KING'S College LONDON

University of London

Final Year Individual Project Automated Team Formation Using Modified Binary Particle Swarm Optimisation

Final Year Project

Author: Anton Pashov

Supervisor: Dr Jeroen Keppens

Student ID: 1642171

April 22, 2020

Abstract

Heterogeneous team formation is still a challenging NP-Hard problem, despite the extensive research performed on the topic. A suggested way of solving team formation problems are the stochastic optimisation methods based on the concept of swarm intelligence. Such a method is Particle Swarm Optimisation, proposed in 1995, it presents a promising and efficient way of optimising different functions. The current project presents a novel approach for solving the team formation problem using a specially modified version of the binary PSO algorithm. The new algorithm is highly customisable, provides mechanics of fine-tuning its behaviour and the focuses on balancing between exploration and exploitation during its execution. It relies on a custom adaptation of the prime PSO components while keeping their meaning and original mechanics. It uses dynamic inertia, changing topology and variable survivability principle inspired by the selectivity concept in Genetic Algorithms. The result is a proposed solution for a specific team formation problem, where a large set of university students have to be assigned to small teams for a very important project. The algorithm is also focused on scalability and provides the users with the full freedom of furtherly optimising it and adapting it to other similar problems.

Originality Avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary. I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this report does not exceed 25,000 words.

Anton Pashov

April 22, 2020

Acknowledgements

First of all, I would like to pay my special regards to my supervisor Dr Jeroen Keppens for his dedicated support and constructive feedback during my work on this project, as well as everybody else from the Faculty of Natural and Mathematical Sciences at King's College London, who was directly or indirectly part of my journey.

Further, I wish to express my deepest gratitude to my family and my friend Nikolay. Thank you for the constant support, guidance and especially for always being there for me. Without you, I would've never made it here.

Contents

List of Figures

Listings

Chapter 1

Introduction

Team formation and teamwork, in general, is a thoroughly studied topic in a wide variety of contexts. The concept of people working together in groups, aiming to reach a collective goal dates back to ancient times, and the aspects in which it was and is still used is enormous, ranging from work/business, through education, to sport and leisure. The first fundamental step towards successfully achieving a mutual goal from a group of people is forming the team. This is what is the main focus of the current project. This report presents in detail a novel approach for solving a team formation problem and its implementation. The software implementation is in the form a Ruby gem. The program gives an automated solution for assigning a comprehensive list of students into small groups of students with heterogeneous distribution of a range of characteristics for each individual, namely: gender, ethnicity and skill measure in the educational field of interest. It also forbids grouping together students, which the user specifies as 'forbidden pairs', i.e. students who for some reason(for example, having worked together on a past project) are said not to be put together. Furthermore, it provides the user with a significant potential for changing the algorithm and the forming criteria for his/her specific purpose, which is explained in the user manual. The implementation uses a modified version of the discrete binary particle swarm optimisation(BPSO) algorithm proposed by J. Kennedy and R.C. Eberhart in 1997[24], which is a successor of the original particle swarm optimisation(PSO) algorithm, invented by the same authors in 1995[23]. The BPSO was specifically modified to fit the purpose and needs of the current project. Concepts from the traditional versions of other evolutionary algorithms were implemented as well to increase performance and accuracy.

1.1 Audience and Benefficiaries

The project has to address a pretty specific domain of the team formation problem. It will focus on assigning university students from King's College London in their second year in the Informatics Department of the Faculty of Natural and Mathematical Sciences, who have a common software engineering group project of high importance in the academic year. Additionally, it should provide a solid base for further improvements and generalisation. The primary beneficiaries of the project are the project supervisors and of course, the students themselves. It provides a reliable mean of forming optimal groups for the important educational task for the people who are in charge of that. On the other hand, it will contribute to maximising the expected benefits each student will get from completing the largest and most demanding, but rewarding assignment they have done so far in the course. Further, it should allow being used for forming other educational teams, even with different problem characteristics. The report and the user guide provide detailed enough explanation of the algorithm, so another user can use it optimise it himself/herself and extend its application.

1.2 Objective

The prime purpose of the project is thoroughly researching the field of swarm intelligence algorithms and especially particle swarm optimisation and providing the project supervisor with reliable and efficient implementation of the PSO for forming optimal groups of students. It should be able to read a file containing the all the necessary data about the students in the class and splitting them into small maximally heterogeneous(by default) teams in a reasonable amount of time and then output list of the generated teams. The program is to be implemented in the form of a Ruby gem, which would allow it to be ran as a part of custom web application in the background, due to the complexity of the problem and the need of plenty of run time. The user should be able to specify custom ways for turning grades into skill, as well as specifying custom and grouping criteria, beyond the default ones of the implementation.

The implementation should also provide a way for validating and handling invalid data. It should detect any invalid parameters, as well as any missing values in the data set. In the case of missing values, an automated way of handling them is needed to facilitate the process for the student furtherly. The way values are handled has to be relevant to the data set, to maximise the chances of successful allocation for the students with missing attributes in the data set.

Furthermore, the algorithm behaviour needs to be fully customisable, allowing the user to

change every single parameter that has an impact on its behaviour. The implementation needs to be also capable of providing statistics for each run. Therefore the user should be able to apply the algorithm to a broader range of problems.

1.3 Scope

As mentioned before, despite the scalability capabilities, the current project is focused on a specific set of constraints, related to the nature of the software engineering group project, or any other similar project group assignment. The program is developed, tested and expected to take data set of size between 250 and 300 students. Furthermore, despite the variety of personal factors which can determine the cumulative set of skills and benefits all the students in the group will acquire, for the scope of the current project, the list of data fields for each student is limited to his/her gender, ethnicity and their educational skill - a grade from a relevant core module from the previous year, as well as optionally a set of students each student is forbidden to work with for reasons determined by the user/project supervisor. That sets a limitation for the data set format. The algorithm will be usable only with problems that have data sets similar, or transformable to the one required for the software engineering project.

Additionally, the implementation will not present a standalone program, but an external library(called Gem in the Ruby language). Therefore it will only be able to be used as a part of user-developed program/web-application. So primary Ruby skills will be a requirement for interacting with the implementation. The primary audience of the project are second year students being allocated into teams for a software engineering group project and the project supervisor. It provides an easier automated way of forming quality teams. Also maximises the expected benefits students will get from their project. Further, the implementation provides other users with full control of the algorithm behaviour so that they can apply it to a broader range of problems

1.4 Report Structure

The rest of this report is structured as follows:

- Chapter 2: Background presents the motivation behind this project, as well as a critical literature review of the related to the topic work.
- Chapter 3: Particle Swarm Optimisation gives the reader more profound insight into the

original PSO and BPSO algorithms, along with a critical review of them and their most popular modifications.

- Chapter 4: Proposed Algorithm presents in detail the designed novel approach to solving the team formation problem, by explaining its mechanics and the problems it addresses.
- Chapter 5: Design and Specification describes in detail the full design of the project, along with all the user and system requirements.
- Chapter 6: implementation shows the full structure of the software product, as well as a detailed explanation of the implementation details and the ways specific problems were tackled.
- Chapter 7: Legal, Social, Ethical and Professional Issues
- Chapter 8: Results and Evaluation presents all the testing and parameter tuning, which was used for optimising and validating the algorithm implementation, the achieved results and critical evaluation of the results towards the specified earlier in the report requirements.
- Chapter 9: Conclusion and Future work summarises my comments on the project, the obstacles I faced and the benefits I got from all the work related to the current project, as well as its capabilities for further work and expansion.
- Chapter 10: User Guide provides a full guide to the user on how to interact with the gem.

Chapter 2

Background

2.1 Motivation

Even though in some particular scenarios, given individual working on his own might contribute to the quality of the results, it is far from the best work pattern in general. Working in teams proved itself as a promising technique for achieving better results regardless of the problem context. With the idea of teamwork, team formation is a direct consequence and the fundamental task that comes first in every situation involving the word "team".

2.1.1 Importance of Team Work for Educational Purposes

The team formation problem applies to many different aspects of the real world, from work projects, sport, different kinds of participatory tasks, to school group projects and learning groups, or any other scenario that requires some kind of collaboration, especially when a distribution of skills is involved[14]. When it comes to education, no matter the stage to which problem relates, group projects bring additional benefits to the apparent advantage of knowledge sharing. Those also include significantly elevated educational results and highly beneficial discussions over the topics relevant to the project[27, 30]. Furthermore, Vygotsky's theory, dating back to 1978, suggests that group work, involving social interactions and peer collaborations, dramatically enhances the learning experience for each of the individuals in the group[38]. Another study also suggests that when specific knowledge and interest levels are taken as the main factor when forming the groups, discussions and interactions inside the team are furtherly improved, which consequently leads to even higher educational results[43].

2.1.2 Team Formation Problem and its Complexity

This approach is widely used nowadays due to the benefits of teamwork. However, the proven importance of careful team formation for maximising educational results adds significant overhead for teachers. The need for the teacher to simultaneously consider many parameters makes the problem exponentially harder for larger groups of students. Thinking about all possible team formations and how the different personal factors influence the quality of the particular team configuration makes the problem NP-Hard.A problem being NP-Hard means that with the increase of the input data size, the time and computational space needed for finding optimal solution increases tremendously and quickly reaches a point where it cannot be obtained in a reasonable amount of time.

Therefore team assignment done by the teacher would be an arduous process, which is very unlikely to produce functional groups. Which may lead to some groups achieving the learning goals much quicker than others, whereas others may not achieve them at all. This suggests the need for an effective algorithm that solves the problem and increases the chances of higher learning results[26].

The actual mathematical complexity of the problem can be expressed in terms of the number of possible solutions, i.e. the size of the search space. If we generalise the problem and think of it as assigning class of n students into teams of size k, the following formula gives the number of possible allocations - $(f(n,k))$:

$$f(n,k) = \frac{\prod_{i=1}^{\frac{n}{k}} C_{ik}^{k}}{\frac{n}{k}!} \tag{2.1}$$

Which means, for example, for a problem, where have to assign 8,12,16,20,24 students to teams of 4, the number of possibilities will be calculated in the following way:

$$f(8,4) = \frac{C_8^4 C_4^4}{2!} = 35 \tag{2.2}$$

$$f(12,4) = \frac{C_{12}^4 C_8^4 C_4^4}{3!} = 5775 \tag{2.3}$$

$$f(16,4) = \frac{C_{16}^4 C_{12}^4 C_8^4 C_4^4}{4!} = 2627625 \tag{2.4}$$

$$f(20,4) = \frac{C_{20}^4 C_{16}^4 C_{12}^4 C_8^4 C_4^4}{5!} = 2546200000 \tag{2.5}$$

$$f(20,4) = \frac{C_{24}^4 C_{20}^4 C_{16}^4 C_{12}^4 C_8^4 C_4^4}{6!} = 4509300000000 \tag{2.6}$$

Clearly, with the increase in the number of students, the number of possible solutions grows enormously. This is the case even for relatively small inputs, compared to the scale of the current problem. Moreover, in the above equations, the number of students is assumed to be a multiple of the team size. In reality, we usually have to deal with a variable team size, which makes the search space is substantially larger. Even if there is only one extra student, to have a mathematically sound definition, we have to solve the problem for $n + k$ students, assigned into teams of size $k + 1$.

2.1.3 Alternative Solution

The complexity of the problem usually leaves the teacher with three choices. The first option is using approach partially or entirely based on randomness for allocating the students into groups. With this approach, the overall quality of groups tends to be low and is likely to obstruct all students to get equal and maximal benefits from the given project/assignment. The second common approach is letting students form groups on their own. That increases the likelihood of getting groups, composed of students who will happily work together and communicate efficiently. However, it is also likely that some students, who are having harder times approaching other people and forming groups will be left alone, which may hinder their overall performance. The third option, though, is looking for an alternative solution to the complicated problem.

A rational candidate for alternative solutions is a computer algorithm. Solving NP-Hard problems has been a vital topic in computer science, and extensive research in the field of AI has been performed for finding possible mechanisms of solving those types of problems. Unfortunately, still, a universal optimal solution is not available for those problems. All proposed algorithm should be thoughtfully designed to facilitate the search for a reasonable solution, due to the complexity of the problem.

Suggested approaches for solving NP-Hard problems are Swarm Intelligence based algorithms. This family of algorithms is inspired by the behaviour of animals that live in social groups such as bats, fish, ant colonies and bees. What is remarkable about them and inspires the algorithms is the simultaneous influence of self-organisation and stigmergy for reaching emergent behaviour in the colony[1]. Examples of such algorithms include Particle Swarm Optimization(PSO), Artificial Bee Colony(ABC), Bat Algorithm(BA), Ant Colony Optimization(ACO).

2.2 Related Work

Due to the importance, wide applicability and the need for an efficient and high quality solution of the team formation problems, extensive research was performed in this field. From the first studies to nowadays, we can see great improvements in the diversity and quality of the suggested methods.

2.2.1 First Team Formation Algorithms

When it comes to using the help of computers for solving team formation problems, the first research works date back to more than 40 years ago. In 1976, Dyer and Maluev[13] were the first to conduct a study on such a topic. They worked on a group assignment problem. However, their main focus was solving the problem from the faculty's point of view, rather than addressing the characteristics of the individual students, despite including student preference in the problem definition. Their optimisation function gave higher importance to problems such as available time slots and the number of courses a lecturer is teaching.

Ten years later, another study came out, that shifted its primary focus towards maximising student experience by aiming to create similar groups of students. Behestian-Ardekani and Mahmood[8] used a greedy algorithm, which calculates within-group scores for each team and aims towards as similar as possible scores for the formed groups. The algorithm had reasonable efficiency. However, the results turned out to be far from optimal. Weitz and Lakshminarayan[41] also used a method that relied on calculating scores for every team. They focused on creating maximally diverse groups by maximising the differences between each pair of students in a given team and tried to maximise the sum of those for all groups. The method used a heuristic approach based on the Very Large Scale Integration(VLSI) design problem. The results of this research pointed to a fundamental potential issue when solving the team

formation problem. Namely the balance between the scores of different groups, as well the ambiguity in transition between the pedagogical and mathematical quality of the solution. Even though the results showed high overall fitness, this did not necessarily guarantee uniform balance in each team. For the sake of getting a mathematically optimal solution, the method would produce some widely diverse groups, while other ones were with homogeneous student distribution. Hence in the development of such an algorithm, the fitness function should be carefully designed, so the satisfaction of the mathematical constraints does not create the risk of compromising the real-world quality of the solution.

2.2.2 New Approaches and Increased Success

With the improvements in computer science and its methodologies, the variety of team formation algorithms started growing in the early 2000s. In 1999 Zakarian and Kusiak[45] developed a methodology, where they formulated the problem as an integer linear program, which was solved using the branch-and-cut technique. Baker and Powell[6] researched different optimisation function for establishing maximally diverse groups. However, as they state in the paper, their work was based on optimisation algorithms, which were used in industry, not specifically tailored for dealing with student assignment problems. Despite the results again producing reasonable, at first glance, groups, this research also brings us back to the point that even if we use methods, which are proven efficient for other purposes, we should still be focused on translating the solution to the specific domain, so it is capable of accommodating the pedagogical constraints. Other studies used various methodologies for solving the problem. Fitzpatrick and Askin's[17] method is based on the individuals' personal qualities like temperament and work drive for calculating the suggested quality of the team. Chen and Lin's[11] team formation model is based on the individuals' teamwork skills, knowledge and relationship with other team members to evaluate team quality. Also, Baykasoglu, Das and Dereli's[7] fuzzy optimisation model, uses the simulated annealing algorithm. Later on, another approach showed good results in multiple studies which were a base for several team formation algorithms[5, 22, 25]. It uses approximation algorithms, which mainly consider communication costs like diameters, minimum spanning trees and distances to the team leader in social networks configurations.

2.2.3 PSO for Solving NP-hard Problems

If we take a step back from team formation algorithms for a second, we can see the increasing popularity of a new novel approach for solving NP-hard problems. After its proposal

in 1995 by Kennedy and Eberheart[23], the Particle Swarm Optimization(PSO) gained increasing popularity in the next few years. In the middle of 2000's several studies used PSO-based algorithms for addressing a variety of NP-hard problems, including vehicle routing, job scheduling, composing test sheets, decision-making, and solving the Travelling Salesman Problem(TSP)[10, 21, 29, 34, 35, 37, 39, 44].

2.2.4 PSO Based Team Formation Algorithms

Despite the interest in PSO for solving NP-hard problems, until recent years, most of the research in the field of team formation has been based on different approaches. Nevertheless, in the past couple years, plenty of papers presenting PSO-based models came out, showing promising results. A research paper proposes the use of this algorithm for composing highly heterogeneous and highly interactive groups for computer-supported collaborative learning[20]. The method produced expertly constructed teams in reasonable execution time by taking into consideration three primary parameters: competence, learning style and student interaction. It focuses on a problem where students who were already familiar with each other, which allowed the use of the student interaction possible for improving the quality of the solution. For the current project, however, most of the students would not have had any work contact with each other, so the use of any interaction based criteria is irrelevant. Another study uses so-called enhanced particle swarm optimisation to automatically compose collaborative learning groups [26]. The optimisation function of this model relies on two factors: each individual's knowledge on each of various topics and its interest in those topics. Even though the algorithm had to deal with different grouping criteria over multiple topics, the tests performed by the authors showed it could obtain near-optimal results in reasonable computation time and proved the robustness of the algorithm. Moreover, a study from 2019 presents an innovative approach for PSO based team formation[14]. The study combines PSO with a swap operator inspired by the crossover process in genetics algorithms(GA). It takes the best from two of the most efficient metaheuristic algorithms to successfully assign experts with different sets of skills to teams, where particular skills are required. The combination of both algorithms led to faster convergence and promising results.

Therefore, the particle swarm optimisation algorithm is the base concept for the development of the team formation algorithm in the current project.

Chapter 3

Particle Swarm Optimisation

A substantial part of the software implementation relies on the concepts of the particle swarm algorithm proposed by James Kennedy and Russell Eberhart in 1995[23]. It is stochastic, metaheuristic algorithm, based on the concept of swarm intelligence. The stochastic optimisation methods are optimisation algorithms, which rely on the random generation of states, values and parameters. On the other hand, metaheuristic algorithms provide a near-optimal solution without exploring the whole search space.

To some degree, It is similar to other evolutionary algorithms such as the genetics algorithms(GA) family. Both optimisation methods randomly generate their initial states, and they represent a population of candidate solutions. At each iteration, the algorithms change the population, to find an optimal solution.GAs use biological concepts such as crossover and mutation to generate new candidate solutions, in PSO the particles are following a trajectory influenced by their inertia, as well the memory of the personal and global best positions found so far.

This chapter presents a description of the original algorithm, which inspired the design and the implementation of the proposed algorithm. Along with that, it gives a critical evaluation of the most popular extensions of the algorithm.

3.1 Algorithm Background

Biological phenomena like flocking birds and schooling fish inspired the algorithm creation. Computer simulations created by Reynolds[31], Heppner and Grenander[19] led to designing the algorithm. It exploits the idea of using social intelligence and knowledge sharing between

the members of the group to reach a high-reward state.

The initial idea of the authors has been to model and simulate social behaviour. However, since simplifying and carefully observing the results, Kennedy and Eberhart realised that it naturally performs optimisation. Their book describes numerous philosophical aspects of PSO and swarm intelligence in general.

