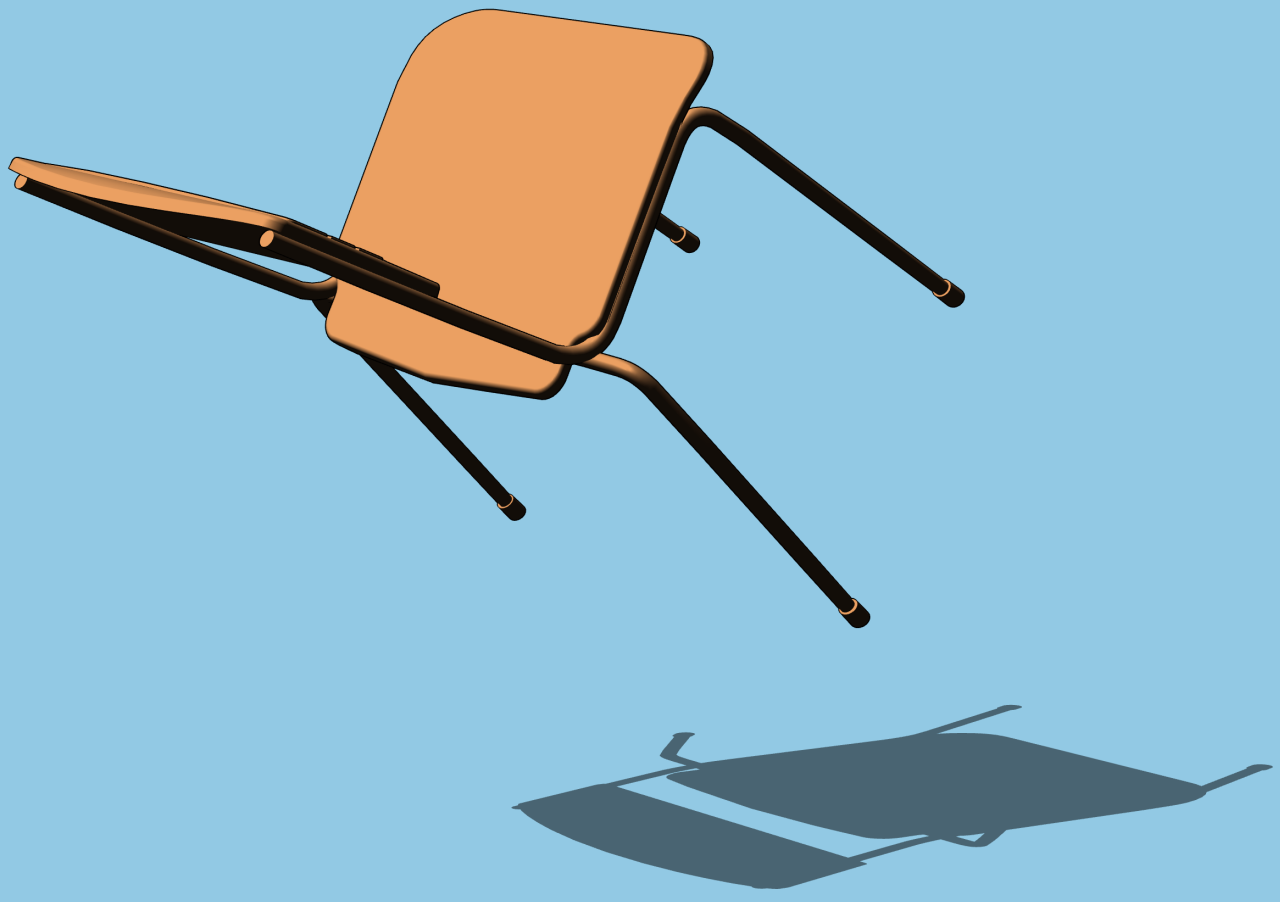
Classroom management at the School of Computer Engineering using Artificial Intelligence methods



Author: Hugo Fonseca Díaz
Supervisors:
Raúl Mencía Cascallana
and Carlos Mencía Cascallana

Universidad de Oviedo

Escuela de Ingeniería Informática
Universidad de Oviedo

Word of thanks

First of all, I would like to thank all the people involved in this project. To Raúl and Carlos, for their contributions and understanding. To Gonzalo de la Cruz, for his previous work, which has made the realisation of this project possible. To the clients/users for their feedback.

Secondly, I would like to thank my friends and colleagues at the School of Computer Engineering of Oviedo, for all these years of fellowship and mutual help. Specially I want to name Kike, Ángel, Edu, Garnacho, Íñigo, Cuesta, Migi and Fidalgo.

Finally, I would like to thank my parents for all the support they have given me over the years, and my brother Neizan, who has contributed to this project with the front and back cover.

Abstract

The main goal of this project is to analyse, formalise and solve the problem of assigning classrooms to groups at the School of Computer Engineering of the University of Oviedo. This is a challenging problem that arises every year in the School and is currently solved manually.

The problem is formalised as an optimisation problem based on a series of meetings with the school management. A greedy algorithm, guided by specific heuristics developed in the project, and a genetic algorithm, which conducts a global search to compute high quality solutions, are proposed to solve the problem.

In addition to providing a theoretical solution, the project builds a software prototype that implements the designed algorithms and on which experiments are carried out to obtain its optimal configuration.

The tool proves to be very effective at solving the problem.

Index

0 Overview **11**

1 Introduction **13**
1.1 Situation overview 13
1.2 Purpose 14
1.3 Scope 15
1.4 Project goals 15

2 Theoretical aspects **17**
2.1 Assignment problem 17
2.2 Greedy algorithms 21
2.3 Heuristics and metaheuristics 25
2.3.1 Evolutionary Computation 26
2.3.2 Evolution Strategies 27
2.3.3 Genetic algorithms 29
2.3.3.1 Selection 30
2.3.3.2 Crossover 34
2.3.3.3 Mutation 39
2.4 Mixing it all together 40

3 Problem definition **42**

4 Proposed solution **45**
4.1 Search space 45
4.1.1 Assignments 45
4.1.2 Solutions 45
4.1.3 States 46
4.1.4 Instances 47
4.2 Collisions 48
4.2.1 Lazy Collision Matrix 48
4.3 Classroom filters 50
4.3.1 Lazy Filter Dictionary 51
4.4 Greedy algorithm 53
4.4.1 Preprocessing 54
4.4.2 Heuristic 55
4.4.3 Repairs 55
4.5 Genetic Algorithm 57
4.5.1 Genome representation 59
4.5.2 Fitness function 60
4.5.3 Operators 61
4.5.3.1 Selection 61
4.5.3.2 Crossover 61
4.5.3.3 Mutation 61

4.5.3.4 Tournament 62

5 Project planning and budget overview **63**
5.1 Planning 63
5.1.1 Project Management 64
5.1.2 Analysis 65
5.1.3 Design 66
5.1.4 Development 66
5.1.5 Documentation 67
5.1.6 Experiments 68
5.1.7 Closure 69
5.2 Budget summary 69

6 Analysis **71**
6.1 System definition 71
6.2 System requirements 71
6.2.1 Functional requirements 71
6.2.2 Non-Functional requirements 74
6.3 Subsystem mapping 75
6.4 Preliminary class diagram 76
6.5 Analysis of use cases 77
6.5.1 Perform the assignments 78
6.5.2 Search for free classrooms 80
6.5.3 Automatically create input files 82
6.6 Analysis of user interfaces 83
6.7 Test plan specification 84

7 System design **87**
7.1 System architecture 87
7.2 Class design 89
7.3 File format design 95

8 System implementation **97**
8.1 License and references 97
8.1.1 License 97
8.1.2 References 97
8.2 Programming languages 97
8.3 Tools and programs used in development 98
8.3.1 Git 98
8.3.2 Vim 98
8.3.3 LaTeX 98
8.3.4 Eclipse 98
8.3.5 PlantUML 98

9 Test development **99**
9.1 Perform the assignments . . . 99
9.2 Search for free classrooms . . . 100
9.3 Automatically create input files . . . 102

10 Experimental results **103**
10.0.1 Instances . . . 103
10.0.2 Fitness function . . . 104
10.0.3 Greedy Algorithm . . . 104
10.0.3.1 Fitness assesment . . . 105
10.0.3.2 Other metrics . . . 106
10.0.4 (Genetic + Greedy) parameter assesment . . . 107
10.0.5 Greedy against (Genetic + Greedy) . . . 108
10.0.6 Further fitness functions . . . 110
10.0.7 Experimental findings . . . 110

11 System manuals **112**
11.1 Installation manual . . . 112
11.2 Execution manual . . . 113
11.3 User manual . . . 117
11.4 Programmer manual . . . 118

12 Budget **121**
12.1 Internal budget . . . 121
12.2 Client budget . . . 123

13 Conclusions and future work **125**
13.1 Final conclusions . . . 125
13.2 Future work . . . 125

14 Annexes **127**
14.1 Definitions and abbreviations . . . 127
14.2 Work Breakdown Structure (WBS) . . . 129
14.3 Submission contents . . . 130
14.3.1 Code . . . 130
14.3.2 Runnable utility . . . 130
14.3.3 Experiment instances . . . 130
14.3.4 System and external files . . . 130

Index

List of Figures

5.1 WBS . . . 63
5.2 Gantt chart . . . 64
5.3 Gantt chart: Project Management . . . 65
5.4 Gantt chart: Analysis . . . 66
5.5 Gantt chart: Design . . . 66
5.6 Gantt chart: Development . . . 67
5.7 Gantt chart: Documentation . . . 68
5.8 Gantt chart: Experiments . . . 69
5.9 Gantt chart: Closure . . . 69
5.10 Internal budget summary . . . 70
5.11 Client budget summary . . . 70

6.1 Preliminary class diagram . . . 76
6.2 Use cases . . . 77
6.3 Sequence diagram: Perform the assignments . . . 79
6.4 Activity diagram: Perform the assignments . . . 79
6.5 Sequence diagram: Search for free classrooms . . . 81
6.6 Activity diagram: Search for free classrooms . . . 81
6.7 Sequence diagram: Automatically create input files . . . 83
6.8 Activity diagram: Automatically create input files . . . 83

7.1 Components diagram . . . 88
7.2 Packages diagram . . . 89
7.3 Class diagram: Alg package . . . 90
7.4 Class diagram: Alg package (Greedy algorithm) . . . 91
7.5 Class diagram: Alg package (Genetic algorithm) . . . 92
7.6 Class diagram: Problem domain . . . 93
7.7 Class diagram: Other business classes . . . 94
7.8 Class diagram: Persistence . . . 94
7.9 Class diagram: Program and CLI . . . 94

12.1 Freelancers description . . . 121
12.2 Amortisation costs . . . 121
12.3 Indirect costs . . . 122
12.4 WBS Budget costs: Project Management and Analysis . . . 122
12.5 WBS Budget costs: Design and Development . . . 122
12.6 WBS Budget costs: Documentation, Experiments and Closure . . . 123
12.7 WBS Budget costs: Summary . . . 123
12.8 Internal budget . . . 123
12.9 Weighting value calculation . . . 124
12.10 Client budget . . . 124

14.1 WBS (Bigger scale) . . . 129

List of Tables

6.1 Use case description: Perform the assignments 78
6.2 Use case description: Search for free classrooms 80
6.3 Use case description: Automatically create input files 82
6.4 Test suite for UC: Perform the assignments 85
6.5 Test suite for UC: Search for free classrooms 86
6.6 Test suite for UC: Automatically create input files 86

10.1 Greedy algorithm: Best fitness summary 106
10.2 Greedy algorithm: Unassigned groups for instance rr_20_21_s1 106
10.3 Greedy algorithm: Unassigned groups for instance cr_21_22_s2 106
10.4 Greedy algorithm: Preferences met for instance rr_20_21_s1 107
10.5 Greedy algorithm: Preferences met for instance cr_21_22_s2 107
10.6 (Genetic + Greedy): Parameter combinations results (Best fitness) . . 108
10.7 Greedy against (Genetic + Greedy): Best fitness summary 109
10.8 Greedy against (Genetic + Greedy): Metrics for instance rr_20_21_s1 . 109
10.9 Greedy against (Genetic + Greedy): Metrics for instance cr_21_22_s2 . 109

List of Algorithms

1 Generic Greedy Algorithm 22
2 Bootaku Greedy Algorithm 23
3 Bootaku Greedy Algorithm BestFreelancerFor 23
4 Abstract Generational Algorithm 26
5 The (μ, λ) Evolution Strategy 28
6 The Genetic Algorithm (GA) 30
7 Random Selection 31
8 Fitness-Proportionate Selection 32
9 Stochastic Universal Sampling Selection 33
10 Tournament Selection 34
11 One-Point Crossover 36
12 Two-Point Crossover 36
13 Uniform Crossover 37
14 Order Crossover (OX) 39
15 Bit-Flip Mutation 40
16 Swap Mutation 40
17 ClassManager Greedy Algorithm 53
18 ClassManager Greedy Algorithm Preprocessing 54
19 ClassManager Greedy Algorithm Assignment Heuristic 55
20 ClassManager Greedy Algorithm Repairing Process 57
21 ClassManager Genetic Algorithm (GA) 58
22 ClassManager GA Next Generation 59
23 ClassManager GA Tournament Selection 62

0 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 José Manuel Redondo [Red]. However, some additional chapters were introduced in order to capture the particularity of the work carried out, inspired by the work of Gonzalo de la Cruz [dlC18].

This document is structured in the following chapters:

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. This chapter delves into the theory supporting the developed system. An 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 Work Breakdown Structure (WBS) on a larger scale 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 chapter. 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 proposed 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.

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 chapters.

1 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 must 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 these 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 after the beginning 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.

1.1 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 computation 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 [dlC18]. 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.

1.2 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 set of partial assignments), 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 Chapter 2. 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 explaining how to install, use, maintain and extend the system.

1.3 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 Section 1.2 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.

1.4 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 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, and evaluate the performance of the proposed solution.
6. Write a set of manuals to cover the essentials of the system.
7. Validate the solution with the users.

2 Theoretical aspects

This chapter provides all the necessary theoretical background for the project, including a general description of assignment problems as well as an account of the algorithms and methods used. As a running example, we will use the following simple problem.

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.

2.1 Assignment problem

The problem described before is an example of an assignment problem. It is the simplest formulation of the assignment problem. It is polynomial-time solvable, while other versions are NP-hard.

The Bootaku problem can be generalised with the following elements:

A set of n freelancers $F = \{F_1, F_2, ..., F_n\}$

A set of m book reviews $R = \{R_1, R_2, ..., R_m\}$

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

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:

$$\sum_{f=1}^{n} \sum_{r=1}^{m} a_{fr} p_{fr} \tag{2.1}$$

The optimal solution consists of 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.

$$F = \{Dante, Virgil, Beatrice\} \tag{2.2}$$

$$R = \{Divina\ Commedia, El\ Quijote, Voyage\ au\ bout\ de\ la\ nuit, Todo\ modo\} \tag{2.3}$$

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

$$A = \begin{array}{c} \\ Dante \\ Virgil \\ Beatrice \end{array} \begin{array}{c} \begin{array}{cccc} DC & EQ & VN & TM \end{array} \\ \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{pmatrix} \end{array} \tag{2.4}$$

$$P = \begin{array}{c} \\ Dante \\ Virgil \\ Beatrice \end{array} \begin{array}{c} \begin{array}{cccc} DC & EQ & VN & TM \end{array} \\ \begin{pmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \end{pmatrix} \end{array} \tag{2.5}$$

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:

$$A1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} \tag{2.6}$$

$$A2 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} \tag{2.7}$$

$$A3 = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \tag{2.8}$$

$$A4 = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} \tag{2.9}$$

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 $A2$

$$P1 = \begin{bmatrix} 9 & 1 & 5 & 4 \\ 2 & 8 & 14 & 2 \\ 7 & 11 & 10 & 6 \end{bmatrix} \tag{2.10}$$

$$P2 = \begin{bmatrix} 9 & 1 & 5 & 4 \\ 13 & 8 & 14 & 2 \\ 7 & 11 & 10 & 6 \end{bmatrix} \tag{2.11}$$

$P1$ and $A1$:

$$Profit = \sum_{f=1}^{n} \sum_{r=1}^{m} a_{fr} p_{fr} = 9 + 11 + 14 + 6 = 40 \tag{2.12}$$

$P1$ and $A2$:

$$Profit = \sum_{f=1}^{n} \sum_{r=1}^{m} a_{fr} p_{fr} = 2 + 11 + 14 + 6 = 33 \tag{2.13}$$

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 $A1$:

$$Profit = \sum_{f=1}^{n} \sum_{r=1}^{m} a_{fr} p_{fr} = 9 + 11 + 14 + 6 = 40 \tag{2.14}$$

$P2$ and $A2$:

$$Profit = \sum_{f=1}^{n} \sum_{r=1}^{m} a_{fr} p_{fr} = 13 + 11 + 14 + 6 = 44 \tag{2.15}$$

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 *task*s 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.

2.2 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

[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.

greedy algorithm checks whether the current solution is valid or not.

