Computer Science
and Mathematics
COC255
B827126

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

Contents

Chapter 1

Literature Review

1.1 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.

1.2 Methods

1.2.1 Genetic Algorithms

Genetic algorithms (GAs) made up a group of search metaheuristics, inspired by Darwin's theory of evolution [**?**]. 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 (date?) 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 expected that it would be too difficult to generate timetables with 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.

1.2.2 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. 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. REF 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
- $M = \{1, \ldots, n_m\}$: set of teachers
- $N = \{1, \ldots, n_n\}$: set of classrooms

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

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

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 (Abdellahi and Eledum, 2017) defined their auxiliary variables as

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

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

for $s \in 1, \ldots, n_k - 1,$ 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 (Abdellahi and Eledum, 2017), 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: (Abdellahi and Eledum, 2017) 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. 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 (Abdellahi and Eledum, 2017).

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

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, 2017) 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:

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

subject to

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

$$\sum_{l\in L}\sum_{k\in K} x_{i,j,k,l,m} \leq 1 \quad \forall i \in I, \forall j \in J, \forall m \in M \tag{Constraint 2}$$

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

(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. By using a penalty function, (Abdellahi and Eledum, 2017) 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.

1.2.3 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.

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 (Awad et al, 2021). They 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.

1.2.4 Answer Set Programming

Answer set programming (ASP) reduces problems to logic programs, and then answer set solvers are used to do the search. 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:

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

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 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.

1.2.5 Bat Inspired Algorithm

The Bat Inspired Algorithm (BA) is a heuristic method originally proposed by Yang (2010). 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 described as:

```
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
End for
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
        End if
        If (rand < Ai) and (f(xi) < f(x*))
            Accept the new solution
            Increase ri and reduce f(xi)
        End if
    End for
End while
Ranks the bats and obtain the current best bat
```

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:

$$f_i = f_{\min} + (f_{\max} - f_{\min})\beta \tag{1.1}$$

$$v_i^t = v_i^{t-1} + (x_i^{t-1} - x_*)f_i \tag{1.2}$$

$$x_i^t = x_i^{t-1} + v_i^t \tag{1.3}$$

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:

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

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, the authors 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 (who?) 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.

1.3 Comparison of Methods

1.4 Chosen Method

Chapter 2

Requirements

2.1 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.

2.2 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 3

Design and Implementation

3.1 Program Overview

3.2 Genetic Algorithm

3.2.1 Initial Population

3.2.2 Fitness Function

3.2.3 Selection

3.2.4 Crossover

3.2.5 Mutation

Chapter 4

Testing

Chapter 5

Evaluation

Chapter 6

Conclusion

References

Appendix A

Source Code

A.1 main.py

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

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



def menu():
    """Print a welcome message to the user, with
       instructions."""
    clear()
    print("Timetable␣Maker\nPlease␣ensure␣that␣config.json␣
       is␣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()
```

A.2 ga.py

```
"""Genetic algorithm base.

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


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(teacher_times)
    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:
    print("No␣timetable␣solution␣found.")
    parent_a, parent_b = p3.select_parents(
       population_fitness)
    # Note that parent_a and parent_b are indices
    offspring = p4.crossover(population[parent_a],
       population[parent_b],
                              len(population[0]),
                                 population_size)
    # check if any of the offspring is a valid solution
    mutated_offspring = p5.mutate(offspring, time_slots
       , rooms, sessions)
    mutated_offspring.append(population[parent_a])
    mutated_offspring.append(population[parent_b])
    population_fitness, valid_solution_bool,
       valid_solution \
        = p2.check_population_fitness(mutated_offspring
           )
    if not valid_solution_bool:
        population = mutated_offspring
        for i in range(2):
            worst_fitness = max(population_fitness)
            worst_fitness_index = population_fitness.
               index(worst_fitness)
            del population_fitness[worst_fitness_index]
            del population[worst_fitness_index]
print("Timetable␣solution␣found.␣Writing␣output␣to␣text
   ␣files...")
```

```
    generate_output_text(valid_solution, "teachers")
    generate_output_text(valid_solution, "student_groups")
    print("Output␣written␣to␣files␣in␣./teacher-timetables␣
       and␣\
␣␣␣␣./student-group-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


def generate_output_text(solution: list, person_type: str):
    """Generate output of a correct timetable to text files
       .