3.2 Original Algorithm

When it comes to optimisation, each bird of the flock is represented by pa article which stands for a candidate solution to the optimisation problem. To find food, the members of the flock share knowledge, and use it as a guide. Similarly, the algorithm always keeps track of the best state of every particle(local best score), and the best of those local bests so far(global best position). Apart from keeping a record of the locations of best states, i.e. best candidate solutions, local and global maximums are influencing the movement of each particle across the search space. Each particle has two key characteristics - position and velocity. The position represents the current candidate solution, while the velocity regulates the movement which the particle will perform in the next iteration. At every iteration both the position and velocity are updated according to the following equations. Position:

$$x_i^{t+1} = x_i^t + v_i^{t+1} \tag{3.1}$$

Velocity:

$$v_i^{t+1} = wv_i^t + c_1 r_1^t (lbest_i^t - x_i^t) + c_2 r_2^t (gbest_i^t - x_i^t) \tag{3.2}$$

Where:

- x_i^t - position of particle i at iteration t
- v_i^t - velocity of particle i at iteration t
- w - the inertia weight, controlling the influence of the previous velocity
- $c1, c2 \geq 0$ - are control constants which are regulating the influence of the local and global bests respectively towards the movement of the particle

- $r_1^t, r_2^t \in [0; 1]$ - local and global random control parameters at iteration t, they are different for each equation
- $lbest_i^t$ - local best position of particle i at iteration t
- $gbest_i^t$ - global best position of particle i at iteration t

The first term of the equation represents the current inertia(velocity) of the particle. Whereas the second is cognitive component, redirecting the inertia to an extent, calculated by the first control constant and the first random number, towards the local best. The third one, the social component, according to the second control constant and the second random number gives the last adjustment to the movement of the particle, making it go towards to global maximum.

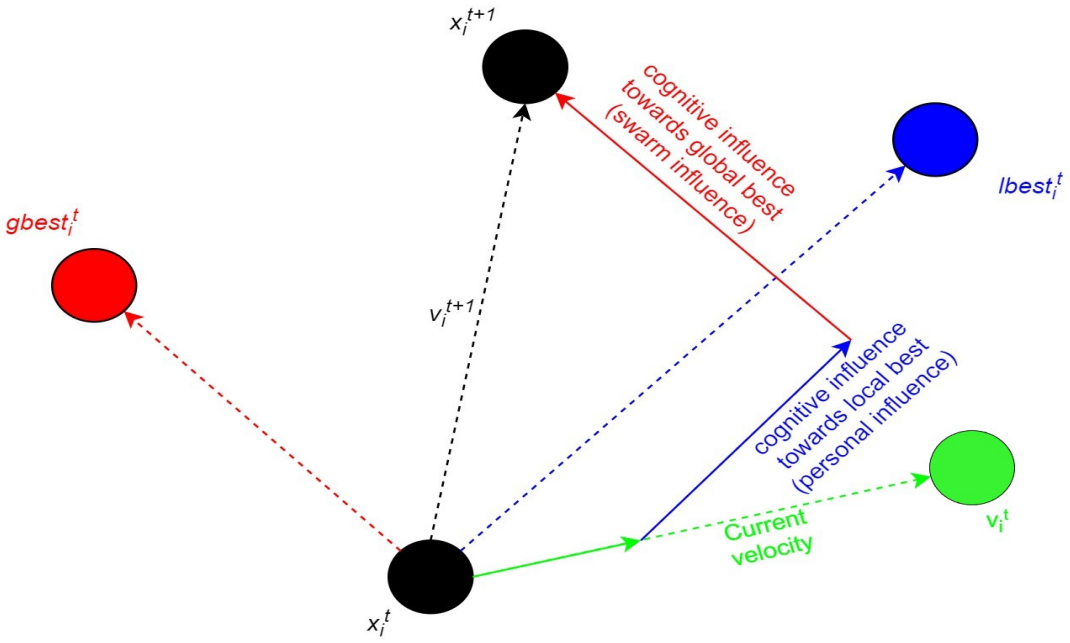

Figure 3.1: *Visual representation of the velocity component*

After each iteration, which in the given context means calculating the new position of each particle, the fitness of each candidate solution is evaluated according to a pre-defined cost function. Also, local and global best are updated accordingly.

The stopping criteria of the algorithm may be one or combination of the following:

- Reaching pre-set maximum number of iterations
- Acceptable solution is found
- No global best improvement over a specified number of past interactions

- Swarm radius approaching 0

3.3 Algorithm Example

This simple example ilustrates PSO with 5 particles, performing an iteration, trying to maximise a simple mathematical function.

Function to be maximised: $f(x) = -2x^2 + 5x - 8$ for $x \in [-30 : 30]$

$w = 0.5; c1 = c2 = 0.5$

Intial positions of particles:

$x_1^0 = -6$

$x_2^0 = -3$

$x_3^0 = -0$

$x_4^0 = 4$

$x_5^0 = 8$

Initial velocities of particles:

$v_1^0 = v_2^0 = v_3^0 = v_4^0 v_5^0 = 0$

Iteration 1 Initial fitnesses of particles:

$f(x_1^0) = -110$

$f(x_1^0) = -41$

$f(x_1^0) = -8$

$f(x_1^0) = -96$

$f(x_1^0) = -110$

Local bests:

$lbest_1^0 = -110$

$lbest_2^0 = -41$

$lbest_3^0 = -8$

$lbest_4^0 = -96$

$lbest_5^0 = -110$

Global best:

$gbest^0 = -8$

Calculating velocities: $v_1^1 = 0.5*0+0.62*0.5(-110-(-110))+0.19*0.5(-110-(-8)) = -10.64$

$v_2^1 = 0.5 * 0 + 0.78 * 0.5(-41 - (-41)) + 0.51 * 0.5(-41 - (-8)) = 8.415$

$v_3^1 = 0.5 * 0 + 0.53 * 0.5(-8 - (-8)) + 0.56 * 0.5(-8 - (-8)) = 0$

$v_4^1 = 0.5 * 0 + 0.21 * 0.5(-96 - (-96)) + 0.9 * 0.5(-96 - (-8)) = -39.6$

$v_5^1 = 0.5 * 0 + 0.51 * 0.5(-110(-110)) + 0.53 * 0.5(-110 - (-8)) = -27.03$

Calcuating new positions:

$x_1^1 = -6 - 10.64 = -16.64$

$x_2^1 = -3 + 8.415 = 5.415$

$x_3^1 = 0 - 0 = 0$

$x_4^1 = 4 - 39.6 = -35.6 => -30$

$x_5^1 = 8 - 27.03 = -19.03$

3.4 Neighbourhoods and Topologies

The classical PSO, explained above, uses the so-called global topology. In global topologies, there is a connection between each pair of particles, and all of them communicate with each other, and there is only one global best. As a result, there is an increased risk of premature convergence around a local minimum if the problem space is not thoroughly explored since all particles are moving towards a single global best. A 2007 study concludes that there is a direct relationship between the overall algorithm performance and the topology choice[15].Some of the most common topologies include[12, 15]:

- Star topology - where all particles are connected and change information(original/gbest PSO)
- Ring topology - each particle exchanges information only with the two neighbouring particles, and the whole swarm form a closed circle.
- Von Neumann topology - each particle has four neighbours, which for easier visual representation and implementational organisation, are usually shown as left, right, top and bottom neighbours.
- Wheel topology - all particles are connected to a single central particle, common for the whole population, which processes the whole information flow.
- Four clusters topology - the whole population is divided into four star topologies, from which there are three particles which are each connected to one of the other three neighbourhoods.

Source: E.M.N. Figueiredo, T.B. Ludermir / Neurocomputing 127 (2014) 4–12

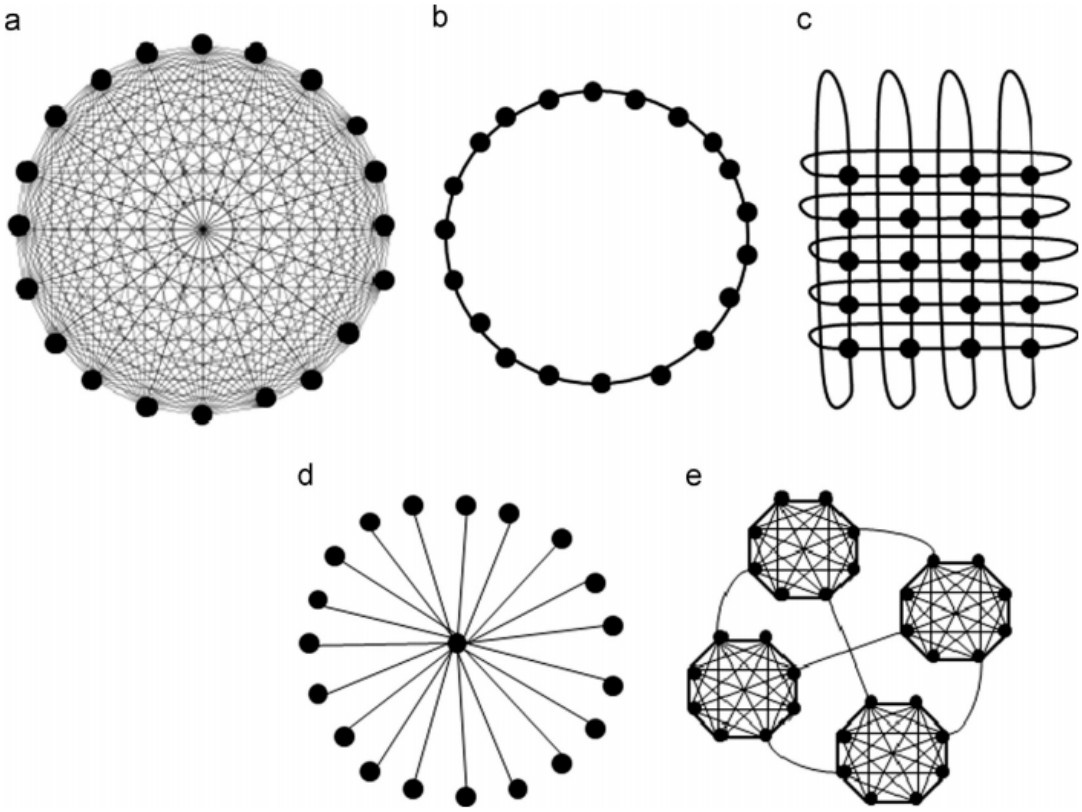

Figure 3.2: Common PSO topologies: (a)Star, (b)Ring, (c)Von Neumann, (d)Wheel, (e)Four Clusters

Those, however, do not exhaustively list all topology configurations. Apart from the listed ones, there are more static configurations, as well as a wide variety of dynamic neighbourhood approaches, a fraction of them listed by Sanchez[33]. Those approaches change the topologies during the run to fine-tune the exploration and exploitation aspects of the algorithm. The result is improved performance.

3.5 Binary PSO

The nature of the original PSO algorithm from 1995, was designed to operate in continuous space. However, a significant amount of optimisation problems have discrete space representation. Hence, two years later, in 1997, the same authors, reworked the algorithm and proposed a discrete binary version[24]. While in the previous version, the particle movement was a numerical change in a set of values, in this work, apart from the position representing a set of binary values, the particle velocities stand for the probabilities that given component in the position

representation will change from 0 to 1, or vice versa.

3.5.1 Formal Representation Differences

Since the velocity of the particles must be in the $[0; 1]$ interval, the velocity is clamped using sigmoid limiting transformation:

$$S(v_{id}) = \frac{1}{1 + e^{-v_i}} \tag{3.3}$$

Moreover, the position is updated according to:

$$x_i(t+1) = \begin{cases} 1 & \text{if } rand() < S(v_{id}(t+1)) \\ 0 & \text{if } rand() > S(v_{id}(t+1)) \end{cases} \tag{3.4}$$

where $rand()$ is a method returning random value in the interval $[0; 1]$.

3.5.2 BPSO Disadvantages

The algorithm's changed mechanics led to several problems related to algorithm implementation and performance which[28] studies. The first problem is related to the sigmoid function for transforming the velocity. In continuous PSO, large velocity values in both positive and negative directions have similar effects on particle position. They both indicate the need for a more significant change in the numerical values. On the contrary, in BPSO, while significant velocity value in the positive direction results in a high change probability, high values in the negative direction lead to very low probabilities of a position change. Additionally, when normal PSO approaches local or global optimum, the velocity goes towards zero, and that translates to reducing particle movement. However with BPSO, if velocity goes to zero, that will give equal chances of the corresponding position bit to be set to either 0 or 1. Consequently, the algorithm improves fitness over the first couple of iterations quickly, but when it approaches the global optimum, the particles start veering away and may get trapped in a local minimum. The second problem arises, from the position update equation, because the update is entirely based on velocity and not on the previous particle position.

Chapter 4

Proposed Algorithm

The algorithm designed for solving the current problem is applying the main principles of the original version of the BPSO and respectively PSO. However, the design of the algorithm is for general purpose, while the current project has a particular scope. Therefore, key algorithm concepts were customised to adapt the algorithm to the problem and increase its performance. Further, a concept used in other algorithms, inspired by biological phenomena, like the family of genetic algorithms - selectivity is implemented to increase the decision quality. The result was a modification of the MBPSO, having linearly updated inertia weight and selectivity parameter, dynamic topology.

4.1 Particle Representation

Hence the original nature of both the original and binary PSO algorithms, one of the vital steps required towards proposing a solution for the current problem was finding suitable particle representation. This was one of the first decisions that had to be taken when designing the algorithm.

The approach used was suggested by Lin et al. [26], which uses a binary representation. Each particle position will be a m x n matrix, where m stands for the total number of students in the class and n is the total number of teams that need to be formed(according to the class and team size). Each row in the matrix represents the team with the corresponding index and will show the students that form it. Whereas, each column will represent a student with the corresponding index in the data set and will denote its team. In other words, the matrix will consist mostly of zeros, apart from where for a given row and column indexes, the corresponding

student is part of the corresponding team. For example, if we have a data set, consisting of 12 students with IDs in the $[0, 11]$ range, which need to form 3 teams of 4 people, the following will be possible particle encodings:

$$x_1^t = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix} \tag{4.1}$$

Team 1 includes students 0,1,2,3; Team 2 includes students 4,5,6,7; Team 3 includes students 8,9,10,11

$$x_1^t = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix} \tag{4.2}$$

Team 1 includes students 0,5,6,11; Team 2 includes students 3,4,7,8; Team 3 includes students 1,2,9,1

4.2 Velocity

One of the crucial factors in PSO implementation is the velocity of a particle. In the original versions, when updating the position of a particle, i.e. generating new candidate solution, the velocity has an impact on the new position generation equal to the impact of the current position, and they are the only two factors involved in the position update equation. To maintain the meaning of the position update equation as a sum of velocity and position, the velocity has to be in the same format as the position, to allow their summation. Furthermore, the positional values in the velocity have to be comparable to those in the position to keep the balanced importance of the two factors. Therefore, the velocity also has the form of m x n matrix. However, the allowed values are floats, rather than 0s or 1s, and they indicate the chance for a particular student to be assigned to a particular team in the next iteration.

4.2.1 Forming Velocity

The factors composing the velocity are kept in nuance with the original algorithm. The velocity is represented as a weighted sum of three components:

1. *Currentvelocity*

2. *Personalcomponent* - formed by random factor and the difference between the current position and the best position the particle has been at so far.

3. *Localcomponent* - formed by another random factor and the difference between the current position and the best position any of the particles in the neighboured has been at so far.

The weights of the components are different tunable values, respectively called inertia weight, personal cognitive parameter and local cognitive parameter.

4.2.2 Inertia weight

The inertia weight is a parameter in the range $[0 : 1]$. It controls the impact the current velocity has on the new updated one. High values are encouraging exploration, while as the values decrease, the exploration tendency shifts to exploitation oriented behaviour. Even though the original algorithm uses constant inertia weight values, a number of studies suggest increased performance with dynamic inertia[3, 3, 4, 36].

Additionally, one of the most important things to consider, while designing the algorithm is the size of the search space. We are dealing with enormously large search space. Hence, the algorithm has to be capable of exploring a substantial part of the possible solutions, especially in the early stages of its run. The explored variety of initial configurations will then guide the swarm into approaching the optimal solution. On the other hand, the current problem does not have a single optimal solution. Because of the large number of students and the relatively small set of attributes and their values, many different combinations will provide solutions of similar quality. Therefore, the exploration problem should be carefully handled. Entirely exploration oriented algorithm will cause the swarm to wander around the search space, and the chances of finding many new solutions, which are not of the desired quality will be as random as the chances of finding the optimal solution. On the contrary, exploitation oriented algorithm will explore a minimal set of allocation variants and will have limited chances of finding optimal solutions, as the desire to follow the local optimum will make the likelihood of getting stuck in local minima very significant. Therefore, we cannot rely on either of those two.

That is why, a dynamic inertia approach was adopted. For the course of the algorithm run, the inertia weight will linearly decrease between pre-set initial and finial inertia values, to stimulate both early exploration and later convergence to a single solution.

4.2.3 Cognitive Terms

To achieve the desired velocity format, the components forming it, should be in the same format, as well. Hence, the cognitive terms also are in the form of m x n matrix. Similarly, the factors that form them will also be represented in the same format.

When it comes to calculating each of the cognitive terms, it is another adaptation of the algorithm to the current problem. According to the original version, when composing the cognitive terms, the random component is multiplied by the difference between current and local/global best and this is one of the primary mechanisms for introducing new information and maintaining the diversity in the population. On the other hand, in the current situation, if we multiply the difference between the positions by a single value, only the bits where the two positions are not equal will have their importance amended. Which is indeed one of the functions of the random parameter. However, the introduction of new information will be completely missing. Therefore, to address this issue, the random parameter is also in the mentioned format. The values inside will represent the chances of random bits to become part of the new position. Then those changes will be summed up with the positional difference.

The last component in forming the cognitive term is the control parameter. In the original version, it is a single value, which, similarly to the random factor is multiplied by the random factor and the positional difference. While the random factor's prime purpose is introducing new information and is supposed to have less influence on the general control of the algorithm behaviour, the control parameters are on the opposite. Together with the inertia weight, they are the main ways of controlling the algorithm performance and characteristics. Similarly to the random factor, just a single value, would not be capable of controlling the behaviour to the desired extent. That is why the control parameters are 3-valued and simultaneously control the introduction of new random information, its importance and the importance of the positional differences. The first value stands for the threshold above which the randomly generated values in the random component is accepted as candidate student allocations. The second value is the probability the candidate allocations will get when updating the position. And the last value will be the probability the positional differences will take.

The proposed control parameters provide another way of balancing and controlling the exploration and exploitation capabilities of the algorithm. To avoid premature convergence, getting stuck in local maximum and endless exploration of low-quality solutions and achieve balanced performance and reaching an optimal solution, the control parameters have to be carefully tuned.

4.3 Fitness Function

Similarly to every other optimisation problem, the fitness function is an essential part of it. It gives meaning to the algorithm implementation by evaluating the quality of each proposed solution. For the current implementation, an improved version of the traditional fitness function used in proposed solutions to the maximally diverse grouping problem(MDGP).