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

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 basic components of assignment problems and not to spend time on explaining difficult restrictions. However, most problems, including the real problem this document defines (to assign classrooms to the groups of the School), may 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 $n \times m$ matrix T with the completion times for all freelancers and reviews.

$$T = \begin{array}{c} \\ Dante \\ Virgil \\ Beatrice \end{array} \begin{array}{c} \begin{array}{cccc} DC & EQ & VN & TM \end{array} \\ \begin{pmatrix} t_{11} & t_{12} & t_{13} & t_{14} \\ t_{21} & t_{22} & t_{23} & t_{24} \\ t_{31} & t_{32} & t_{33} & t_{34} \end{pmatrix} \end{array} \tag{2.16}$$

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. With these additional constraints the Bootaku problem becomes an example of the so-called Generalised Assignment Problem, which is NP-hard.

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 limitation 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 does not have 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 best 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 *metaheuristics*.

2.3 Heuristics and metaheuristics

Solving a search problem involves a state space, represented by a tree or graph (depending on the situation) in which each node, called a state, represents an intermediate situation in the construction of the solution.

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 Evolutionary Computation method, a subtype of Population-based methods.

2.3.1 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 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. Two common EAs are *Genetic Algorithms* and *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 this kind of algorithms is usually 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 re-

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

peated 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 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 reason for this is that these individuals may introduce bias in the search, limiting its exploration capabilities.

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.

2.3.2 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 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 with a number of λ 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 number of λ/μ children are generated. A

mutation operation is performed to create the offspring from a copy of their parents. This whole process is done until we arrive at a solution with good enough quality 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 on 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, and it can terminate with a premature convergence.

2.3.3 Genetic algorithms

Genetic Algorithms (GAs) [Luk13] are 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 a 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.

2.3.3.1 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 greater the fitness of an individual, 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 values* will fall in the range of one of the individuals, thus selecting such an individual.

Algorithm 8 Fitness-Proportionate 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: **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.
4: **if** $\vec{f}$ contains only zeros **then**
5: Substitute all items of $\vec{f}$ with ones
6: **end if**
7: **for** i from 2 to l **do** ▷ Convert $\vec{f}$ into a cumulative distribution.
8: $f_i \leftarrow f_i + f_{i-1}$
9: **end for**
10: **end procedure**
11: **procedure** FitnessProportionateSelection(P)
12: $n \leftarrow$ random number from 0 to f_i inclusive
13: $selected \leftarrow p_1$
14: **for** i from 2 to l **do**
15: **if** $f_{i-1} < n \leq f_i$ **then**
16: $selected \leftarrow p_i$
17: **end if**
18: **end for**
19: **return** $selected$
20: **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.

2.3.3.2 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.

$$P_a = \{1, 5, 3, 6, 2, 7, 4\} \tag{2.17}$$

$$P_b = \{2, 5, 7, 3, 4, 6, 1\} \tag{2.18}$$

One-Point crossover 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.

$$C_a = \{1, 5, 3, 6, \ 4, 6, 1\} \tag{2.19}$$

$$C_b = \{2, 5, 7, 3, \ 2, 7, 4\} \tag{2.20}$$

In the case of Two-Point crossover, 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.

$$C_a = \{1, 5, 7, 3, 2, 7, 4\} \tag{2.21}$$

$$C_b = \{2, 5, 3, 6, 4, 6, 1\} \tag{2.22}$$

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 crossover, 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 than 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 crossover 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 $p_1 - 1$). OX is very effective when the encoding of the chromosomes are permutations. 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.

$$P_a = \{1, 5, 3, 6, 2, 7, 4\} \tag{2.23}$$
$$P_b = \{2, 5, 7, 3, 4, 6, 1\} \tag{2.24}$$

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

$$C = \{4, 1, 3, 6, 2, 7, 5\} \tag{2.25}$$

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

$$P_a = \{1, 5, 3, 6, 2, 7, 4\} \tag{2.26}$$
$$P_b = \{2, 5, 7, 3, 4, 6, 1\} \tag{2.27}$$
$$C = \{1, 5, 2, 3, 6, 7, 4\} \tag{2.28}$$

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

2.3.3.3 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 describe 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
```

2.4 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 2.2. 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.

$$Individual\ A = \{DC, EQ, VN, TM\} \tag{2.29}$$

$$Individual\ B = \{VN, EQ, DC, TM\} \tag{2.30}$$

$$Individual\ C = \{TM, VN, EQ, DC\} \tag{2.31}$$

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 to use a Tournament Selection, OX for the crossover operator and 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.

3 Problem definition

The School of Computing Engineering of the University 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 Chapter 2, there are more *tasks* (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.

$$C = \{c_1, c_2, ..., c_n\} \tag{3.1}$$
$$G = \{g_1, g_2, ..., g_m\} \tag{3.2}$$

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.

$$T = \{t_1, t_2, ..., t_p\} \tag{3.3}$$
$$L = \{l_1, l_2, ..., l_q\} \tag{3.4}$$

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.

$$W_i = \{w_{i1}, w_{i2}, ..., w_{ir}\} \tag{3.5}$$

$$H_i = \{h_{i1}, h_{i2}, ..., h_{is}\} \tag{3.6}$$

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.

$$S = \{s_1, s_2, ..., s_t\} \tag{3.7}$$

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

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.

$$R_i = \{r_{i1}, r_{i2}, ..., r_{iu}\} \tag{3.8}$$

$$P_i = \{p_{i1}, p_{i2}, ..., p_{iv}\} \tag{3.9}$$

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 should 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 should 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.

The goal is to assign a classroom to each group, satisfying all the hard constraints and fulfilling the soft constraints to the greatest possible extent.

[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.

4 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.

4.1 Search space

4.1.1 Assignments

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

$$(G_i, C_j) \tag{4.1}$$

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.

$$TotalNumberOfAssignments = |G| \tag{4.2}$$

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

4.1.2 Solutions

A *solution* for this problem is represented by a set of all the assignments that must be performed for the semester. For a solution to be valid, all hard constraints must

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

be satisfied. As presented in the previous section, the total number of assignments equals the total number of groups in that semester. So we have the next statement.

$$Solution = \{(G_1, C_x), (G_2, C_y), ..., (G_m, C_z))\} \tag{4.3}$$

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.

$$IncompleteAssignment = (G_i, -) \tag{4.4}$$

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

$$EmptySolution = \{(G_1, -), (G_2, -), ..., (G_m, -))\} \tag{4.5}$$

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.

4.1.3 States

A *state* represents a phase in the problem. We distinguish three kinds of 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 solution of assignments and work from there. Now we will discuss how we can

jump from one state to the next one, 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 at most the number of classrooms. So we have the following.

$$TotalNumberOfCandidates = |C| \tag{4.6}$$

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 Section 4.3). The important thing to note at this moment is that, because of these filters, not all states may be generated.

4.1.4 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 could be expected to 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.

4.2 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.

4.2.1 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* (LCM) comes in.

Imagine that we have the following group set.

$$G = \{G_1, G_2, G_3\} \tag{4.7}$$

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

$$LCM = \begin{array}{c} \\ G_1 \\ G_2 \\ G_3 \end{array} \begin{array}{c} \begin{array}{ccc} G_1 & G_2 & G_3 \end{array} \\ \begin{pmatrix} & -1 & -1 \\ -1 & & -1 \\ -1 & -1 & \end{pmatrix} \end{array} \tag{4.8}$$

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.

$$LCM = \begin{array}{c} \\ G_1 \\ G_2 \\ G_3 \end{array} \begin{array}{c} \begin{array}{ccc} G_1 & G_2 & G_3 \end{array} \\ \begin{pmatrix} & 1 & -1 \\ 1 & & -1 \\ -1 & -1 & \end{pmatrix} \end{array} \tag{4.9}$$

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.

$$LCM = \begin{array}{c} \\ G_1 \\ G_2 \\ G_3 \end{array} \begin{array}{c} \begin{array}{ccc} G_1 & G_2 & G_3 \end{array} \\ \begin{pmatrix} & 1 & -1 \\ 1 & & 0 \\ -1 & 0 & \end{pmatrix} \end{array} \tag{4.10}$$

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.

$$LCM = \begin{array}{c} \\ G_1 \\ G_2 \\ G_3 \end{array} \begin{array}{c} \begin{array}{ccc} G_1 & G_2 & G_3 \end{array} \\ \begin{pmatrix} & g_{12} & g_{13} \\ g_{21} & & g_{23} \\ g_{31} & g_{32} & \end{pmatrix} \end{array} \tag{4.11}$$

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 accesses the corresponding location in the LCM, and because it contains a 1, the greedy algorithm concludes that they indeed collide. These operations are done in amortised constant time, as the matrix is coded as a dictionary of dictionaries. If instead the greedy algorithm 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.

4.3 Classroom filters

A classroom filter is a function. It receives as input either the set C of classrooms or a subset $I \subseteq C$, 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, described in Section 4.2.1). 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 set F.

This reduces the number of classrooms that the greedy algorithm 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 need to 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.

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

$$M = \{M_1, M_2, ..., M_n\} \tag{4.12}$$

$$G = \{G_1, G_2, G_3\} \tag{4.13}$$

$$C = \{C_1, C_2\} \tag{4.14}$$

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 algorithm will try to assign a classroom for all groups. As a result of the values in the dictionary being $null$, the greedy algorithm 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 $F_i \subseteq C$ being the filtered classrooms for each group. Example of values for the F sets follow.

$$F_1 = \{C_1\} \tag{4.15}$$
$$F_2 = \{C_1, C_2\} \tag{4.16}$$
$$F_3 = \{\} \tag{4.17}$$

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 some hard constraints are changed. Lastly, a final remark. We can say that from now on, until the prototype terminates, the greedy algorithm will not have to perform the filtering operations for any of the groups again, as the results are already stored in the dictionary.

4.4 Greedy algorithm

The greedy algorithm (see Section 2.2 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 classroom for an assignment, and if found, it assigns it not only to that group, but to all groups to which that group is related (unless collisions occur, in which case these groups are left unassigned until the greedy algorithm iterates over them). 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.

4.4.1 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 classrooms for a given group contains only one element, i.e. whether only one classroom 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 classroom 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 classroom to remain unassigned, so priority is given to such groups.

4.4.2 Heuristic

The heuristic used by the greedy algorithm is as follows. Classes are filtered by type, capacity and constraints (positive and negative).

Once filtered, they are sorted according to two criteria. If there are preferences for that group, they are ordered in such a way that first the classes included in the positive preferences are iterated, then the classes with no preference, and finally the classes included in the negative preferences. Each of these three subsets is ordered according to the number of available seats, from lowest to highest. If there are no preferences for that group, the classes are simply ordered by number of available seats, also from lowest to highest.

Then, the filtered classes 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
```

