0 Index

List of Figures

List of Algorithms

1 Overview

This document presents all the important information regarding the *Classroom management at the School of Computer Engineering using Artificial Intelligence methods* end-of-degree thesis.

It is important to note that the structure of the contents for this document is done following the criteria and recommendations of the template document for Degree's and Master's Thesis of the School of Computing Engineering of Oviedo (version 1.4) by Redondo [Red]. However, some additional chapters were introduced in order to capture the particularity of the work carried out, inspired by the research of de la Cruz [dlC18].

Introduction. Here we explain in a simple way the problem we want to solve, what reasons are behind the development of the project and give a description of the current situation of the School with regard to this and other similar problems. We also provide a broad outline of the scope for the project and what objectives need to be met for the project to succeed.

Theoretical aspects. The first chapter delving into the theory supporting the developed system. One example of an assignment problem is presented and solved using a greedy algorithm. Then, an in-depth study of heuristics and metaheuristics is carried out, covering a wide range of evolutionary computation algorithms. Finally, an explanation is given on how to combine the greedy and genetic algorithms to solve the example problem, which mirrors on a small scale the overall work done for the project.

Problem definition. Here the formulation of the problem as an assignment problem is elaborated. We present the information we have of the problem and simultaneously carry out a structural analysis of its components.

Proposed solution. This chapter lays down in detail the proposed solution to the previously defined assignment problem. It is in this chapter where we define the search space of the problem, basic concepts to understand the behaviour of the algorithms and the techniques previously identified in the theoretical aspects section that we use to solve the real problem.

Project planning and budget overview. This outlines the project planning and the

internal and client budgets. The WBS can be found in the annexes and the fully detailed budget is given in its own section at the end of the document.

Analysis. An analysis of the system, with the system requirements, draft diagrams, use cases and the test plan specification.

System design. The technical details of the system analysed in the previous section. Here we include the finalised diagrams, as well as the arquitecture of the system and the in-depth test plan.

System implementation. Details of the development of the software. The programming languages, standards and tools used to code the system, and all the relevant information gathered in the process of creating the system.

Test development. A rundown of all the testing done for the system, with explanations for every test and the obtained results.

Experimental results. The conclusions reached after experimenting with different input data and the optimal configuration for the parameters of the genetic algorithm.

System manuals. All system manuals are provided, with explanations and guided steps in the required level of detail to help the target readers of each manual to complete their work..

Conclusions and future work. I express here the findings that I have obtained through the development of this project and also point out a series of project-related topics that may be carried out in the future by other colleagues.

Budget. The full details for the elaboration of the internal and client budgets are shown, with all the intermediate steps that we took to calculate them.

Annexes. The glossary of definitions and abbreviations and a small commentary on the submitted files. In addition to this, you can find here additional information on the project that was not necessary to elaborate on in the different sections.

Source code. Here we compare the pseudocode of the algorithms explained in the problem solution with the real implemented algorithms.

2 Introduction

The School of Computing Engineering of the University of Oviedo has more than twenty classrooms, including theory classrooms and laboratory classrooms. Each semester there are over three hundred groups, each with their type (theory, seminar or laboratory), subject and schedule. The timetable of the groups varies on a weekly basis, this means that not all groups have to attend classes all weeks, and some of them do not even have repeating patterns.

This makes assigning classrooms to groups a complicated task, since there can be no temporal collisions. When various other constraints enter the equation, such as minimising the number of labs used by a subject or assigning classrooms to Spanish groups that are different from English groups, things become much more complex.

All this assignments are done *manually* by one person. The number of enrolled students can only be *guessed* when this process is done. This means that groups can be created, modified or removed once the semester has already started, so more assignments are usually made, checking once again all the restrictions. These new assignments are difficult to manage as there is not much room for flexibility to change those made before the semester. This is due to the fact that both students and teachers already use the initial assignments as a reference.

This project provides the supervisor of this process with a tool to help them perform the assignments, reducing their workload. Not only does it generate assignments for all the groups of the semester, but can use previous assignments, total or partial, to calculate a subset of assignments (for example, the assignments for the new groups created in the middle of the semester). On top of that, the prototype developed in this thesis makes finding a set of free classrooms to hold events easy and fast, using the assignments generated previously by the system itself.

2.1 Situation overview

At the beginning of each semester, the School opens a process in which the person in charge takes the list of groups for the semester, their schedules and the list of classrooms, and performs a manual compilation of all the assignments.

There are a number of other similar procedures, like the creation of the exam timetable or the assignments of enrolled students to subject groups. However, some are not manual, but automated by a system, like the previously mentioned procedure of assigning students to groups. Seeing the potential of such tools, I was given the task of automating the assignment of classrooms to subject groups by similar means.

The procedure of assigning the classrooms is done after configuring the student groups for the semester and knowing their schedules. Even though it is a manual process, the supervisor does not start making the assignments from scratch. First, they have the knowledge of previous years, and then they have a list of preferences or premade assignments. For example, certain laboratories can only be assigned to specific groups, like the ones from the Electronic Technology of Computers subject. The system described in this document preserves these sources of information and builds on top of them.

2.2 Purpose

This project aims to help the personnel of the School manage their classrooms. It will address two main functionalities, the automation of the process of assigning classrooms to all the groups of a given semester (starting from scratch or using a previous partial or total assignment), and a tool that searches for gaps in a previous set of assignments for single or multi-day events in one or more classes.

The implementation of this system is intended to assist in the work of the supervisor for this process, and provide an efficient and flexible tool that expands the possibilities of such work. To do so, the program executes two algorithms, a genetic algorithm and a greedy algorithm (the genetic being *guided* by the greedy, more on that later). For a more detailed view on these algorithms the reader might refer to 3. Once the assignments have been calculated, the system will allow the users to find classrooms to hold specific events in the middle of the semester.

Along with the system, the system manuals are submitted. These have the purpose of teaching how to install, use, maintain and extend the system.

2.3 Scope

The project needs to formally define the problem of assigning classrooms of the School to all the groups of the semester, conduct a study on the problem and propose a solution.

A development of a software prototype that solves the problem is planned, designed, implemented and tested. This prototype will solve the two main functionalities indicated in 2.2 and will consist of a command line application that takes input data in plain text files and outputs the solution to plain text files. The program is configured by different configuration files depending on the functionality being executed. An experimental study on the results of the software system is carried out, finding the most fitting default values for the configuration files. The project also contains the system manuals of the application, which consist of the installation, usage, user and programmer manuals.

Finally, the prototype also has a module for automating the creation of the input files required for the main functionalities to work. It uses a format agreed with the client and will parse files previously used by the School, making it easier to integrate with other systems already in use.

2.4 Project goals

We can identify from the scope the following objectives. They need to be met in order to close the project successfully:

1. Formally define the problem of assigning classrooms to the groups of the School.
2. Study the problem and the means to solve it.
3. Define the proposed solution.
4. Build a prototype that solves the problem using the algorithms described in the proposed solution.
    (a) It will receive plain text input files with the required data.
    (b) It will output the solution to plain text files.

(c) It will be able to make the assignments starting from scratch or from a total or partial set of assignments.

(d) It will be able to search a set of free classrooms for a specific event in one or more days.

(e) It will be able to automate the creation of the input plain text files for the main functionalities.

5. Make a set of experiments to find the best default values for the configuration files.
6. Write a set of manuals to cover the essentials of the system.
7. Validate solution with the users.

3 Theoretical aspects

A digital magazine Bootaku works with three freelancers. Dante, Virgil and Beatrice. Together they write a section about book reviews. Gathering data from previous sections, Bootaku wants to define and solve a problem of efficiently assign all reviews to the three critics so that the section gets the highest profit. For the assignments, Bootaku wants every book review to have one (and only one) associated freelancer. If a freelancer ends up with no reviews, the assignments are still valid if and only if the previous condition is met.

3.1 Assignment problem

The problem described before is an example of an assignment problem. It can be generalised with the following elements:

A set of n freelancers f

A set of m book reviews r

An assignment matrix of $n \times m$ assignments a_{fr} such that $a_{fr} = 0$ when freelancer f is not assigned to book review r and $a_{fr} = 1$ when freelancer f is assigned to book review r.

A profit matrix of $n \times m$ profits p_{fr} which indicate the profit obtained when assigning freelancer f to book review r and that $p_{fr} > 0$.

A valid solution is defined as a matrix of assignments where all the book reviews have a freelancer assigned to them and no book review has more than one associated freelancer.

The profit for all the assignments will then be:

$$\sum_{f=1}^{n} \sum_{r=1}^{m} a_{fr} p_{fr} \tag{3.1}$$

The optimal solution consists on having a set of assignments such that the sum of all the profits for the current assignments is maximised.

For example, imagine that for the next month's section, we have the following data. The information is represented by means of two sets: F for the freelancers and R for the reviews.

$$F = \{Dante, Virgil, Beatrice\} \tag{3.2}$$

$$R = \{Divina\ Commedia, El\ Quijote, Voyage\ au\ bout\ de\ la\ nuit, Todo\ modo\} \tag{3.3}$$

Then, our assignments and profits will be represented by the A and P matrices.

$$A = \begin{array}{c} \\ Dante \\ Virgil \\ Beatrice \end{array} \begin{array}{c} \begin{array}{cccc} DC & EQ & VN & TM \end{array} \\ \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{pmatrix} \end{array} \tag{3.4}$$

$$P = \begin{array}{c} \\ Dante \\ Virgil \\ Beatrice \end{array} \begin{array}{c} \begin{array}{cccc} DC & EQ & VN & TM \end{array} \\ \begin{pmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \end{pmatrix} \end{array} \tag{3.5}$$

Where each row represents a freelancer and each column represents a book review. So freelancer 1 is Dante, 2 is Virgil and 3 is Beatrice. The same goes for the book reviews. Book review 1 is *Divina Commedia*, 2 is *El Quijote*, 3 is *Voyage Au Bout De La Nuit* and 4 is *Todo Modo*.

Now, we are going to study valid and non-valid solutions. As we explained before, a solution is valid if every book review has a freelancer assigned to it, and no more than one.

We will analyse four sets of values for the A matrix:

$$A1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} \tag{3.6}$$

$$A2 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} \tag{3.7}$$

$$A3 = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \tag{3.8}$$

$$A4 = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} \tag{3.9}$$

From these matrices, we can deduce that $A1$ and $A2$ are valid solutions, because they have one freelancer for each book review. We are not concerned with a freelancer having no book reviews assigned. However, a book review without an associated freelancer represents a non-valid solution. That is precisely the case for $A3$, the book review for *El Quijote* has not an assigned freelancer. In the case of $A4$, the fact that *El Quijote* has two freelancers assigned makes it a non-valid solution.