4.3.1 MDGP Fitness Function

Weitz and Lakshminarayanan defined Formal representation of the MDGP problem and its objective function in 1997[40], which many studies used later[9, 16, 18, 32]. It is formulated as the following quadratic integer program:

$$max : \sum_{p=1}^{G} \sum_{i=1}^{N-1} \sum_{j>1}^{N} d_{ij} x_{xp} x_{jp} \tag{4.3}$$

subject to:

$$max : \sum_{p=1}^{G} x_i p = 1 \ for \ i = 1, 2, ..., N \tag{4.4}$$

$$max : \sum_{i=1}^{N} x_i p = S \ for \ p = 1, 2, ..., G \tag{4.5}$$

where:

- N - number of students.
- G - number of groups that should be formed.
- S - number of students in each group, equal to $\frac{N}{G}$.
- d_{ij} - difference/distance function between students i and j, which most commonly is the weighted Euclidean distance between the students, represented as vectors of their characteristics.
- x_{ip}/x_{jp} - a function that equals 1 if student i/j is in group p, or 0 otherwise.

4.3.2 Current Problem Fitness Function

The fitness function which the algorithm uses is very similar to the one presented above. Every student is represented as a set of three characteristics - $[gender, ethnicity, skill]$. The distance function used is the euclidean distance between each pair of student representations. According to the Euclidean distance definition, the distance between student i and student j is given by:

$$d_{ij} = \sqrt{(gender_i - gender_j)^2 + (ethnicity_i - ethnicity_j)^2 + (skill_i - skill_j)^2} \tag{4.6}$$

The first problem that arises is calculating the difference between the non-numeric parameters(gender and ethnicity). Even though they have numeric value representations in the data set, their meaning does not imply a direct subtraction. Therefore, the subtraction is replaced by a function, and the distance function takes the following form:

$$d_{ij} = f_1(f_2(gender_i, gender_j) + f_3(ethnicity_i, gender_j) + (skill_i - skill_j)^2) \tag{4.7}$$

where:

$$f_1(x) = \begin{cases} \sqrt{x}, & \text{if} x \geq 0 \\ -\sqrt{x}, & \text{otherwise} \end{cases} \tag{4.8}$$

$$f_1(x, y) = \begin{cases} gender\_weight, & \text{if } x = y \\ 0, & \text{otherwise} \end{cases} \tag{4.9}$$

$$f_2(x, y) = \begin{cases} ethnicity\_weight, & \text{if } x = y \\ 0, & \text{otherwise} \end{cases} \tag{4.10}$$

and $gender\_weight$ and $ethnicity\_weight$ are parameters of the algorithm(in section 6.1 their values are addressed).

By using those two difference functions, the distance between the attributes will always have a static value, instead of different values for different attribute pairs, as the difference between two given ethnicities is not larger or smaller than the difference between two other ethnicities. Furthermore, this representation allows flexibility in the search criteria. If the values of both or any of the parameters are negative, the algorithm will search for homogeneous team allocation in terms of the attributes having a negative weight parameter.The function f_1 was introduced to avoid any problematic behaviour in case of $f_2(gender_i, gender_{1j}) + f_3(ethnicity_{2i}, ethnicity_{2j}) > (skill_i - skill_j)^2$.

At the algorithm design stages, empirical tests on different algorithm versions suggested Another problem with the direct implementation of the MDGP fitness function. During tests using only the distance between students, a trend of grouping extremely diverse teams while leaving others with two or more students with the same skill in the team was observed. Henceforth, a penalty system for duplicating skills is implemented. It uses three different penalty values. The smallest one is applied when a team has two duplicating values, slightly firmer one is used one for three duplicating values, and a very severe one is applied if all the students in the team have equal skill indicators.

4.4 Topology

The meaning of the term exploration concerning the current problem can be interpreted as each particle rather following unique trajectory than aiming to reach a place that many other particles are trying to reach at the same time. One way to achieve that is to choose parameter values such that the global maximum has almost no impact on the particle velocity and respectively position update. Nevertheless, there is also another approach - using more than one global maximum, i.e. more than one neighbourhood. If the particles start in several neighbourhoods with a low number of particles in each one, only small subsets of particles will be influenced by the same global maximum. However, the vital influence of the global maximum will still be present when updating velocity and position.

The algorithm starts with many neighbourhoods, each of which having a small number of particles inside them. That in addition to the random initial allocations, and large inertia weight value will let each small subset of particles to explore the search space independently. After a given iteration interval, some particles are moving to other neighbourhoods, until we end up with one neighbourhood, where the particles will try to converge towards the optimal solution collectively. The figures below illustrate some of the stages for moving from the initial

neighbourhood configuration to global topology.

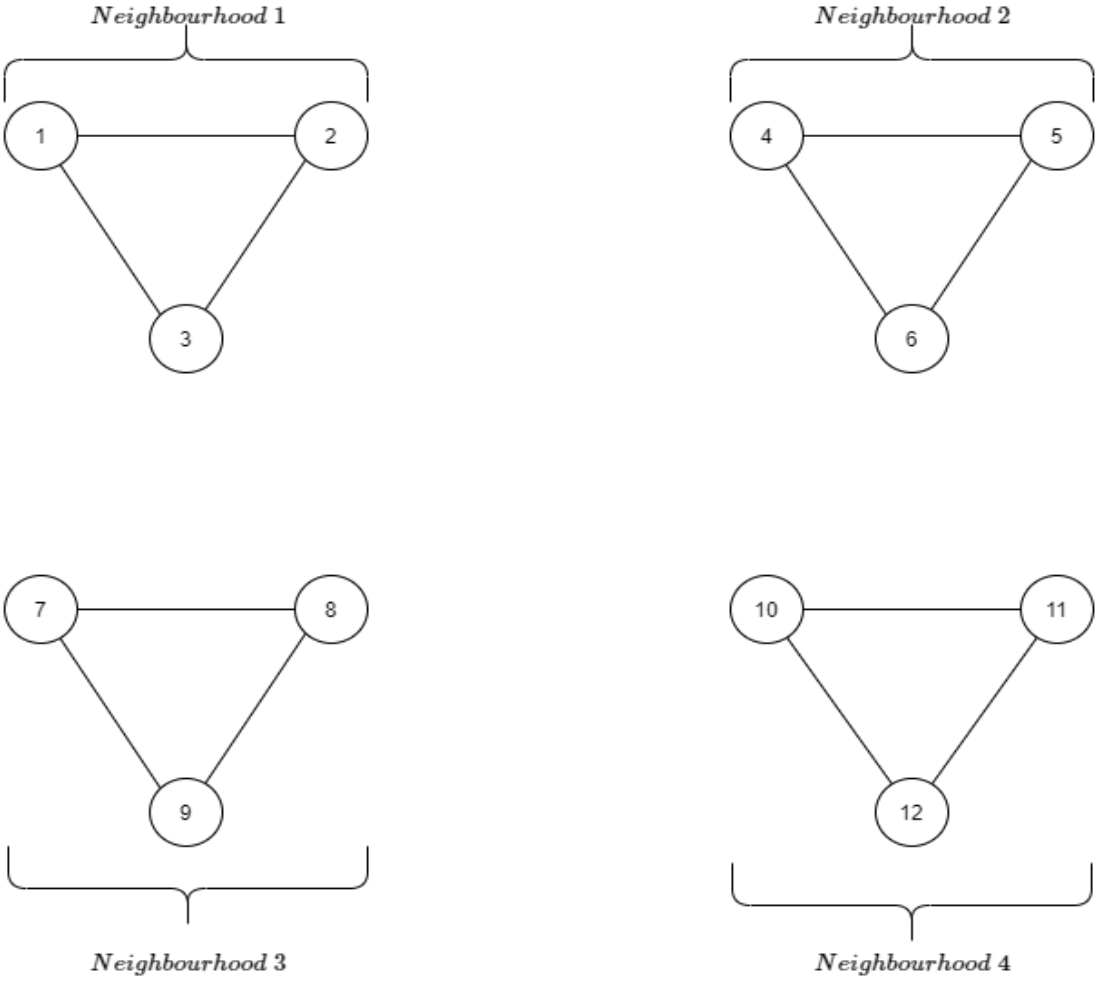

Figure 4.1: Example initial neighbourhood configuration

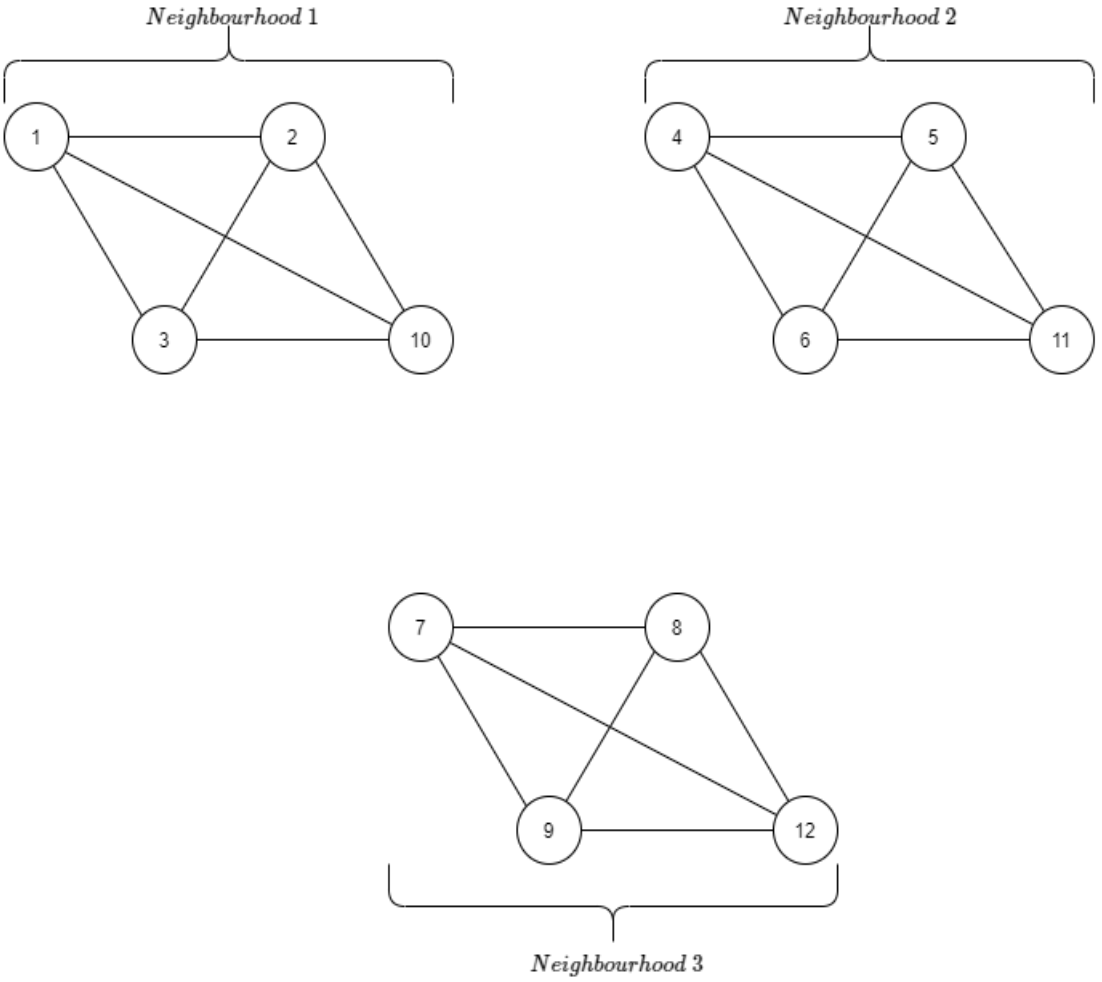

Figure 4.2: Temporary neighbourhood configuration example

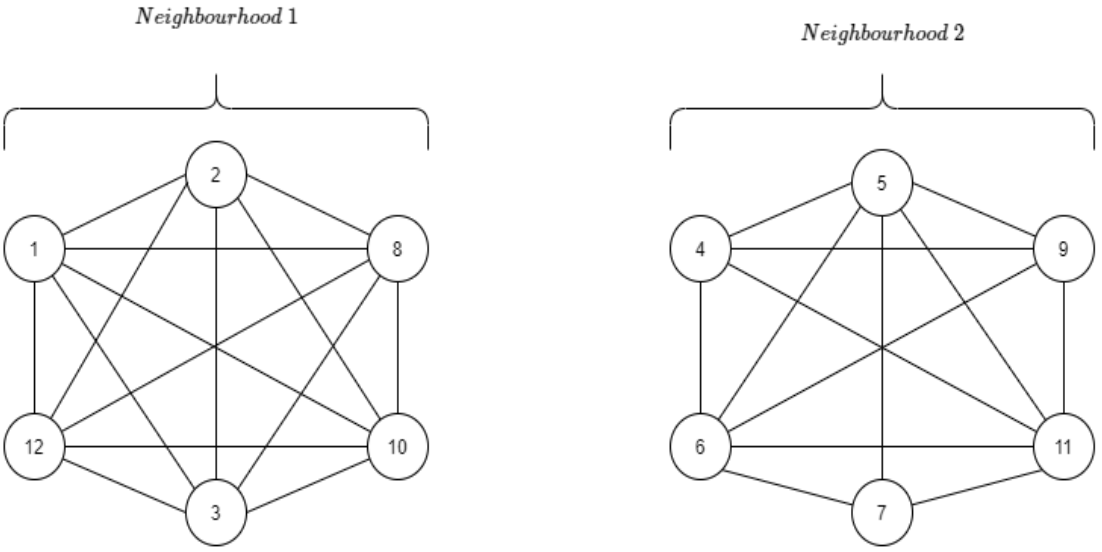

Figure 4.3: Temporary neighbourhood configuration example 2

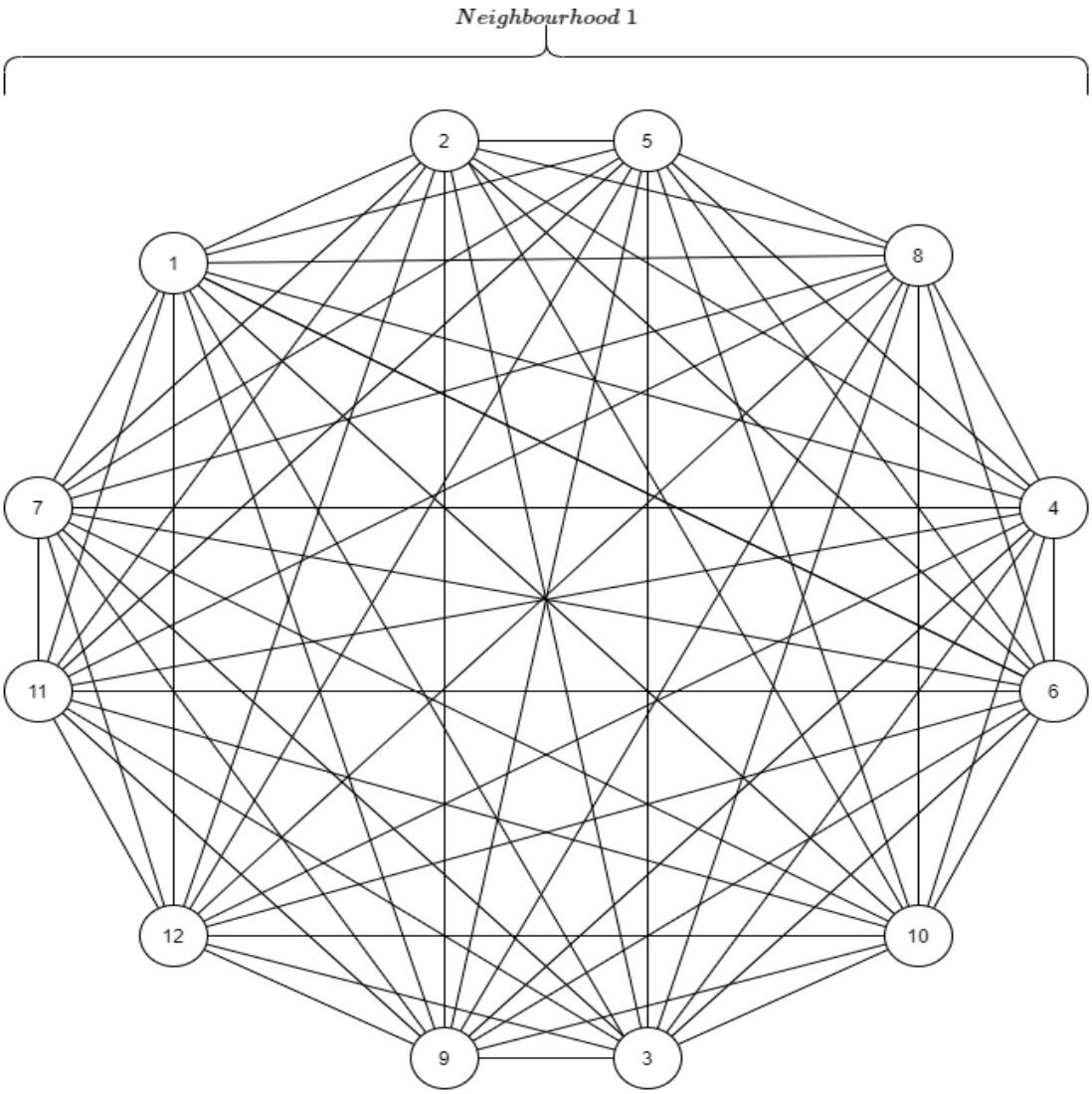

Figure 4.4: Final neighbourhood configuration example

4.5 Selectivity

In genetic algorithms, selection has an essential role. One of their main characteristics is that only a given number of the chromosomes(candidate solutions), i.e. the n fittest candidates, are selected for mating and producing new offsprings. This concept inspired the introduction of what is called 'survivability' for the scope of this project.

The combination of the particle's new velocity and current position is the final probability matrix on which position update depends. If the allocation occurs only based on those probabilities, the observations in the early stages of the algorithm development, showed that at each iteration, a large number of students were changing their teams. This has its purpose, however

only when extreme exploration is needed. In this case, what is needed is a controlled behaviour, that is capable of migrating from exploration-focused search to exploitation-focused one. The survivability concept is implemented to tackle this issue. Two values define it - the survival number and the survival threshold. The former stands for the number of probabilities left in the probability matrix before position update that will be considered for student swaps and is one of the algorithm parameters. Whereas the latter represents the probability, for which there are only as much as the survival number values left in the probability matrix and is calculated at each iteration for each particle.

Therefore, an additional mean of enforcing initial global search that steadily turns into a final local search in a controlled manner is introduced. The algorithm starts with a high survival number, which is linearly decreased over time to a small value, which encourages fewer swaps at the final iterations and stimulates the collective convergence around the global maximum.