4.4.3 Repairs

To conclude with the greedy algorithm components, we must describe the assignment repair process, which is probably the most complex of all components. As previously explained, in the case of not finding an appropriate classroom 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 classrooms of the group and, for each classroom, it obtains the groups that are in conflict with the evaluated group. If there is only one conflict for a classroom, the assignment of the conflicting group is removed and the classroom is assigned to the group under repair. Then, an attempt is made to obtain a new classroom for the conflicting group. If successful, both end up with an assigned classroom and the repair succeeds. If unsuccessful, the previous classroom is reassigned to the conflicting group and the process is repeated for each conflicting group in each classroom. If, after the execution of this process, the repair is unsuccessful, the group is finally left 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
```

4.5 Genetic Algorithm

The Genetic Algorithm (GA) (see Section 2.3.3 for details) generates sets of assignments, empty or partial, with different orderings. These 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 kept into 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 than or equal to the probability, the crossover operator 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**

4.5.1 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 algorithm by generating a bunch of different orderings and therefore testing which order produces the best solutions.

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.

$$I = \{A_1, A_2, ..., A_m\} \tag{4.18}$$

Where m is the number of groups for that semester.

To convert the individual representation into an actual list 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 list with the order given by the representation.

4.5.2 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 Chapter 3), and has a weight associated with it.

The formula for the fitness function is therefore given by:

$$fitness = \sum_{i=1}^{n} V_i W_i \tag{4.19}$$

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

4.5.3 Operators

We will briefly describe the selected operators for the ClassManager GA (see Section 2.3.3 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 Algorithm 22).

4.5.3.1 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 Algorithm 7.

4.5.3.2 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 twice, with the roles of the parents inverted (see the $NextGeneration$ pseudocode), 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 Algorithm 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 than or equal to the predetermined crossover probability.

4.5.3.3 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 than or equal to the mutation probability. The pseudocode for Swap Mutation is indicated in Algorithm 16.

4.5.3.4 Tournament

Finally, we introduce the specific version of Tournament Selection implemented for the ClassManager GA. It differs from the generic Tournament Selection (see Algorithm 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 Project planning and budget overview

So far we have discussed the theory that will function as a pillar of the software system to be developed. With this chapter we begin the sections focused on the software engineering used to define and build such a system. In this first chapter we present the planning and WBS of the software project, defining and explaining its phases, and finally showing an overview of the internal and customer budget.

5.1 Planning

The planning of the class management system, identified as Classmanager, has the following phases. They are shown with a Gantt diagram and with the WBS of the project. Due to the size of the Gantt chart, it is shown in its full version and split into phases. We operate with working hours of 3 hours on average every day of the week.

The following phases were established for the planning: *Project Management*, *Analysis*, *Design*, *Development*, *Documentation*, *Experiments* and *Closure*.

Figure 5.1: WBS

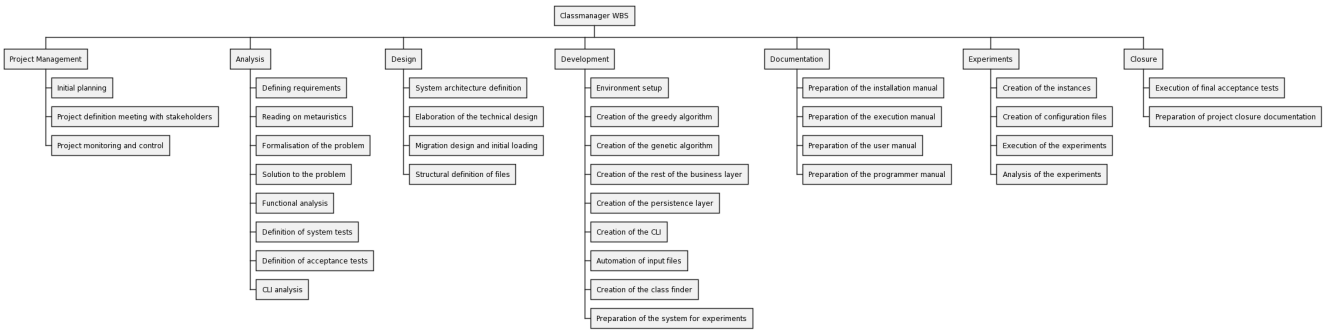


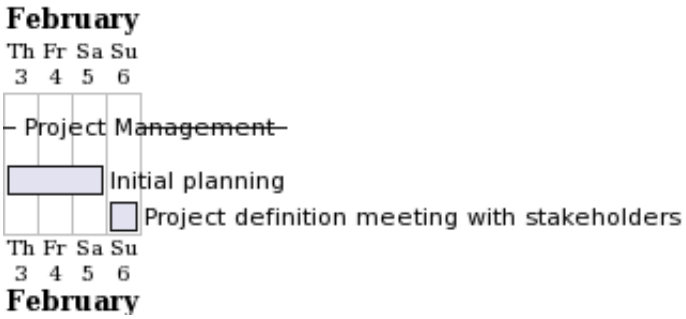
Figure 5.2: Gantt chart

5.1.1 Project Management

- **Initial planning**: Reading the project synopsis, first discussions with supervisors and establishing the project plan.

- **Project definition meeting with stakeholders**: Requirements gathering meeting with the client, definition of objectives and scope of the project.
- **Project monitoring and control**: Regular meetings with supervisors and clients to discuss and review project status.

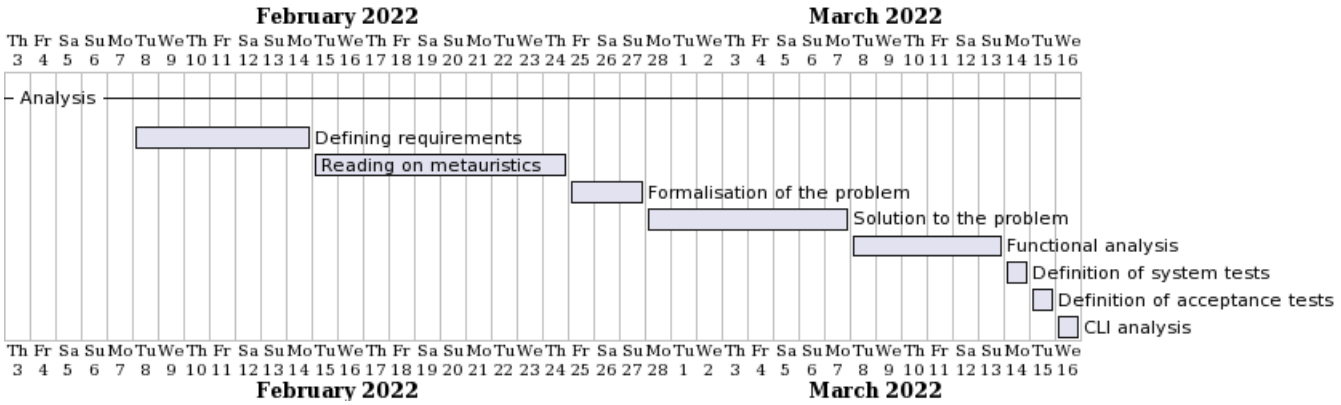
Figure 5.3: Gantt chart: Project Management



5.1.2 Analysis

- **Defining requirements**: Write down and review the requirements previously obtained in the meeting with the clients.
- **Reading on metaheuristics**: Reading diverse literature on greedy algorithms, genetic algorithms and metaheuristics.
- **Formalisation of the problem**: Formal definition of the problem as an assignment problem, explaining its components and constraints.
- **Solution to the problem**: Theoretically elaborating the solution to the previously defined problem.
- **Functional analysis**: Analysis of use cases, functionality and other aspects of the software prototype.
- **Definition of system tests**: Tests on input files, configuration files and system functionality.
- **Definition of acceptance tests**: Checking the quality of the results obtained by the system.
- **CLI analysis**: Study on how to run the utility and the format of its arguments.

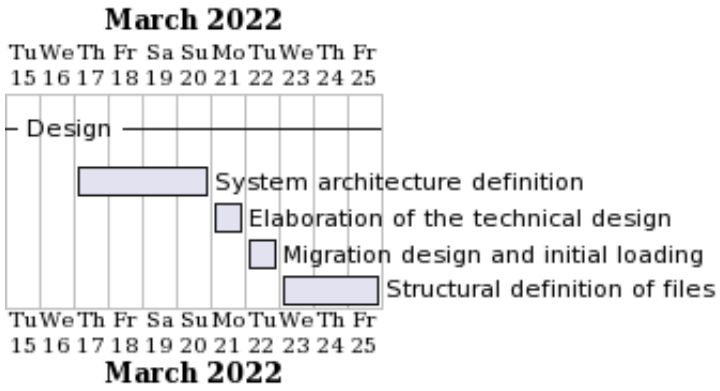
Figure 5.4: Gantt chart: Analysis



5.1.3 Design

- **System architecture definition**: Definition of subsystems, packages and the relationships between them.
- **Elaboration of the technical design**: Final class diagrams.
- **Migration design and initial loading**: Study on the conversion of planning and enrolment files into files with a format that can be processed by the system.
- **Structural definition of files**: Elaborating the final format of the input, output and configuration files.

Figure 5.5: Gantt chart: Design

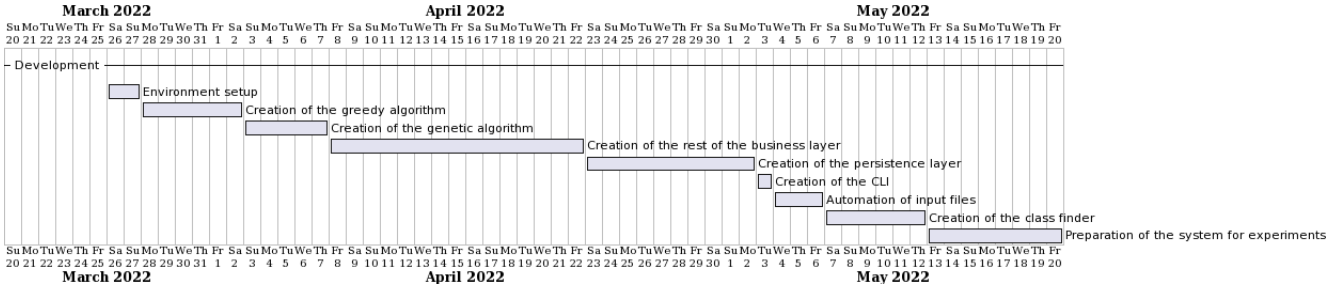


5.1.4 Development

- **Environment setup**: Creation of code and documentation repositories, configuration of text editors and tools to run and test the system.

- **Creation of the greedy algorithm**: Define the LCM, LFD, and the Greedy Algorithm components.
- **Creation of the genetic algorithm**: Define the operators of the Genetic Algorithm and its components.
- **Creation of the rest of the business layer**: Implement data infrastructure, log management, error handling and other business layer items.
- **Creation of the persistence layer**: Develop DataAccess and file management.
- **Creation of the CLI**: Utilities for all CLI content to be centralised in a single component.
- **Automation of input files**: Functionality of automating the files previously used by the School.
- **Creation of the class finder**: Functionality of searching classrooms with free time slots.
- **Preparations of the systems for experiments**: Adapt the system to make it easier to run many instances in parallel.

Figure 5.6: Gantt chart: Development

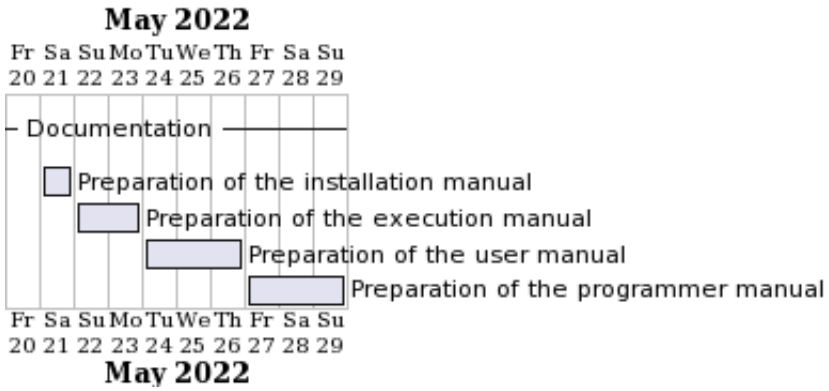


5.1.5 Documentation

- **Preparation of the installation manual**: Required dependencies and setup required to be able to run the application.
- **Preparation of the execution manual**: Available options for launching the software.

- **Preparation of the user manual**: Instructions on how to execute each functionality, explanation of how to interpret the output files and recommended configurations.
- **Preparation of the programmer manual**: Instructions on how to maintain the program, where to find the main parts of the code and how to interpret the log file.

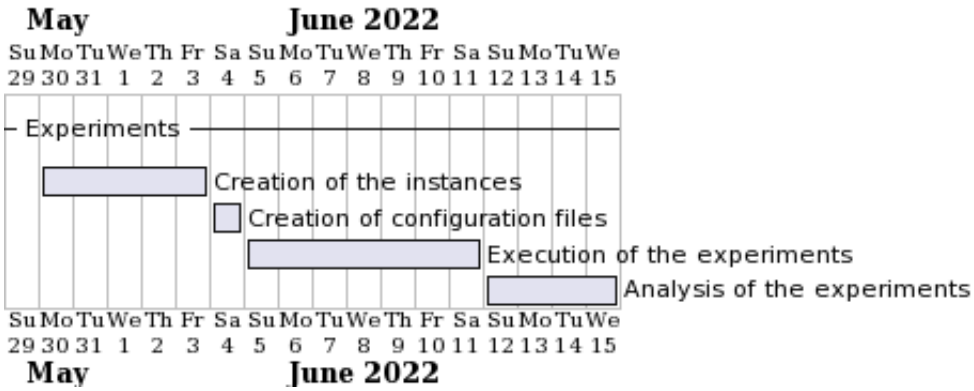
Figure 5.7: Gantt chart: Documentation



5.1.6 Experiments

- **Creation of the instances**: Instances are created based on actual data from past courses by using the automation functionality of the files used by the School.
- **Creation of configuration files**: Define the configuration files with the different combinations of values and parameters.
- **Execution of the experiments**: Execution of the experiments using a computer specialised in parallel runs.
- **Analysis of the experiments**: Analysis of the results and obtaining the best values in order to obtain the highest possible quality results.

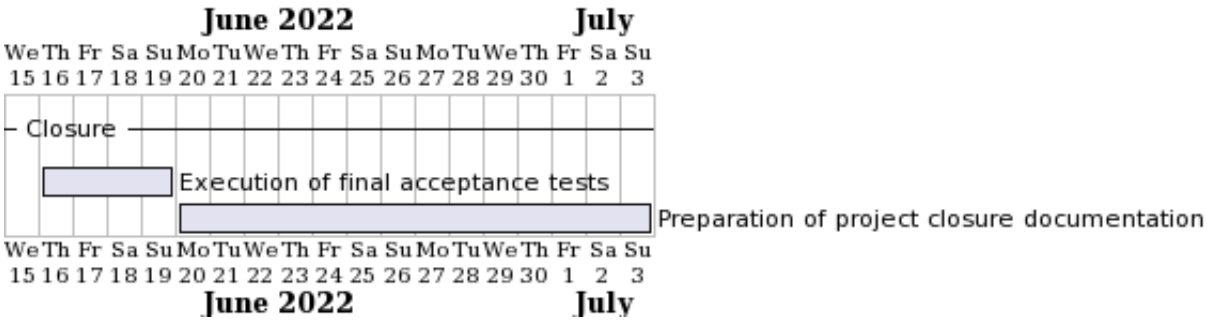
Figure 5.8: Gantt chart: Experiments



5.1.7 Closure

- **Execution of final acceptance tests**: Execution of tests to verify the quality of the results.
- **Preparation of the closure documentation**: Theoretical and technical project documentation.

Figure 5.9: Gantt chart: Closure



5.2 Budget summary

This section shows the internal and client buget summary of the project. They can be found in Figures 5.10 and 5.11. For the budget breakdown the reader is referred to Section 12.

Figure 5.10: Internal budget summary

Item	Total
Project Management	443,31 €
Analysis	1.822,51 €
Design	443,31 €
Development	2.758,39 €
Documentation	443,31 €
Experiments	788,11 €
Closure	837,37 €
Indirect Costs	2.620,00 €
Amortisations	112,50 €
	10.268,82 €

Figure 5.11: Client budget summary

Item	Total
Analysis	4.278,68 €