Now, we will give values to the P matrix in order to discuss possible optimal solutions. We will compare them with the assignment matrices $A1$ and $A2$

$$P1 = \begin{bmatrix} 9 & 1 & 5 & 4 \\ 2 & 8 & 14 & 2 \\ 7 & 11 & 10 & 6 \end{bmatrix} \tag{3.10}$$

$$P2 = \begin{bmatrix} 9 & 1 & 5 & 4 \\ 13 & 8 & 14 & 2 \\ 7 & 11 & 10 & 6 \end{bmatrix} \tag{3.11}$$

$P1$ and $A1$:

$$Profit = \sum_{f=1}^{n} \sum_{r=1}^{m} a_{fr} p_{fr} = 9 + 11 + 14 + 6 = 40 \tag{3.12}$$

$P1$ and $A2$:

$$Profit = \sum_{f=1}^{n} \sum_{r=1}^{m} a_{fr} p_{fr} = 2 + 11 + 14 + 6 = 33 \tag{3.13}$$

We can observe that for $P1$, the assignments defined in $A1$ are better than those in $A2$, because they result in a better profit. Another important remark about $A1$ is that it is the optimal solution to the problem, because it assigns the book reviews to the freelancers with the best profit value for their assigned books. Now $P2$ will be evaluated.

$P2$ and $A1$:

$$Profit = \sum_{f=1}^{n} \sum_{r=1}^{m} a_{fr} p_{fr} = 9 + 11 + 14 + 6 = 40 \tag{3.14}$$

$P2$ and $A2$:

$$Profit = \sum_{f=1}^{n} \sum_{r=1}^{m} a_{fr} p_{fr} = 13 + 11 + 14 + 6 = 44 \tag{3.15}$$

In the case of the profits values of $P2$, the situation is reversed. $A2$ is now the optimal solution and therefore better than $A1$.

The important thing to notice here is that the values for the P matrix right now may appear as having no meaning whatsoever. But we need to think of P as the results obtained from a profit funtion. Then, we can interpret $P1$ as values of profit in a context where Virgil has just expressed an opinion on social media about the *Divina Commedia* and caused a massive controversy. We can then say that $P1$ is a function which gives more importance to public relations and so the profit p_{21} is very low, whereas $P2$ gives more importance to views and so the profit p_{21} is higher. Of course, in a real problem you know what the function is calculating, but this shows how we can add meaning to a set of symbols in order to understand the data more

efficiently.

With this, we have discussed an assignment problem, looked at its main components and analysed its non-valid, valid and optimal solutions. Some final remarks about the relation between a general assignment problem and the Bootaku problem follow. The actors that perform the jobs, in this case the freelancers that *write* the reviews, are called the *agents*. The *tasks* to be performed are, in the Bootaku problem, the book reviews. Nevertheless, the agents in an assignment problem do not need to be persons (or even things that carry out actions), they can be machines, warehouses, or classrooms. The same can be said for the tasks.

3.2 Greedy algorithms

One way of solving the Bootaku problem described earlier can be found in *greedy algorithms*. A greedy algorithm [GV98] will try to find a subset of candidates that meet the problem constraints and that form the optimal solution. To do so, the algoritm is run iteratively. In each iteration, it will select the best candidate for that precise moment, neglecting future consequences (that is why they are called *greedy*)[1]. Before adding a candidate to the solution, the algorithm will determine if it is promising. If the answer is yes, then the candidate is added to the solution. Otherwise, the candidate is no longer evaluated. Each time a candidate is added to the solution, the greedy checks whether the current solution is valid or not.

With this in mind, here follows the *pseudocode* of the generic Greedy Algorithm.

[1]For example, let's say I'm walking down the street and I get thirsty. On my mental list of candidate drinks, water has a value of 5 points, lemonade 3 and tea 1. The first vending machine I come across on the street only sells lemonade and tea, so I buy lemonade. However, on the next street I find another vending machine that sells water, but as I am no longer thirsty I don't buy any more drinks. That is, I found a valid solution but not the optimal solution.

Algorithm 1 Generic Greedy Algorithm

```
1:  procedure GreedyAlgorithm(candidates)
2:      x ← ϵ
3:      solution ← {}
4:      found ← false
5:      while !isEmpty(candidates) and !found do
6:          x ← selectCandidate(candidates)
7:          if isPromising(x, candidates) then
8:              addToSolution(x, solution)
9:              if isSolution(solution) then
10:                 found ← true
11:             end if
12:         end if
13:     end while
14:     return solution
15: end procedure
```

As we can see, the generic greedy algorithm has a very simple and elegant definition. However, even though greedy algorithms are easy to implement and can obtain efficient solutions, they are not perfect. Their main flaw relies on their selection function. It is difficult to design a function that can simultaneously find a good local result and translate it into a good global result. That is, the best candidate in some iteration may not be part of the optimal solution.

Next, we will solve the Bootaku problem using a greedy algorithm. The greedy algorithm that we are going to use is inspired by the one defined in [GV98] for solving the *Assignment of tasks* problem.

Algorithm 2 Bootaku Greedy Algorithm

```
1: procedure GreedyBootaku(profits, assignments)
2:     best ← ϵ
3:     for each freelancer F_i ∈ F do
4:         for each review R_j ∈ R do
5:             assignments[F_i, R_j] = false    ▷ We initialise the assignments matrix
   (to false or cero, it does not matter).
6:         end for
7:     end for
8:     for each review R_j ∈ R do
9:         best ← bestFreelancerFor(profits, assignments, R_j)
10:        assignments[best, R_j] = true    ▷ Again, it can be true or one, depending
   on the implementation.
11:    end for
12:    return assignments
13: end procedure
```

Algorithm 3 Bootaku Greedy Algorithm BestFreelancerFor

```
1: procedure BestFreelancerFor(profits, assignments, review)
2:     best ← ϵ
3:     min ← maximum integer value
4:     for each freelancer F_i ∈ F do
5:         if profits[F_i, review] < min then
6:             min ← profits[F_i, review]
7:             best ← F_i
8:         end if
9:     end for
10:    return best
11: end procedure
```

Those are the two procedures needed in order to solve the Bootaku problem. In the *Assignments of tasks* problem described in the book, the authors define an extra function that checks if the worker is already assigned to another task. However, because our problem is not balanced (we have a different number of tasks and agents), it means that we can have a freelancer writing more than one book review, so that extra function is not required.

The Bootaku problem is really simple to solve because of its lack of constraints. This is done deliberately to focus more on the components of assignment problems and not to waste time on explaining difficult restrictions. However, most problems, including the real problem this document defines (to assign classrooms to the groups of the School), have a lot of constraints.

One way to complicate the Bootaku problem would be to assign completion times to each review. We would have a $n \times m$ T matrix with the completion times for all freelancers and reviews.

$$T = \begin{array}{c} \\ Dante \\ Virgil \\ Beatrice \end{array} \begin{array}{c} \begin{array}{cccc} DC & EQ & VN & TM \end{array} \\ \begin{pmatrix} t_{11} & t_{12} & t_{13} & t_{14} \\ t_{21} & t_{22} & t_{23} & t_{24} \\ t_{31} & t_{32} & t_{33} & t_{34} \end{pmatrix} \end{array} \tag{3.16}$$

Then we could have a maximum time per freelancer. This would force the greedy algorithm to perform a check before assigning a review to a freelancer. If the time it takes to write the review surpases the maximum time available for that freelancer, the assignment cannot be made.

Let's say that Beatrice has been assigned to the book review for *El Quijote*, so she has already spent a total time of $t_{32} \leq maxTime$. In a future iteration the greedy algorithm evaluates Beatrice for reviewing *Todo Modo*. Even if she has the greatest profit for *Todo modo*, it is still not enough. The greedy algorithm first has to check in the $BestFreelancerFor$ procedure if $t_{32} + t_{34} \leq maxTime$ and, if the condition is true, then the assignment is performed.

We can notice in the Beatrice example the main failure of greedy algorithms. She was assigned to *El Quijote* for a profit p_{32} and then evaluated again for *Todo modo* with a profit p_{34}. Imagine that she has not enough time left to be able to review the second book and that p_{34} is *way bigger* than p_{32}. This is where assigning the best local result a_{32} would end up ruling out the possibility of assigning the better global result a_{34}. Later in this chapter we will see a possible way of fixing this problem with the help of genetic algorithms, but before that we will conduct a study on *metaheuristic*s.

3.3 Heuristics and metaheuristics

Search strategies in a search problem can be *informed* or *uninformed*. Informed strategies use knowledge specific to a given problem but that is outside of its definition, making this type of strategies more efficient than uninformed strategies. The main way of applying our knowledge of a given problem into the search algorithm designed to solve said problem is by means of *heuristic functions*. An heuristic function $h(n)$ [RN10] represents an estimation of the minimum cost of getting to the objective state from the state given by node n. To expand a node, the algorithm also makes use of an *evaluation function*. An evaluation function $f(n)$ analyses the non-expanded nodes and selects the one with the lowest cost. In the case of greedy algorithms, the evaluation function of a node n is equivalent to the heuristic function of the same node. So we have that $f(n) = h(n)$.

Now that we have explained what heuristic functions are, one question remains. What are *metaheuristics*? Analysing the word, one could think that the *meta* prefix implies that metaheuristics are *heuristics about heuristics*, in the same way *metadata* is *data about data*. However, as Luke [Luk13] points out, this is not the case at all. He defines metaheuristics as:

> ... a rather unfortunate term often used to describe a major subfield, indeed the primary subfield, of **stochastic optimization**. Stochastic optimization is the general class of algorithms and techniques which employ some degree of randomness to find optimal (or as optimal as possible) solutions to hard problems. Metaheuristics are the most general of these kinds of algorithms, and are applied to a very wide range of problems.

There are many methods of designing algorithms based on *metaheuristics*. In this project we will focus on the Evolutionary Computation method, a subtype of Population-based methods.

3.3.1 Evolutionary Computation

Evolutionary Computation (EC) [Luk13] takes inspiration from population biology, genetics and evolution [2]. We are interested in the types of algorithms designed using

[2]Because this method uses vocabulary from these fields of biology, we have followed Luke's approach and defined these terms one by one. A list of definitions of the most commonly used terms in

this method, known as Evolutionary Algorithms (EAs). An EA may be (usually) either a *generational algorithm* or a *steady-state algorithm*. A generational algorithm creates a new population of individuals, based on the previous one, in each iteration. Moreover, a steady-state algorithm changes a subset of individuals in each iteration, but not the entire population. The most common EAs are the *Genetic Algorithms* and the *Evolution Strategies*, and there are generational and steady-state versions of the two.