4.6 Position Update

Updating the particle's position is represented by the second main equation defining the fundamental functionality of the PSO algorithm. However, as described in 3.5.1, the position update mechanics in classic MBPSO are different. Though, for the current problem, neither of the two approaches is directly applicable. First of all, the velocity needs to be combined with the current position of the particle. The problem comes from their representation. We need to combine a matrix of probabilities in the $[0 : 1]$ range with a matrix of 0s and 1s, which contains precisely four(or whatever the size of teams is) 1s in each row and only one in each column. Practically, the values we have for a significant number of entries in the position variable is the largest possible value an entry can have in the velocity variable. Furthermore, this value will likely be present once or twice, as due to the normalisation, only the highest(and its duplicates if any are present) will take the value of 1. Therefore, if we just sum them up, this will have an enormous impact on the position update. It will lead to a position update which is close to entirely similar to the current position. In other words, this will give very high chances of the algorithm getting stuck, probably not even in a local maximum. Because of that, before summing the variables, the entries in the position variable, having a value of 1 are multiplied by a random variable in the $[0 : 1]$ range to reduce the impact it will have on the position update. Also, it will stay in nuance with the general idea behind the velocity and position update - everything being represented as suggestions about which students should swap teams and what are the probabilities of each swap happening. This position update approach, combined with

the velocity update, suggests a solution for handling the problems described in 3.5.2.

4.7 Pseudo Code

Algorithm 1 MBPSO Algorithm

```
1: iteration ← 0
2: neighbourhood list ← initialise neighbourhoods
3: for all neighbourhoods in neighbourhood list do
4:    particles list ← initialise particles
5: end for
6: while iteration < max iterations do
7:    for all neighbourhoods in neighbourhood list do
8:       for all particlesinparticles list do
9:          personal_random ← generate personal random component
10:         local_random ← generate local random component
11:         personal_difference ← XOR(position, personalbest)
12:         local_difference ← XOR(position, localbest)
13:         personal_term ← personal_random + personal_difference
14:         local_term ← local_random + local_difference
15:         new_velocity ← velocity * inertia + personal_term + local_term
16:         normalisenew_velocity
17:         velocity ← new_velocity
18:         velocity ← velocity + position * random
19:         calculatesurvivalthreshold
20:         assign students with probabilities for all teams lower than the threshold
21:         for all unassigned students do
22:            sorted_indices ← Sort the indices of teams according to the probabilities for joining them
23:         end for
24:         i ← 0
25:         while there are unassigned students do
26:            for all unassigned students do
27:               if the team sorted_indices[i] points to has free slots then
28:                  assign the student to the team
29:               else
                     keep in the list of unassigned students
30:               end if
31:            end for
32:            i ← i + 1
33:         end while
34:         calculate fitness
35:         update personal best
36:      end for
37:      update local best
38:      if no change local best then
39:         iterations without update counter is incremented
40:      else
            iterations without update counter is reset to 0
41:      end if
42:   end for
43:   if it is time to update any of the dynamic parameters then
44:      update the needed parameters
45:   end if
46:   if algorithm has converged then
47:      format and return fittest allocation
```

48: **end if**
49: **end while**
50: *format and return fittest allocation* =0

Chapter 5

Requirements and Design

5.1 User Requirements

The implementation should allow the user to easily request a team allocation from the readily optimised algorithm if the problem is close to the one for which the algorithm is optimised. Additionally, it should allow controlling, observing and optimising the algorithm behaviour to transfer it to other team allocation problems. It should also provide automated functionalities for validating the input and an automated way of handling invalid values, however only with the user's confirmation.

The implementation should allow the user to:

- **UR1** Install the gem from remote and local sources
- **UR2** Validate the data set and all other input before giving them as input to the algorithm
- **UR3** Easily run the algorithm for the software engineering project problem by providing only the data set
- **UR4** Separately handle missing values in the data set
- **UR5** Choose if the missing values should be automatically handled
- **UR6** Provide custom grade to skill mapping
- **UR7** Fully customise the algorithm behaviour
- **UR8** Request statistics to facilitate optimisation
- **UR9** Adapt the algorithm for other team allocation problems by optimising the parameters

5.2 System Requirements

On the other hand, apart from the algorithm specifics, the implementation should alow:

- **SR1** Platform Independant use
- **SR2** Remote installation
- **SR3** Optimised performance for the specific problem
- **SR4** Suggest solution only by getting a data set
- **SR5** Validate all user input
- **SR6** Notify the user immediately for any invalid values and explain what is the problem
- **SR7** Detect and handle missing values
- **SR8** Notify the user for any missing values and provide a way of the user deciding if the values should be automatically handled
- **SR9** Flexible and fully customisable
- **SR10** Provide algorithm behaviour insights
- **SR11** Solution validation according to a reliable heuristic
- **SR12** Provide solid base for future development into general team formation problem solver

5.3 Platform

The algorithm will be entirely implemented in the Ruby programming language, using version 2.6.5. It is an interpreted high level, general-purpose programming language, designed in the mid-'90s by the Japanese computer scientist Yukhiro "Matz" Matsumoto[42]. Its goal is providing the opportunity of quick and straightforward creation of web applications. It is dynamically typed, supports a wide variety of design paradigms, and as well as the very elegant syntax, which allows using it for a wide variety of applications and produces stylish but powerful code. The final software product will be in the form of a Ruby Gem. The RubyGems package manager gives a standard format for creating, distributing and using libraries, written in the Ruby language. Those software packages are called gems. They have extensive use for extending the functionalities of Ruby projects and quickly distributing reusable functionalities.

5.4 Development Methodology

The whole project was undertaken using an agile approach. As the project is about presenting a novel approach to solving a specific problem, using unfamiliar at that point method, the chances of things not going according to plan was significant. Which turned out to be the case, to some extent. Therefore, this approach has been chosen over the traditional waterfall method, where the design needs to be finalised before any implementation takes place. Extensive research in the field of team formation algorithms, swarm intelligence and their combinations was performed, and initial design outline was defined. However, the development process was performed based on constant and thorough analyses on what is achieved so far, more profound research and updates on the design and implementation goals. The overall work on the project was separated in sprints, each of which usually lasted around two weeks. Before each sprint, clear goals have been set. During the sprints, thorough notes on what has been achieved, what is yet to be achieved and any new, unexpected factors were taken. After the sprint, all the notes have been analysed, and an updated plan for the new sprint has been created, along with any amendments to the design and implementations requirements. This approach allowed for easier and successful adaptation to changes, new information and ideas and their implementation, as well as tackling unexpected roadblocks.

5.5 Design Specifications

5.5.1 Use Cases

When it comes to the use cases the implementation has to satisfy, the set of provided functionalities is not very long, due to the scope of the project. The primary use case is allocating the students to teams, which can be achieved either by using the tested default or custom values for each parameter. However, the program provides three additional features, that check and prepare the data in advance before being used as in input in the algorithm.

First of all, the user will be able to check whether the data set he/she will provide as an input to the algorithm has a valid format and content. Furthermore, validation for all the rest of the optional parameters will be available. It will be most needed, if the user wants to use and verify a custom way of specifying the relationship between skill and grade, since providing grade to skill mapping containing missing or duplicate corresponding skill values to a given grade, may cause unintended behaviour. For example, if there are seven skill levels$[1 : 7]$ and the grades are in the $[1 : 100]$ range, if a there is a grade with no corresponding skill, the value

will remain as it is, and generate significant distances between students with that grade and any other students they are matched with in the candidate solutions. Furthermore, since the validation for all possible parameters will be implemented, to ensure the smooth run of the algorithm, it will also be provided to the user to be used.

The other mean of the user interacting with the implemented software is to fill any missing values in the dataset before the algorithm starts. There are plenty of reasons why there may be a missing field in a student's characteristics. Those include student deciding to not provide information about the given attribute(even though the "prefer not to say" option is present for both gender and ethnicity), any problems with the data acquisition and data set generation, occuring further back in the overall team formation cycle, as well as missing grades, which may be simply result of a student moving from other department/university with unknown or irrelevant past marks.

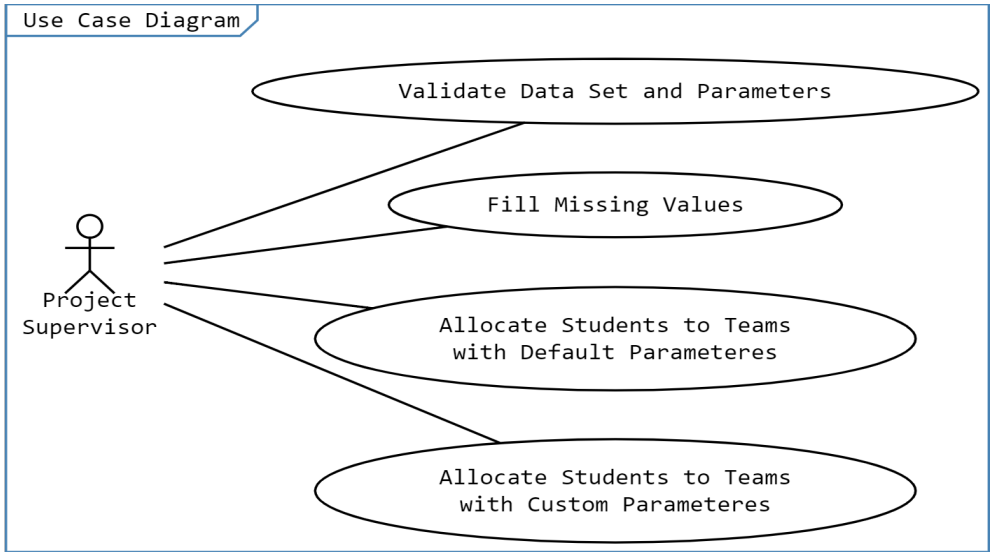

Figure 5.1: Use case diagram

5.5.2 Class Structure

The class structure of the implementation was designed following the use case diagram(figure 5.1) and the algorithm specifics, to create relevant, self-explanatory and use-suggesting naming. First of all, the data validation and the handling of the missing values are encapsulated in a separate class, and then the algorithm structure is separated in three classes - one responsible for instantiating the needed objects and running the algorithm, and the others representing the main objects in the algorithm - neighbourhood and particle.

MBPSO Class

The MBPSO class is the class responsible for controlling the whole workflow of the program. It is the class that will need to be instantiated by the user when adding the gem functionality to the desired external application/web application. It has only one public method, which is responsible for starting, controlling and finishing the run of the algorithm and is to be called by the user when the team allocation process should begin. Its function is supported by additional private methods, encapsulating the rest of the needed functionalities.

Regarding the variables, there is one for each parameter passed, as well some additional ones responsible for different control mechanics needed for the class. All of the variables in this class are with private visibility. Everything needed by other classes is passed to them as a parameter, either when instantiating the object, or when calling a function from the given class.

Validation Class

The $Validation$ class has a single instance created in the $MBPSO$ class but is also available to be instantiated by the user if needed. It provides several public methods covering the validation of all parameters and returning them if they are matching the validity criteria. There is also one private method - $raise_arg_error$, which is used by all public methods for throwing exceptions when necessary but does not have a reasonable use outside of that.

MVH Class

The MVH(missing values handler) class provides ways for detection and handling missing values in the data set. Similarly to the $Validation$ class, it has a single instance in the main class and can also be used by the user. It has only one public callable method - $fill\_missing\_values$, which according to the user's preference indicator, handles the missing values and returns the most frequent genders and ethnicity, as well as the mean and the standard deviation of the grades in the class. To achieve that, it uses two separate private methods - $check\_missing\_values$ and $calculate\_stdev$. The former iterates over the whole data set and signals if there are any missing values, along with their locations in the data set. Whereas the latter calculates and returns the mean and the standard deviation of the set of grades.

Neighbourhood Class

The other two classes are directly related to the actual algorithm implementation. The first one and the one which has at least one instance in the algorithm driver class is the $Neighbourhoodclass$. It provides the $MBPSO$ with methods for manipulating the particles that belong to it, as well iterating them and changing their inertia and control parameters.

Particle Class

The functionality of the $Particle$ class relies on two public methods and several separate private ones, which are encapsulating specific behaviours, needed by the public ones. The methods available to the owners of objects of this class, which represent all the needed functionality of the class are $update\_velocity$ and $update\_position$. The functions of this class are the most complicated part of the implementation. Therefore, the method structure had to be as transparent as possible with a sensible separation of responsibilities between classes and optimal reusability of code.

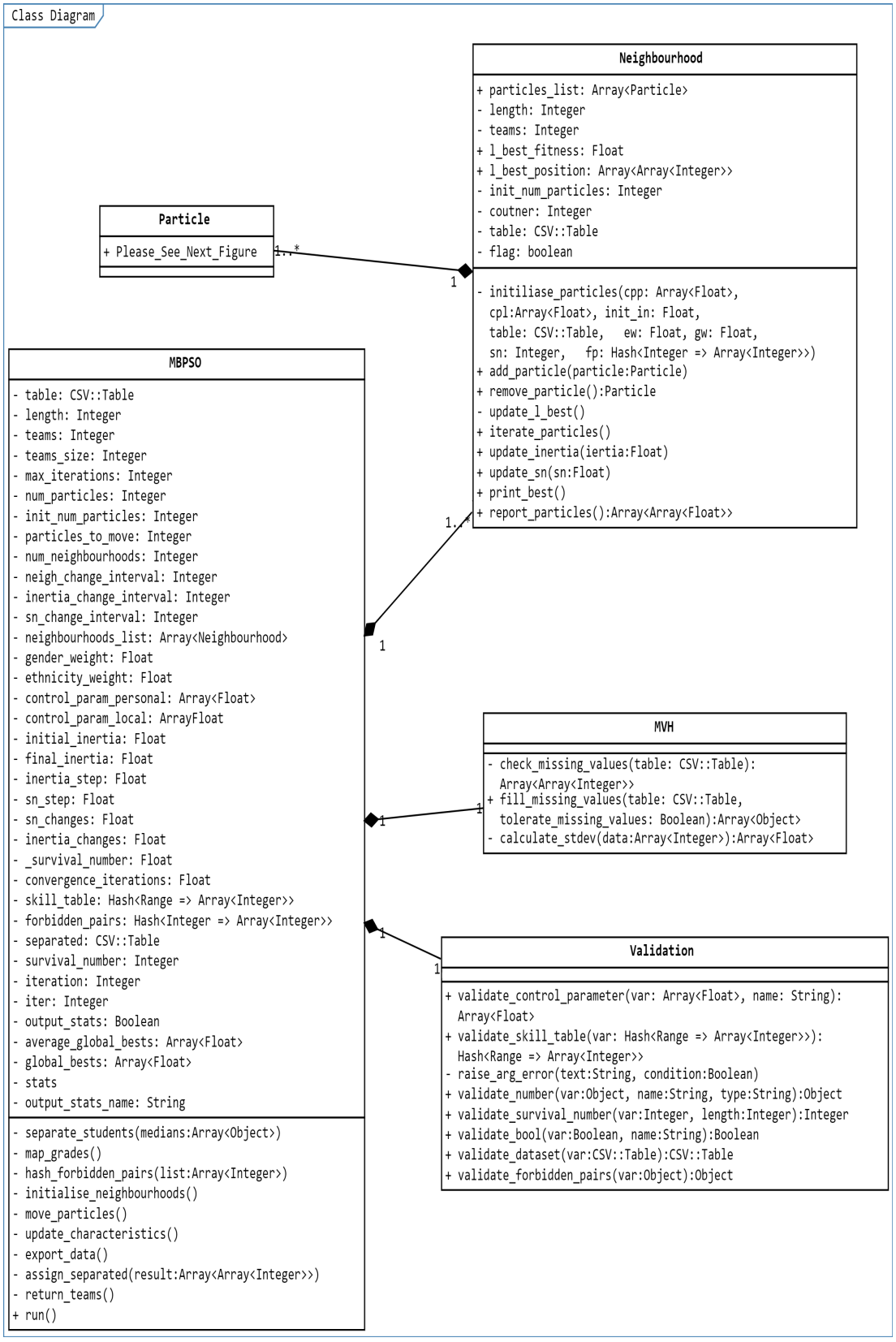

Figure 5.2: Class Diagra

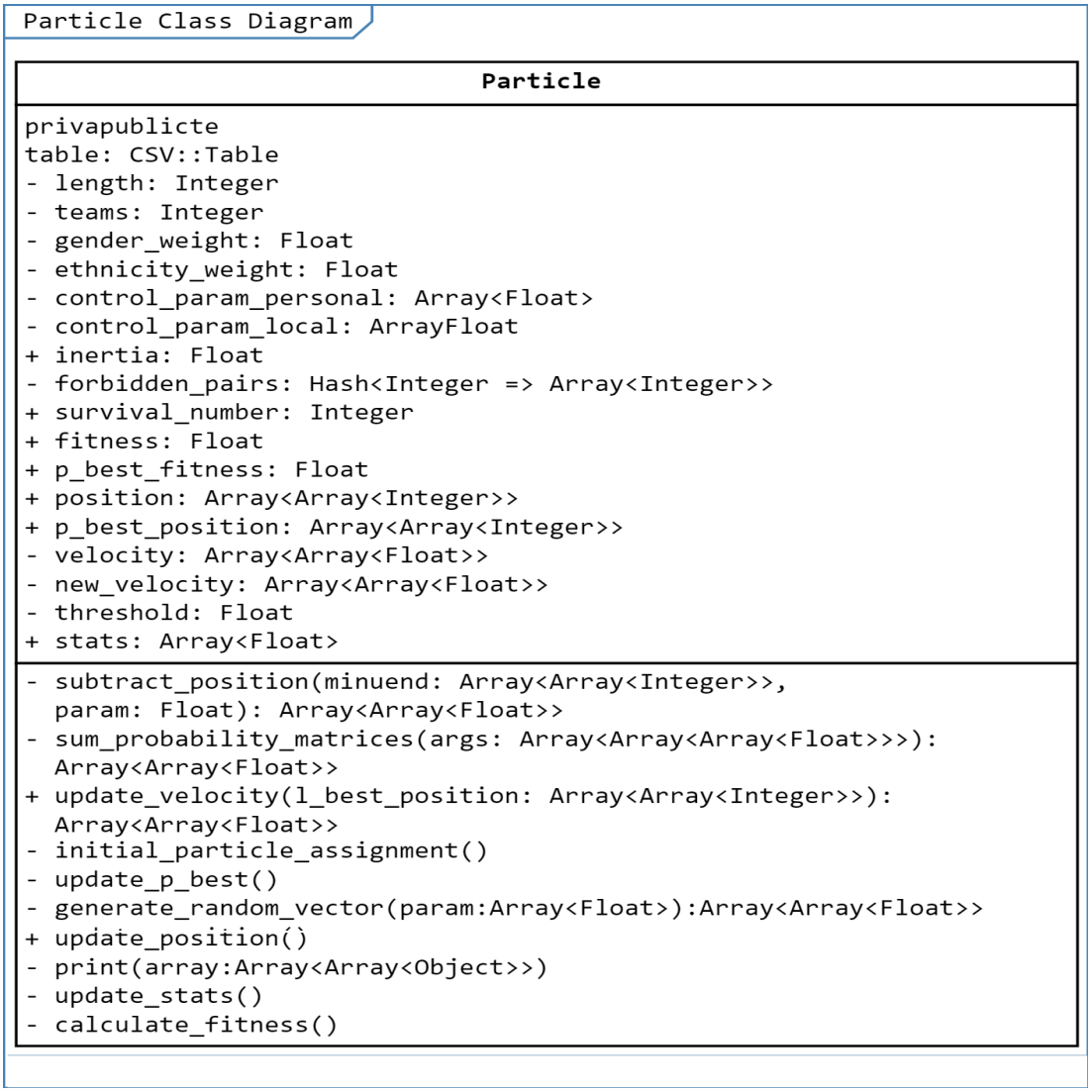

Figure 5.3: Particle class structure(part of the class duagram)

5.5.3 Sequence Diagram

The sequence diagram is important part of each well designed piece of software. It depicts the interaction between the different objects involved in running the given software, as well as the order in which interaction and actions occur. It provides further insight into how the program works and when certain actions happen.