    Args:
        solution (list): A correct timetable solution.
        person_type (str): Who the timetable is for,
           student_groups or teachers
    """
    dictionary = {}
```

```
file = open("data.json", "r", encoding="utf-8")
data = json.load(file)
file.close()

for person in data[person_type]:
    dictionary.update({person["id"]: []})
for session in solution:
    if person_type == "teachers":
        session_person = session[4]
    else:  # session_person == "student_groups"
        session_person = session[2]
    session_person_list = dictionary.get(session_person
       )
    session_person_list.append(session)
    dictionary.update({session_person:
       session_person_list})

print("Creating␣directory␣for␣" + str(person_type) + "␣
   timetables...")
try:
    os.mkdir("./" + str(person_type) + "_timetables")
    print("Directory␣'" + str(person_type) + "
       _timetables'␣created.")
except FileExistsError:
    print("Directory␣already␣exists.")
for session_person in dictionary:
    file = open(str(session_person) + ".txt", "w",
       encoding="utf-8")
    person_sessions = dictionary[session_person]
    for session in person_sessions:
        file.write(str(session))
    file.close()
print("Timetable␣output␣files␣created.")
# TODO: formatting?
```

A.3 initial population.py

```
"""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.
```

```
"""
print("Reading␣data␣file")
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["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 = []
    for day in teacher["available_times"]:
        match day:
            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 _:
                day_number = 0
        for start_time in day["start_times"]:
            time_number = day["start_times"][start_time
               ]
            teacher_time.append(str(day_number) + str(
               time_number))
    teacher_times.append(teacher_time)

return sessions, rooms_id, time_slots_id, teacher_times


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.
    """
    print("Generating␣initial␣population...")
    population = []
    for i in range(population_size):
        solution = create_complete_solution(sessions, rooms
           , time_slots)
        population.append(solution)
    print("Initial␣population␣generated.")
    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
```

A.4 fitness function.py

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

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


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).
    """
    print("Calculating␣the␣fitness␣of␣individuals...")
    population_fitness = []
    valid_solution_bool = False
    valid_solution = None
    for solution in population:
        sol_fitness = calculate_fitness(solution,
           teacher_times)
        population_fitness.append(sol_fitness)
        if sol_fitness == 0:
            valid_solution_bool = True
            valid_solution = solution
            break

    print("Fitness␣of␣individuals␣calculated.")
    return population_fitness, valid_solution_bool,
       valid_solution


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:
        float: Fitness of the solution.
    """
    # Sort by time slot
    solution.sort(key=lambda x: x[0])
    problem_count = 0
    for i in range(len(solution) - 1):

        # Check if sessions are happening at the same time
        if solution[i][0] == solution[i + 1][0]:

            # Room clash:
            if solution[i][1] == solution[i + 1][1]:
                problem_count += 1

            # Student group clash:
            if solution[i][2] == solution[i + 1][2]:
                problem_count += 1

            # Teacher clash:
            if solution[i][4] == solution[i + 1][4]:
                problem_count += 1

    for i in range(len(solution)):

        # check teacher preferences
        time_slot = solution[i][0]
        teacher_num = int(solution[i][4])

    if problem_count == 0:
        fitness = 0
    else:
        fitness = 1 / problem_count
    return fitness
```