Below is the pseudocode for an abstract generational algorithm.

Algorithm 4 Abstract Generational Algorithm

```
1:  procedure AbstractGenerationalAlgorithm(maxTime)
2:      P ← create the initial population
3:      best ← ϵ
4:      currentTime ← get current time
5:      while !idealSolution(P) and currentTime ≤ maxTime do
6:          evaluate(P)                    ▷ Calculate the fitness of all individuals.
7:          for each individual P_i ∈ P do
8:              if best = ϵ or Fitness(P_i) > Fitness(best) then
9:                  best ← P_i
10:             end if
11:         end for
12:         P ← newGeneration(P, breed(P))
13:         currentTime ← update time
14:     end while
15:     return best
16: end procedure
```

The initial population in these kinds of algorithms is created by adding random individuals to a set until the maximum population size is reached. Some good practices for this process follow. The most important thing is not to generate repeated individuals. This can be done with a dictionary in which we store the individuals as keys. For every new randomly generated individual we check if it is not already contained in said dictionary before adding it to the population set. Finally, it is possible to include individuals designed by hand into the initial population (this is called *seeding* the

EC has been created in the annex of definitions and abbreviations.

population). However, the use of EAs already implies that finding a good heuristic for the problem is not trivial. So even if we think that the individuals we design may be going in the right direction, it is very likely that they will end up producing poor results.

The main difference between generational EAs relies on how they create the new generation. This process is done by means of different operations such as *selection*, *crossover* and *mutation*. Also, some EAs simply discard all the parents in the new generation and others include them again if they have an acceptable fitness. We will take a look at two specific EAs in the following sections, Evolution Strategies and Genetic Algorithms.

3.3.2 Evolution Strategies

Evolution Strategies (ES) [Luk13] are a type of Evolutionary Algorithms. They make use of a selection operator called *Truncation Selection*. This operator consists of selecting individuals from the highest to the lowest fitness value until a predetermined number of selected candidates is reached. For creating the new generation, ES simply use the mutation operator, without combining it with a crossover operator.

The only ES to be covered in this section is the $(\mu,\ \lambda)$ algorithm. We have chosen it because it is one of the simplest ES and therefore easier to understand. The design of $(\mu,\ \lambda)$ is essentially one version of the Abstract Generational Algorithm that details the way in which the new generation is built. This new generation is constructed by using both μ and λ parameters.

In this algorithm, we have an initial population of λ number of individuals which are randomly generated. Then, we evaluate the fitness of all individuals, as we did in the Abstract Generational Algorithm, and we calculate the best individual in the generation. The next step is to create the new generation. To do so, $(\mu,\ \lambda)$ performs the Truncation Selection on the parents, selecting the μ number of parents with greatest fitness, and for each parent a λ/μ number of children are generated. A mutation operation is performed to create the offspring from a copy of their parent. This whole process is done until we arrive at an optimal solution or the maximum time runs out. The pseudocode of the $(\mu,\ \lambda)$ algorithm is shown as follows.

Algorithm 5 The $(\mu,\ \lambda)$ Evolution Strategy

```
1:  procedure MuLambdaES(μ, λ, maxTime)
2:      best ← ϵ
3:      currentTime ← get current time
4:      P ← {}
5:      for λ times do
6:          P ← P ∪ {new random individual}
7:      end for
8:      while !idealSolution(P) and currentTime ≤ maxTime do
9:          evaluate(P)                    ▷ Calculate the fitness of all individuals.
10:         for each individual P_i ∈ P do
11:             if best = ϵ or Fitness(P_i) > Fitness(best) then
12:                 best ← P_i
13:             end if
14:         end for
15:         Q ← μ individuals with the highest to lowest fitness
16:         P ← {}
17:         for each individual Q_j ∈ Q do
18:             for λ/μ times do
19:                 P ← P ∪ {mutation(copy(Q_j))}
20:             end for
21:         end for
22:         currentTime ← update time
23:     end while
24:     return best
25: end procedure
```

Knowing how to give values to the parameters λ, μ and the mutation probability is very important in this algorithm. In the case of λ, as it approaches ∞ the algorithm starts behaving as a random search algorithm, so it is best if it does not have an excessively high value. For μ, if we give it a very low value, the algorithm becomes very selective and focuses only of a specific type of individual with a high fitness value. This may result in *premature convergence* of the algorithm and thus end the execution with a locally optimal rather than a globally optimal solution. Lastly, because the mutation operation determines the similarity between parents and offsprings, if the mutation probability is very high, the new population would be very different from the

previous one. Therefore, it would make children resemble random individuals.

To conclude the section, simply note that there is another algorithm, very similar to this one, called $(\mu + \lambda)$. The only difference between $(\mu + \lambda)$ and $(\mu,\ \lambda)$ is that while $(\mu,\ \lambda)$ discards the parents when creating the new generation, $(\mu + \lambda)$ makes a union between parents and children. This makes each new generation of the $(\mu + \lambda)$ algorithm having a size of $\mu + \lambda$, where μ is the number of parents and λ the number of new offspring. Because very fit parents can survive for several generations, $(\mu + \lambda)$ behaves like a $(\mu,\ \lambda)$ with a very low μ number, it can terminate with a premature convergence.

3.3.3 Genetic algorithms

The Genetic Algorithm (GA) [Luk13] is a type of Evolutionary Algorithms with a strong similarity towards the $(\mu,\ \lambda)$ Evolution Stategy. What separates the two algorithms the most is the selection operation and the way a new generation is created. In $(\mu,\ \lambda)$, candidates were chosen by Truncated Selection. Once the candidates are selected, the next generation is populated by mutations of the parental copies. However, in the GA, a pair of parents is selected and their offspring are immediately created, adding them to the new generation. This process of simultaneous selection and reproduction occurs until the population reaches the maximum number of individuals set. Further discussion of the GA will follow, once we have seen its pseudocode.

Algorithm 6 The Genetic Algorithm (GA)

```
1:  procedure GeneticAlgorithm(popsize, maxTime)
2:      best ← ϵ
3:      currentTime ← get current time
4:      P ← {}
5:      for popsize times do
6:          P ← P ∪ {new random individual}
7:      end for
8:      while !idealSolution(P) and currentTime ≤ maxTime do
9:          evaluate(P)                          ▷ Calculate the fitness of all individuals.
10:         for each individual P_i ∈ P do
11:             if best = ϵ or Fitness(P_i) > Fitness(best) then
12:                 best ← P_i
13:             end if
14:         end for
15:         Q ← {}                               ▷ Here the GA begins to differ from (μ, λ).
16:         for popsize/2 times do
17:             Parent P_a ← selectWithReplacement(P)
18:             Parent P_b ← selectWithReplacement(P)
19:             Children C_a, C_b ← crossover(copy(P_a), copy(P_b))
20:             Q ← Q ∪ {mutate(C_a), mutate(C_b)}
21:         end for
22:         P ← Q
23:         currentTime ← update time
24:     end while
25:     return best
26: end procedure
```

As we said before, the GA is differentiated from the $(\mu,\ \lambda)$ ES by means of its selection, crossover and mutation operators. We will now elaborate on these operators, defining them and explaining some of their implementations.

3.3.3.1 Selection

We begin with the selection operator. Even if the selection variant in the GA is different from the one in $(\mu,\ \lambda)$, the concept of selection is equivalent in both algorithms.

In other words, both use the operation of selection to obtain the parents who will produce the children of the future generation. There can be multiple ways to implement the selection operator, including *Random Selection*, *Fitness-Proportionate Selection*, *Stochastic Universal Sampling*, *Tournament Selection* and a variant of the GA which includes *elitism*.

The Random Selection variant, as its name implies, picks two random parents, removing them from the population. After generating the offspring, another pair of parents is selected and so on, until the new population is complete. In the GA designed to solve the problem of classroom management, we used this variant combined with a tournament between the randomly selected pair of parents and their two generated children to select the best two individuals out of the four (in terms of fitness).

Algorithm 7 Random Selection

1: **procedure** RandomSelection(P)
2: $selected \leftarrow$ random individual from population P
3: $remove(selected,\ P)$ ▷ The individual is removed from the population.
4: **return** $selected$
5: **end procedure**

Next, we will address the topic of Fitness-Proportionate Selection, also known as *Roulette Selection*. In this variant, all individuals are dimensioned according to their fitness. If we think of this process as a lottery, the larger the size of an individual, i.e. the greater the fitness, the more likely the individual is to win the lottery prize. In this case that prize is to be selected to be a parent. This means that a random number n such that $0 \leq n \leq$ *the total sum of all fitness value*s will fall in range of one of the individuals, thus selecting such an individual.

Algorithm 8 Fitness-Proportionate Selection

procedure GenerationPreparations(P) ▷ Executed only one time at the start of each generation.
    global $\vec{p} \leftarrow \langle p_1, p_2, ..., p_l \rangle$ ▷ Vector which copies all individuals in P.
    global $\vec{f} \leftarrow \langle f_1, f_2, ..., p_l \rangle$ ▷ Vector with the fitness values of all the individuals in $\vec{p}$, keeping the order.
    if $\vec{f}$ contains only zeros **then**
        Substitute all items of $\vec{f}$ with ones
    end if
    for i from 2 to l **do** ▷ Convert $\vec{f}$ into a cumulative distribution.
        $f_i \leftarrow f_i + f_{i-1}$
    end for
end procedure
procedure FitnessProportionateSelection(P)
    $n \leftarrow$ random number from 0 to f_i inclusive
    $selected \leftarrow p_1$
    for i from 2 to l **do**
        if $f_{i-1} < n \leq f_i$ **then**
            $selected \leftarrow p_i$
        end if
    end for
    return $selected$
end procedure

We can observe that this operator uses two procedures. The first one is an auxiliary procedure named *GenerationPreparation*s, which defines the global vector variables $\vec{p}$ and $\vec{f}$ representing the individuals of the population and their fitness values. The second and main one is the actual selection. In this main procedure we obtain a random number and extract the individual whose range of values contains that chosen random number.

