer][locus_inner:]
            right_a = parent_b[locus_outer + 1:]
            left_a.append(centre_a)
            child_a = left_a + right_a
            offspring.append(child_a)

            # Crossover of child b
            left_b = parent_b[:locus_outer]
            centre_b = parent_b[locus_outer][:locus_inner] + \
                parent_a[locus_outer][locus_inner:]
            right_b = parent_a[locus_outer + 1:]
            left_b.append(centre_b)
            child_b = left_b + right_b
            offspring.append(child_b)

    return offspring, parent_list_copy
```

B.7 mutation.py

```python
"""Phase 5 (mutation) of genetic algorithm.

Functions:
    mutate(offspring_in: list, time_slots: list, rooms: list, sessions: list,
        mutation_chance: int) -> list
"""
import copy
import random


def mutate(offspring_in: list, time_slots: list, rooms: list, sessions: list,
           mutation_chance: int) -> list:
    """Chance for offspring to mutate.

    Args:
        offspring_in (list): The pre-mutation offspring.
        time_slots (list): The list of time slots.
        rooms (list): The list of rooms.
        sessions (list): The list of sessions.
        mutation_chance (int): The chance of mutation.

    Returns:
        list: List of mutated offspring.
    """
    mutation_count = 0
    offspring = copy.copy(offspring_in)
    for i in range(len(offspring)):  # For each solution
        solution = offspring[i]
        for j in range(len(solution)):  # For each session
            for k in range(3):
                mutate = random.randint(1, mutation_chance)
                # Mutation does occur
                if mutate == 1:
                    mutation_count += 1
                    match k:
                        # Mutation of time slot
                        case 0:
                            new_time_slot = random.choice(time_slots)
                            offspring[i][j][0] = new_time_slot
                        # Mutation of room
                        case 1:
                            new_room = random.choice(rooms)
                            offspring[i][j][1] = new_room

                        # Mutation of session
                        case _:
                            new_session = random.choice(sessions)
                            offspring[i][j][2] = new_session[0]
                            offspring[i][j][3] = new_session[1]
                            offspring[i][j][4] = new_session[2]

    return offspring
```