Design	1.040,76 €
Development	6.475,83 €
Documentation	1.040,76 €
Classmanager	**12.836,03 €**

6 Analysis

In this second chapter of software engineering, we will present the analysis conducted on the system to be built, with a brief general definition of what is intended to be done, recapitulating what was previously discussed in the theoretical sections.

The system requirements, both functional and non-functional, are also introduced, followed by an identification of the internal components of the system, the subsystems. An outline of class design is shown, focusing on the relationships between classes and data, which continues with an analysis of the identified use cases and the user interface. The chapter ends with a discussion of the tests to be carried out on the system.

6.1 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.

6.2 System requirements

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

6.2.1 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 configuration files can be split into any number of files.

RInput3.2 The configuration files must include the information required by each functionality.

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.

6.2.2 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.

6.3 Subsystem mapping

In order to better understand the system, a study has been carried out on the subsystems that make up the system. The following subsystems have been identified:

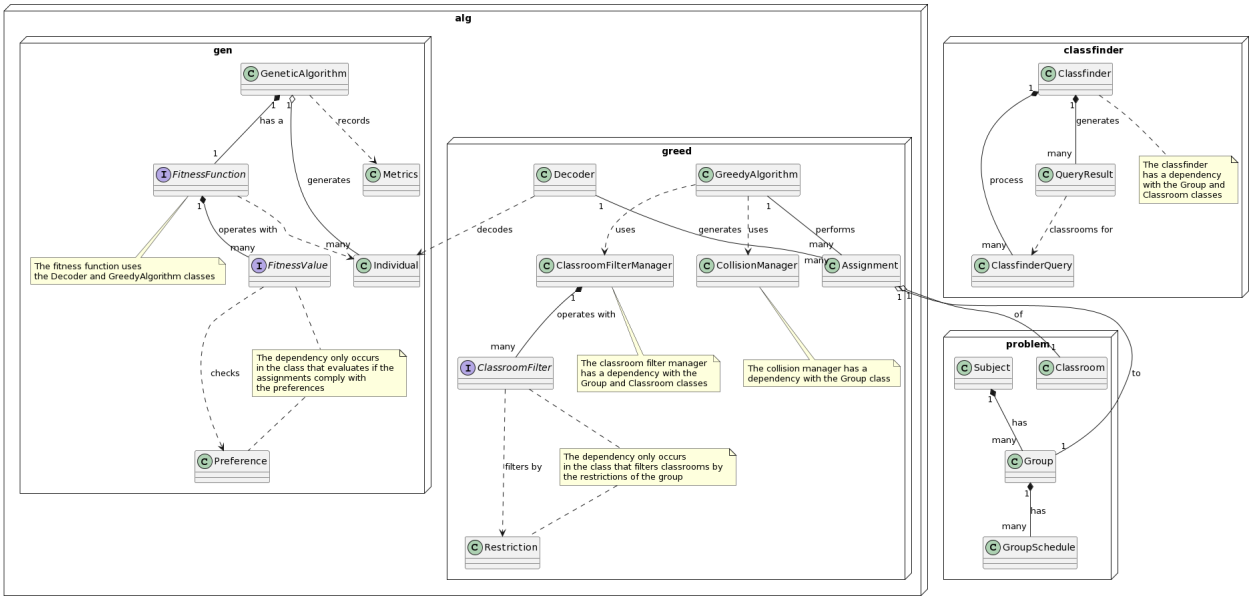
- **Algorithm subsystem:** Contains the greedy and genetic algorithms, as well as their operators and procedures.
- **Classfinder subsystem:** Contains functionality related to the search for free classes in a time and schedule range.
- **Configuration subsystem:** Parses configuration files and keeps them in memory.
- **ErrorHandler subsystem:** It addresses the errors encountered and differentiates the expected from the unexpected.
- **Log subsystem:** It is in charge of creating and updating the system log.

- **Problem domain subsystem:** Defines the abstractions of the problem in the form of object types.
- **Central subsystem:** In charge of connecting the other subsystems to carry out the different functionalities of the system.
- **Persistence subsystem:** Responsible for parsing input files and constructing output files.
- **CLI subsystem:** Handles all text output to the console.

6.4 Preliminary class diagram

The preliminary class diagram is shown in Figure 6.1.

Figure 6.1: Preliminary class diagram



Some notes on the previously shown class draft. It only contains classes from the business layer, because it is the most important part of the prototype.

The *alg* package contains all classes related to the genetic and greedy algorithms. They are both interconnected by means of the fitness function of the GA, which uses the Decoder in the *greed* package to get the assignments in the order specified by an individual and then calls the greedy algorithm with such a list of assignments. In

the case of the greedy algorithm, some of its components, like the ClassroomFilterManager (which models the LFD) and the CollisionManager (which models the LCM) have a dependency with some classes of the problem domain package. That is also the case for the Assignment class, which represents an assignment of a classroom to a group.

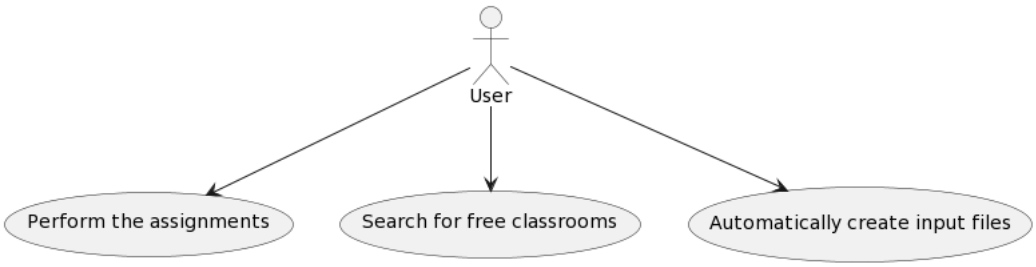
The *classfinder* package is a little more isolated than the alg package, but nevertheless has dependencies with some classes of the problem domain.

Finally, the *problem* package models the problem domain, and contains the information of the abstracted models of the School and its environment. It is a key package not only for the different packages of the business layer, but also for the persistence layer, which depends on it for implementing the DataAccess to each abstraction.

6.5 Analysis of use cases

This section presents the analysis of the use cases of the system. Three use cases have been identified (see Figure 6.2), each one reflecting a main functionality of the prototype. Therefore we have a use case for the assignment of classes to groups, the search for free classes, by means of specific queries, and the automatic generation of the system input files.

Figure 6.2: Use cases



For each use case, a table with information about the use case is presented. These tables contain the preconditions for the use case, the postconditions reached after its execution, a description of its steps, and a section with other variants of the use case. These tables are followed by a sequence and activity diagram, which show the flow of the use case in a more visual way.

6.5.1 Perform the assignments

Table 6.1: Use case description: Perform the assignments

PERFORM THE ASSIGNMENTS	
PRECONDITIONS	*The configuration files containing information about the GA and the input files need to be correctly formatted and must include the required data. Moreover, the input files with the classrooms, subjects, groups, group schedules, group academic weeks, restrictions, preferences and initial assignments must be correctly formatted.*
POSTCONDITIONS	*A text file with a report on the calculated assignments is generated. Also, a csv file with the assignments in the required format and a text file per classroom indicating its schedule in a timetable format are also written.*
DESCRIPTION	• *The user executes the program with the option flag signaling the calculation of the assignments and the path to the required configuration files.* • *The system parses the configuration files.* • *The system parses the required and optional files indicated in the configuration files, as well as the GA parameters.* • *The system executes the algorithms.* • *The system outputs the best individual's information into a report text file, a csv file with the assignments and a text file for each classroom with its timetable.*
SECONDARY SCENARIOS (VARIANTS)	**Variant 1. Errors on parsing.** If the system encounters any errors while parsing the input files, it will notify the user and store the information about such errors in the log file. **Variant 2. Not enough permissions.** If the system cannot read from or write into a folder because of insufficient permissions, it will notify the user and store the information about the error in the log file.

Figure 6.3: Sequence diagram: Perform the assignments

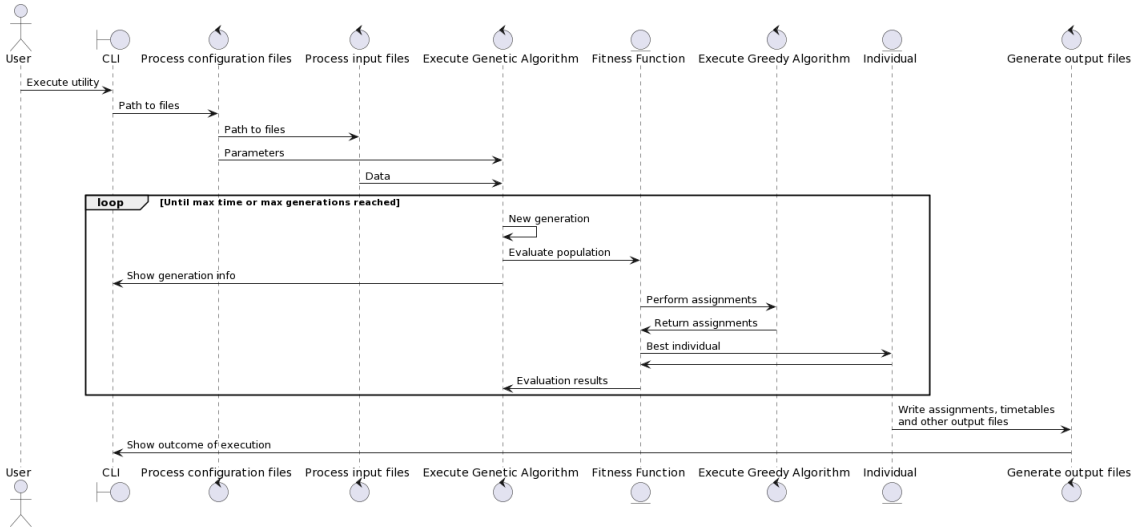
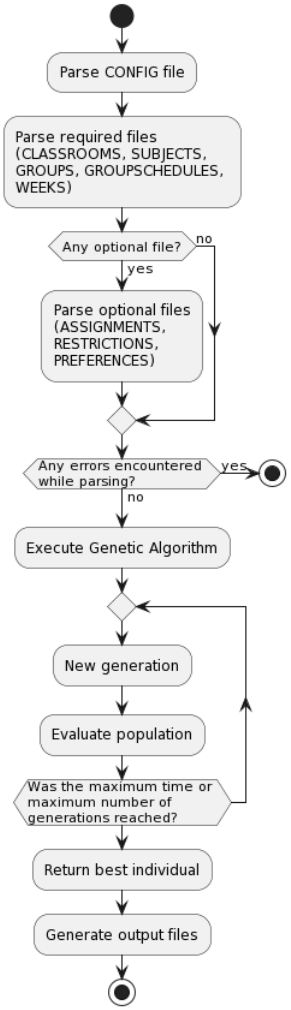


Figure 6.4: Activity diagram: Perform the assignments



6.5.2 Search for free classrooms

Table 6.2: Use case description: Search for free classrooms

SEARCH FOR FREE CLASSROOMS	
PRECONDITIONS	*The configuration files containing information about the input files needs to be correctly formatted and must include the required data. Moreover, the input files with the classrooms, subjects, groups, group schedules, group academic weeks, assignments and queries must be correctly formatted.*
POSTCONDITIONS	*A text file with the query results is generated.*
DESCRIPTION	• *The user executes the program with the option flag signaling the execution of the queries for finding free classrooms and the path to the required configuration files.* • *The system parses the configuration files.* • *The system parses the required files indicated in the configuration files.* • *The system executes the queries.* • *The system outputs the result of all queries into a text file.*
SECONDARY SCENARIOS (VARIANTS)	*Same as in Table 6.6*

Figure 6.5: Sequence diagram: Search for free classrooms

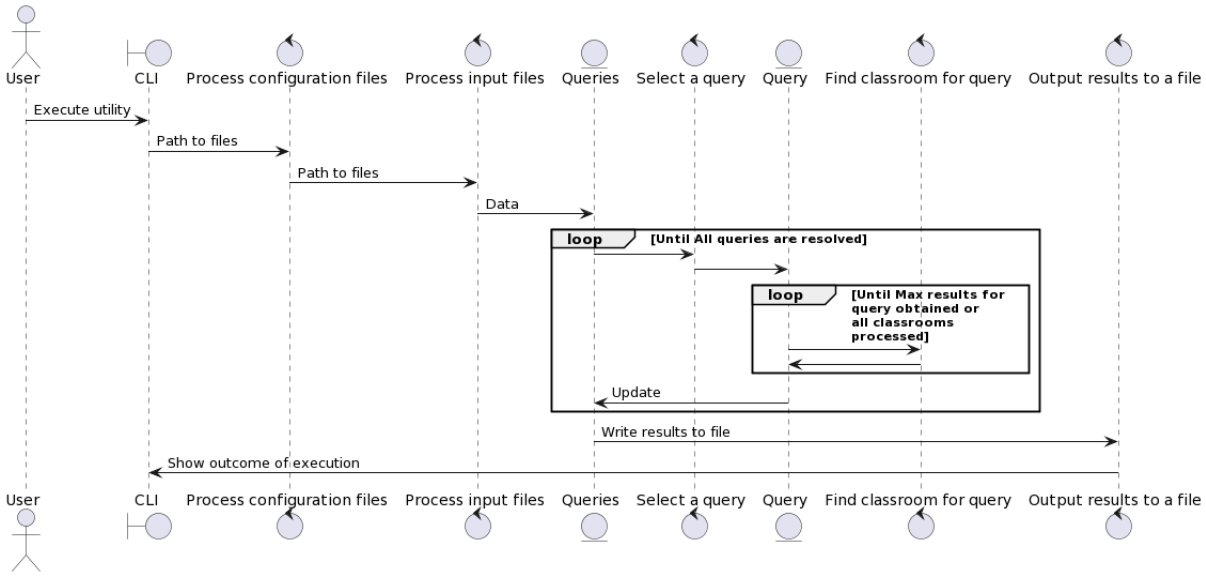
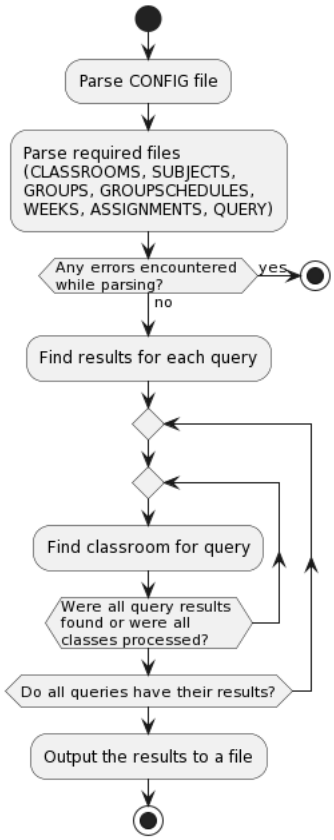


Figure 6.6: Activity diagram: Search for free classrooms



6.5.3 Automatically create input files

Table 6.3: Use case description: Automatically create input files

AUTOMATICALLY CREATE INPUT FILES	
PRECONDITIONS	*The configuration files containing information about the input files needs to be correctly formatted and must include the required data. Moreover, the input files with the planning for the semester and the number of enrolled students for each group must be correctly formatted.*
POSTCONDITIONS	*The files with the data of the groups, group schedules and group academic weeks are generated.*
DESCRIPTION	• *The user executes the program with the option flag signaling the generation of the input files and the path to the required configuration files.* • *The system parses the configuration files.* • *The system parses the required files indicated in the configuration files.* • *The system transforms the input files into the required format.* • *The system outputs the result of each transformation in their own files.*
SECONDARY SCENARIOS (VARIANTS)	*Same as in Table 6.6*

Figure 6.7: Sequence diagram: Automatically create input files

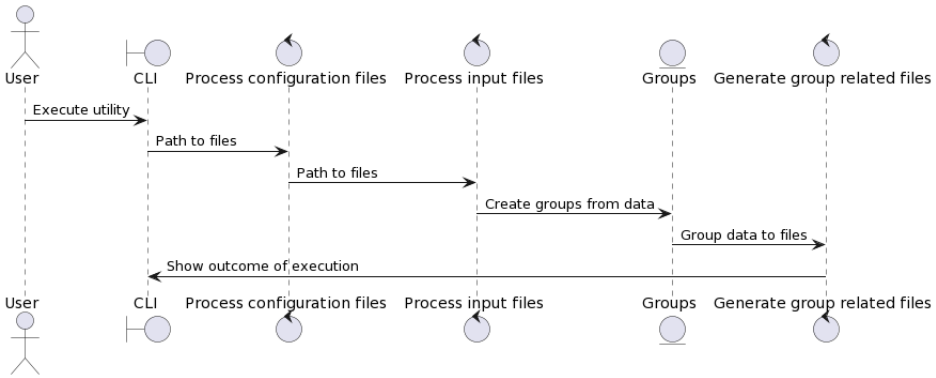
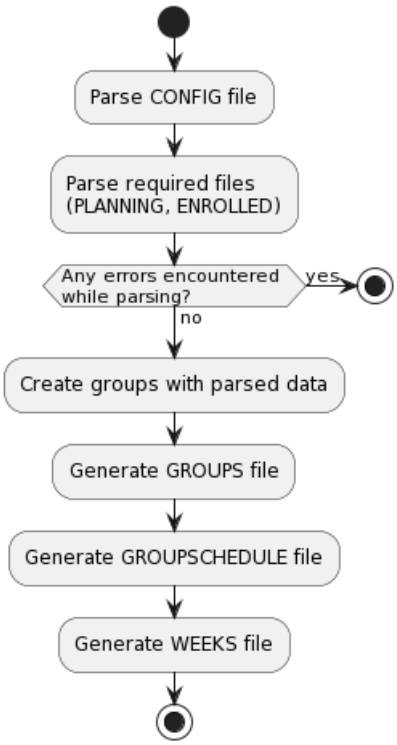


Figure 6.8: Activity diagram: Automatically create input files



6.6 Analysis of user interfaces

The prototype will have a Command Line Interface (CLI). The only interaction with the user occurs when they are calling the utility in the shell, indicating the option they want the prototype to execute and the necessary configuration files. As we explained before in the document, there are three main options for calling the prototype. One for performing the assignments, the second for finding free classrooms and the third

for automating the creation of the input files.