A derivative of the Fitness-Proportionate Selection operator mentioned at the beginning of the section is called Stochastic Universal Sampling (SUS). SUS has two very similar procedures to the ones shown before, one executed one time each generation (normally), which defines some global variables and a main procedure which performs the selection. The main difference comes in how the selection is

performed. Let s be the sum of all fitness values and l be the population size. A random number generated between 0 and s/l will select the individual in that range. For every remaining selection the position value, which started in the random number, is increased by s/l and a new selection is carried out (up to a maximum of l times).

Algorithm 9 Stochastic Universal Sampling Selection

1: **procedure** GenerationPreparations(P) ▷ Executed only one time at the start of each generation.
2: **global** $\vec{p} \leftarrow \langle p_1, p_2, ..., p_l \rangle$ ▷ Vector which copies all individuals in P.
3: $\vec{p} \leftarrow shuffle(\vec{p})$
4: **global** $\vec{f} \leftarrow \langle f_1, f_2, ..., p_l \rangle$ ▷ Vector with the fitness values of all the individuals in $\vec{p}$, keeping the order.
5: **global** $index \leftarrow 0$
6: **if** $\vec{f}$ contains only zeros **then**
7: Substitute all items of $\vec{f}$ with ones
8: **end if**
9: **for** i from 2 to l **do** ▷ Convert $\vec{f}$ into a cumulative distribution.
10: $f_i \leftarrow f_i + f_{i-1}$
11: **end for**
12: **global** $value \leftarrow$ random number from 0 to f_l/l inclusive
13: **end procedure**
14: **procedure** SUS(P)
15: **while** $f_{index} < value$ **do**
16: $index \leftarrow index + 1$
17: **end while**
18: $value \leftarrow value + f_l/l$
19: **return** P_{index}
20: **end procedure**

The main advantage of Stochastic Universal Sampling over Fitness-Proportionate selection lies in the fact that while an individual with a high fitness (higher than s/l) might never be chosen in a Fitness-Proportionate selection, in the Stochastic Universal Sampling variant its selection is guaranteed.

The last variant we will explain is the Tournament Selection. In contrast with these past two variants, the Tournament Selection is a very straightforward algorithm. In it, a t number of candidates are selected and the fittest is returned, like a sports

competition. Every time a t_i candidate is selected, it is removed from the population. The pseudocode for this variant is shown below.

Algorithm 10 Tournament Selection

```
1:  procedure TournamentSelection(P, t)
2:      best ← random individual from population P
3:      remove(best, P)
4:      for i from 2 to t do
5:          next ← random individual from population P
6:          remove(next, P)
7:          if fitness(next) > fitness(best) then
8:              best ← next
9:          end if
10:     end for
11:     return best
12: end procedure
```

This variant is both simple and flexible. Its flexibily comes from the variable size of candidates for the tournaments. Some considerations for the value of t follow. If t is very low, the operator behaves like a random search. However, if t is very high, the individual with the greatest fitness value will have a much higher likelihood of showing up and getting picked every time. As we stated when talking about Random Selection, a combination of both Random and Tournament selections were implemented in the final design of the GA used in the prototype.

To conclude, we will explain the concept of elitism in the AG. Elitism simply takes a predetermined number of individuals from the previous generation sorted by fitness and includes them directly in the next one before spawning offspring. This decreases the number of offspring created to maintain the fixed population number. A GA with elitism is therefore similar to the $(\mu + \lambda)$ ES and can terminate with premature convergence if not set up correctly.

3.3.3.2 Crossover

The crossover operator mixes the genomes of a pair of parents to produce new children. We saw how in $(\mu,\ \lambda)$ the offsprings of the μ selected parents were created by mutating copies of their parents, without ever using the crossover operator. This is

not the case for the GA, in this algorithm the mutation occurs after the genome of a child is created from the mixture of the genome of its parents. There are several variants of this operator. The following alternatives are briefly outlined in this section: *One-Point Crossover*, *Two-Point Crossover*, *Uniform Crossover* and *Order Crossover*.

The One-Point and Two-Point crossover work in a similar fashion. They choose random numbers and swap sections of the genomes of the parents to create the genomes of their offspring. We will give an example before showing the pseudocode of both variants.

We have the following chromosomes, representing the parents.

$$P_a = \{1, 5, 3, 6, 2, 7, 4\} \tag{3.17}$$

$$P_b = \{2, 5, 7, 3, 4, 6, 1\} \tag{3.18}$$

One-Point picks a value at random from 0 to 6 and the outcome is 4. The position at $4 - 1$ (the number selected marks the end of the section and is excluded from it) is the point in which the crossover between parents is produced. The operator then generates these two children.

$$C_a = \{1, 5, 3, 6, \ 4, 6, 1\} \tag{3.19}$$

$$C_b = \{2, 5, 7, 3, \ 2, 7, 4\} \tag{3.20}$$

In the case of Two-Point, two values are randomly selected, once again from 0 to 6. These values indicate the ends of the section that both parents will exchange, the first being the initial position included in the section and the second being the final position excluded from the section. The selected numbers are 2 and 4, which leads to the generation of the following offspring.

$$C_a = \{1, 5, 7, 3, 2, 7, 4\} \tag{3.21}$$

$$C_b = \{2, 5, 3, 6, 4, 6, 1\} \tag{3.22}$$

The pseudocode for these two variants is presented below.

Algorithm 11 One-Point Crossover

```
1:  procedure OnePoint(P_a, P_b, popsize)
2:      x ← random integer from 0 to popsize − 1
3:      C_a ← copy(P_a)
4:      C_b ← copy(P_b)
5:      if x ≠ 0 then
6:          for i from 0 to x − 1 do
7:              Swap values of C_ai and C_bi
8:          end for
9:      end if
10:     return C_a and C_b
11: end procedure
```

Algorithm 12 Two-Point Crossover

```
1:  procedure TwoPoint(P_a, P_b, popsize)
2:      x ← random integer from 0 to popsize − 1
3:      y ← random integer from 0 to popsize − 1
4:      C_a ← copy(P_a)
5:      C_b ← copy(P_b)
6:      if x > y then
7:          Swap x and y
8:      end if
9:      if x ≠ y then
10:         for i from x to y − 1 do
11:             Swap values of C_ai and C_bi
12:         end for
13:     end if
14:     return C_a and C_b
15: end procedure
```

The main problem with both algorithms is that it is common to break the *linkage* (or *epistasis*) between the elements in the chromosome [Luk13]. Imagine that a pair of elements of an individual produces a high fitness with certain element values. This pair is considerably separated on the chromosome, so it is most likely that it will be split when executing the parental crossover. This implies that a section of the chromosome that could give a good fitness value is broken in two or more pieces, and each piece is given to the offspring produced by the crossover, which could overall produce a worse fitness for the new individuals. In this respect, Two-Point is better than One-Point, but still faces the same problem.

To reduce linkage breaks in the chromosome we can take a look at the Uniform Crossover variant. In this algorithm, all the elements of the chromosome are iterated and, for each one, a random choice of a number from 0.0 to 1.0 is made. If the number chosen is less or equal to a previously defined probability, the parent elements at that position are swapped. The pseudocode of the Uniform Crossover is provided below.

Algorithm 13 Uniform Crossover

```
1:  procedure UniformCrossover(P_a, P_b, popsize, probSwap)
2:      C_a ← copy(P_a)
3:      C_b ← copy(P_b)
4:      for i from 0 to popsize − 1 do
5:          if probSwap ≥ random number from 0.0 to 1.0 inclusive then
6:              Swap values of C_ai and C_bi
7:          end if
8:      end for
9:      return C_a and C_b
10: end procedure
```

Finally, we will explain the operator used in the designed algorithm for the classroom management problem, the Order Crossover (OX) [Dav85]. The OX operator was first introduced in 1985 at a key Artificial Intelligence conference in Los Angeles, California [Jos85]. In this algorithm, two random numbers are picked, similarly to the Two-Point Crossover. Then, the selected section from the first parent is copied onto the offspring, keeping order and position (from point p_1 to $p_2 - 1$). It is now that OX and Two-Point differ. In Two-Point, all but the replaced section is kept the same in the offspring. Yet, in OX, the elements of the second parent which are still not present in the offspring are copied to it *in the same order as they appear*, from

the end to the start of the section (in a circular way, from point p_2 to $p_1 - 1$). We will illustrate this algorithm with some examples and then provide its pseudocode.

We will use the same parents as before. Imagine that the random numbers (from 0 to 6) result in $p_1 = 2$ (included in the section) and $p_2 = 6$ (excluded from the section). We then have the following.

$$P_a = \{1, 5, 3, 6, 2, 7, 4\} \tag{3.23}$$
$$P_b = \{2, 5, 7, 3, 4, 6, 1\} \tag{3.24}$$

Therefore, the offspring generated by this pair of parents is as described below.

$$C = \{4, 1, 3, 6, 2, 7, 5\} \tag{3.25}$$

To give another example, let's say the numbers chosen are $p_1 = 5$ and $p_2 = 2$. The result is as follows.

$$P_a = \{1, 5, 3, 6, 2, 7, 4\} \tag{3.26}$$
$$P_b = \{2, 5, 7, 3, 4, 6, 1\} \tag{3.27}$$
$$C = \{1, 5, 2, 3, 6, 7, 4\} \tag{3.28}$$

Displayed below is the pseudocode for the OX.

Algorithm 14 Order Crossover (OX)

procedure OX(F, S, len) ▷ F and S represent first and second parents, len is the individual length
  $p_1 \leftarrow$ random integer from 0 to $popsize - 1$
  $p_2 \leftarrow$ random integer from 0 to $popsize - 1$
  $C \leftarrow copy(F)$ ▷ First parent is copied to the offspring.
  $overflow \leftarrow 0$
  if $p_2 \leq p_1$ **then**
    $overflow \leftarrow len$
  end if
  $k \leftarrow p_2$ ▷ $k =$ Offspring position pointer.
  for i from 0 to $len - 1$ **do** ▷ $i =$ Second parent position pointer
    $j \leftarrow p_1$ ▷ $j =$ First parent position pointer.
    while $j < p_2 + overflow$ and $S_i \neq F_{j \mod len}$ **do**
      $j \leftarrow j + 1$ ▷ Iterate section p_1 to $p_2 - 1$
    end while
    if $j = p_2 + overflow$ **then** ▷ If S_i is not in the section
      $C_{k \mod len} \leftarrow S_i$
      $k \leftarrow k + 1$
    end if
  end for
  return C
end procedure

3.3.3.3 Mutation

We end these sections on operators by discussing the mutation operator in the GA. Mutation takes an individual and modifies its genome. This can be done in many ways. We will talk about two, *Bit-Flip Mutation* and swapping elements.