The actions sequence of the current algorithm is described by figure5.4 and the other figures it references - 5.5, 5.6, 10.2, 5.8, 5.9. Due to the large size of the full diagram, as well the goal of an easily readable and understandable diagram, it was separated in the given way.

Figure 5.4 outlines the general sequence of the actions involved in the team formation

process. It starts with validating and preparing all the data, parameters and object instances needed[5.5]. When all data and objects are ready, the actual team formation process is invoked. Its principal part consists of a loop, the length of which is governed by the stopping criteria of the algorithm - desired maximum allowed number of iterations and stagnation check. At each iteration, the $MBPSO$ objects invokes for each neighbourhood the method that makes each of its particles to perform an iteration. That step consists of another loop, iterating over each particle in the neighbourhood and invoking the needed methods for performing a full iteration. First, the velocity of the current particle of interest is updated[5.9]. This is a prerequisite for updating its position, which is the next action performed. With the particle having its new position, the new fitness is calculated, and the personal best fitness and fitness are updated if it is higher than the current best one. When this is completed, the particle reports its fitness to the neighbourhood object it belongs. Subsequently, when all the particles in the neighbourhood have reported their new fitness, the new local best fitness and position are checked if they need to be updated. If that is the case, they're updated with the new bests in the population. When the required the algorithm finished iterating the particles, the resulting teams are returned to the user.

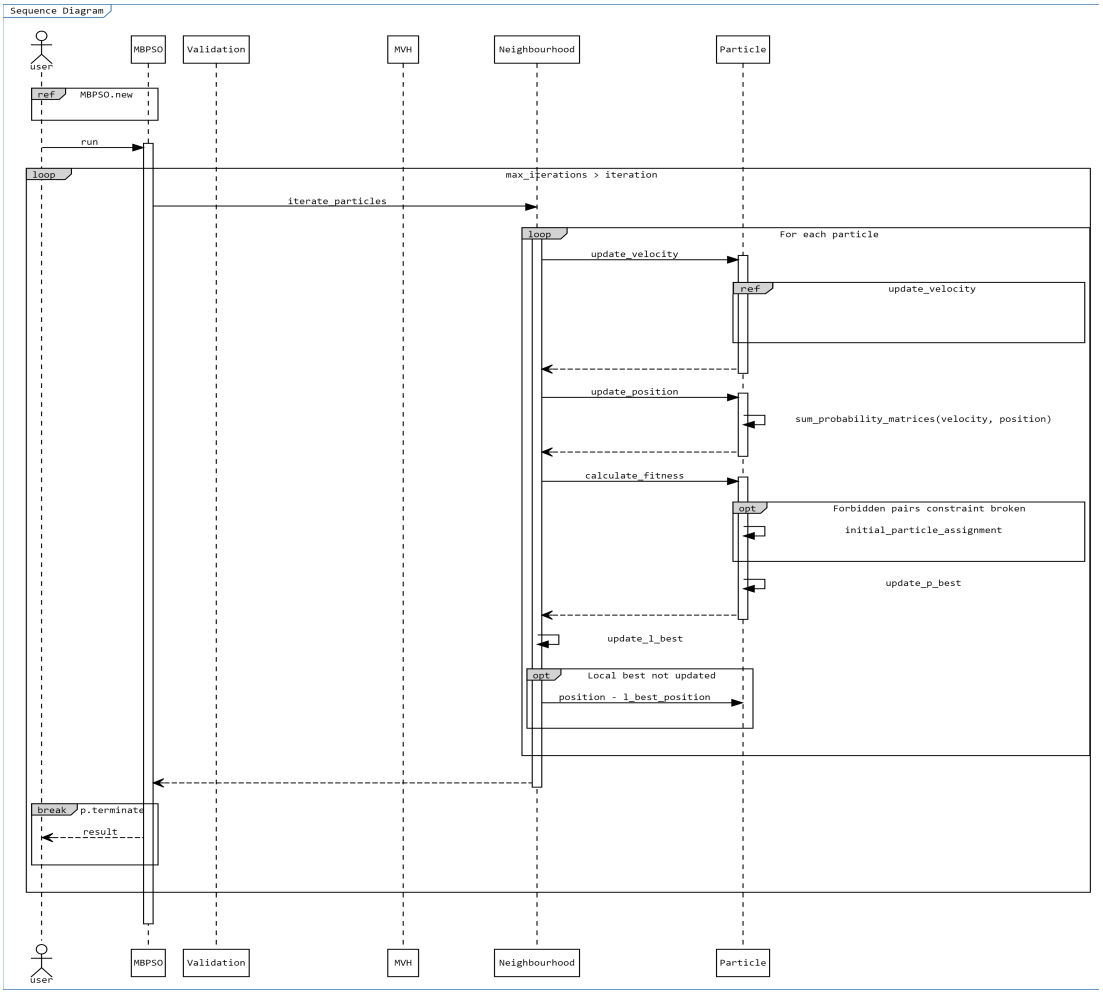

Figure 5.4: Sequence diagram showing the overall sequence structure of full run of the algorithm

The module instantiation and the data preparation that comes with it consist of three main parts - input validation, data preparation and neighbourhoods and particles initialisation, which occur in the same sequence.

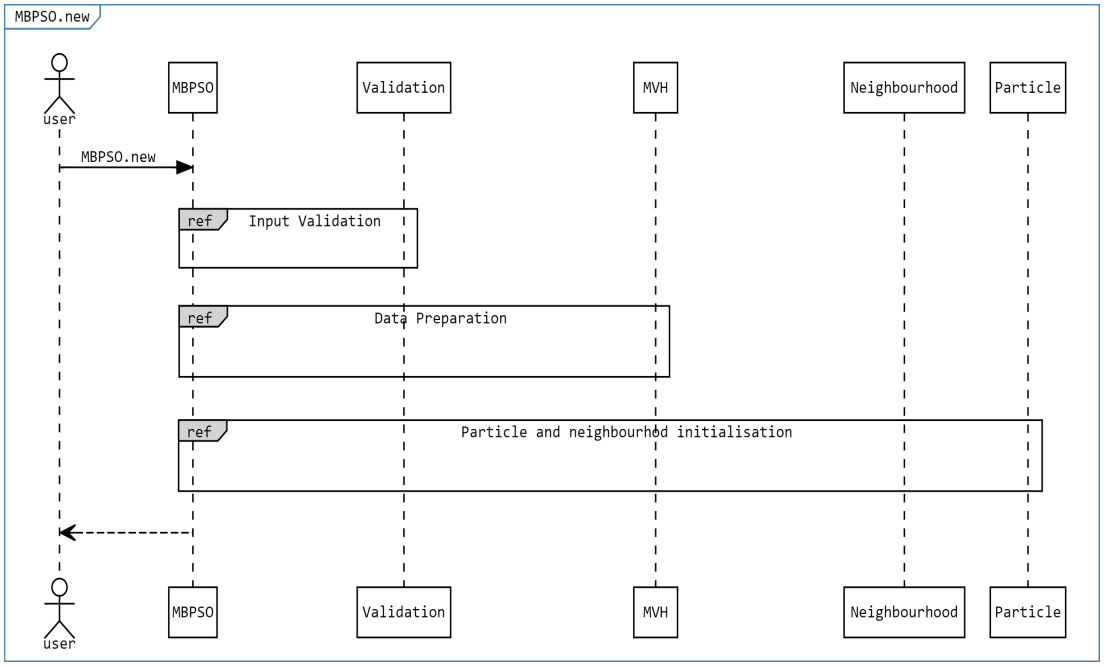

Figure 5.5: Sequence diagram showing the initialisation of an instance of the $MBPSO$ class

The validation of the data set and the parameters are the very first thing that happens when creating the object and respectively running the algorithm. Ensuring all potential problems are eliminated as early as possible in the program execution has high importance. Hence, the program hs that particular design. The validation itself consists of several method calls, checking all the input to, first of all, avoid any unintended behaviour, as well as omit the need of defensive style of implementing the functionalities. The first variable to be checked is the data set which is the only required parameter with no default value. Which is also the parameters that is validated towards the most extensive set of criteria. The rest of the parameters follow..

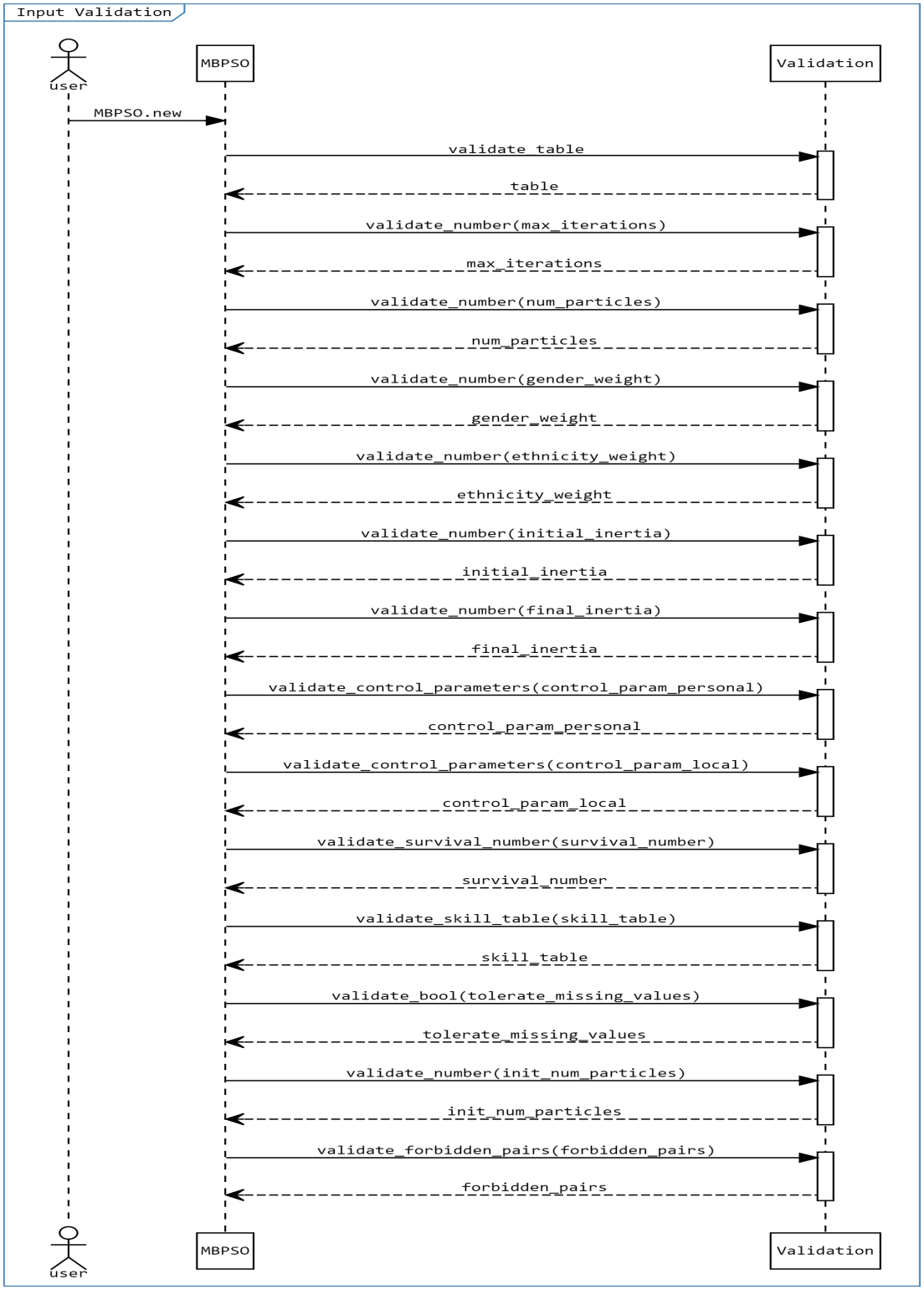

Figure 5.6: Sequence diagram showing the input validation part of the *MBPSO* initialisation process

The input validation is followed by calls to two methods - $hash\_forbidden\_pairs$, which turns

the list of forbidden pairs to a hash the required format for using it, as well the *map_grades*, which turns the grade values to skill values, accordin to the user input or default value.

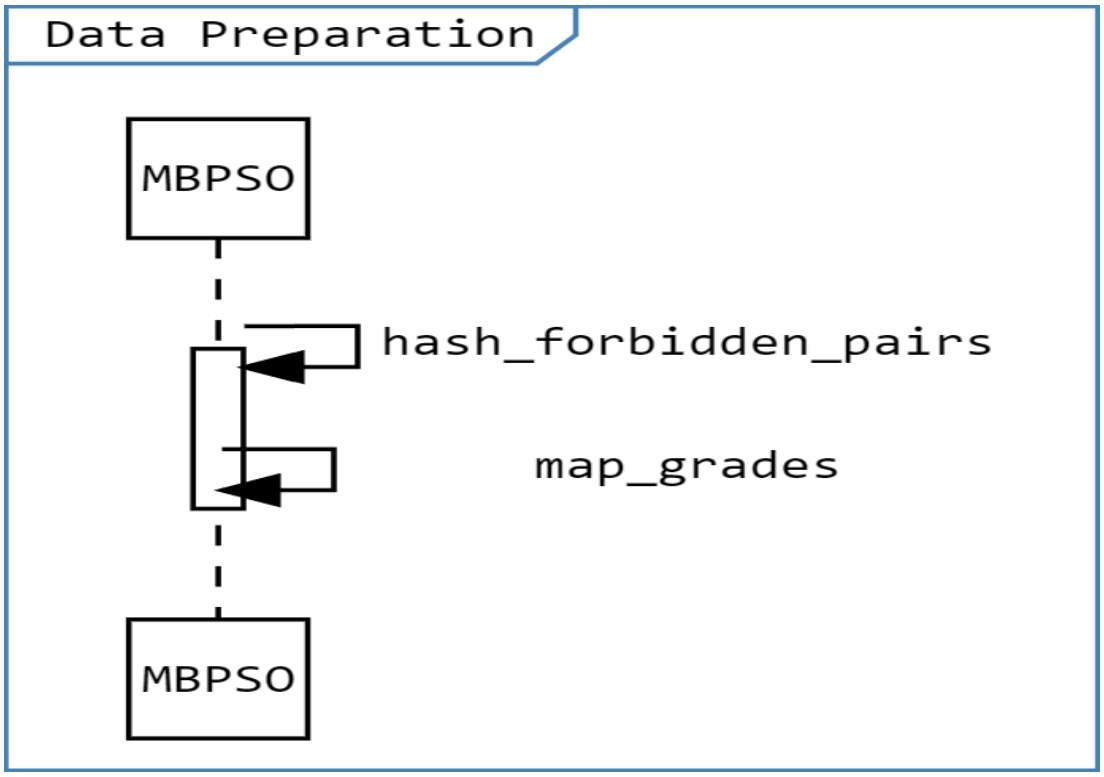

Figure 5.7: Sequence diagram showing the data preparation part of the $MBPSO$ initialisation process

The last precondition for running the algorithm is the initialisation of the required number of neighbourhoods and particles. Since each one belongs to a neighbourhood, a method initialises the particles, which the neighbourhood constructor calls. Therefore, the $MBPSO$ constructor invokes the creation of the needed neighbourhoods, and then with the creation of each one, the needed number of particles are constructed as well.

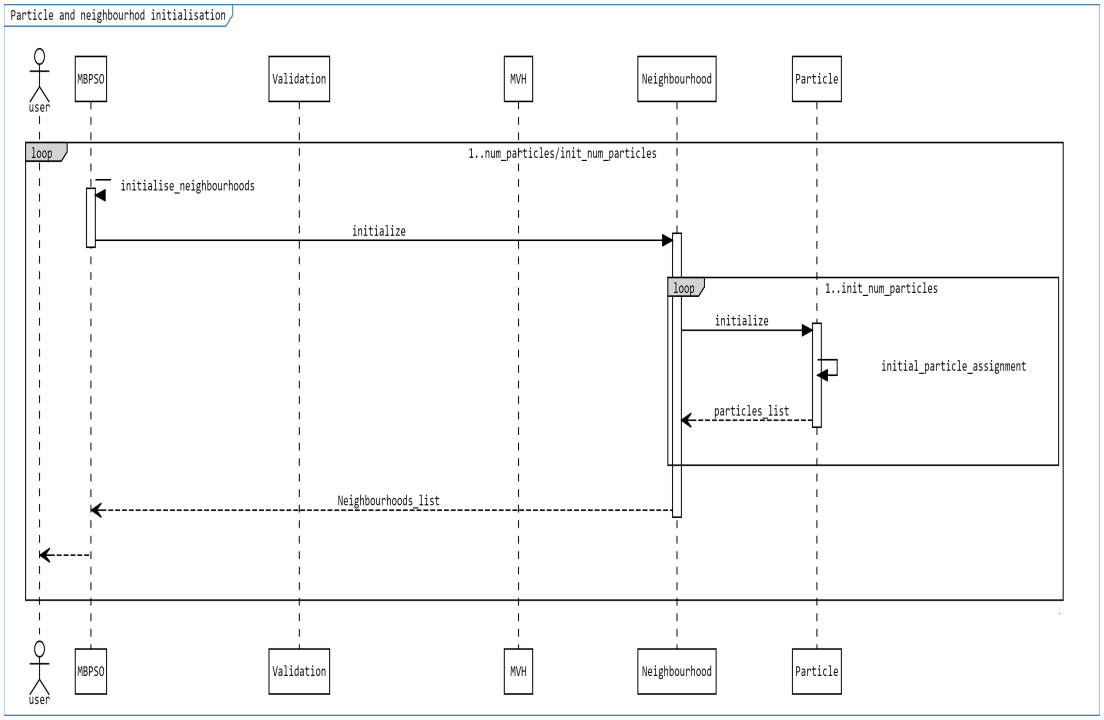

Figure 5.8: Sequence diagram showing the neighbourhood and particles initialisation part of the $MBPSO$ initialisation process

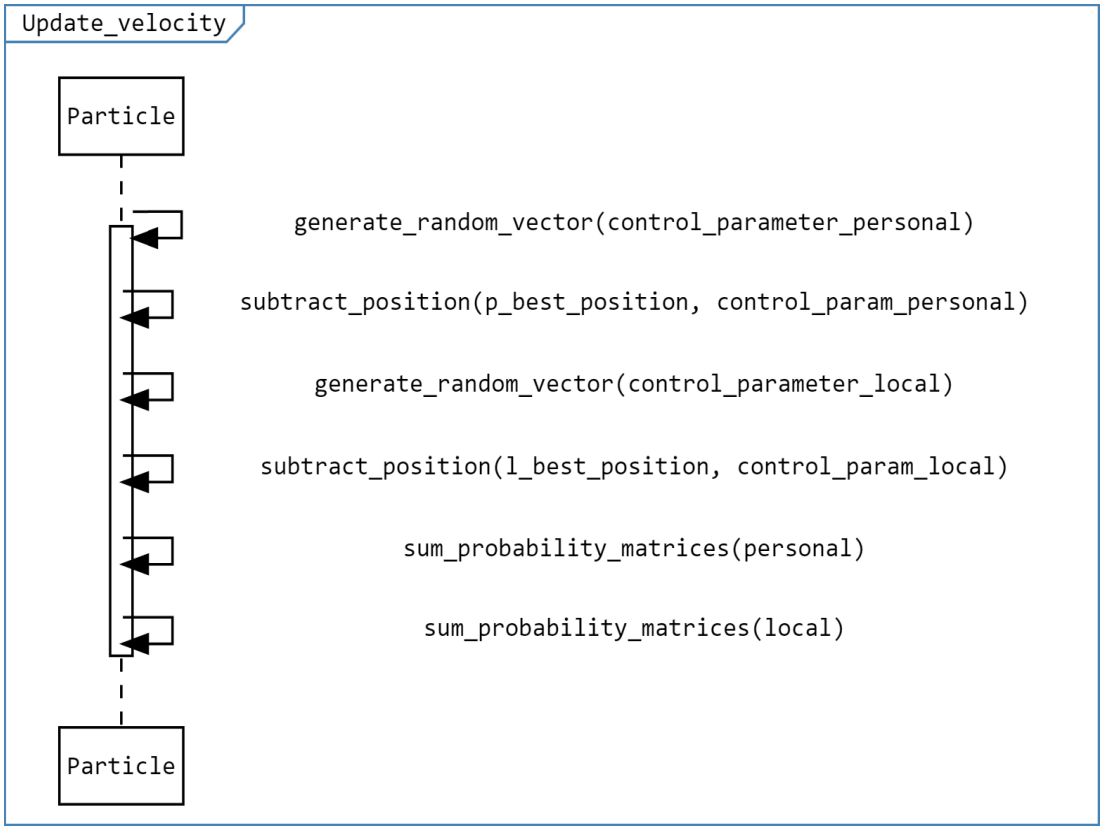

Figure 5.9: Sequence diagam showing the update of a particle's velocity

Chapter 6

Implementation

6.1 MBPSO Class

The MBPSO class acts as the main class of the implementation structure. To interact with the gem, the user has to instantiate the gem. When instantiating it, it takes one compulsory argument *table* - the dataset in a $CSV :: Table$ format and can also take several optional parameters, each of which having pre-set default values, or the so-called in Ruby named parameters. Hence, the user has the option only to provide the dataset and stick to the tested default parameters and behaviour, or to customise the algorithm. The optional parameters and their default values are below.