Once the execution starts, the CLI will show relevant information to the user about the currently running processes (name and status). In the case of the GA execution, the selected parameters will be listed, as well as data on the results of each generation. The amount of information shown in the CLI about the GA can be controlled by the user through the configuration files.

6.7 Test plan specification

Testing will focus on the parsing phase and the quality of the solutions obtained. In order to verify that both areas operate as expected, a battery of functional tests and an experimentation with various scenarios and instances will be carried out. Further details of the conducted experimentation are given in its own section (see Section 10), here we will focus on the test suite.

Table 6.4: Test suite for UC: Perform the assignments

PERFORM THE ASSIGNMENTS	
Test	**Expected result**
No optional files	The assignment process will start from scratch and no restrictions or preferences are to be considered.
Preferences	The assignment process will start from scratch and no restrictions are to be considered. Positive and negative preferences will be evaluated as soft constraints.
Restrictions	The assignment process will start from scratch and no preferences are to be considered. Positive and negative restrictions will be evaluated as hard constraints.
Partial assignments	The assignment process will start from the initial assignments and no preferences or restrictions are to be considered. The initial assignments will remain as they were once the allocation process is completed.
All optional files	The assignment process will start from the initial assignments and the restrictions and preferences will be taken into account.
Incorrect config format	The system will notify the error to the user.
Incorrect input file format	The system will notify the error to the user.
Incorrect input data	The system will notify the error to the user

Table 6.5: Test suite for UC: Search for free classrooms

SEARCH FOR FREE CLASSROOMS	
Test	**Expected result**
Theory query	The system will look for theory classes that meet the requirements of the query.
Lab query	The system will look for laboratories that meet the requirements of the query.
Theory and lab query	The system will look for theory classes for one request and laboratories for the other. Each search must meet the requirements of its associated query.
Incorrect config format	The system will notify the error to the user.
Incorrect input file format	The system will notify the error to the user.
Incorrect input data	The system will notify the error to the user

Table 6.6: Test suite for UC: Automatically create input files

AUTOMATICALLY CREATE INPUT FILES	
Test	**Expected result**
Automation	The system will generate the input files necessary for the GA execution.
Incorrect config format	The system will notify the error to the user.
Incorrect input file format	The system will notify the error to the user.
Incorrect input data	The system will notify the error to the user

7 System design

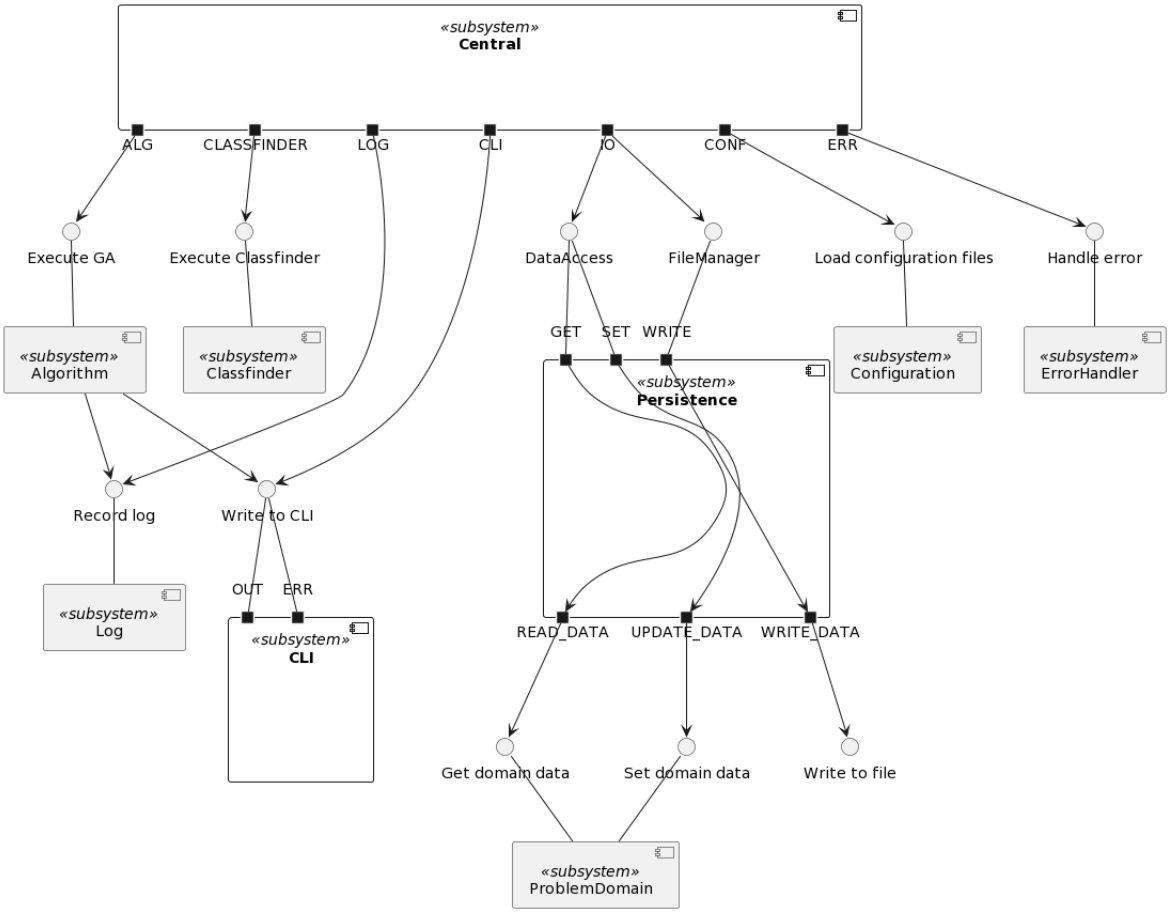
Until now we have seen a theoretical view of the algorithms, with some pseudocode, and an abstracted technical view of the system with the analysis discussed in the previous chapter. This chapter presents to the reader, with the help of different diagrams, a more technical side to the prototype in which the actual code of the utility will be based. The software architecture, the detailed class diagram and the format of the files used by the system follow.

7.1 System architecture

The architecture of the system is laid out in two diagrams, the component diagram, which models the relationship between the subsystems and their interfaces; and the package diagram, which shows the logical layout of the code.

The component diagram is depicted below (see Figure 7.1).

Figure 7.1: Components diagram

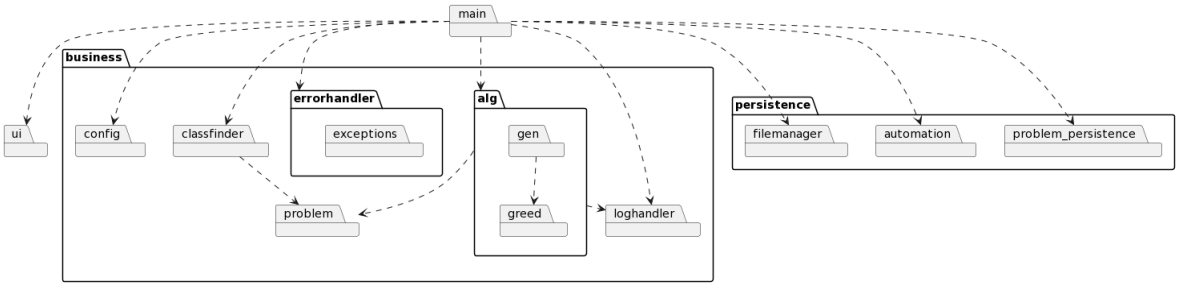


It can be seen that the central subsystem is responsible for connecting the major subsystems with each other, especially those belonging to the different code layers. Only the central subsystem and the algorithm subsystem have direct access to the log and the CLI.

The relationship between the persistence layer and the problem domain subsystem is due to the fact that it is the former that creates the data for the latter, and once the central subsystem receives this data, it is responsible for providing said data to the different components that require it.

The package diagram follows (see Figure 7.2).

Figure 7.2: Packages diagram



What is noteworthy of this diagram is the appearance of new elements that the initial class diagram did not have. For example, the error handler contains both its logic and the definition of the prototype's own exceptions. The log handler contains a reference to the internal Java log logic, and acts as a wrapper that facilitates communication with other modules. Finally, the persistence layer has three main packages: the file manager, in charge of writing and reading files; the automation functionality of the files from the School to our files; and finally the DataAccess to the files of the domain.

7.2 Class design

This section presents the different class diagrams that make up the whole system, without showing trivial details and without repeating elements with similar layouts, in order to enhance readability.

Figure 7.3: Class diagram: Alg package

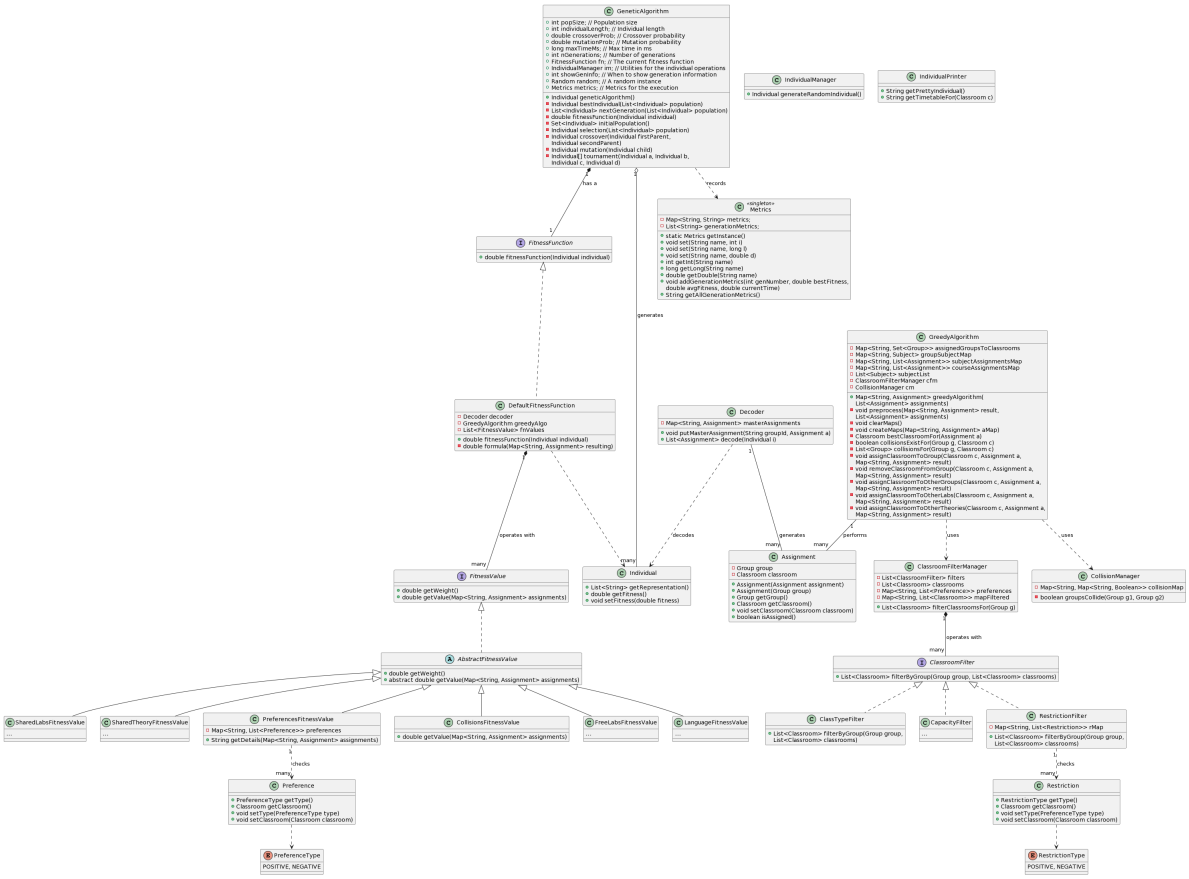


Figure 7.4: Class diagram: Alg package (Greedy algorithm)

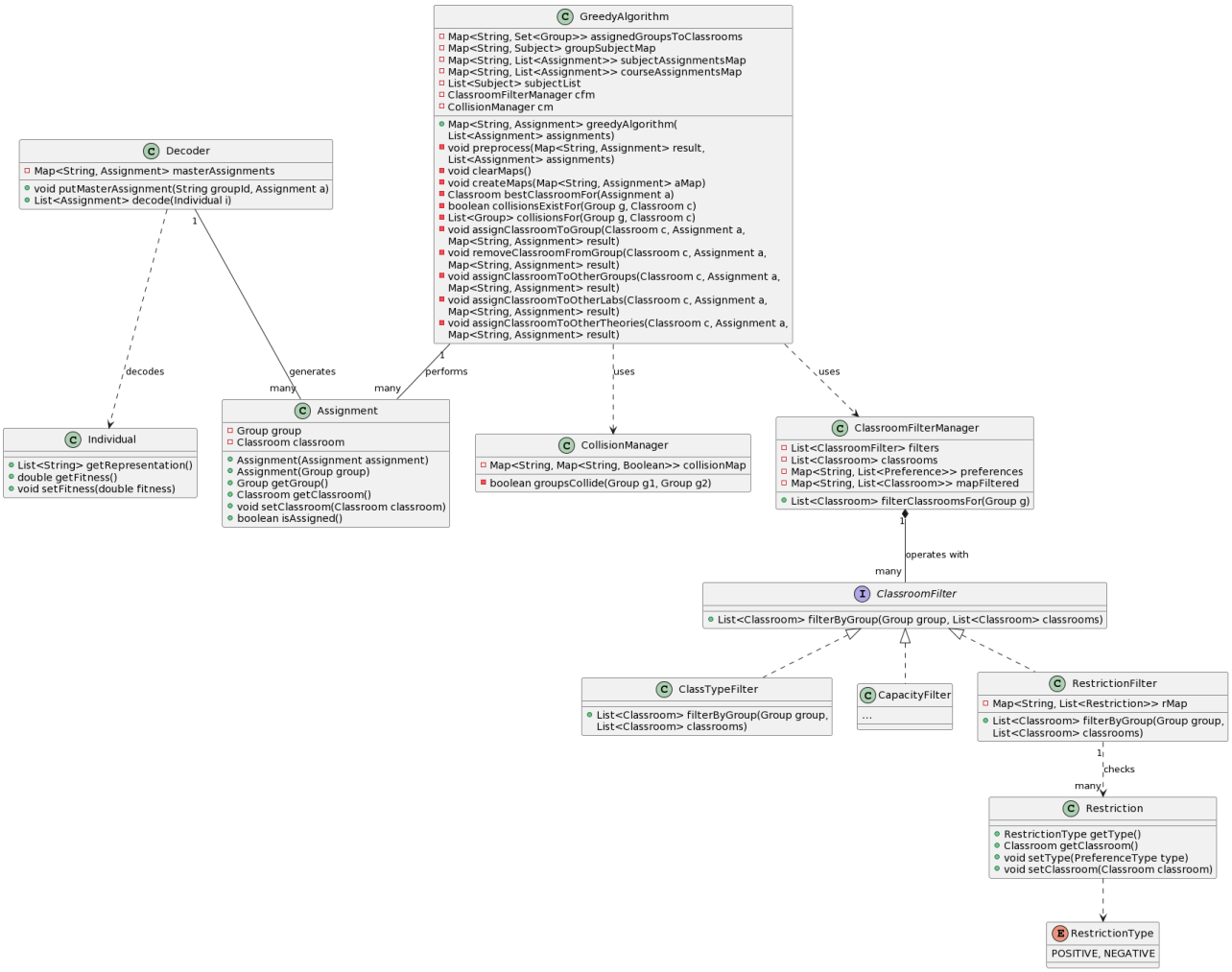


Figure 7.5: Class diagram: Alg package (Genetic algorithm)

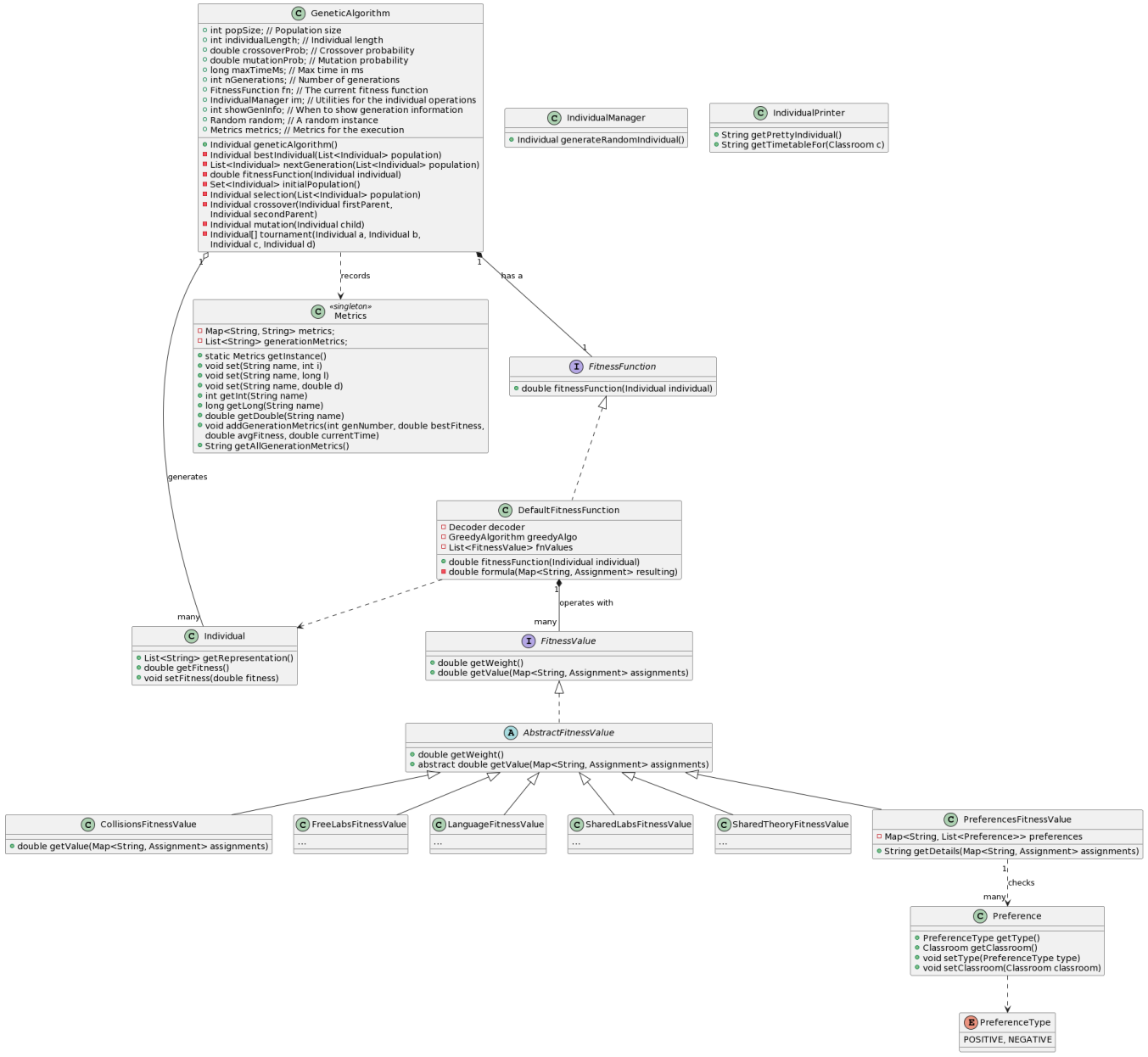


Figure 7.6: Class diagram: Problem domain

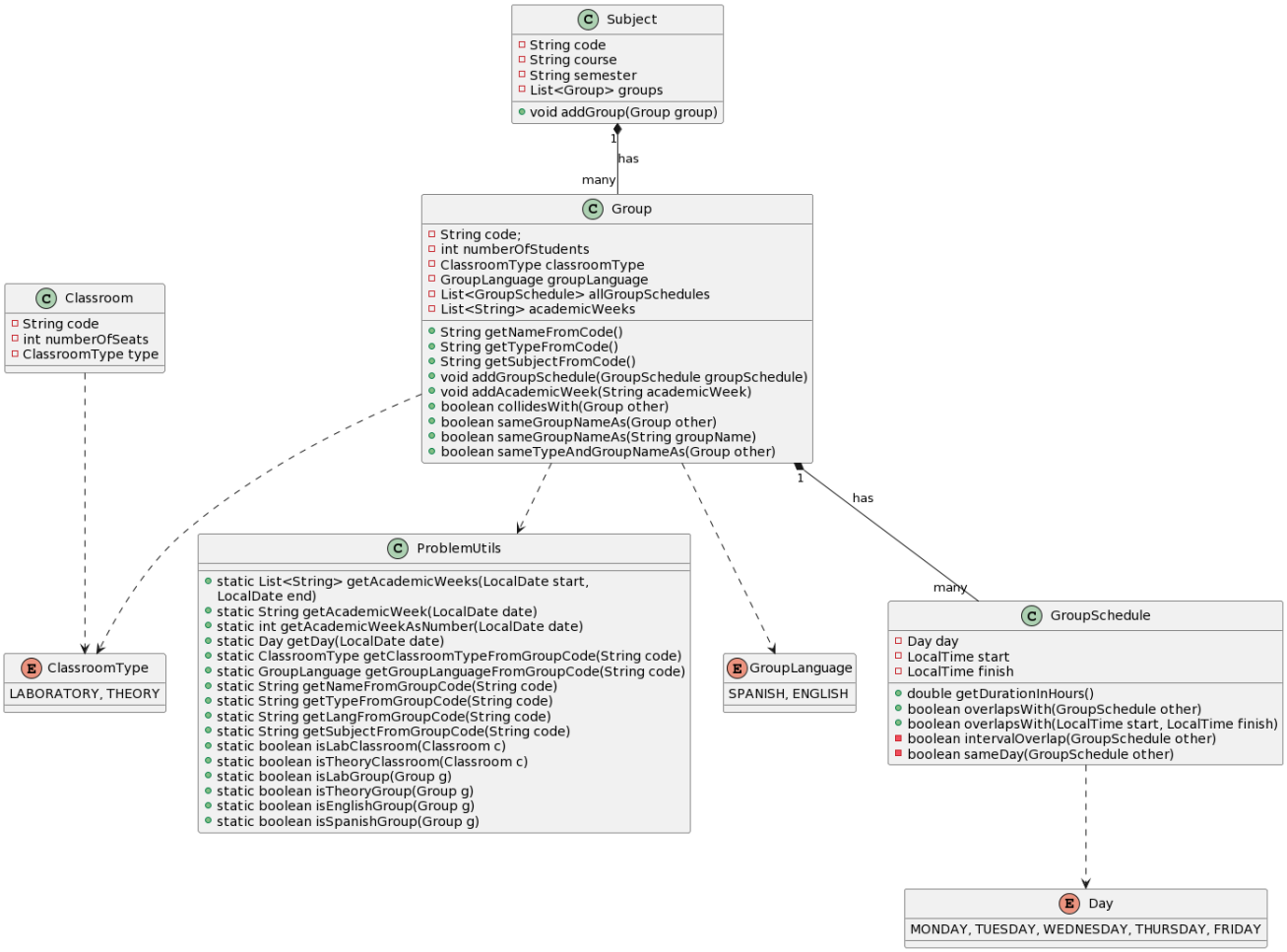


Figure 7.7: Class diagram: Other business classes

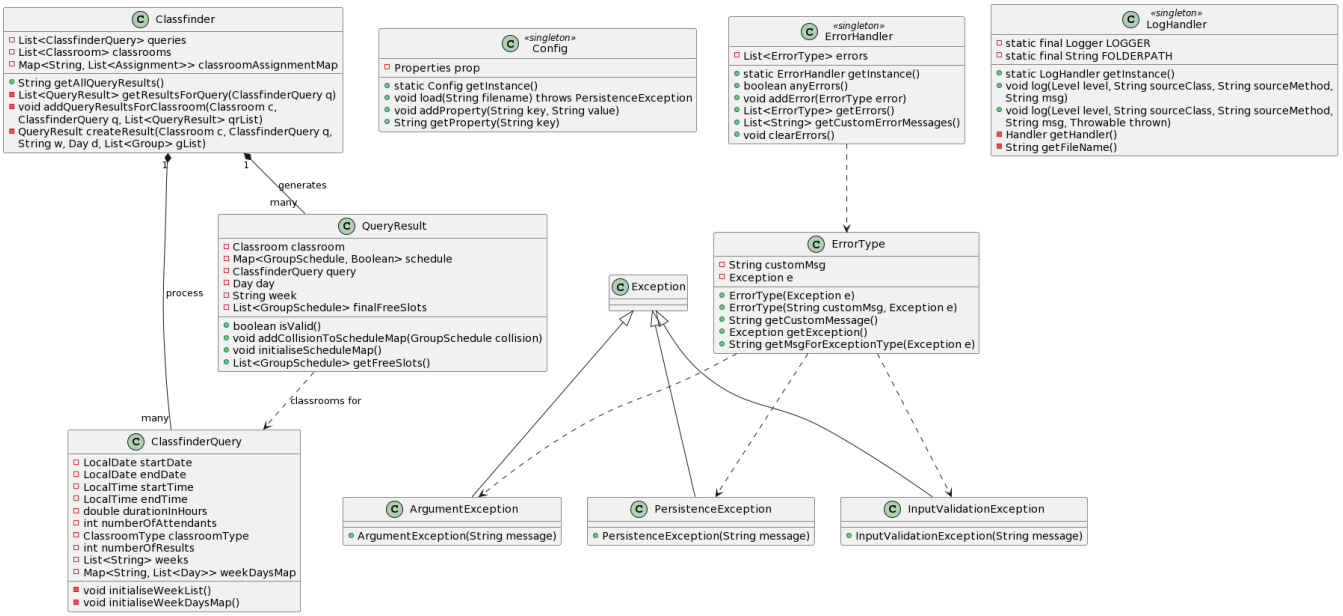


Figure 7.8: Class diagram: Persistence

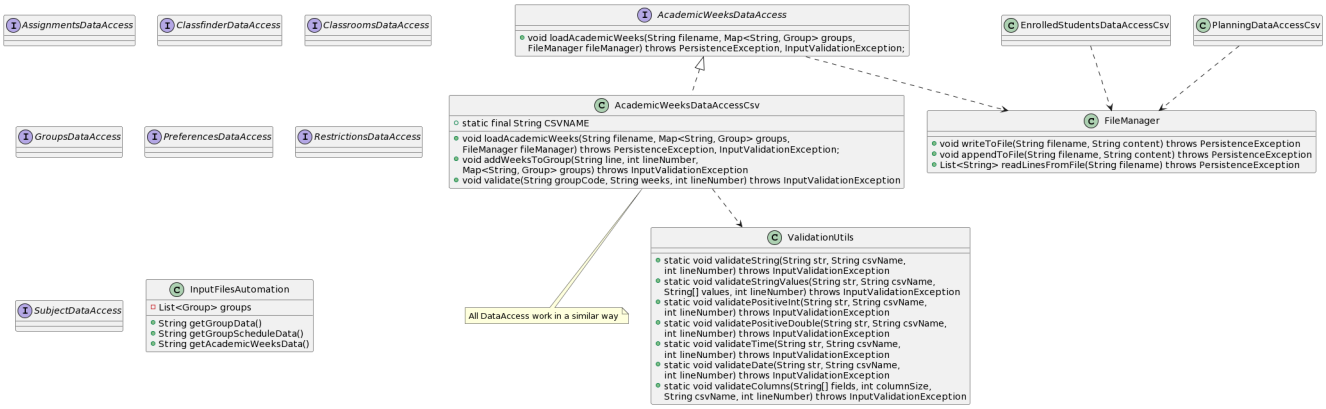
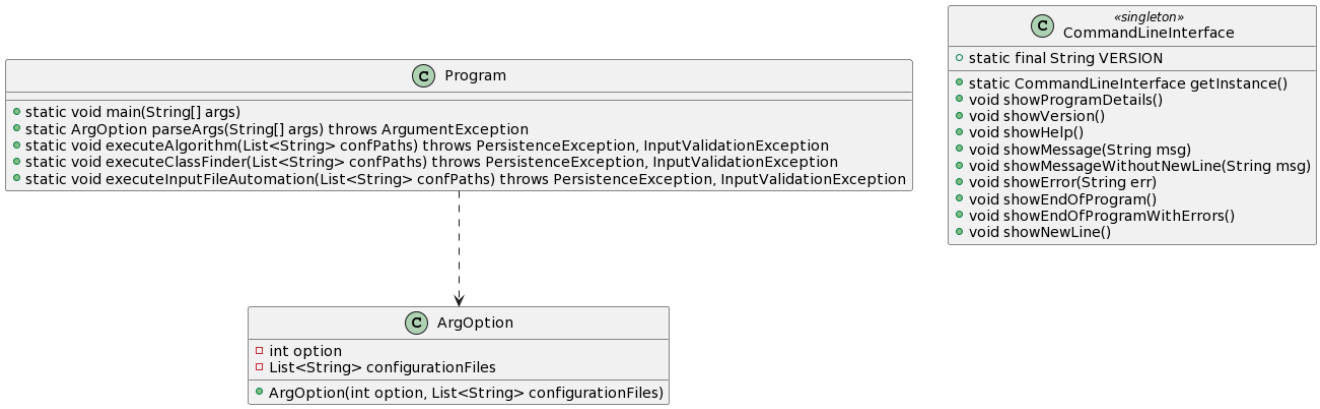


Figure 7.9: Class diagram: Program and CLI



7.3 File format design

We can differentiate files into system files and external files. System files are those that we have originally defined in order for the system to function correctly. External files are those files whose existence precedes this project.

All files are plain text files, and can have the extensions CSV, TXT or PROPERTIES. TXT files contain elements of the solution that are understandable by the user but not processable by the system. CSV and PROPERTIES files are usually input files, in most cases, or output files, in some other cases.

It should be noted that the utility can process as many PROPERTIES files as the user wants. Since these configuration files contain keys and values, the user is free to define them in as many PROPERTIES files as desired, as long as all the required keys are present.

In the case of CSVs, it is expected that these are always separated by semicolons, and that they always include a header with the column names, as the utility always ignores the first line except in special cases. Empty lines will cause errors when parsing, so they must be removed.

All the files involved in the system are now displayed. For real examples, the reader is referred to Annex 14.3.4.

Input / Output:

- Classrooms (System CSV file)
- Subjects (System CSV file)
- Groups (System CSV file)
- Group schedules (System CSV file)
- Group academic weeks (System CSV file)
- Preferences (System CSV file)
- Restrictions (System CSV file)
- Assignments (System CSV file)

- Assignments Summary (System TXT file)
- Classroom Timetable (System TXT file)
- Classfinder queries (System CSV file)
- Classfinder query results (System TXT file)
- Plan file (External CSV file): It must be manually edited to change the separator to a semicolon and remove its empty lines.
- Enrolled students (External CSV file): It must be manually created from the Excel file with the enrolled student tables.

Configuration:

- Algorithm configuration (System PROPERTIES file): for the *perform the assignments* use case.
- Classfinder configuration (System PROPERTIES file): for the *search for free classrooms* use case.
- Automation configuration (System PROPERTIES file): for the *automatically create input files* use case.

It is again stressed that these configuration files are an agglomeration of all the keys needed for each use case. If the user prefers to split it into other files, as long as they contain all the keys between them, they can do so.

8 System implementation

8.1 License and references

8.1.1 License

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

8.1.2 References

- *Java Code Conventions*. Set of guidelines and conventions for programmers to consider when using the Java programming language.
- *Linux kernel coding style*. [1] Set of guidelines and conventions for programmers to consider when programming in the Linux kernel. The stylistic choices in this guide have, for the most part, been the ones adopted for formatting the prototype code, as we believe that the readability of the code is preferable to the Java Code Conventions guidelines.

8.2 Programming languages