Bit-Flip mutation can only work with genomes represented by boolean values (0 and 1). This operator works in a somewhat comparable fashion to the Uniform Crossover operator. We define a flip probability and select a random number from 0.0 to 1.0. If the number is lower or equals to the flip probability, we flip the boolean value in that position of the chromosome.

Algorithm 15 Bit-Flip Mutation

```
1: procedure BitFlip(C, popsize, probSwap)
2:     for i from 0 to popsize − 1 do
3:         if probSwap ≥ random number from 0.0 to 1.0 inclusive then
4:             C_i ← invertValue(C_i)
5:         end if
6:     end for
7:     return C
8: end procedure
```

Swap mutation consists of taking a pair of values and exchanging their positions. The advantage of this variant over Bit-Flip is that Swap Mutation can be used with chromosomes consisting of non-boolean values.

Algorithm 16 Swap Mutation

```
1: procedure SwapMutation(C, popsize)
2:     x ← random number from 0 to popsize − 1
3:     y ← random number from 0 to popsize − 1
4:     while x = y do
5:         y ← random number from 0 to popsize − 1
6:     end while
7:     Swap positions of C_x and C_y
8:     return C
9: end procedure
```

3.4 Mixing it all together

In this chapter we have discussed assignment problems, greedy algorithms, heuristics, metaheuristics and evolutionary computation. We studied the Bootaku problem and gave a solution with a greedy algorithm. However, although we have mentioned several times in the document that the actual algorithm developed in this thesis contains a genetic algorithm guided by a greedy algorithm, we have not gone into the specifics of what this combination involves.

In this section we will solve the Bootaku problem with the completion times ex-

tension explained in 3.2. First of all, we need to define a chromosome representation for the Bootaku problem. Because the tasks of the problem are the book reviews, the chromosome is represented by the book codes, in this case each code consists of two initials to uniquely identify a book review.

Some possible genomes for the problem will have the following structure.

$$Individual\ A = \{DC, EQ, VN, TM\} \tag{3.29}$$

$$Individual\ B = \{VN, EQ, DC, TM\} \tag{3.30}$$

$$Individual\ C = \{TM, VN, EQ, DC\} \tag{3.31}$$

This genomes represent *the order in which the book reviews will be assigned*. As each freelancer can only write a number of book reviews that do not exceed a maximum completion time, the order in which these reviews are assigned is a very important factor to consider if we want to obtain the optimal solution. The use of different orders helps to overcome the main problem of greedy algorithms, which is the difficulty of converting local optimum to global optimum.

Next, we choose the operators which the Genetic Algorithm will use. We decide that we want a Tournament Selection, an OX for the crossover operator and a Swap Mutation. The initial generation is created by generating random individuals and verifying every time that they are unique, before adding them to the population. The function for evaluating fitness will execute the Greedy Algorithm with the assignments performed in the order specified by the chromosome. It will then calculate the total sum of the profits obtained, which will indicate the fitness of the individual. Finally, the parameters of the genetic algorithm will be calculated by experimenting with different problem instances and checking that the assignments made are of the expected quality.

In conclusion, we can say that the Bootaku problem helps us to understand, on a smaller scale, a similar (but simpler) problem that we face in this project, that of assigning classes to groups in the school. In the following sections, we will replace the Bootaku problem with the real problem, building on what we have discussed throughout this chapter.

4 Problem definition

The School of Computing Engineering of Oviedo must find a classroom for each group of a given semester. In most cases of this particular problem, just like in the Bootaku situation described in 3, there are more *task*s (groups/book reviews) than *agents* (classrooms/freelancers). And, in the same way, a valid solution implies that all groups have one (and only one) classroom assigned.

The data for the classrooms and groups can be represented by two sets C and G.

$$C = \{c_1, c_2, ..., c_n\} \tag{4.1}$$
$$G = \{g_1, g_2, ..., g_m\} \tag{4.2}$$

Where C is the set of n elements representing all the classrooms of the School, and G a set of m elements representing the groups for a given semester.

A classroom can be a laboratory or a theory classroom. Groups, on the other hand, can be taught in English or Spanish, and have three types. Laboratory, theory and seminar groups. In this problem, because we are only interested in the classrooms that can be assigned to groups, we only consider two types. Laboratory and theory. That is, we consider that the types of classrooms are *identical* to the types of groups.

$$T = \{t_1, t_2, ..., t_p\} \tag{4.3}$$
$$L = \{l_1, l_2, ..., l_q\} \tag{4.4}$$

Therefore, each classroom c and group g have a type t. In the case of groups, they are also taught in language l.

Each group has a set of academic weeks and of group schedules. A group can attend classes weekly, every two weeks or on a non-trivial pattern, and may be taught on one or several days.

$$W_i = \{w_{i1}, w_{i2}, ..., w_{ir}\} \tag{4.5}$$

$$H_i = \{h_{i1}, h_{i2}, ..., h_{is}\} \tag{4.6}$$

Therefore, every group i has a set of weeks W_i and a set of schedules H_i. A schedule consists of a triplet in the form $(DayOfTheWeek, start(hh : mm), finish(hh : mm)$.

Every group belongs to a subject.

$$S = \{s_1, s_2, ..., s_t\} \tag{4.7}$$

A subject s is related to a subset of G groups.

With these definitions, we have all the data we need in order to solve the problem. Now we shall address the problem constraints that we have to fulfill. We call *hard constraints* those which are imperative for the solution to be valid, and *soft constraints* the ones that reflect on the overall quality of the solution but are not mandatory. Before listing them, two new concepts are presented. Restrictions and preferences.

$$R_i = \{r_{i1}, r_{i2}, ..., r_{iu}\} \tag{4.8}$$

$$P_i = \{p_{i1}, p_{i2}, ..., p_{iv}\} \tag{4.9}$$

Restrictions and preferences can be positive or negative. A group i must be assigned to a classroom in the set of its positive restrictions, and cannot be assigned to a classroom in the set of its negative restrictions. It is prefered that the group i is assigned to a classroom in the set of its positive preferences, and preferably not in the set of its negative preferences. With that in mind, the constraints are defined next.

Hard constraints:

Laboratory groups can only be assigned to laboratories.

Theory and seminar groups can only be assigned to theory classrooms.

A group cannot be assigned to a classroom whose capacity is less than the number of students in the group.

A group with a set of positive restrictions must be assigned to one of those classrooms.

A group with a set of negative restrictions cannot be assigned to one of those classrooms.

A group cannot be assigned to a classroom if that classroom was already assigned to another group and both groups collide (they overlap in week and schedule).

Soft constraints:

Laboratory groups of the same subject must all attend the same laboratory classroom, and if not possible, at least minimise the number of laboratories assigned to them.

Theory groups of the same name and course work in the same way, but being assigned to theory classrooms [1].

English and Spanish groups should go to different classrooms.

Every hour a number of laboratories must be empty. To cover for emergencies.

A group with a set of positive preferences should be assigned to one of those classrooms.

A group with a set of negative preferences should not be assigned to one of those classrooms.

[1] All the groups in the School have the format *subject.type.name*. For example the group *Com.T.1* refers to *theory* group *1* of the *Computability* subject. So all theory groups named 1 would be assigned to the same theory classroom, if possible.

5 Proposed solution

We have already studied the theoretical concepts necessary to solve this type of problem, and we have formalised the problem to be solved. In this section we explain which components, techniques and algorithms we have designed and used to solve the problem of assigning classes to groups. To do so, we will define the search space, introduce the concepts of collisions and class filters, and comment on the pseudocode of the proposed algorithms.

5.1 Search space

5.1.1 Assignments

An assignment is a tuple which associates a group with a classroom.

$$(G_i, C_j) \tag{5.1}$$

Because a group can only have *one* classroom assigned, an assignment can be identified by the *code* [1] of the group. For example, the assignment for group SI.T.1 can be identified by the code SI.T.1 as well.

Assigning just *one* classroom to each group means that the total number of assignments is calculated by the following expression.

$$TotalNumberOfAssignments = |G| \tag{5.2}$$

This implies that there are as many assignments as the number of groups for the semester.

5.1.2 Solutions

A *solution* for this problem is represented by a set of all the assignments that must be performed for the semester. As presented in the previous section, the total number

[1] The name convention previously mentioned: *subject.type.name* (e.g. Com.T.1).

of assignments equals the total number of groups in that semester. So we have the next statement.

$$Solution = \{(G_1, C_x), (G_2, C_y), ..., (G_m, C_z))\} \tag{5.3}$$

Where m is the total number of groups and x, y and z are the index for the classrooms assigned to the groups. Note that the classrooms are not sequential (e.g x could represent C_{12} and y represent C_3).

An *empty solution* is represented by a set of all the assignments where each assignment is *incomplete*. We mean that an assignment is incomplete when the group has no classroom assigned.

$$IncompleteAssignment = (G_i, -) \tag{5.4}$$

So, for the empty solution, we have a set with the following format.

$$EmptySolution = \{(G_1, -), (G_2, -), ..., (G_m, -))\} \tag{5.5}$$

Finally, a *partial solution* is one in which not every assignment was performed, and a *complete solution* is defined by a set in which all the assignments have been performed and each group has a classroom associated with it.

5.1.3 States

A *state* represents a phase in the problem. There can be three states. The *initial state*, which stands for the empty solution of non allocated assignments. The *final state*, which represents a complete solution with all the assignments performed. And the *intermediate states* portraying partial solutions.

A key concept to understand our design is the following. Although by default we start the execution of the algorithms with an initial state, it is also possible to start the execution with an intermediate state. This is because we can receive as input a partial or total solution of assignments and work from there. Now we will discuss how

we can jump from one state to the next, which is normally called *state expansion*.

To expand a state, one of the non performed assignments in the solution is executed. This means that every time a classroom is assigned to a group the state is being expanded. To perform an assignment, the number of possible candidates is the same as the number of classrooms. So we have the following.

$$TotalNumberOfCandidates = |C| \tag{5.6}$$

Nevertheless, as there exist constraints that indicate whether or not the solution is valid, there are filters which reduce the number of available classrooms for a group. This allows for optimized and easy to retrieve calculations in the execution of the greedy algorithm (this will be explained later in 5.3). The important thing to note at this moment is that, because of these filters, not all states can be expanded.