A.5 selection.py

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

```
Functions:
    select_parents(population_fitness: list) -> [int, int]
    normalise_values(fitness_values: list) -> list
    choose_parent(range_limits: list) -> int
"""
import random


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

    Args:
        population_fitness (list): The fitness values of
           the population.

    Returns:
        int: Index of the first parent.
        int: Index of the other parent.
    """
    print("Selecting␣parents␣for␣crossover...")
    fitness_values = population_fitness
    print(len(fitness_values))  # debugging TODO: remove
       later
    range_limits_a = normalise_values(fitness_values)
    print(range_limits_a)  # debugging TODO: remove later
    parent_a = choose_parent(range_limits_a)
    print("First␣parent␣selected.")
    fitness_values.remove(fitness_values[parent_a])

    print(len(fitness_values))  # debugging TODO: remove
       later
    range_limits_b = normalise_values(fitness_values)
    parent_b = choose_parent(range_limits_b)
    print(str(parent_b))  # debugging TODO: remove later
    print("Second␣parent␣selected.")
    return parent_a, parent_b


def normalise_values(fitness_values: list) -> list:
    """Normalise the fitness values so that they summate to
        1.

    Args:
        fitness_values (list): The fitness values of the
           population.

    Returns:
        list: Cumulative sums of the normalised fitness
           values.
    """
    fitness_total = 0
    for value in fitness_values:
        fitness_total += value
    range_limits = []
    for i in range(len(fitness_values)):
        value_norm = fitness_values[i] / fitness_total
        range_limits.append(value_norm)
        if i != 0:
            range_limits[i] += range_limits[i - 1]
    range_limits[len(fitness_values) - 1] = 1
    range_limits.insert(0, 0)
    return range_limits  # limits are upper limits?
```

```
def choose_parent(range_limits: list) -> int:
    """Perfom a binary search to get the index of the
       chosen parent.

    Args:
        range_limits (list): Cumulative sums of the
           normalised fitness values.

    Returns:
        int: The index of the chosen parent.
    """
    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 = middle
        elif range_limits[middle] < choice:
            lower = middle
        else:
            upper = middle
    return parent_index
```

A.6 crossover.py

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

Functions:
    crossover(parent_a: list, parent_b: list,
       num_of_sessions: int,
        population_size: int) -> list
"""
import random
import math


def crossover(parent_a: list, parent_b: list,
   num_of_sessions: int,
             population_size: int) -> list:
    """Crossover parents to produce offspring.

    Args:
        parent_a (list): The first selected parent.
        parent_b (list): The other selected parent.
        num_of_sessions (int): How many lesson sessions are
            needed to make up
            the timetable.
        population_size (int): The size of the population.

    Returns:
        list: The offspring.
    """
    print("Producing␣offspring...")
    offspring = []
    half_pop = math.ceiling(population_size / 2)
    for i in range(half_pop):
        # locus_outer: session that contains the split
        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)  # TODO:
               hard coding
        else:
            locus_inner = random.randint(0, 2)  # TODO:
               hard coding

        # 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)

    # Remove last child if the populaton size is an odd
       number
    if half_pop != math.floor(population_size / 2):
        del offspring[-1]

    print("Offspring␣produced.")
    return offspring
```

A.7 mutation.py

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

Functions:
    mutate(offspring_in: list, time_slots: list, rooms:
       list, sessions: list,
        mutation_chance: int) -> list
"""
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.
    """
    print("Mutating␣offspring...")
    mutation_count = 0
    offspring = offspring_in
```

```
for solution in offspring:
    for session in solution:
        for i in range(3):
            mutate = random.randint(1, mutation_chance)

            # Mutation does occur
            if mutate == 0:
                mutation_count += 1
                match i:

                    # Mutation of time slot
                    case 0:
                        new_time_slot = random.choice(
                           time_slots)
                        offspring[solution][session][0]
                            = new_time_slot

                    # Mutation of room
                    case 1:
                        new_room = random.choice(rooms)
                        offspring[solution][session][1]
                            = new_room

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

```
                                    = new_session[2]
print("Offspring␣mutated.␣" + str(mutation_count) +
       "␣mutation(s)␣occured.")
return offspring
```