The use of C or Java for programming the system was discussed. In the end we opted for Java for two reasons. The first is simple, I, the developer, have more experience in Java than in C (although I have used both in this School). Perhaps C would be a suitable language to implement the algorithms described in this document more efficiently, but the extra time I would have to spend learning the language in an advanced way makes it unmanageable for this project. The second reason is also obvious. If this project is to be continued by our colleagues at the School, it would be preferable if it were written in the language that has been learnt throughout the degree courses, i.e, Java.

Apart from Java, the experimentation phase has made use of the Python language to extract the data from the executions and to present this data in a useful manner for further analysis.

[1] Available at https://www.kernel.org/doc/Documentation/process/coding-style.rst

8.3 Tools and programs used in development

8.3.1 Git

Git is a distributed version control software created by Linus Torvalds. It is used in this project to control changes to the code and documentation thanks to two repositories hosted on Github servers.

8.3.2 Vim

Vim is a text editor created by Bram Moolenaar. In this project Vim is used to write the documentation, define the system files and any other non-Java code.

8.3.3 LaTeX

LaTeXis a software system used for creating documents. We made use of this technology to create the documentation from scratch, inspired by other templates such as those mentioned at the beginning of the document. One of the main advantage of using LaTeXto create the documents is the possibility of integrating it into a version control system such as Git, which we have done in this project.

8.3.4 Eclipse

Eclipse is an integrated development environment (IDE) for Java. The vast majority of the system was programmed using this tool, employing its debugging tools and a plugin that emulates the behaviour of Vim in the Eclipse editor.

8.3.5 PlantUML

PlantUML is a UML diagram generator derived from plain text. It has been used for all diagrams in the documentation, as well as for creating the WBS and Gantt charts.

9 Test development

As previously stated, the testing phase will focus on checking the correct functioning of the system, especially in the persistence layer. The overall quality of the solutions is assessed in the experimentation stage.

In this chapter, a battery of tests is presented for each use case, each test containing the following information: ID, description, expected outcome, real outcome.

9.1 Perform the assignments

- **ID**: PA-01-NOF
  - **Description**: The utility is executed with a configuration that disables the optional files.
  - **Expected outcome**: The assignment process will start from scratch and no restrictions or preferences are to be considered.
  - **Real outcome**: OK
- **ID**: PA-02-PREFS
  - **Description**: The utility is executed with a configuration that considers preferences.
  - **Expected outcome**: The assignment process will start from scratch and no restrictions are to be considered. Positive and negative preferences will be evaluated as soft constraints.
  - **Real outcome**: OK
- **ID**: PA-03-RES
  - **Description**: The utility is executed with a configuration that considers restrictions.
  - **Expected outcome**: The assignment process will start from scratch and no preferences are to be considered. Positive and negative restrictions will be evaualted as hard constraints.
  - **Real outcome**: OK

- **ID**: PA-04-PAR
  - **Description**: The utility is executed with a configuration that considers a partial list of assignments previously made by the user of the system.
  - **Expected outcome**: The assignment process will start from the initial assignments and no preferences or restrictions are to be considered. The initial assignments will remain as they were once the allocation process is completed.
  - **Real outcome**: OK
- **ID**: PA-05-CONF
  - **Description**: The utility is executed with a configuration file with missing properties.
  - **Expected outcome**: The system will notify the error to the user.
  - **Real outcome**: OK
- **ID**: PA-06-INFOR
  - **Description**: The utility is executed with a configuration that points to input files with incorrect format.
  - **Expected outcome**: The system will notify the error to the user.
  - **Real outcome**: OK
- **ID**: PA-06-INDATA
  - **Description**: The utility is executed with a configuration that points to input files with incorrect information (e.g a group schedule with a reference to a group code that is not present in the system).
  - **Expected outcome**: The system will notify the error to the user.
  - **Real outcome**: OK

9.2 Search for free classrooms

- **ID**: SF-01-THEO
  - **Description**: The utility is executed with a configuration that points to a query file that contains one request for finding a theory classroom.

  - **Expected outcome**: The system will look for theory classes that meet the requirements of the query.
  - **Real outcome**: OK

- **ID**: SF-02-LAB
  - **Description**: The utility is executed with a configuration that points to a query file that contains one request for finding a laboratory.
  - **Expected outcome**: The system will look for laboratories that meet the requirements of the query.
  - **Real outcome**: OK

- **ID**: SF-03-ALL
  - **Description**: The utility is executed with a configuration that points to a query file that contains two requests. One to consult free classrooms and another to consult free laboratories.
  - **Expected outcome**: The system will look for theory classrooms for one request and laboratories for the other. Each search must meet the requirements of its associated query.
  - **Real outcome**: OK

- **ID**: SF-04-CONF
  - **Description**: The utility is executed with a configuration file with missing properties.
  - **Expected outcome**: The system will notify the error to the user.
  - **Real outcome**: OK

- **ID**: SF-05-INFOR
  - **Description**: The utility is executed with a configuration that points to input files with incorrect format.
  - **Expected outcome**: The system will notify the error to the user.
  - **Real outcome**: OK

- **ID**: SF-06-INDATA

  - **Description**: The utility is executed with a configuration that points to input files with incorrect information (e.g a group schedule with a reference to a group code that is not present in the system).
  - **Expected outcome**: The system will notify the error to the user.
  - **Real outcome**: OK

9.3 Automatically create input files

- **ID**: AC-01-AUTO
  - **Description**: The utility is executed with the correct configuration.
  - **Expected outcome**: The system will generate the input files necessary for the GA execution.
  - **Real outcome**: OK
- **ID**: AC-02-CONF
  - **Description**: The utility is executed with a configuration file with missing properties.
  - **Expected outcome**: The system will notify the error to the user.
  - **Real outcome**: OK
- **ID**: AC-03-INFOR
  - **Description**: The utility is executed with a configuration that points to input files with incorrect format.
  - **Expected outcome**: The system will notify the error to the user.
  - **Real outcome**: OK
- **ID**: AC-04-INDATA
  - **Description**: The utility is executed with a configuration that points to input files with incorrect information (e.g a record of enrolled students of a group that does not exist in the planning file).
  - **Expected outcome**: The system will notify the error to the user.
  - **Real outcome**: OK

10 Experimental results

This chapter shows the experimental study carried out to evaluate the quality of the prototype and its algorithms. We advance the main thesis we have arrived at after this study: The greedy algorithm is able to solve the problem without the help of the genetic algorithm. However, it is not powerful enough to find the best solutions obtained with other configurations. Only when coupled with the best version of the genetic algorithm does the software perform at its best.

The experiments were run on a Linux cluster (Intel Xeon 2.26 GHz, 128 GB RAM).

10.0.1 Instances

Four scenarios have been identified for this phase. One scenario is a combination of a group loading level and a constraint loading level. We have groups from the first and second semester of two different academic years, and we have restrictions and preferences obtained from client meetings.

A charge level has two states: regular and charged. Regular groups are simply those found in the academic course planning. Regular constraints are those indicated by the clients. Loaded groups introduce new groups artificially, generating them automatically by code. The same applies to the constraints.

For the charged load level, one group per subject was created for each instance with a 10% probability, and one restriction or preference with a 20% probability. Once created, the results were checked for incorrect data or contradictions.

We therefore have the following scenarios:

- **Charged groups - Charged constraints**
- **Charged groups - Regular constraints**
- **Regular groups - Charged constraints**
- **Regular groups - Regular constraints**

Each scenario has instances created from the first and second semester of the academic years 20-21 and 21-22.

10.0.2 Fitness function

The fitness function is created based on the client's expectations of results. The highest weight is given to all assignments being performed, followed by the fulfilment of preferences, then the remaining fitness values.

The values for the fitness function follow.