5.1.4 Instances

The complexity of the calculations and completion time depend on many factors. Some of those factors follow. First, the number of groups for the semester, which directly translates into the number of assignments to be made. Second, the number of classrooms. If there are more classrooms, it is easier to avoid collisions. Obviously, the number of groups is much more volatile between semesters than the number of classes, which is likely to change very occasionally, if at all. Lastly, the case of starting the prototype with an intermediate state. This means that the set of assignments represents a partial solution given as input, and since the number of calculations decreases in direct proportion to the assignments already made, the completion time would be lower.

Because of the constraints of the problem, there are some groups where class allocations are more straightforward. For example, a group with only one positive restriction is either going to have that classroom assigned to it or, if it collides with other group, end up unallocated. This is why all the groups that just have one available classroom are assigned first. Also, the groups which have less students have more available classrooms than those with a large number of members.

5.2 Collisions

A collision is an overlap of the timetable of two different groups. For a collision to occur, the groups must clash at least once in the same week, day and time. Collisions are an essential part of this problem, as we cannot assign a classroom to a group if another group was previously associated with that classroom and both groups collide.

5.2.1 Lazy Collision Matrix

Due to the large number of assignments that have to be made throughout the execution cycle of the genetic algorithm, the chosen data structures were properly analysed. This is where the *Lazy Collision Matrix* comes in.

Imagine that we have the following group set.

$$G = \{G_1, G_2, G_3\} \tag{5.7}$$

Then, our initial Lazy Collision Matrix would be represented by the expression below.

$$LCM = \begin{array}{c} \\ G_1 \\ G_2 \\ G_3 \end{array} \begin{array}{c} \begin{array}{ccc} G_1 & G_2 & G_3 \end{array} \\ \begin{pmatrix} & -1 & -1 \\ -1 & & -1 \\ -1 & -1 & \end{pmatrix} \end{array} \tag{5.8}$$

First of all, the diagonal is empty because we never compare one group against itself. Then, we can observe that the rest of values are defaulted to -1. Why? Because there are *not yet evaluated*. That is the reason behind the name of the matrix. It is *lazy* because the collisions are only calculated when needed.

Continuing with this example, imagine that we assign classroom C_x to group G_1. Then, G_2 also tries to have C_x assigned to it, so we check if both groups collide. We find out that they do, so we update the matrix.

$$LCM = \begin{array}{c} \\ G_1 \\ G_2 \\ G_3 \end{array} \begin{array}{c} \begin{array}{ccc} G_1 & G_2 & G_3 \end{array} \\ \begin{pmatrix} & 1 & -1 \\ 1 & & -1 \\ -1 & -1 & \end{pmatrix} \end{array} \tag{5.9}$$

Therefore, the values are updated with a 1, which indicates that both groups *collide*. This results in a different classroom C_y being assigned to G_2. Now, G_3 has that classroom also available, so we check if it clashes with G_2. We learn that they do not collide, so we update the matrix again.

$$LCM = \begin{array}{c} \\ G_1 \\ G_2 \\ G_3 \end{array} \begin{array}{c} \begin{array}{ccc} G_1 & G_2 & G_3 \end{array} \\ \begin{pmatrix} & 1 & -1 \\ 1 & & 0 \\ -1 & 0 & \end{pmatrix} \end{array} \tag{5.10}$$

The value for non-collision is 0, as observed. Because G_3 does not clash with G_2, they are both allocated in the same classroom.

We can now generalise the LCM as in the next expression.

$$LCM = \begin{array}{c} \\ G_1 \\ G_2 \\ G_3 \end{array} \begin{array}{c} \begin{array}{ccc} G_1 & G_2 & G_3 \end{array} \\ \begin{pmatrix} & g_{12} & g_{13} \\ g_{21} & & g_{23} \\ g_{31} & g_{32} & \end{pmatrix} \end{array} \tag{5.11}$$

Where a value g_{ij} can be

$$\begin{cases} 1, & \text{if } G_i \text{ collides with } G_j \\ 0, & \text{if } G_i \text{ does not collide with } G_j \\ -1, & \text{if the collision has not yet been evaluated} \\ \epsilon, & \text{otherwise} \end{cases}$$

The main advantage of this design is that we do not have to calculate all collisions.

For example, a collision between a laboratory group and a theory group would be pointless to calculate because they would never be allocated in the same classroom. Therefore we alleviate the number of calculations.

As there is only one Lazy Collision Matrix, all calculations performed by the greedy algorithm across all populations in all generations are stored in just one place. This means that all collisions are being calculated only when necessary and only once. Think of the previous example. In a future iteration the greedy algorithm wants to check if groups G_1 and G_2 collide. It access the corresponding location in the LCM, and because it contains a 1, the greedy concludes that they indeed collide. This is done with a $O(1)$ complexity, as the matrix is coded as a dictionary of dictionaries. If instead the greedy wanted to check if groups G_1 and G_3 collide, because the LCM has a -1 in that position, the greedy would have to perform the collision check and then update the matrix.

5.3 Classroom filters

A classroom filter is a function. It receives as input either the C set or a subset $I \subset C$, and outputs a new subset $F \subset C$ with the available classrooms for a given group. It filters out the classrooms that do not comply with the hard constraints for that particular group (except collisions, which are calculated later with the LCM, refer to 5.2.1). All groups have the *same* classroom filters.

For example, let us consider a group G_i with type T_j and x students. A type filter for G_i would remove from the result set all the classrooms with a type different from T_j. A capacity filter would then take the set resulting from the previous type filter and use it as input. Then it would eliminate from the set all classrooms with a capacity lower than x and return the new F set.

This reduces the number of classrooms that the greedy has to evaluate and therefore decreases the complexity of the calculations. Furthermore, the filters are deterministic, that is, the execution of the filters of a group will always give the same results. This means that, if needed, the filters are only performed once per group per execution.

This may lead us to this question. *If all classroom filters are executed only once*

per group every time you run the prototype, why bother using a lazy approach for storing them? Would it not be better to perform and store them in a dictionary at the start of the execution? The answer is no. It is true that if we start from an empty solution the lazy approach does not present a big advantage. Nonetheless, in the case of partial solutions, it reduces the number of calculations. For example, if we want to assign a classroom to a new group created in the middle of the semester, the only thing we care about is if the group collides with any other group. The filters for the rest of the groups are, in that situation, irrelevant. This is due to the fact that they have already been allocated to their corresponding classrooms.

5.3.1 Lazy Filter Dictionary

The filters work in a similar way to the Lazy Collision Matrix. Again, the election of the data structures is crucial to optimise the execution times. That is why the classroom filters are coded using a dictionary of sets. Once more, we will explain this with an example.

We have a M set of n classroom filters, three groups G_1, G_2 and G_3, and two classrooms C_1 and C_2.

$$M = \{M_1, M_2, ..., M_n\} \tag{5.12}$$

$$G = \{G_1, G_2, G_3\} \tag{5.13}$$

$$C = \{C_1, C_2\} \tag{5.14}$$

Then, we define the dictionary as a function $Dict : Keys \rightarrow Values \cup \{\epsilon\}$, where ϵ is the $null$ character. That is, $\epsilon \notin Values$.

The keys are represented by the groups, so $Keys \equiv G$. The values depict the different sets of available classrooms for each group. Because the dictionary is as lazy as the LCM, the calculations are performed as needed, so the initial set of values are by default ϵ. Then we can say that $Values = \{\epsilon, \epsilon, \epsilon\}$.

Accordingly, we have the following cases.

$$Dict(x) = \begin{cases} \epsilon, & \text{if} x = G_1 \\ \epsilon, & \text{if} x = G_2 \\ \epsilon, & \text{if} x = G_3 \\ \epsilon, & \text{otherwise} \end{cases}$$

We are now in the first execution of the greedy algorithm. Because we start from an empty state and not from an intermediate step, the greedy will try to assign a classroom for all groups. As a result of the values in the dictionary being $null$, the greedy knows that it must execute the filters, updating the dictionary. This is the LFD after the first execution for this case.

$$Dict(x) = \begin{cases} F_1, & \text{if} x = G_1 \\ F_2, & \text{if} x = G_2 \\ F_3, & \text{if} x = G_3 \\ \epsilon, & \text{otherwise} \end{cases}$$

With each $F_i \subset C$ being the filtered classrooms for each group. Example of values for the F sets follow.

$$F_1 = \{C_1\} \tag{5.15}$$
$$F_2 = \{C_1, C_2\} \tag{5.16}$$
$$F_3 = \{\} \tag{5.17}$$

We can observe that the for G_1, classroom C_2 was filtered out. In the case of G_2, both classrooms comply with all constraints. And for G_3, there are no classes available. This is very important, because it implies that with these filters, G_3 will *always* end up without a classroom. Therefore, a complete solution cannot be found for this case unless the filters are changed. Lastly, a final remark. We can say that from now on, until the prototype terminates, the greedy will not have to perform the filters for any of the groups again, as the results are already stored in the dictionary.

5.4 Greedy algorithm

The greedy algorithm (see 3.2 for details) performs the assignments taking care not to infringe any hard constraint. To meet this objective, the LFD and the LCM are used. Its pseudocode is as follows.

Algorithm 17 ClassManager Greedy Algorithm

```
1:  procedure GreedyAlgorithm(assignments)
2:      solution ← copy(assignments)
3:      repairs ← {}
4:      solution ← preprocess(solution)
5:      for i from 0 to length(solution) − 1 do
6:          if !isAssigned(A_i) then
7:              c ← bestClassroomFor(A_i)
8:              if c ≠ ϵ then
9:                  assignClassroomToGroup(c, A_i, solution)
10:                 assignClassroomToSameGroups(c, A_i, solution)
11:             else
12:                 addToRepairs(A_i, repairs)
13:             end if
14:         end if
15:     end for
16:     solution ← repair(repairs, solution)
17:     return solution
18: end procedure
```

As can be noted, the algorithm receives a set of empty or partial assignments as an input parameter and returns another set with the assignments made. It iterates over all assignments and calculates those that are not already completed. It does this by obtaining the best class for an assignment, and if found, it assigns it not only to that group, but to all groups to which that group is related. So what does it mean if one group is related to another? It's simple, if the group is a lab group, the classroom is assigned to that group and to all lab groups belonging to its subject. If, on the other hand, the group is a theory group, the classroom found is assigned to that group and to all the theory groups in its course that share the same name.

If no class is found for a group, the group is added to the repair set. Groups

belonging to that set are attempted to be fixed at the end of the procedure, before returning the set with the solution.