Table 6.1: Optional Parameters for PSO Class

Parameter Name	Type	Default Value	Role
$team_size$	Positive $Integer$	4	Size of each team
$max\_iterations$	Positive $Integer$	10000	The maximum number of iterations
$convergence\_iterations$	Positive Integer	300	The number of iterations without update that will cause the algorithm to terminate

num_particles	Positive *Integer*	20	Total number of particles in the population
particles_to_move	Positive Integer	2	The number of particles that will be moved at each neighbourhood update
neigh_change_interval	Positive Integer	100	The iteration interval at which neighbourhood change will occur
gender_weight	Non-negative *Float* or *Integer*	9	The weight of the gender difference when evaluating the particle's fitness
ethnicity_weight	Non-negative *Float* or *Integer*	9	The weight of the ethnicity difference when evaluating the particle's fitness
initial_inertia	Non-negative *Float* or *Integer*	0.9	The initial value of the inertia weight
final_inertia	Non-negative *Float* or *Integer*	0.2	The value which the inertia will reach over time
inertia_changes	Positive Integer	300	The number of times inertia will be updated before reaching the final value

$inertia\_change\_interval$	Positive Integer	10	The iteration interval at which inertia will be updated
$control\_param\_personal$	Array, consisting of 3 non-negative $Float$	$[0.2, 0.4, 0.4]$	Set of parameters controlling the behaviour and impact of the personal cognitive term on the velocity
$control\_param\_local$	Array, consisting of 3 non-negative $Float$	$[0.6, 0.2, 0.9]$	Set of parameters controlling the behaviour and impact of the local cognitive term on the velocity
$survival\_number$	Integer $\in [2 : numberofstudents]$	nil (if it remains nill, it becomes equal to the number of students)	Maximum number of swaps at each position update
$final\_survival\_number$	Positive $Integer$	8	The final values the survival number will reach over time
$sn\_change\_interval$	Positive Integer	10	The iteration interval at which inertia will be updated

sn_changes	Positive Integer	300	The number of times survival number will be updated before reaching its final value
skill_table	Hash, having *Integer* Ranges as keys and positive *Integer* as values	{0..39 => 1, 40..49=> 2, 50..59 => 3, 60..69 => 4, 70..79 => 5, 80..100 => 6}	Controlling the behaviour of the grade to skill mapping
forbidden_pairs	*CSV* :: *Table*, having 2 columns	*nil*	Specifying the ID's of students who cannot be paired together
tolerate_missing_values	Boolean	true	Indicates whether the user agrees on automatic replacement of missing if any are present
$init_num_particles$	Positive Integer	3	The initial number of particles in each neighbourhood
output_stats	Boolean	false	Flag indicating whether statics should exported
output_stats_name	String	'stats.csv'	The name of the file where the data will be written

The purpose of the class is to validate all input(more in section validation), make everything ready for algorithm execution(more on data set preparation in section 6.3), initialise the required number of neighbourhoods by passing them the needed values, so they can successfully do their part. After neighbourhoods and respectively particles creation, the PSO class is responsible for starting the algorithm and controlling both the behaviour via supervising the dynamic concepts and termination criteria of the algorithm. Finally, it is also responsible for algorithm termination and output formatting.

6.1.1 Grade to Skill Mapping

After all the parameters are proven to be in the correct desired format, and there are no cells with missing values, there is one more step left before going to the execution of the actual algorithm. That is, transforming the grades of all students into numerical values, indicating their skill. The method responsible for that is *map_grades*. It uses the *skill_map* hash, either its default value, shown in figure 6.1, or the one specified by the user as an optional argument. The method iterates over the 'Grades' column of the data set and compares the grades with the keys in the hash. When it finds the correct value related to the key, changes the grade with the corresponding skill indicator. However, in the data set that the program uses, the column header remains 'Grade'.

Furthermore, the algorithm uses a duplicate of a the passed *table* variable. If the program directly takes the passed value, assign it to a field and manipulate it, it would remain with skill values. If the user reruns the algorithm, without acquiring the data set again, the results of the run will be practically useless.

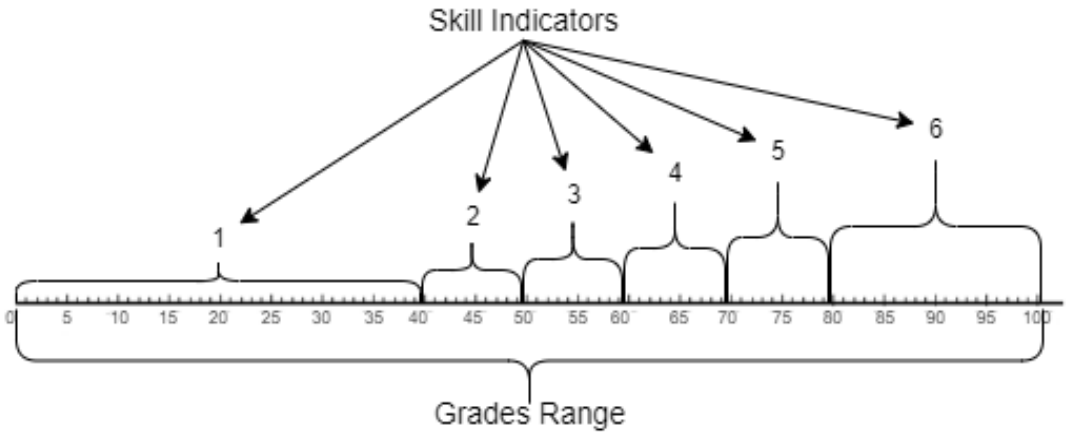

Figure 6.1: Default Grade to Skill Transfer Table

6.1.2 Separating Extra Students

As explained in 2.1.2, considering all possible variants of assigning a number of students that is not a multiple of team size and the number of teams, tremendously increases the search space. To avoid that complication, the program provides a way of handling the extra student. The current problem has a limited variety of attributes for each student and their values. Therefore, in large data sets, there are many students with similar representations. The *separate_students* method checks if the division of the class size by the teams size has a remainder. If there is one, it looks for that many students with 'average' characteristics. Average means, having the most frequent gender and ethnicity in the data set, as well as grade in the range $[\mu - \sigma : \mu + \sigma]$(where μ is the mean of the data set, and σ is the standard deviation). It looks for such students and removes them from the data set before it is passed to the neighbourhoods. Those values are taken from the MVH class, described in section 6.3. If the method does not find enough eligible students, it starts a new search, checking only for the grade criteria. In the worst-case scenario, if more students still need to be separated, random students are taken from the data set. Those students are stored in the *separated* field for later use.

6.1.3 Creating Neighbourhoods

Neighbourhoods are instantiated via the *initialise_neighbourhoods* method and are stored as entries in the *neighbourhoods_list* array. The number of neighbourhoods to be created is calculated as the result of the integer division of the total number of particles by the initial number of particles. That gives the number of "full" neighbourhoods. Full neighbourhood means that it has as many particles as specified by the corresponding parameter. Then it is checked if the division had a remainder. If that is the case, an additional neighbourhood is created, containing the remaining particles.

6.1.4 Running the Algorithm

The *run* method of the $MBPSO$ class works as the main method in the implementation. The user has to call it to start the algorithm. It provides overall control over the algorithm run. It consists of two parts - controlling the iterations and preparing the output. The iterations are controlled by a while loop, iterating until it reaches the maximum number of iterations, or until the algorithm converges. However, the latter is only possible after all particles have moved to a single neighbourhood, and the inertia weight and the survival number have reached their final values, to avoid premature convergence.

Iterating Particles

From *MBPSO*'s point of view, the particles iteration is pretty straightforward. It iterates over the list of neighbourhoods and calls the *iterate_particles* for each one. For testing and statistics exporting purposes, the algorithm also keeps track of the local bests. It adds local best fitness for each neighbourhood into a temporary array, which has its value reset at each iteration, due to purposely falling out of scope. This array holds the local bests for each neighbourhood. They are used for evaluating the average local best, as well as the current global best. At the end of the iteration, they are pushed into the *average_global_bests* and *global_bests* arrays respectively.

Controlling Algorithm

At the end of each iteration, the algorithm checks two things - whether it has converged and whether any control parameters should be updated. It checks for convergence by checking the *counter* field of the first neighbourhood, as the run can terminate only when a single neighbourhood is left. If the neighbourhood signals for convergence, the iterations counter is made equal to the maximum allowed number of iterations, which makes that the last iteration performed. When it comes to controlling the dynamic parameters, the control of their updates is encapsulated in the $update_characteristics$ method. It compares the current iteration number with the iterations at which inertia, survival number and topology are to be changed. If the topology needs to be changed, the *move_particles* method is called, after that if the inertia or survival number, need an update, their values are decremented by the *inertia_step*, and the sn_step values. Then for each neighbourhood, the *update_inertia* and *update_sn* are invoked, which forward the parameters to all their variable. Also, the counter of the first neighbourhood is set to 0, to avoid termination before the parameters reach their final values.

Moving Particles

If the iteration number suggests particle moves, the *move_particles* is called. It consists of a loop, that iterates until it moves the required number of particles, or there is only neighbourhood left. It uses a pointer, stored in the *iter* and that makes it possible to move particles to new neighbourhoods on a roulette principle. That means it does not always start moving to a neighbourhood with a fixed index(for example 0) but keeps adding to the neighbourhood following the one where it last added a particle, and when all neighbourhoods have an equal number of particles, the pointer goes back to a value of zero. The implementation uses the

remove_particle method from the *Neighbourhood* class. It uses it to remove the last element of the last neighbourhood. If it returns a particle(meaning the neighbourhood is not empty), it moves the particle and increments the pointer. Though if *nil* is returned(meaning the neighbourhood is empty), the last element in the list of neighbourhoods is removed and starts over. An example of a full transition from initial configuration with $num\_particles = 11$, $init\_num\_particles = 3$ and moving two particles at a time is illustrated in figure 6.2. Also, the method sets the counter of the first neighbourhood to zero.

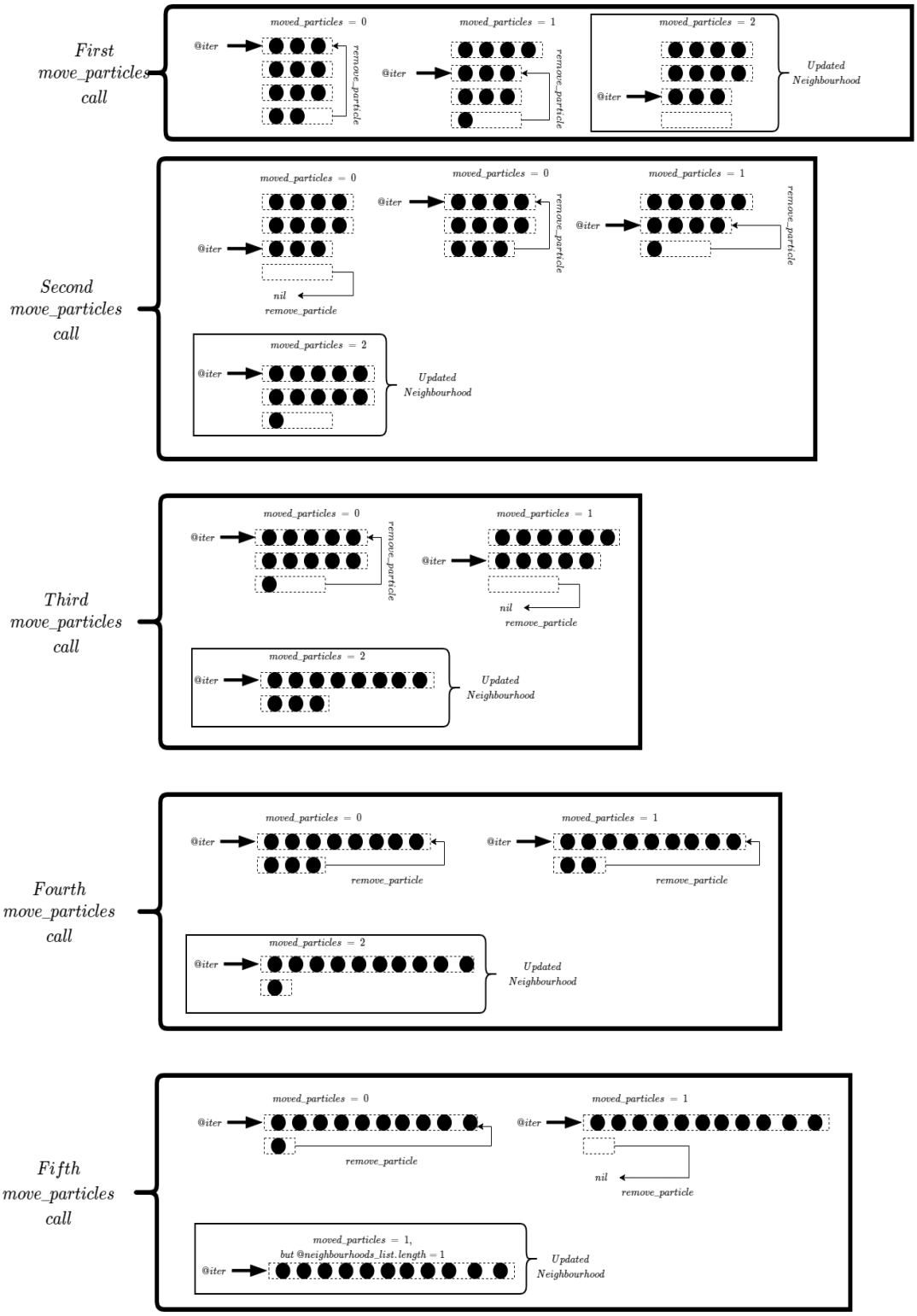

Figure 6.2: Moving particles example

Exporting Statistics

When the algorithm terminates, it checks if the user requested statistics export. If that is the case, the *export_data* method creates a new .csv file in a *data* folder, inside the current directory of the algorithm with the specified name(the default is *stats*). This functionality allows the user to implement an automated way of testing multiple algorithm configurations. The algorithm can run in the background, performing many runs and storing the data about them in separate files. Code listing 10.5 provides an example of a script reading configuration from an external file and saving the statistics in files with relevant names. This is how the majority of the empirical tests were performed while implementing the algorithm. In the resulting file, each column represents the data for a single iteration. Also, the first row will contain the global best for each iteration, while the second row will represent the average of the local bests for each iteration. All other rows represent the fitness along the run for each particle. Listing 10.4 provides Matlab code, that automatically visualises all the exported data, regardless of the number of particles and iterations performed.

Assigning Separated Students and Returning Results

Finally, the run method checks whether there are any students in the list of separated students. If there are any, it randomly assigns them to different teams. Assuming their attributes have all the most frequent values and optimal teams are formed, that should not contaminate the solution quality.

The last step in an algorithm run is to format the produced solution. The *return_teams* method is called, which takes the best position of the only neighbourhood left and transforms it into an array. The array contains an equal to the number of teams sub-arrays, each containing the IDs of students that belong to the given teams. And also prints the attributes of the students for each team.

Figure 6.3: *Small part of the output, showing its format*

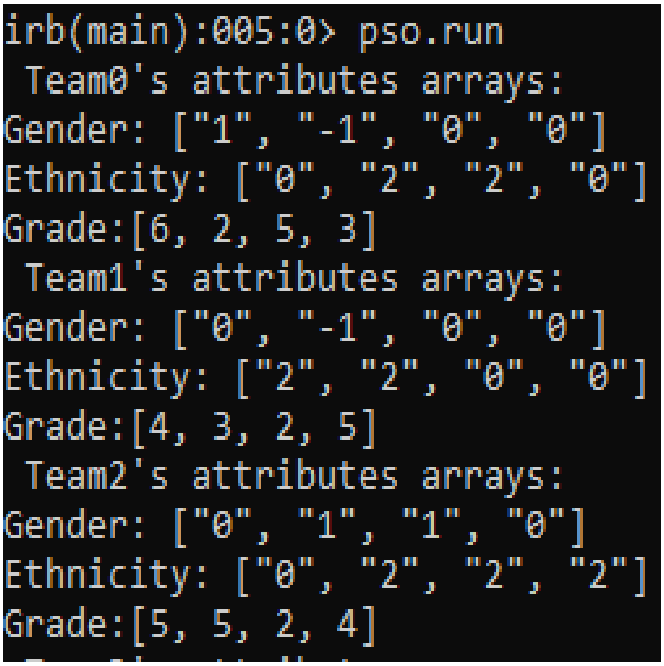

Figure 6.4: *Small part of the printed stats about the teams*

6.2 Validation class

Chapter 6.1 briefly mentions input validation, which is described in this section. To guarantee a smooth run , the algorithm validates the data set and all optional parameters according to their desired format. This functionality is held inside the *Validation* class. It has its methods automatically called from inside the *initiliaze* method of the *MBPSO* class, i.e. when the user instantiates the gem. The goal is to notify the user about any problems before performing any other actions and wasting time. If it finds an uncompliant parameter, *ArgumentError* exception is thrown, along with a short description explaining the problem. Even though this part of the program follows a hard-coded sequential execution of all steps, all the functionalities are available for the user to validate everything needed separately. All validation methods, regardless of which particular parameter they check, if it is in the required format, returned the value of the parameter so that it can be directly assigned to a field or manipulated in another way.