```
# Fitness weights

# COL_WEIGHT: Weight for the collisions fitness value
COL_WEIGHT = 1.0

# FREE_LABS_WEIGHT: Weight for the free labs fitness value
FREE_LABS_WEIGHT = 0.25

# LANG_WEIGHT: Weight for the group language fitness value
LANG_WEIGHT = 0.25

# SHARED_LABS_WEIGHT: Weight for the shared labs fitness value
SHARED_LABS_WEIGHT = 0.25

# SHARED_THEORY_WEIGHT: Weight for the shared theory classes fitness value
SHARED_THEORY_WEIGHT = 0.25

# PREFS_WEIGHT: Weight for the preferences fitness value
PREFS_WEIGHT = 0.5
```

10.0.3 Greedy Algorithm

With this fitness function, we proceed to experiment with the following versions of the greedy algorithm:

- **Base greedy**: Algorithm without repairs and biases.
- **Base greedy + Repairs**: Algorithm with repairs but without biases.
- **Base greedy + Repairs + Biases**: Algorithm with repairs and biases towards preferences and shared classrooms.

Experiments are launched ten times per run. Each run executes the greedy algorithm another ten times and returns the best result. An overview of the results will now be given.

Each run stores a number of metrics related to the fitness values. These metrics are listed and defined below.

- Average CPU time.
- Best fitness.
- Average fitness.
- Assignments without classroom.
- Average number of free labs by hour.
- Preferences met.
- Fitness which evaluates whether the Spanish and English groups go to different classes.
- Fitness that assesses whether the laboratory groups of a subject go to the same laboratory.
- Fitness which measures whether theory groups with the same name and course go to the same classroom.

10.0.3.1 Fitness assesment

Although the greedy does not factor in fitness to execute its operations, it can be an indicative value of the quality of a version of the algorithm.

The inclusion of the repair process does not seem to influence the results of the greedy algorithm a lot. However, the biases introduce a substantial improvement, as can be seen in the following summary table of the best fitness values (see Table 10.1).

Table 10.1: Greedy algorithm: Best fitness summary

Version	Best fitness summary
Base greedy	149.31
Greedy + Repairs	149.81
Greedy + Rep + Bias	166.69

10.0.3.2 Other metrics

Evaluating the results, we can see a clear difference between the behaviours of these three versions of the greedy algorithm in terms of assignments and preferences. The algorithms behave similarly in terms of number of unassigned groups (see Tables 10.2 and 10.3).

Table 10.2: Greedy algorithm: Unassigned groups for instance rr_20_21_s1

RegGroups-RegConstraints Course 20-21 Semester 1	
Version	**Average groups unassigned (out of 340)**
Base greedy	11
Greedy + Repairs	9
Greedy + Rep + Bias	8

Table 10.3: Greedy algorithm: Unassigned groups for instance cr_21_22_s2

CharGroups-RegConstraints Course 21-22 Semester 2	
Version	**Average groups unassigned (out of 314)**
Base greedy	6
Greedy + Repairs	5
Greedy + Rep + Bias	5

However, when we look at the preferences respected by each version, the picture changes. There is a clear advantage of the algorithm with bias over the other two versions. The tables for the previous two instances are now shown except that they now refer to preferences (see Table 10.4 and 10.5).

Table 10.4: Greedy algorithm: Preferences met for instance rr_20_21_s1

RegGroups-RegConstraints Course 20-21 Semester 1	
Version	**Average preferences met (out of 18)**
Base greedy	1
Greedy + Repairs	1
Greedy + Rep + Bias	10

Table 10.5: Greedy algorithm: Preferences met for instance cr_21_22_s2

CharGroups-RegConstraints Course 21-22 Semester 2	
Version	**Average preferences met (out of 43)**
Base greedy	10
Greedy + Repairs	12
Greedy + Rep + Bias	17

In the shared classroom metrics, an improvement is also observed in the algorithm with bias, since its heuristic contemplates this situation. In the rest of the metrics the difference is not so pronounced

10.0.4 (Genetic + Greedy) parameter assesment

Considering the results of the previous section, the greedy algorithm with repairs and bias is chosen for the genetic algorithm experiments. In this section we will describe the tests performed to tune the GA parameters.

Twenty-seven combinations were considered for the Course 20-21 instances. The values chosen are as follows.

Population size: $\{100, 250, 500\}$

Crossover probability : $\{0.8, 0.9, 1\}$

Mutation probability : $\{0.05, 0.1, 0.2\}$

The results are summarised in Table 10.6.

Table 10.6: (Genetic + Greedy): Parameter combinations results (Best fitness)

Comb	rr_20_21_s1	rr_20_21_s2	rc_20_21_s1	rc_20_21_s2	cr_20_21_s1	cr_20_21_s2	cc_20_21_s1	cc_20_21_s2
Comb01	191.58	175.19	179.98	174.46	190.14	174.1	178.63	172.49
Comb02	192.33	176.0	180.64	174.27	189.78	173.92	179.36	172.99
Comb03	192.07	175.24	180.13	174.66	190.53	174.33	179.95	173.12
Comb04	193.07	175.63	180.66	174.67	190.03	174.96	179.23	173.7
Comb05	193.34	176.18	180.81	175.13	190.32	174.33	179.4	173.1
Comb06	192.44	175.46	180.99	175.05	190.96	174.31	179.55	173.39
Comb07	193.05	176.07	181.08	174.27	190.59	174.51	179.89	174.24
Comb08	193.07	176.08	181.14	175.2	190.99	175.18	180.11	174.24
Comb09	192.93	176.07	180.87	174.84	190.73	174.56	179.69	174.12
Comb10	191.69	174.86	179.93	174.2	190.01	173.67	179.14	172.54
Comb11	192.89	174.84	180.47	175.28	190.13	174.37	179.61	173.54
Comb12	192.46	175.05	180.99	174.55	190.19	174.24	179.8	172.72
Comb13	192.81	175.43	181.41	174.51	190.43	173.99	180.16	173.81
Comb14	193.03	176.35	180.57	174.73	190.96	174.44	179.64	173.64
Comb15	192.67	175.43	180.22	174.51	191.17	175.48	179.92	174.95
Comb16	193.31	175.75	180.81	175.65	190.9	174.6	179.28	174.17
Comb17	193.19	176.18	180.47	176.19	191.14	175.08	180.39	173.85
Comb18	192.9	175.88	180.78	174.96	190.83	174.91	179.87	173.99
Comb19	191.79	175.06	180.65	175.07	189.48	175.37	179.44	172.87
Comb20	193.41	175.5	180.65	174.8	190.26	174.62	179.83	172.98
Comb21	192.89	175.35	180.14	174.08	190.55	174.23	179.89	173.02
Comb22	192.47	175.65	180.31	174.98	190.68	175.04	179.91	173.68
Comb23	193.2	175.9	180.87	174.54	190.98	174.39	180.21	173.45
Comb24	191.95	175.54	180.95	174.31	190.18	174.13	179.72	172.93
Comb25	192.38	175.64	180.83	174.85	190.68	175.96	180.05	173.84
Comb26	192.86	176.26	181.85	175.44	190.91	174.69	179.46	173.32
Comb27	192.73	175.55	181.08	173.75	190.22	174.83	179.76	173.16

We decided on combination 17 as it had the best fitness results, although there is not much of a difference between the different configurations. The content of this combination are shown below.

```
POP_SIZE = 250
CROSS_PROB = 0.9
MUTA_PROB = 0.2
```

10.0.5 Greedy against (Genetic + Greedy)

We will now use the parameters in the combination 17 for the GA with the initial fitness function, testing the three versions of the greedy algorithm in combination with the genetic algorithm. For this comparison, we will use the previous tables, adding the values obtained with the combination of both algorithms.

We begin by showing the values of the best fitness found (see Table 10.7).

Table 10.7: Greedy against (Genetic + Greedy): Best fitness summary

Version	Best fitness summary
Base greedy	149.31
Genetic + Base greedy	169.50
Greedy + Repairs	149.81
Genetic + (Greedy + Repairs)	168.75
Greedy + Rep + Bias	166.69
Genetic + (Greedy + Rep + Bias)	179.63

Next, the two algorithms will be compared on the other metrics. The four tables in the greedy algorithm evaluation are summarised in two tables (See Tables 10.8 and 10.9).

Table 10.8: Greedy against (Genetic + Greedy): Metrics for instance rr_20_21_s1

RegGroups-RegConstraints Course 20-21 Semester 1		
Version	**Average groups unassigned (out of 340)**	**Average preferences met (out of 18)**
Base greedy	11	1
Genetic + Base greedy	7	7
Greedy + Repairs	9	1
Genetic + (Greedy + Repairs)	6	6
Greedy + Rep + Bias	8	10
Genetic + (Greedy + Rep + Bias)	6	13

Table 10.9: Greedy against (Genetic + Greedy): Metrics for instance cr_21_22_s2

CharGroups-RegConstraints Course 21-22 Semester 2		
Version	**Average groups unassigned (out of 314)**	**Average preferences met (out of 43)**
Base greedy	6	10
Genetic + Base greedy	4	18
Greedy + Repairs	5	12
Genetic + (Greedy + Repairs)	5	18
Greedy + Rep + Bias	5	17
Genetic + (Greedy + Rep + Bias)	4	18

From these tables we can conclude that the greedy algorithm still fares very well when compared to the combination of the greedy and genetic algorithms, but that ultimately the best option is to combine the two.

10.0.6 Further fitness functions

As a final experiment, we decided to perform a comparison between different values of the fitness weights. For this we considered four scenarios:

- **Shared labs and theories**: In this scenario we test whether the genetic algorithm achieves better results by ignoring language separation and focusing on clustering groups in the same classrooms, according to the criteria set out earlier in this document.
- **Group separation by language**: The reverse of the first scenario. Here we focus on separating the groups by language, ignoring whether or not they are grouped in the same classrooms.
- **Equal weights**: All fitness value have a weight of one.
- **Weights for fitness values of collisions and preferences**: Only the unassigned classrooms (collisions) and the preferences get a weight value of 1.0 and 0, respectively. The rest are set to zero.

Analysing the results, we can see that the worst configuration is the one where the weights are equal. Granted, it has higher fitness, but in the other metrics it is far inferior to the rest of the configurations.

Naturally, scenario 2 (group separation by language) ends up with results whose language fitness is higher than scenario 1 (shared labs and theories), but lower fitness in shared classes compared to the latter. In terms of assignments without classrooms and fulfilled preferences, both configurations work in a similar way.

If we compare scenarios 1 and 2 with scenario 4, we see that the difference is not so overwhelming as to justify not assigning weights to language or shared class values.

10.0.7 Experimental findings

Reflecting on the previous experiments, one can come to a conclusion that was already anticipated at the beginning of the chapter. This problem is not so complex, therefore, a greedy algorithm with a good heuristic can achieve an acceptable solution in short times.

But, in order to really obtain the best possible solution with these algorithms, it is necessary to use the genetic algorithm with the right combination of parameters and weights.

11 System manuals

11.1 Installation manual

SETUP

In order to be able to execute the **classmanager** utility, two things are required. The first is Java 8 or higher, the second is the creation of the runtime environment. This environment simply consists of having the utility at the same level (in terms of folders) as a folder that the user must create, called **classmanager_log**.

FOLDER TREE

When we run the jar file, we will do it from this folder. If the configuration files have relative paths, they must be relative to the jar file. An example of this is shown below.

Classmanager/ (FOLDER)

classmanager.jar (Executable JAR)

classmanager_log/ (Required FOLDER)

Some logs...

inputfiles/ (User FOLDER)

classrooms.csv

...

output/ (User FOLDER)

config/ (User FOLDER)

myConfig.properties

We have the previously described folder in our computer. Let's say we are creating the configuration file *myConfig.properties*, and we want to give a path to the CLASSROOMS_FILE_PATH property. Should we specify the relative path from the configuration file or from the jar? As said before, the path is **relative to the jar file**, so in our case, `CLASSROOMS_FILE_PATH = ./inputfiles/classrooms.csv`.

11.2 Execution manual

NAME

classmanager - manages the classrooms and groups of the School of Computing Engineering of Oviedo.

SYNOPSIS

classmanager *OPTION* [*FILE*...]

DESCRIPTION

The *classmanager* utility assigns classes to groups in the School for a given semester. In addition, it allows queries to be made on the classrooms to obtain those available in a range of dates and schedules. Finally, the utility has a built-in tool that allows for automatic conversion of the files currently used by the school into files compatible with the system.

A *FILE* represents a configuration file. Each option requires a set of properties with values of different types. As long as these properties are given to the utility, the user is free to split them in as many configuration files as they want. If a file is not provided, or if the file cannot be parsed due to lack of permissions, the utility will notify the user of the error. The same goes for the required data not submitted in the configuration files.

OPTIONS

Generic Program Information

-h, --help

Output a usage message and exit.

-v, --version

Output the version of **classmanager** and exit.

Functionalities

-a, --algorithm

Perform the assignments, output the result into the expected files and exit.

-q, --query

Search for available classrooms, output the results into the expected files and exit.

-t, --transform

Transform the School files into compatible files, output the generated files and exit.

REGULAR EXPRESSIONS

Three types of regular expressions designed for this project can be used in the *PREFERENCES* and *RESTRICTIONS* files. The three types are taken into account when using them in the name section of a group's code (*subject.type.**name***).

The first type involves using the asterisk (*) to indicate that the preference or restriction applies to both English and Spanish groups of a particular type of class (e.g. CVVS.L.*).

The second type uses the addition (+) to indicate that the preference or restriction should apply only to the Spanish groups (e.g. CVVS.L.+).

The last type employs the question mark (?) in the same way as the second type except that the groups concerned are English groups (e.g. CVVS.L.?).

EXIT STATUS

The program can terminate in two states: *OK* and *ERROR*. The OK status indicates that the execution was completed without problems, while the ERROR status implies that the operation could not be carried out as expected due to an error. The detailed error is recorded in the *LOG* file generated by the utility.

NOTES

Make sure that all CSV files have a header with the column names, because the program expects such a header to exist. Otherwise it will skip the first row of the CSV. Also, the CSV files cannot have empty rows or the utility will complain. Finally, the utility expects the separator of *all* CSVs to *always* be the semicolon.

COPYRIGHT

classmanager - manages the classrooms and groups of the School of Computing Engineering of Oviedo. Copyright (C) 2022 Hugo Fonseca Díaz

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.

BUGS

Occasionally an error message will be displayed *after* the message indicating that the program has terminated with error status. It should appear *before* this message. However, repeating the execution may make it appear where expected.

If you find a bug, you can create an Issue in this repository or send an email to UO258318 AT uniovi DOT es.

EXAMPLES

Show help:

```
java -jar classmanager.jar -h
```

```
java -jar classmanager.jar --help
```

Show version:

```
java -jar classmanager.jar -v
```

```
java -jar classmanager.jar --version
```

Performing the assignments:

```
java -jar classmanager.jar -a algFolder/algorithm.properties
                ioFolder/io.properties
```

Note that the user has decided to split the properties in two files, but they could have used only one if they wanted.

Another way of performing the assignments:

```
java -jar classmanager.jar --algorithm all.properties
```

Searching for classrooms:

```
java -jar classmanager.jar -q classfinder.properties
```

```
java -jar classmanager.jar --query one.properties
                two.properties three.properties
```

Transforming the files of the School into the files used by the utility:

```
java -jar classmanager.jar -t automation.properties
```

```
java -jar classmanager.jar --transform automation.properties
```

SEE ALSO

You can view the updated documentation at this repository. If you wish to view the code, you can do so by visiting this other repository.

11.3 User manual

This manual explains the different files involved in the three possible use cases. See Annex 14.3.4 for examples of each use case.

PERFORM THE ASSIGNMENTS

CONFIG

A default configuration is provided for the greedy algorithm and for the genetic algorithm. The only notable thing about this property file is that you can specify the path to the optional files but disable their use with boolean keys.

INPUT

Files submitted for classes and subjects can be reused. It is important to keep the headers of each CSV file and not to have empty rows, as discussed in the execution manual.

If a group or subject code is referenced in another csv file, a check will be made to see if that code exists. However, the codes will not be checked to see if they are well-formed in the original files, so special care must be taken.

The format of dates and times should also be taken into account, so it is recommended to study the files already submitted for reference. The utility will check if they are incorrectly formatted and will warn the user of any problems.

If you start from an existing mapping file, note that there can be no classless mappings or the utility will interpret this as an error when parsing the file.

OUTPUT

Several types of files are generated. It can be categorised into three types, the csv file of assignments, the txt file with the summary of assignments and the files with the timetable in tabular form.

The summary file indicates the assignments sorted by course and subject. The timetables show in table format the timetable of each class. If there is more than one group in a cell, it means that both groups attend classes at the same time, but

it is understood that they never overlap in their academic weeks.

SEARCH FOR FREE CLASSROOMS

INPUT

As the queries are entered via CSV, as many queries as the user wants can be introduced. Care should be taken not to enter a start date that is later than the end date.

OUTPUT

A txt file is generated. This file contains, for each query, its parameters and results. The results include a class, a week, a day and a series of ranges of free slots for that classroom.

AUTOMATICALLY CREATE INPUT FILES

INPUT

Input files must be correctly adapted to the system. This involves using the semicolon separator and deleting all empty lines. A header must also be created for both files (see example files provided in the Annex).

OUTPUT

The group, group schedules and group academic weeks are generated. Since the system does not contain the subjects in memory for this use case, the entries are sorted by the code of the groups.

11.4 Programmer manual

LOG

```
DETALLADO: START Persistence logic
jul 05, 2022 12:23:15 AM main.Program main
GRAVE: Wrong integer in GROUPS csv file (cero or negative: 0), line 42
business.errorhandler.exceptions.InputValidationException: Wrong integer in GROUPS csv file (cero or negative: 0), line 42
        at persistence.problem.csv.utils.ValidationUtils.validatePositiveInt(ValidationUtils.java:56)
        at persistence.problem.csv.GroupsDataAccessCsv.validate(GroupsDataAccessCsv.java:106)
        at persistence.problem.csv.GroupsDataAccessCsv.lineToGroup(GroupsDataAccessCsv.java:51)
        at persistence.problem.csv.GroupsDataAccessCsv.loadGroups(GroupsDataAccessCsv.java:29)
        at main.Program.executeExperiments(Program.java:855)
```

```
        at main.Program.main(Program.java:94)

jul 05, 2022 12:24:50 AM main.Program main
DETALLADO: START Persistence logic
jul 05, 2022 12:24:50 AM main.Program main
GRAVE: Non existing code for group in PREFERENCES csv file (CPM.L.+), line 6
business.errorhandler.exceptions.InputValidationException: Non existing code for group in PREFERENCES csv file (CPM.L.+), line 6
        at persistence.problem.csv.PreferencesDataAccessCsv.lineToPreferences(PreferencesDataAccessCsv.java:97)
        at persistence.problem.csv.PreferencesDataAccessCsv.loadPreferences(PreferencesDataAccessCsv.java:34)
        at main.Program.executeExperiments(Program.java:883)
        at main.Program.main(Program.java:94)

jul 05, 2022 12:25:21 AM main.Program main
DETALLADO: START Persistence logic
jul 05, 2022 12:25:21 AM main.Program main
DETALLADO: END Persistence logic
jul 05, 2022 12:25:21 AM main.Program main
DETALLADO: START Business logic
jul 05, 2022 12:25:21 AM business.alg.gen.logic.GeneticAlgorithm geneticAlgorithm
DETALLADO: START Genetic Algorithm
jul 05, 2022 12:25:21 AM business.alg.gen.logic.GeneticAlgorithm geneticAlgorithm
MUY DETALLADO: Parameters:

        -> Max number of generations: 10
        -> Max time (ms): 360000
        -> Mutation probability: 0.5
        -> Crossover probability: 0.9
        -> Population size: 200
        -> Fitness function: business.alg.gen.logic.fitness.DefaultFitnessFunction
        -> Individual length: 311
jul 05, 2022 12:25:29 AM business.alg.gen.logic.GeneticAlgorithm geneticAlgorithm
DETALLADO: END Genetic Algorithm
jul 05, 2022 12:25:29 AM main.Program main
DETALLADO: END Business logic
```

The above is an extract from the log file. We can see a trace of the flow of the application in three different executions. The first two fail, each with a different error, in the groups and preferences files. The last one occurs after the errors have been fixed and the parameters of the genetic algorithm used for the execution can be observed. Once the execution of the genetic is finished, an entry indicating this fact and another one reporting the completion of the business logic are created.

UTILITIES

CSV and problem domain validation utilities can be used to lighten and de-code new implementations.

If a new exception type is to be introduced, it must be defined in the business layer error handler package and specified as an expected exception by instructing the ErrorType.

NOTES

Some notes follow.

- New use cases should be executed in the Program class as a unique function with a series of steps (see the other use cases).
- All printing to the terminal must be via the CommandLineInterface class.
- Changes to the problem domain classes and to the utility classes should be reviewed carefully, as much of the system depends on such classes.
- If more GA operators are to be developed, a Strategy Design Pattern approach, similar to the FitnessFunction, is recommended.
- New fitness values for the GA should follow the current procedure of returning a value between `0` and `100`.
- If data collections are shared between objects, copies of the collections must be shared. If the objects in these collections are to be modified, the shared collection must contain cloned objects (this can be seen in the Decoder class).

12 Budget

This chapter presents the budget for the work. First the internal budget will be calculated taking into account the environment in which we work, and then the budget of the client will be prepared with the necessary items and the benefits applied.

The working time concept is reiterated. Each week has seven working days of three hours each. It is important to bear in mind this concept of working day in order to understand some sections of the budget.

12.1 Internal budget

To calculate the internal budget we must first define our situation. We are two freelancers, a junior and a senior software engineer (see Figure 12.1).

Figure 12.1: Freelancers description

Role name	Role ID	Units	Salary (hour)	Taxes	Cost per hour
Junior Software Eng.	J	1	12,63 €	30,00%	16,42 €
Supervisor Software Eng.	S	1	18,94 €	30,00%	24,62 €

We have the following rental and amortisation expenses related to the project (see Figure 12.2).

Figure 12.2: Amortisation costs

Product	Type	Period (years)	Unit	Quantity	Cost (per unit)	Total cost
Laptop	Amortisation	4	Month	5	12 €	60 €
Computer for experimentation	Rental		Hour	21	2,50 €	52,50 €
						112,50 €

Indirect costs are shown below (see Figure 12.3).

Figure 12.3: Indirect costs

Product	Unit	Quantity	Cost (per unit)	Total cost
Electricity	Month	5	160 €	800 €
Internet	Month	5	50 €	250 €
Taxes	Month	5	300 €	1.500 €
Office supplies	Miscellaneous			70 €
				2.620,00 €

Finally, we calculate the costs of carrying out the tasks defined in the WBS.

Figure 12.4: WBS Budget costs: Project Management and Analysis

Task	Hours	Working days (3hours)	Worker	Cost (hour)
Project Management	**27**	**9**	**J**	443,31 €
Initial planning	9	3	J	147,77 €
Project definition meeting with stakeholders	3	1	J	49,26 €
Project monitoring and control	15	5	J	246,29 €
Analysis	**111**	**37**	**J**	1.822,51 €
Defining requirements	21	7	J	344,80 €
Reading on metauristics	30	10	J	492,57 €
Formalisation of the problem	9	3	J	147,77 €
Solution to the problem	24	8	J	394,06 €
Functional analysis	18	6	J	295,54 €
Definition of system tests	3	1	J	49,26 €
Definition of acceptance tests	3	1	J	49,26 €
CLI analysis	3	1	J	49,26 €

Figure 12.5: WBS Budget costs: Design and Development

Task	Hours	Working days (3hours)	Worker	Cost (hour)
Design	**27**	**9**	**J**	443,31 €
System architecture definition	12	4	J	197,03 €
Elaboration of the technical design	3	1	J	49,26 €
Migration design and initial loading	3	1	J	49,26 €
Structural definition of files	9	3	J	147,77 €
Development	**168**	**56**	**J**	2.758,39 €
Environment setup	6	2	J	98,51 €
Creation of the greedy algorithm	18	6	J	295,54 €
Creation of the genetic algorithm	15	5	J	246,29 €
Creation of the rest of the business layer	45	15	J	738,86 €
Creation of the persistence layer	30	10	J	492,57 €
Creation of the CLI	3	1	J	49,26 €
Automation of input files	9	3	J	147,77 €
Creation of the class finder	18	6	J	295,54 €
Preparation of the system for experiments	24	8	J	394,06 €

Figure 12.6: WBS Budget costs: Documentation, Experiments and Closure

Task	Hours	Working days (3hours)	Worker	Cost (hour)
Documentation	**27**	**9**	**J**	443,31 €
Preparation of the installation manual	3	1	J	49,26 €
Preparation of the execution manual	6	2	J	98,51 €
Preparation of the user manual	9	3	J	147,77 €
Preparation of the programmer manual	9	3	J	147,77 €
Experiments	**48**	**16**	**J**	788,11 €
Creation of the instances	12	4	J	197,03 €
Creation of configuration files	3	1	J	49,26 €
Execution of the experiments	21	7	S	517,06 €
Analysis of the experiments	12	4	J	197,03 €
Closure	**51**	**17**	**J**	837,37 €
Execution of final acceptance tests	9	3	J	147,77 €
Preparation of project closure documentation	42	14	J	689,60 €

So we are left with the following cost of the WBS phases (see Figure 12.7).

Figure 12.7: WBS Budget costs: Summary

Task	Hours	Working days (3hours)	Worker	Cost (hour)
Classmanager	**459**	**153**		7.536,32 €

With the addition of all the costs calculated so far, we obtain the internal budget for the project.

Figure 12.8: Internal budget

Item	Total
Project Management	443,31 €
Analysis	1.822,51 €
Design	443,31 €
Development	2.758,39 €
Documentation	443,31 €
Experiments	788,11 €
Closure	837,37 €
Indirect Costs	2.620,00 €
Amortisations	112,50 €
	10.268,82 €

12.2 Client budget

This project is expected to yield a profit of 25%. With this information we will proceed to calculate the client's budget. It is important to note that the client will not be

shown a budget with all the items, only the generic ones that describe the work to be done. Therefore, we must calculate the amount of the items not shown added to the benefits, and thus obtain a weighting value with which to calculate the client's final budget.

Figure 12.9 shows the calculation of this weighting value.

Figure 12.9: Weighting value calculation

Profit margin	25%
Internal budget	10.268,82 €
Profit	2.567,21 €
Items not displayed in the client budget	4.801,29 €
Profit + Items not displayed	7.368,50 €
Internal budget - Items not displayed	5.467,53 €
Weighting value	1,35

Each item is added to its own value multiplied by the weighting value and entered into the client's final budget (See Figure 12.10). This means that a cost c_i shown in the internal budget will be transformed into a new cost if the item appears in the client budget. This transformation is given by $x_i = c_i + c_i w$, where w is the weighting value.

Figure 12.10: Client budget

Item	Total
Analysis	4.278,68 €
Design	1.040,76 €
Development	6.475,83 €
Documentation	1.040,76 €
Classmanager	**12.836,03 €**

13 Conclusions and future work

13.1 Final conclusions

All the objectives set for the project have been met. A study and formalisation of the problem has been carried out and the problem was solved, both theoretically and experimentally, thanks to a software tool that was developed in this project.

The files necessary for the software to work have been designed and created. They are provided in the annexes, many of which can be reused by the user of the application. In addition, an experimental study has been conducted on the developed solution in order to provide the customer with the default configuration of the system to ensure quality results in various possible scenarios.

The developed system is licensed under a free software licence and may be modified and distributed by users who wish to do so.

With all this in mind, the project comes to a close with satisfaction on my part, although the work does not end here. In the next section, some possible lines for the continuation of the work developed in this project are explained.

13.2 Future work

Although the objectives of the project have been met, nothing changes the fact that the software designed in this document is a prototype. Of course, a fully usable prototype with interesting functionalities, but there is room for improvement. Listed below are some lines of research and development to be pursued on the basis of this project.

- **Expansion of the class search functionality to return results that support multiple events**. Currently the user can browse the results looking for free classes at the same time, but perhaps it would be useful to have a feature that allows the users to choose the number of classes they are seeking for an event that are available at the same time.
- **Increased validation of input files**. The current validation is perfectly valid,

and can help the user to find most of the problems. However, more specific checks such as the formatting of a group/class/subject code, that the end dates of queries are not less than the start dates, etc, would provide a better user experience.

- **Possibility of assigning different classes to each group depending on the day**. This is *very complicated*, as it affects the main pillar of the whole theory that a group can only have one classroom associated with it. However, if one wanted to rethink the problem from scratch, this approach could be taken into account. Hopefully, such an approach would serve to further reduce the number of unassigned groups.

- **Rewrite the code in a language such as C or Go, more focused on algorithms**. It should be noted that the code described here, as mentioned above, is only a prototype. It can and does work correctly, but if new functionalities appear that greatly affect the original design, one should not be afraid to rewrite the code, and in that case, another language could be used. Shorter computation times could be expected if the system is implemented in this kind of languages.

14 Annexes

14.1 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.

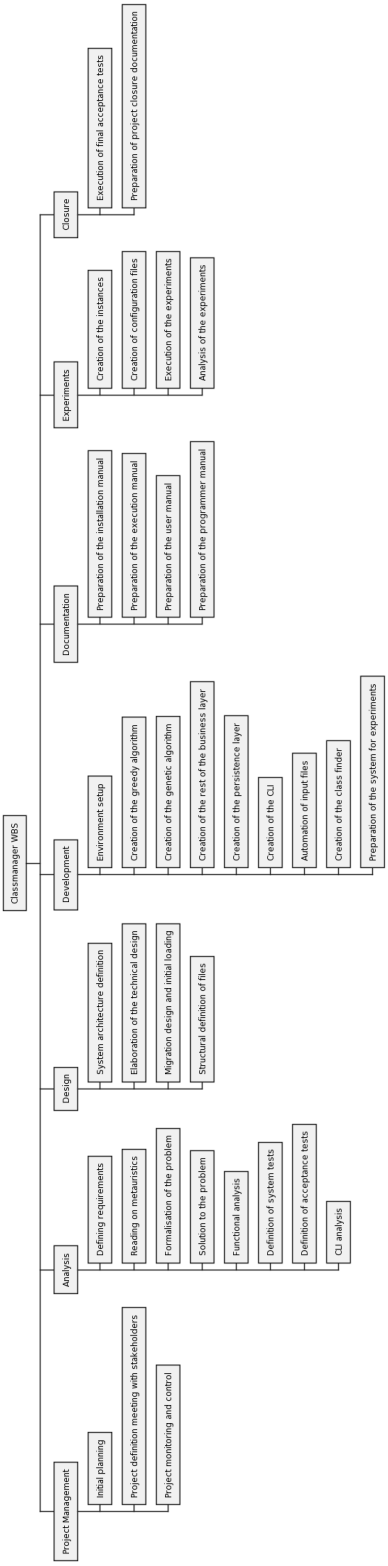
- **Greedy algorithm:** algorithm that builds the solution in successive steps, always trying to take the optimal solution for each step
- **Heuristic:** function that gives value to each path from a intermediate state to the goal state. Applied in search algorithms, heuristics are based on knowledge 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 4.2.1).
- **LFD:** Lazy Filter Dictionary (see 4.3.1).
- **UC:** Use case.
- **TXT:** Text. Refers to the text file format.
- **WBS:** Work Breakdown Structure.

14.2 Work Breakdown Structure (WBS)

Figure 14.1: WBS (Bigger scale)



14.3 Submission contents

14.3.1 Code

This Annex includes a compressed ZIP file with the source code of the prototype.

14.3.2 Runnable utility

This Annex includes the executable JAR file of the utility.

14.3.3 Experiment instances

This Annex includes the experimental files used for tuning the genetic algorithm configuration. It serves as an example to see the instances that we worked with.

14.3.4 System and external files

This Annex includes compressed ZIP file with examples of all files involved with the system. Also included is the JAR tool with instructions on how to run the various use cases (the configuration files are already prepared). Its structure is as follows.

annexfileformat/

- **ucassignments/**
  - **config/**
  - **input/**
  - **output/**
- **ucfreeclassrooms/**
  - **config/**
  - **input/**
  - **output/**
- **ucautomation/**
  - **config/**
  - **input/**
  - **output/**
- **classmanager_log/** (LOG examples)

classmanager.jar (Runnable JAR utility)

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.

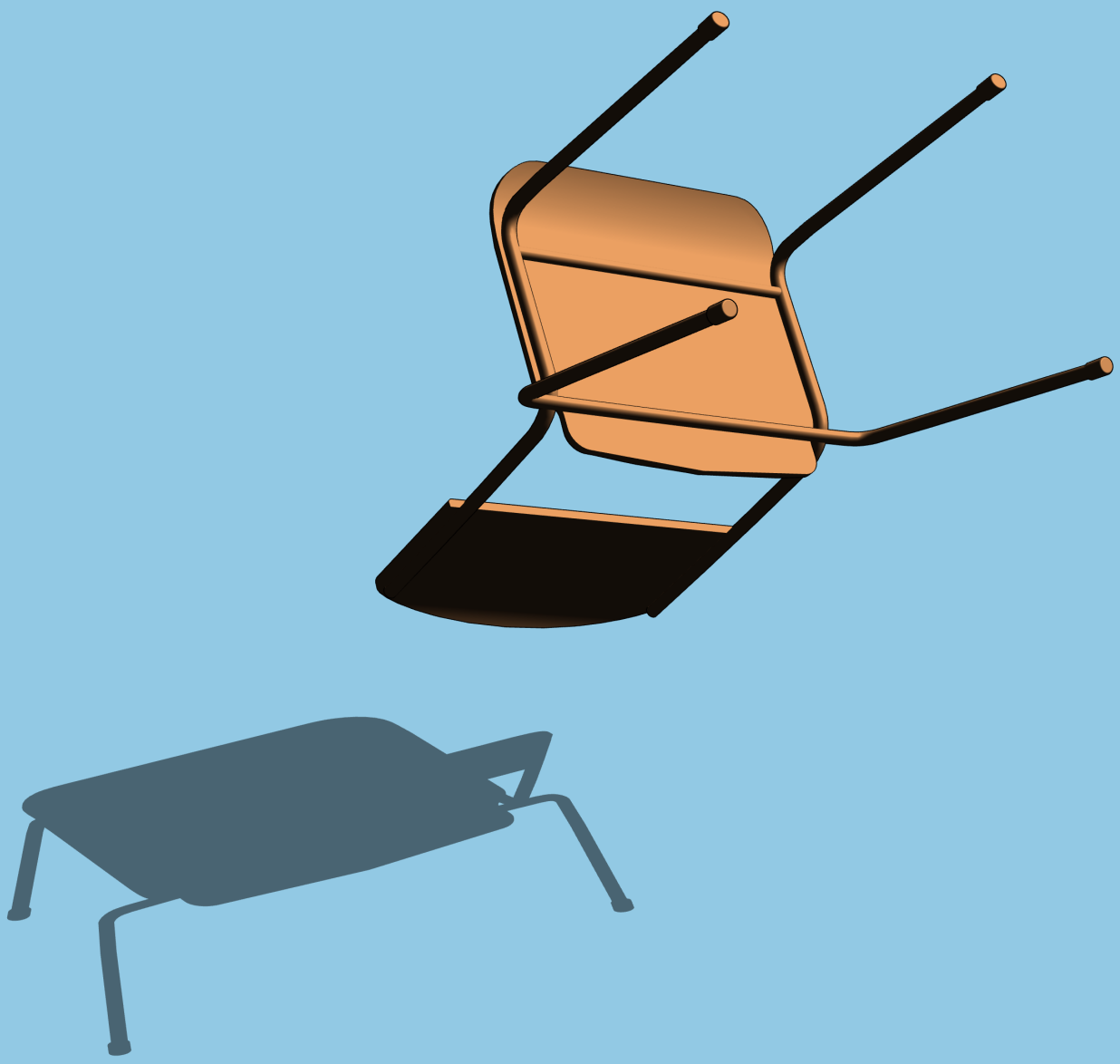
[GV98] Rosa Guerequeta and Antonio Vallecillo. *Técnicas de Diseño de Algoritmos*. Servicio de publicaciones de la Universidad de Málaga, 1998.

[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.



Universidad de Oviedo

Escuela de Ingeniería Informática
Universidad de Oviedo