5.4.1 Preprocessing

The greedy algorithm preprocesses the input set with the assignments, working on those that are not yet complete. The first thing it looks at is whether the set of filtered classes for a given group contains only one element, i.e. whether only one class can be assigned to that group. If this is the case, an attempt is made to make the assignment directly, checking that there are no conflicts.

Algorithm 18 ClassManager Greedy Algorithm Preprocessing

```
1:  procedure Preprocess(solution)
2:      for i from 0 to length(solution) − 1 do
3:          if !isAssigned(A_i) then
4:              filtered ← filterClassroomsFor(group(A_i))
5:              if length(filtered) = 1 then
6:                  c ← bestClassroomFor(A_i)
7:                  if c ≠ ϵ then
8:                      assignClassroomToGroup(c, A_i, solution)
9:                  end if
10:             end if
11:         end if
12:     end for
13:     return solution
14: end procedure
```

We can observe that in this procedure the class found is not assigned to the groups related to the one we are evaluating, as it is done in the greedy algorithm itself later on. The reason for this is that additional assignments may cause another group with only one filtered class to remain unassigned, so priority is given to such groups.

5.4.2 Heuristic

The heuristic used by the greedy algorithm is very simple. Classes are filtered by type, capacity and constraints (positive and negative). Once filtered, they are iterated until

one is found that does not cause a collision between the evaluated group and the rest of the groups previously assigned to that classroom.

Algorithm 19 ClassManager Greedy Algorithm Assignment Heuristic

```
1:  procedure BestClassroomFor(A_i)
2:      selected ← ϵ
3:      filtered ← filterClassroomsFor(group(A_i))
4:      for j from 0 to length(filtered) − 1 do
5:          selected ← F_j                          ▷ Filtered classroom j
6:          if !collisionExistsFor(group(A_i), selected) then
7:              break out of the filtered classrooms for loop
8:          end if
9:          selected ← ϵ
10:     end for
11:     return selected
12: end procedure
```

5.4.3 Repairs

To conclude with the greedy algorithm components, we must talk about the assignment repair process, which is probably the most complex of all components. As previously explained, in the case of not finding an appropriate class for a group, such a group is put into the repair set. The repair process iterates each group in this set and tries to fix its assignment. For this purpose, it obtains the filtered classes of the group and, for each class, it obtains the groups that are in conflict with the evaluated group. If there is only one conflict for a class, the assignment of the conflicting group is removed and the class is assigned to the group under repair. Then, an attempt is made to obtain a new class for the conflicting group. If successful, both end up with an assigned class and the repair succeeds. If unsuccessful, the previous class is reassigned to the conflicting group and the process is repeated for each conflicting group in each class. If, after the execution of this process, the repair is unsuccessful, the group is finally unassigned.

The pseudocode for the repairing procedure follows.

Algorithm 20 ClassManager Greedy Algorithm Repairing Process

```
1: procedure Repair(repairs, solution)
2:     for i from 0 to length(repairs) − 1 do
3:         filtered ← filterClassroomsFor(group(A_i))
4:         for j from 0 to length(filtered) − 1 do
5:             collisions ← collisionsFor(group(A_i), F_j)
6:             if length(collisions) > 1 then
7:                 break out of the filtered classrooms for loop   ▷ Assignment could not be repaired.
8:             end if
9:             group ← firstElement(collisions)
10:            a ← assignmentFor(group)
11:            removeClassroomFromGroup(F_j, a, solution)
12:            assignClassroomToGroup(F_j, A_i, solution)
13:            c ← bestClassroomFor(a)
14:            if c ≠ ϵ then
15:                assignClassroomToGroup(c, a, solution)   ▷ Assignment repaired.
16:            else
17:                removeClassroomFromGroup(F_j, A_i, solution)   ▷ Assignment could not be repaired.
18:                assignClassroomToGroup(F_j, a, solution)
19:            end if
20:        end for
21:    end for
22:    return solution
23: end procedure
```

5.5 Genetic Algorithm

The Genetic Algorithm (GA) (see 3.3.3 for details) generates sets of assignments, empty or partial, with different ordering. This sets are later piped into the greedy algorithm, which performs and returns the final assignments to the GA. The GA then evaluates the solution by calculating its fitness value. This is done until a predetermined time or number of generations is reached, returning the best order of assignments, that is, the best individual found.

Algorithm 21 ClassManager Genetic Algorithm (GA)

```
1:  procedure GeneticAlgorithm(popsize, numgen, maxTime, crossProb, mutProb)
2:      best ← ϵ
3:      currentTime ← get current time
4:      gen ← 0
5:      P ← {}
6:      for popsize times do
7:          P ← P ∪ {new random individual}
8:      end for
9:      repeat
10:         evaluate(P)                    ▷ Calculate the fitness of all individuals.
11:         for each individual P_i ∈ P do
12:             if best = ϵ or Fitness(P_i) > Fitness(best) then
13:                 best ← P_i
14:             end if
15:         end for
16:         Q ← nextGeneration(P, popsize, crossProb, mutProb)
17:         P ← Q
18:         currentTime ← update time
19:         gen ← gen + 1
20:     until gen ≥ numgen or currentTime ≥ maxTime
21:     return best
22: end procedure
```

This pseudocode is very similar to the general GA, except for two things. Here, instead of having a function that checks whether or not the individual represents a valid solution, our GA simply iterates through a number of generations, stopping when it reaches that number or when time runs out. The other thing in which they differ is that the operations carried out for creating the next generation are performed in a different procedure. This procedure selects the parents, checks whether or not the offspring should result from a crossover of its parents [2] and also if the offspring is mutated [3]. A tournament is then performed between the two pairs of parents and children and the two best, in terms of fitness, are added to the next generation. The

[2] This check is performed by calculating a random number that is compared with the crossover probability. If the selected number is lower or equal to the probability, the crossover is executed.

[3] Same check as for the crossover operator but comparing the number with the mutation probability.

pseudocode for this procedure follows.

Algorithm 22 ClassManager GA Next Generation

1: **procedure** NextGeneration(P, popsize, crossProb, mutProb)
2: $Q \leftarrow \{\}$
3: **for** $popsize/2$ times **do**
4: Parent $P_a \leftarrow selectAndRemove(P)$
5: Parent $P_b \leftarrow selectAndRemove(P)$
6: Child $C_a \leftarrow copy(P_a)$
7: Child $C_b \leftarrow copy(P_b)$
8: **if** $crossProb \geq$ random number from 0.0 to 1.0 inclusive **then**
9: $C_a \leftarrow crossover(copy(P_a),\ copy(P_b))$
10: $C_b \leftarrow crossover(copy(P_b),\ copy(P_a))$
11: **end if**
12: **if** $mutProb \geq$ random number from 0.0 to 1.0 inclusive **then**
13: $C_a \leftarrow mutation(copy(C_a))$
14: $C_b \leftarrow mutation(copy(C_b))$
15: **end if**
16: Winners $W_a, W_b \leftarrow tournament(P_a,\ P_b,\ C_a,\ C_b)$
17: $Q \leftarrow Q \cup \{W_a, W_b\}$
18: **end for**
19: **return** Q
20: **end procedure**

5.5.1 Genome representation

The individuals of the GA are represented by a vector chromosome of group codes. Each code uniquely identifies a group, therefore an assignment, and the genome tells the greedy algorithm in which order it must perform that assignment. The rationale behind this random ordering is clear if we think back on what we discussed about greedy algorithms. Their main flaw relies on not always finding the correct solution because they focus on local optimum instead of global ones. The GA helps the greedy by generating a bunch of different orderings and therefore testing which order gets closer to the optimal solution.

A vector chromosome looks like this. Let's say that I represents an individual, represented by a set of group codes in a random order. Codes identify assignments,

so they are indicated by A_i.

$$I = \{A_1, A_2, ..., A_m\} \tag{5.18}$$

Where m is the number of groups for that semester.

To convert the individual representation into an actual set of assignments in the specified order, we have designed a *decoder*. The decoder has a dictionary given by the function $Dict : Keys \rightarrow Values \cup \{\epsilon\}$. The group codes work as keys, and the assignments related to each group are the corresponding values. When an individual is decoded, the decoder simply takes each group code, obtains the value associated to them by looking in the dictionary, and then returns a set with the order given by the representation.

5.5.2 Fitness function

The fitness function of the GA has the following responsibilities. First, it passes the individual representation to the decoder in order to obtain the corresponding set of assignments. Then, the function gives the assignments to the greedy algorithm and receives the solution set. Finally, it returns the value resulting from applying its formula to the solution set.

The formula of the fitness function is given by the sum of all fitness values multiplied by their corresponding fitness weights. Each fitness value represents a soft constraint described in the problem definition (see 4), and has a weight associated with it.

The formula for the fitness function is therefore given by:

$$formula = \sum_{i=1}^{n} V_i W_i \tag{5.19}$$

Where n is the number of fitness values defined, V_i is fitness value i and W_i the weight associated with V_i.

5.5.3 Operators

We will briefly talk about the selected operators for the ClassManager GA (see 3.3.3 for a more in-depth explanation of all the operators discussed here). The order and use of these operators can be clearly illustrated with the help of the pseudocode of procedure $NextGeneration$ (see 22).

5.5.3.1 Selection

The GA uses Random Selection. This operator selects and removes a random individual from the population. Since at the end of the offspring creation cycle there is a tournament between parents and offspring, a random selection of individuals is perfectly valid in this case. The pseudocode of this operator is displayed in 7

5.5.3.2 Crossover

As we explained when we introduced this operator, the Order Crossover (OX) is the crossover operator for the ClassManager GA. As in our case it only returns one child per pair of parents, OX is called two times, with the positions of the parents inverted (check the $NextGeneration$ pseudocode to see what I mean), that is, parent a acts as the first parent in the first run and as the second parent in the second run (and vice versa for parent b). See 14 for its pseudocode.

It is also important to note that the offspring are only created from a crossover if a random number between 0.0 and 1.0 is lower or equal to the predetermined crossover probability.

5.5.3.3 Mutation

For the mutation operator the GA uses Swap Mutation. This operator selects two group codes at random and swaps their positions. As in the crossover operator, the mutation only occurs if a randomly selected value is lower or equal to the mutation probability. The pseudocode for Swap Mutation is indicated in 16.

5.5.3.4 Tournament

Finally, we introduce the specific version of Tournament Selection implemented for the ClassManager GA. It differs from the generic Tournament Selection (see 10) in

that the candidates for the tournament are not a t number of randomly selected individuals, but instead the two pairs of parents and children generated after executing the rest of operators. These four individuals are compared by their fitness and the best two survive, joining the new generation. Below we indicate the pseudocode for this version of Tournament Selection.