6.2.1 Throwing errors

All of the variables validated using this class are supposed to be used as parameters when creating an instance of the *MBPSO* class.Therefore, all the exceptions that any method of this

class would throw are of the *ArgumentError* type. However, every exception has a different explanation, aiming to facilitate troubleshooting. A sensible approach for implementing that in a neat and readable manner is extracting that functionality into a separate method. Hence, the simple $raise_arg_error$ was implemented. It takes two arguments - *text* and *condition*. Both of them are formed in the described below methods and passed to it. The former one represents the error message to be shown, which is dependant on the validated valued and the exact problem found. Whereas, the second one is a logical condition, again generated according to the validated value, which the current method evaluates and raises an error accordingly.

6.2.2 Data Set

The data set containing the information about the student is the most important and the only compulsory parameter for instantiating the gem. Therefore, it is inevitable to validate its format and content. This is the very first thing that the happens when calling $MBPSO.new()$ and is achieved by calling the *validate_table* method, which undertakes the validation and iteratively uses the $raise_arg_error$ method to check all the needed conditions and raise errors with the relevant messages if needed. The method also allows the data set to have missing values, the handling of which is explained later in this section.

The first things it checks are if the variable passed is an instance of the $CSV :: Table$ class, whether the headers list contains the values $'id', 'Gender', 'Ethnicity', 'Grade'$ and if there are duplicating student ID's.

After ensuring the data set is in the same format and all the ID's are unique, the method validates all the values in the dataset, according the following regular expressions:

```
gender_regex = /^(-1|0|1)$/
ethnicity_regex = /^(-1|[0-4])$/
grade_regex = /^(100|[1-9]?[0-9])$/
```

Listing 6.1: Regular expressions validating data set values

Which ensure that:

1. Gender indicator values are integers in the range $[-1 : 1]$ or empty, 0 for male, 1 for female and -1 for student who prefers not to say.

2. Ethnicity indicator values are integers in the range $[-1 : 4]$ or empty, where 0 stands for white, 1 for mixed/multiple ethnic groups, 2 for Asian or Asian British, 3 for Black/Af-

frican?Carribean/Black British, 4 for other ethnic group and -1 for students prefer not to say.

3. Grade indicator values are integers in the range [0 : 100] or empty.

Because of the large expected size of the data set, when it throws an exception, it also indicates the ID of the problematic student entry, along with an indication of which value caused the problem to aid quickly finding the invalid value.

```
Invalid ethnicity value for student with ID = 46.Required: integer in the range [-1:4]. (ArgumentError)
```

Figure 6.5: *Example exception thrown for invalid value inside the dataset*

6.2.3 Numerical parameters

There are two types of numerical parameters according to their allowed values - positive integers and non-negative floats or integers. The $validate_number$ method contains the validation functionality for both types. The method takes three parameters:

- *var* - the variable that is to be validated
- *name* - the variable name, so it can be added to the error message*
- *type* - value indicating what is the required variable format - *pos_int* should be specified for positive integer and nn_um should be specified for a non-negative number.

**A much neater solution would be to extract the variable name from the passed variable and avoid passing additional parameters. However, the only way that could Ruby can achieve that is by using binding() and the eval() method. The latter is a potential security threat when used for evaluating user input as it is sent directly to the system. If such an approach is used, the method has to increase the security level and perform checks for tainted user input. To avoid that kind of overhead, the method(and plenty of the other validation methods) are implemented by using a parameter for the variable name.*

The method uses a switch condition, which according to the specified *type* generates the needed *text* and *condition* and raises an exception if needed, telling the user that the input is invalid. In listing 6.2 the implementation can be seen, as well as the general structure of generating error message text, condition and calling the $raise\_arg\_error$ method.

Furthermore, there is a third particular case of allowed values for a particular parameter - survival number. However, it is separated in another method and is be discussed further in

this section.

Positive integers

There are plenty of parameters, which are required to be positive integers. If the method encounters any of them being an instance of a different class, rather than *Integer* or having non-positive value, the specified condition is not met, and it throws and saying that the specified parameter is not a valid integer.

Non-negative numbers

Similarly, for the float parameters, the same type of exception gets thrown, unless the parameter is an instance of the *Float* or *Integer*(as for the given scenario integer values are perfectly fine) classes and has a non-negative value, saying that non-negative float or integer is required.

```
/pso.rb:25:in `initialize': Argument 'ethnicity_weight' is not a valid integer or float (ArgumentError)
```

Figure 6.6: *Example exception thrown for invalid single-valued parameter*

```
def validate_number(var, name, type)
      case type
      when 'pos_int'
        text = "Argument '#{name}' is not a valid positive
            Integer"
        condition = (var.is_a?(Integer) && var.positive?)
      when 'nn_num'
        text = "Argument '#{name}' is not a valid non-negative
            integer or float"
        condition = ((var.is_a?(Integer) || var.is_a?(Float)) &&
            (var >= 0))
      else
        text = 'Invalid validation call'
        condition = false
      end
      raise_arg_error(text, condition)
      var
```

```
    end
```

Listing 6.2: Validating numeric values

6.2.4 Survival Number

Validation works in the same way for the *survival_number* parameter as well, with the only difference, the allowed values for it are between 2 and whatever the number of students is. That requires an additional argument to be passed - *length*, which indicates the upper bound value for this parameter and the condition is generated accordingly.

6.2.5 Cognitive Parameters

There are two cognitive parameters that the algorithm needs for execution - *control_param_personal* and *control_param_local*. Both of them are required to be represented as arrays, containing three floats in the range $[0 : 1]$ (Integer values are also allowed, to avoid any unnecessary exceptions if 0 or 1 are passed, instead of 0.0 or 1.0). The validation is performed via a call to the *validate_control_parameters* method. It generates such conditions, so the raising error method can in case of invalid value to throw an exception, explaining which of the two parameters has an issue and what is the required format. To allow that, apart from the variable value, the method takes a *name* argument, along with the *var* one.

```
Argument 'control_param_personal' is not in the required format. Array with 3 floats in the range [0;1] expected (ArgumentError)
```

Figure 6.7: *Example exception thrown for invalid cognitive parameter*

6.2.6 Skill Table

Should the user want to specify a custom way of mapping grade to skill, he/she should specify this desired way via a hash. In the hash, the keys represent a range of grades, whereas the values they point to stand for the skill group the students falling within the corresponding range of grades will be classified. Hence, ensuring the grade range is fully covered, as well as not having overlapping ranges, is crucial for avoiding unintended behaviour. The $validate_skill_table$ method is responsible for that, which initiates raising an error if needed.

The first part of the transfer function validation consists of ensuring the keys of the hash represent ranges that unambiguously cover the full range of possible values for grades, i.e.

[0 : 100] by checking if the keys are valid integer ranges(instances of the *Range* class, for which the first and the last elements are integers, as Ruby allows other types of ranges, such as Integer ones), then the ranges are expanded and added to an array. Then it sorts the array in ascending order and checks whether:

1. The list is has 101 elements
2. The first item in the list is 0
3. The last item in the list is 100

The simultaneous satisfaction of those three conditions ensures that every value in the range [0 : 100] is included, without any duplicates.

After that, all the values, indicating the skill group of interest, are checked for being valid positive integers.

```
Argument 'skill_table' has invalid value and/or invalid coverage of the grade range (ArgumentError)
```

Figure 6.8: *Example exception thrown for invalid grade to skill mapping*

Forbidden Pairs

To validate the list of forbidden pairs, the $validate\_forbidden\_paird$ maethods checks two thins. First of all, whether the passed variable is a $CSV :: Table$ or an $Array$ and whether it has only two values per row.

6.3 MVH(Missing Values Handler) Class

The current implementation also provides an automated way of handling missing values in the data set. The goal is that the empty fields in the data set are replaced with values relatively sound with the statistical distribution of the data. Gender and ethnicity indicators are replaced with the most frequent indicators among all students, while any missing grades are replaced via approximation of the normal distribution of all grades in the data set. There are three methods involved in that step: $check\_missing\_values$, $calculate\_stdev$ and $fill\_missing\_values$.

The mentioned above methods are cascadingly invoked in the same order when $fill\_missing\_values$ is called by the $MBPSO$ constructor, or via separate user call to the method. The method $check\_missing\_values$ traverses the whole data set and records in an array the indices which point to the cells with missing values in the original data set. That array consists of three

elements, each of which is an array on his own, one for each attribute, containing the indices where the empty cells can be found for the given attribute. Then the sizes of those three arrays are checked, if all three of them are empty, it directly returns the variable value, and skips any further action. However, if there are any elements added to any of those sub-arrays, the whole two-dimensional array is returned.

The output of the check for empty cells is used for deciding what the next steps are. If it returns false, no handling is needed, and the execution of the handling of the missing value is terminated. On the contrary, if the returned value is not $false$, a warning indicating the presence of any cells to be filled is printed to notify the user that the program will handle them .

```
WARNING! There are missing values in the data set, which will be automatically handled.
```

Figure 6.9: *Example warning printed when entries with missing values are found*

The method handles the missing indicator for gender and ethnicity in a similar manner. The corresponding sub-array returned from the checking method is having its size checked, and if there are indeed missing values for that attribute, it scans the whole data set to find the most frequently occurring indicator. Then all empty cells are filled with that value.

```
if missing_values[0].length > 0
      frequencies = @table['Gender'].inject(Hash.new(0)) { |h, v|
          h[v] += 1; h }
      most_frequent_value = @table['Gender'].max_by { |v|
         frequencies[v] }

      missing_values[0].each do |x|
        @table[x]['Gender
        '] = most_frequent_value
      end
    end
```

Listing 6.3: Handling missing values in 'Gender' column

Handling gender and ethnicity indicators is pretty straightforward, due to the relatively small variety of possible values. When it comes to grades, things are a bit more complicated, due to the opposite reason. Replacing values only with the one with the most frequent one or

the rounded mean all grades would be very irrelevant. Henceforth, an alternative approach was used - replacement according to Gaussian distribution approximation. The implementation assumes that the grades in the data set have a uniform distribution. which however is not guaranteed to be the case.

According to the original definition of the normal distribution, $\approx 68\%$ of the data should lie within the $\mu \pm \sigma$ interval, $\approx 95\%$ should lie withing the $\mu \pm 2\sigma$ interval and $\approx 99.7\%$ of the data should lie within the $\mu \pm 3\sigma$ interval. Where μ stands for the mean of the data and σ represents the standard deviation of the data given by:

$$\sigma = \sqrt{\frac{\sum(x_i - \mu)^2}{N}} \tag{6.1}$$

where x_i stands for the i-th value in the population and N represents the population of the size.

Source:https://towardsdatascience.com/understanding-the-68-95-99-7-rule-for-a-normal-distribution-b7b7cbf760c2

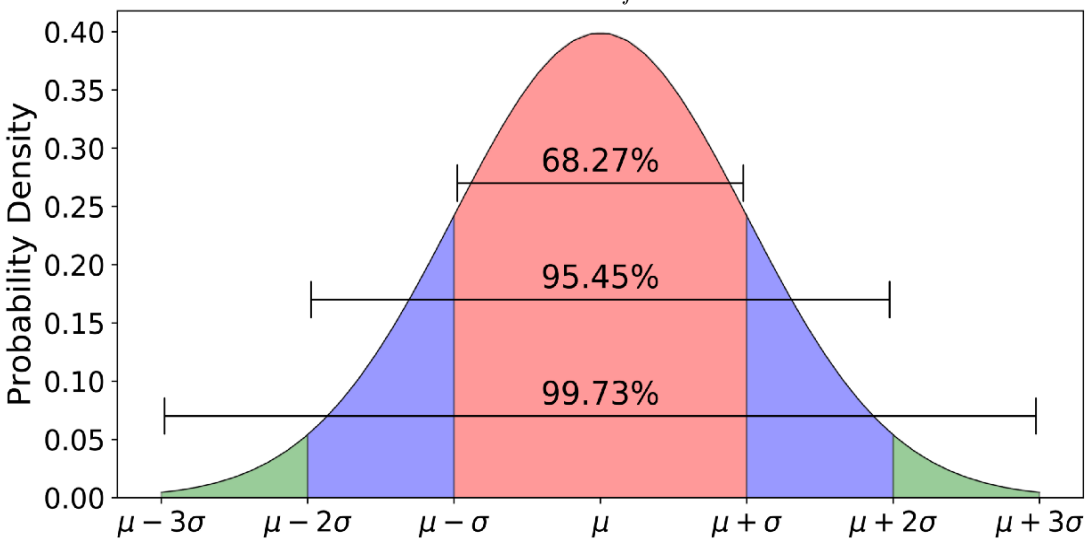

Figure 6.10: Normal Distribution

Calculating mean and standard deviation is implemented in the *calculate_stdev* method, which takes the array with all grades in the data set and returns a $(mean, stdev)$ pair.

```
def calculate_stdev(data)
    data = data.compact.map(&:to_i)
```

```
    mean = data.sum / data.size
    sum = 0
    data.each { |v| sum += (v - mean) ** 2 }
    stdev = Math.sqrt(sum / data.size)
    [mean, stdev]
  end
```

Listing 6.4: Implementation of calculating mean and standard deviation

For the current implementation approximation approach is taken. The middles of the colour regions, shown in figure 6.10, are taken as the boundaries, specifying which values will be added to the empty cells, according to randomly generated float. More precisely:

- $if rand \in [0:1] \Rightarrow \mu - 3\sigma$
- $if rand \in [2:9] \Rightarrow \mu - 2\sigma$
- $if rand \in [10:33] \Rightarrow \mu - \sigma$
- $if rand \in [34:66] \Rightarrow \mu$
- $if rand \in [67:91] \Rightarrow \mu + \sigma$
- $if rand \in [92:99] \Rightarrow \mu + 2\sigma$
- $if rand \in [99:10] \Rightarrow \mu + 3\sigma$

By using this approximation, along with a random number indicating the skill the student will get, and statistical data about the rest of the population, the automatically added grades will have more realistic values and distribution. Most of the added values will stay close to the mean. However, there will be still a chance that empty cells will take varying values. That will help for maintaining the validity of the allocation, even when dealing with missing data. That is also another way of increasing the scaling capabilities of the project.

Note that the cases which involve values $\pm 3\sigma$ and $\pm 2\sigma$ are clamped to stay within the $[0; 100]$ range. Test on the implementation with different data sets showed that with some, especially smaller data sets, the standard deviation may be calculated, so the values exceed the desired range. Clamping them, ensures the validity of the replaced values, regardless of the likelihood of a problem occurring.

6.4 Neighbourhood

6.4.1 Particles list manipulation

Even though the class contains other functionalities, its primary responsibilities are maintaining and controlling the particles that belong to the given neighbourhood. The particles are held in an instance variable called $particles_list$. It is an array containing the needed number of *Particle* objects. Due to the dynamic topologies, the algorithm is using, this list is regularly changed. The neighbourhood changes are occurring via adding and removing methods.

Those methods present very simple and almost unchanged implementation of the classic array manipulation methods *add* and *pop*. They are just encapsulating those functionalities with more sensible naming related to the program. The *add_particle* method adds a particle to the particles array, while the *remove_particle* removes a particle from the list and returns it if it is present or returns *nil* value if the list is empty, which is used to indicate empty neighbourhoods when altering the topologies.

6.4.2 Iterating Particles

Similarly to the $MBPSO$ class, requesting an iteration from all neighbourhoods, particle iteration from the point of view of the $Neighbourhood$ class is just forwarding the iteration request to its particles, without any complicated functionality. The method iterates over its list of particles and consecutively calls their *update_velocity*, *update_position*, $calculate\_fitness$ and *update_stats* methods, which outlines a full particle iteration. After all the particles have completed their iterations, their new fitnesses are compared inside the *update_l_best* method and the local best fitness and position are updated if needed. Furthermore, this method has an additional important function. That is keeping track of the number of iterations without a local best update. That number is stored in the *counter*, which is incremented at every iteration, during which the local best hasn't improved. On the contrary, if a better solution has arisen during the iteration, the number of iterations without an update is set to zero.

6.4.3 Reporting Particle Stats

When the algorithm terminates, if any statistics have to be, the *export_method* needs to gather the information for the fitnesses of the particles during the algorithm execution, as in the $MBPSO$ class record is kept only of the local and global bests. To acquire this data, the export method sends a request to the last $Neighbourhood$ object left by calling the *report_particles*

method. It forwards the request to all particles, acquires their *stats* fields, which contains the required data and returns it.

6.5 Particle

6.5.1 Particle Position and Initial Position Generation

The matrix that represents the particles position was implemented as an two dimensional array, or an array of array in other word. The main array - *position* is of length equal to the number of students. In each slot of this array is held another array, which has the length of the number of the team, which is visualised in figure 6.11.

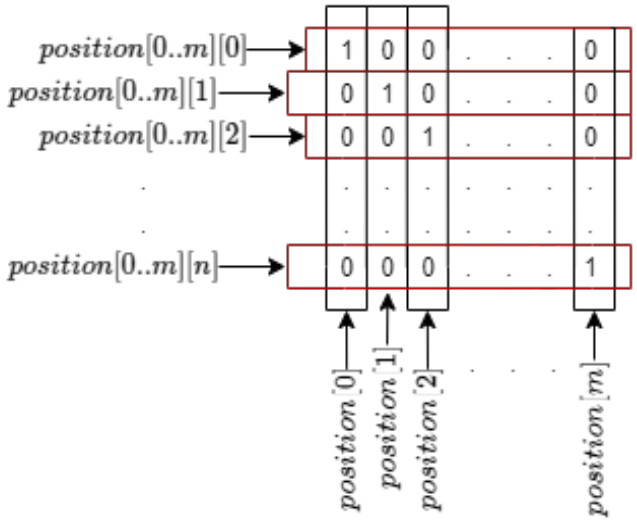

Figure 6.11: 2D array representing particle position

With this representation, a cell in the matrix is accessed by $position[x][y]$, where x is the student of interest, and y is the team of interest.