Algorithm 23 ClassManager GA Tournament Selection

```
1: procedure TournamentSelection(A, B, C, D)
2:     I ← {}
3:     I ← I ∪ {A, B, C, D}
4:     sort I by fitness, from highest to lowest
5:     return I_1 and I_2
6: end procedure
```

5.5.4 Parameters

TODO: AFTER EXPERIMENTS

6 Project planning and budget overview

6.1 Planning

6.2 Budget summary

7 Analysis

7.1 System definition

The prototype defined in this document consists of a command line application which receives a list of text files as input and, by processing them, can perform three operations. The first and main one is the calculation of all the assignments required for the semester, starting from scratch or by using a previous list of assignments generated by the system itself. The second operation consists of finding holes in the schedule, that is, to find free classrooms which comply with a set of constraints. This is useful for assigning a class to a specific event even if the exact time or day of such an event is not known and can only be guessed. The last operation is the automation of the creation of the input files, which is done by converting some other files previously used by the School into new files which the system is able to process.

7.2 System requirements

The functional and non-functional requirements of the system. The non-functional requirements include the technological, system manuals and response time requirements.

7.2.1 Functional requirements

RCLI The system must implement a command line interface (CLI).

- **RCLI1** The CLI must show information about the current process being executed.
- **RCLI2** The CLI must show information about the GA.
  - **RCLI2.1** About the parameters of the GA.
  - **RCLI2.2** About the generations of the GA.
  - **RCLI2.3** About the execution time of the GA.
- **RCLI3** The CLI must show information about the result of the execution.
  - **RCLI3.1** If the system terminated successfully.
  - **RCLI3.2** If the system terminated with errors, they will be also notified to the user.

RInput1 The system must receive the required data as input.

- **RInput1.1** The system must receive as input the classrooms of the School.
- **RInput1.2** The system must receive as input the groups of the semester.
- **RInput1.3** The system must receive as input the schedule of the groups.
- **RInput1.4** The system must receive as input the weeks in which the groups have classes.
- **RInput1.5** The system must receive as input the subjects of the semester.
- **RInput1.6** The system must receive as input the queries with the constraints for finding free classrooms in the schedule.
- **RInput1.7** The system must receive as input the previously used files for the automation of the system input files.

RInput2 The system might receive optional data as input.

- **RInput2.1** The system might receive as input a total or partial list of assignments.
- **RInput2.2** The system might receive as input a list of classroom preferences.
- **RInput2.3** The system might receive as input a list of classroom restrictions.

RInput3 The system must receive the required configuration files as input.

- **RInput3.1** The system must receive as input a configuration file with the parameters of the GA.
- **RInput3.2** The system must receive as input a configuration file with the path to the input and output files.

RConf The system must be configured by plain text files.

- **RConf1** System configuration must allow the user to control the parameters of the GA.
- **RConf2** System configuration must allow the user to specify the paths of the input files.
- **RConf3** System configuration must allow the user to specify the folder paths for the output files.

RAssign The system must perform the assignments by using AI algorithms.

RAssign1 The assignments should prioritise that Spanish and English groups go to different classes.

RAssign2 The assignments may start from an initial set of assignments which must remain the same.

RAssign3 The assignments must maximize the number of groups of the same name and course assigned to the same theory classroom.

RAssign4 The assignments must maximize the number of groups of the same subject assigned to the same laboratory.

RAssign5 The assignments must leave a number of free laboratories in each time slot.

RAssign6 The assignments should prioritise that a laboratory does not end up with a large number of unused computers.

RClassFinder The system must be able to find free classrooms for an event given some constraints.

RClassFinder1 The constraints must include the date range for the search.

RClassFinder2 The constraints must include the range of hours for the search.

RClassFinder3 The constraints must include the duration of the event (in hours) for the search.

RClassFinder4 The constraints must include the number of attendants to the event.

RClassFinder5 The constraints must include the type of classroom to hold the event in.

RClassFinder6 The constraints must include the maximum number of results to obtain.

RAutomation The system must be able to automate the creation of the input files.

RAutomation1 The system must receive the planning for the semester.

RAutomation2 The system must receive the number of enrolled students for each group.

RLog The system must keep a log of its operations.

RLog1 The log must indicate the date and time of every record.

RLog2 The log must indicate the log level of every record.

RLog3 The log must record the complete information of encountered errors.

RLog4 The log must record basic information of the flow of the application.

7.2.2 Non-Functional requirements

RTech The system requires a specific setup to be executed.

RTech1 The system requires Java 8 to be installed in the computer which executes it.

RTech2 The system requires that the folders where the input files are located have sufficient permissions for the system to be able to read these files.

RMan The system manuals must provide the readers with appropriate information for carrying out their tasks.

RMan1 The installation manual must explain the setup needed before the execution of the program.

RMan1.1 It is aimed both at the user and the developers of the application.

RMan2 The execution manual must explain the syntax for executing each functionality of the system.

RMan2.1 It is aimed both at the user and the developers of the application.

RMan3 The user manual must explain all the functionality of the system.

RMan3.1 It is aimed at the user of the application.

RMan3.2 It must provide examples of usage with step by step instructions.

RMan4 The programmer manual must briefly explain the structure of the code and its components.

RMan4.1 It is aimed at the developers and maintainers of the application.

RMan4.2 It must provide examples of possible changes with some directions to implement them.

RMan4.3 It must provide an explanation on how to interpret the log of the application.

RResponse The system must perform the assignments of a semester in less than a day.

RResponse1 The assignments must have the expected quality. An assignment is said to be of quality if it meets all hard and as many soft constraints as possible.

7.3 Subsystem mapping

7.4 Preliminary class diagram

7.5 Analysis of use cases

7.6 Analysis of user interfaces

7.7 Test plan specification

8 System design

8.1 System architecture

8.2 Class design

8.3 Interaction and state diagrams

8.4 Activity diagram

8.5 Interface design

8.6 Technical specification of the test plan

9 System implementation

9.1 Standards and references

9.1.1 Standards

9.1.2 Licenses

The software of this project is licensed under the GNU General Public License v2.0.

9.1.3 Other references

Java Code Conventions. Set of guidelines and conventions for programmers to consider when using the Java programming language.

9.2 Programming languages

There were two programming languages considered for the implementation of the system, **C** and **Java**.

Considering that:

- The author and only developer of the system has worked with Java throughout his university studies, but only used C in one subject and in some of his personal projects.
- Java is probably less efficient than C when executing the genetic and greedy algorithms.
- Java code is more easy to run in other systems than C code.
- The program is going to be executed only a few times a year.

For this reasons, even if C would be faster in execution, because the program will not be running every day, and taking into account the other two advantages, Java was the language of choice for implementing the system.

9.3 Tools and programs used in development

9.4 System development

10 Test development

10.1 Unit tests

10.2 Integration and system tests

10.3 Usability and accessibility tests

10.4 Performance tests

11 Experimental results

12 System manuals

12.1 Installation manual

12.2 Execution manual

12.3 User manual

12.4 Programmer manual

13 Conclusions and future work

13.1 Final conclusions

13.2 Future work

14 Budget

14.1 Internal budget

14.2 Client budget

15 Annexes

15.1 Definitions and abbreviations

Listed below is a glossary of definitions and abbreviations used in the document whose meaning may not be obvious.

Glossary of definitions:

- **Evolutionary Computation:** method of designing a metaheuristic algorithm. It is a subtype of Population-based methods. The definitions for the common components of evolutionary computation follow [Luk13].
  - **Breeding:** the act of creating one or more children from a population of parents by combining the crossover and mutation operators.
  - **Chromosome:** a specific type of genome consisting of a fixed-length array.
  - **Child and parent:** both are individuals. A child being a possible modification of its parent.
  - **Crossover:** operator that creates childs from parents by means of combining sections of the genomes of the parents.
  - **Evaluation:** calculating the fitness of an individual.
  - **Fitness:** quality of an individual.
  - **Generation:** the population of a given iteration of the algorithm. The next generation is created by means of the different operations defined by said algorithm.
  - **Genome:** the data structure that defines an individual.
  - **Individual:** candidate solution for the problem.
  - **Mutation:** operator that modifies the genome of an individual.
  - **Population:** set of individuals.
  - **Selection:** operator that elects individuals from the population based on some criteria.
- **Genetic algorithm:** metaheuristic search and optimization algorithm.

- **Greedy algorithm:** algorithm that builds the solution in successive steps, always trying to take the optimal solution for each step
- **Heuristic:** function that gives value to each path from a intermediate state to the goal state. Applied in search algorithms, heuristics are based on knowledge outside the problem definition.
- **Java:** general-purpose, high-level, object-oriented programming language.
- **Metaheuristic:** algorithm that uses randomness to find a possible optimal solution to a hard problem. They are part of the stochastic optimization field.

Glossary of abbreviations:

- **CSV:** Comma-Separated Values. Refers to a text file format.
- **CLI:** Command Line Interface.
- **EA:** Evolutionary Algorithm.
- **EC:** Evolutionary Computation.
- **ES:** Evolution Strategies.
- **GA:** Genetic Algorithm.
- **LCM:** Lazy Collision Matrix (see 5.2.1).
- **LFD:** Lazy Filter Dictionary (see 5.3.1).
- **TXT:** Text. Refers to the text file format.

15.2 Submission contents

16 Source code

Bibliography

[Dav85] Lawrence Davis. Applying adaptative algorithms to epistatic domains. In Joshi [Jos85], pages 162–164.

[dlC18] Gonzalo de la Cruz. Metaheuristics for the assignment of students to class groups. End-of-degree thesis, School of Computing Engineering of Oviedo, 2018.

[GV98] Rosa Guerequeta and Antonio Vallecillo. *Técnicas de Diseño de Algoritmos*. Servicio de publicaciones de la Universidad de Málaga, 1998.

[Jos85] Aravind K. Joshi, editor. *Proceedings of the 9th International Joint Conference on Artificial Intelligence. Los Angeles, CA, USA, August 1985*. Morgan Kaufmann, 1985.

[Luk13] Sean Luke. *Essentials of Metaheuristics*. Lulu, second edition, 2013. Available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/.

[Red] Jose Manuel Redondo. *Documentos-modelo para Trabajos de Fin de Grado/Master de la Escuela de Informática de Oviedo*. Escuela Universitaria de Ingeniería Técnica en Informática de Oviedo, 1.4th edition.

[RN10] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, third edition, 2010.