When it comes to defining the initial position of the particle, the *initial_particle_assignment* is responsible for that. Given the fundamental importance of a particle having a valid position before the first iteration, the method is called inside the constructor(as much as we can relate the Ruby's *initialize*() method to the constructor paradigm in other programming languages), when the particles are initialised. Initially, the 2d array is created and filled with zero values. The position is then generated randomly, using the $shuffle()$ method, which applied to an array, randomly shuffles the elements inside. The array used, is an array holding the values in the $[0 : numberofstudents - 1]$ range. A loop iterates through the values between 0 and the

number of students. At every index, the iteration's index is used for choosing which student is assigned to which team. This index takes the corresponding number from the shuffled list of students, and this student is assigned to the team with indicator equal to the remainder of the division of the loop index by the total number of teams. This ensures that all teams will get an equal number of students. So the student at index 0 in the shuffled list of students will go to team 0, as(if we assume that there are four teams) 0%4 is 0. The next student - at index 1 will go to team $1\%4 = 1$ and so on, up to index 3. Then when we reach index 4, again $4\%4 = 0$, for 5 we have $5\$4 = 1$ and so forth. This guarantees the constraints for an equal number of students in each team and a student being assigned to only one team remain fulfiled.

```
def initial_particle_assignment
    array = 0.upto(@length - 1).to_a
    array = array.shuffle
    (0..@length - 1).each do |x|
      student = array[x]
      @position[student][x%@teams] = 1
    end
  end
```

Listing 6.5: Generating random initial particle position

6.5.2 Calculating and Updating Velocity

Particle velocity is one of the fundamental implementation parts of the algorithm, due to the fact it has the most significant responsibility for updating the position. Furthermore, the scope of the current project requires a highly customised approach for calculating and updating this component. Even though the implementation is kept as similar as possible to the concepts of the original PSO algorithm, there were some significant problems that had to be overcome. The steps involved in successfully updating the velocity are: subtracting positions, generating random factors, and calculating the velocity. This is done by the *update_velocity* which calls a couple of separate methods, encapsulating smaller functionalities.

Note: For all the examples in the diagrams below, the assumed problem is assigning populations of size 12 into 3 teams of 4, with personal and local control parameters of $[0.8, 0.8, 0.6]$

Subtracting Positions

In PSO's velocity update equation, after the breakdown of the equation structure, the first calculation that has to be done is subtracting the current position from the local and global bests(the underlined parts). This functionality is held inside the $subtract_position$ method.

$$v_i^{t+1} = wv_i^t + c_1 r_1^t(\underline{lbest_i^t - x_i^t}) + c_2 r_2^t(\underline{gbest_i^t - x_i^t}) \tag{6.2}$$

For our current needs, the position subtraction has been done using the exclusive OR(XOR) logical operator(denoted by the $\bigoplus$ symbol), as suggested by[2]. XOR is a binary logical operator, returning true, only if both parameters are different.

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	0

Table 6.2: Logical XOR truth table

By using this approach, we are getting an indication only of bits in the particle's representation, where we have differences between the current position and the personal or local best. We are not interested in students who are in the same team as the best allocation of interest. We call those differences *swapping suggestions*, in other words, the places where the best allocation suggests swapping students with respect to the current allocation. After the method finds the swapping suggestions, it replaces each one of them with a probability specified as the third element in the control parameter variables. Figure 6.12 illustrates this process.

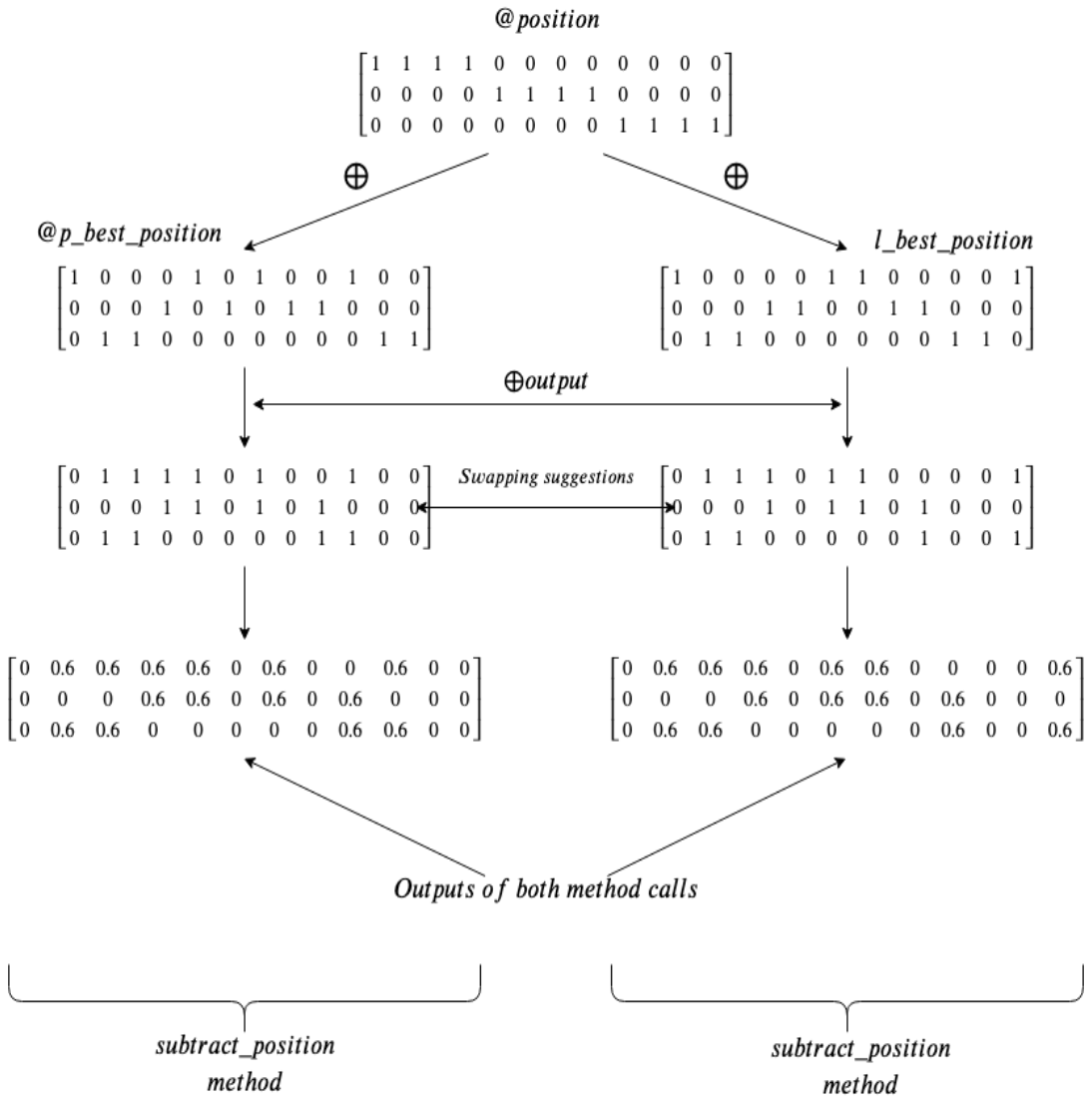

Figure 6.12: Subtracting position from personal and local bests example.

Generating Random Factor

The next action to be performed is multiplying the position difference by a random factor, which is the way exploration is made possible, rather than only going towards the local and global maximum. Because of the problem format, this action also has a custom implementation. The random factor is a matrix of the same size as the particle's position, containing values different from zero, only where Ruby's random number generator suggested a student swap. Initially, the method gives random values to all cells, however only the ones above a given threshold are kept as potential swapping suggestions. It uses the first float in the control parameter variables as a threshold parameter. Then it assigns all swapping suggestions values equal to the second value in the corresponding control parameter. This is implemented in the *generate_random_vector*

method and is illustrated in figure 6.13.

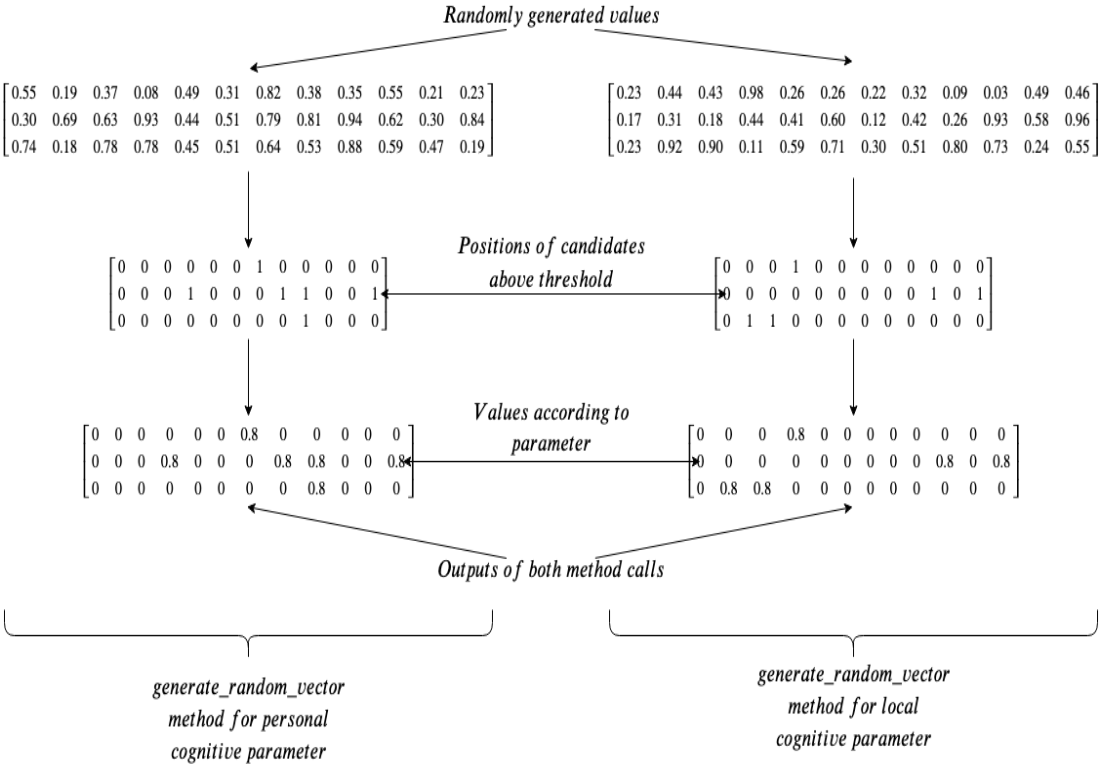

Figure 6.13: Generating random factor.

Calculating Velocity

After the subtracting the positions and generating the random factors, we can now calculate our alternatives of the second and third term in the velocity update equation. This is done in the *generate_cognitive_parameter* method, which sums the probabilities inside the matrices generated by *generate_random_vector* and *subtract_positions* methods.Then it calculates the non-normalised new velocity by summing the products of old velocity and inertia weight, and the personal and local cognitive parameters.The final step is normalising the new velocity and the survival threshold by dividing all values by the maximum probability present in the matrix. Example execution of *update_velocity* is shown in figure 6.14.

Figure 6.14: Updating velocity example

6.5.3 Updating Position

This is the most complicated part of the implementation, along with the velocity update. Apart from the nature of the problem that complicates the design and implementation, for this bit, there are some further constraints that need to be satisfied, which make the situation even trickier, explained in 4.6. Initially, the current position is randomised, and the @*velocity* variable is updated as the positional summations of itself and the randomised position.

Survival threshold is the last component needed before starting the actual allocation of students. Without it, there was observed a high number of changes at every iteration, keeping the algorithm highly explorative. This may be of good use, but only if it works in a controlled manner, as explained in 4.5 . Not controlling which students can move between teams and assigning students by simply iterating over the position, led to some students(especially ones

with lower indexes in the data set) to fill up the slots for teams, where students with very low suggested probability to move should have been allocated, this leads to those students getting allocated to random teams, filling the slots there. This would make all the calculations and efforts to controlling the behaviour of the algorithm useless, as a significant part of the population has been allocated, wherever there are free slots, regardless of its probability for doing so. While 4.5 explains the conceptual reasons for implementing selectivity, this was the implementation reason that solidified the decision. The *survivability_threshold* gives a threshold, above which in the list of probabilities after summing velocity with the randomised position, there are only @*survival_number* students left. That makes it possible to assign all the students with lower possibilities first and then make swaps only with students above the given threshold. The snippet below calculates the threshold 6.6.

```
@threshold = @velocity.flatten.max(@survival_number+1).last
```

Listing 6.6: Calculating survivability threshold

With all the aforementioned in place, the allocation is ready to begin. There are two important constraints when updating position, namely - team size, in other words, having only four students assigned to a team or only four values of 1 in each row and student getting allocated to only one team or having only a single value of 1 in each column. They are enforced by using the $free\_slots$ and *unassigned_students* variables. The former one is an array with an element for each team, initially having the value of 4. The corresponding value is decremented each time a student is assigned to a team, and also, this value being positive is a requirement before a student is allocated to a given team. The latter is filled with the indices of all unassigned students after the first stage(explained below). After the firsts stage of allocations, only the students in this list are considered for allocating, also when a student gets allocated, it is dropped from the list. The first stage starts with iterating over the columns in the @*velocity* variable and checking if there is an element, which is above the threshold. If that is the case, then this index, which equals the student of interest, is added to the list of unassigned students. If there are no such values in the column, it means that the student does not have high enough probabilities, according to the survivability principle, so it gets allocated to its old team, and the capacity of that team is decremented. That stage should end up with all student with lower probabilities of moving teams getting assigned to their old teams, without the risk of random assignment due to a lack of free slots. Furthermore, it should leave us with @*survivab_number* of elements in the *unassigned_students* list unless one or more

students have a very high probability for joining more than one team. However, that still will not interfere with the idea of controlled swapping.

The next step is sorting the probabilities in each of the remaining columns, corresponding to the unassigned students. However in the sorted list, we do not have the probabilites, but instead, the indices, which point to those probabilities, i.e. the first element will not be the value of the probability in the @*velocity* variables, but rather the index at which this highest probability is in the particular column. This index will point to the team that this student had the highest probability of joining. Consequently, in the new @*probabilities_indices* variable, which is a 2D array, we have sorted probability indices for each student in each column and the teams with highest probabilities in the first row, the second-highest probabilities in row 2 and so forth. It calculated by applying code snippet 6.7 to each column of the velocity table which the index is present in the list of unassigned students. That allows the algorithm to iterate over a row at a time until the students are assigned. In other words, it will first check the team for which students have the highest probabilities with respect to the free slots, and keep track of who has been assigned and who is yet to be assigned. In the case of remaining students for assignment, the next row is checked. The indexes of those columns are kept in the *index* temporary variable, which is incremented after iterating over the list of unassigned students.

```
unassigned_students.each do |x|
      probabilities_indices[x] = @velocity[x].map.with_index.sort
          .map(&:last)
      ...
      ...
      ...
    end
```

Listing 6.7: Creating the table with sorted probabilities indices

For performance purposes, this iteration over the list of unassigned students is combined with comparing the indexes of the highest proabilities with the corresponding team capacities. This is done with couple simple if statements, where the dots are in listing 6.7.

The rows are iterated however only up to the fifth row. Observations of the values and behaviour of the algorithm indicated that at this point, the indices point to unsiginificant probabilities and it only keeps using execution time for the last at most two or three students. Hence, if the *index* variables reaches a value above 4, each of the students in the unassigned

list is automatically assigned to the first team with available slots. The whole position update process is illustrated in figure 6.15. The @*survival_number* assumed in the example is equal to 4.

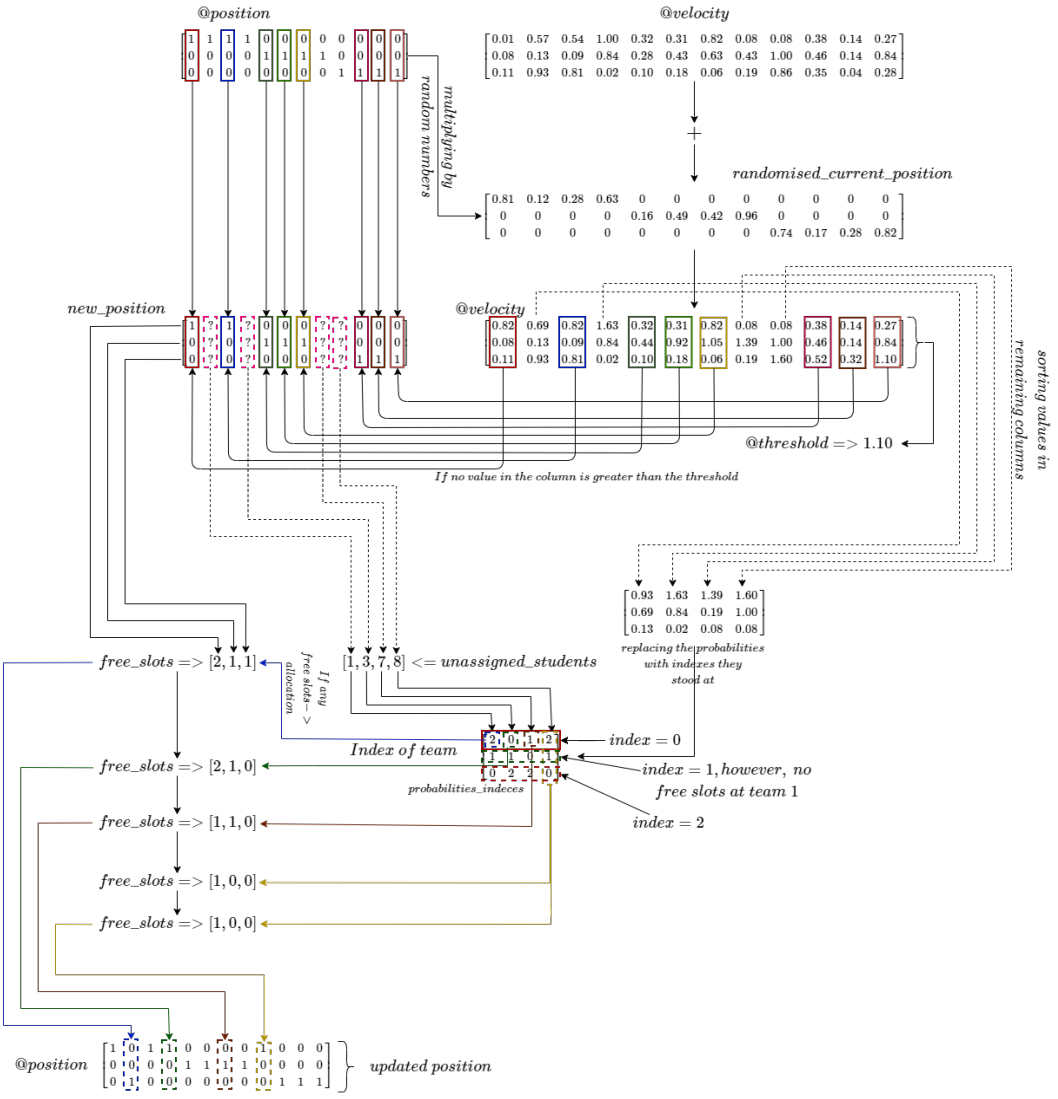

Figure 6.15: Updating Position Example

6.5.4 Calculating Fitness

The *calculate_fitness* method is responsible for evaluating the quality of the solution the particle represents. To do that, it iterates over the *position* field, a team at a time. For each team, the attributes of the participating students are extracted at three arrays. An array for each attribute. After that, for all possible pairs of students, the distances are calculated according to the fitness function described in 4.3.2 and is added to the total fitness of the

particles. When the distances between all pairs of students are calculated, the number of unique entries in the list of skills is checked. A number of unique entries, lower than the number of students in the team indicates for duplicating skill values. Depending on the number of duplicate values, a specified penalty is deducted from the fitness. Finally, the *update_p_best* method is invoked.

Handling Forbidden Pairs

An additional feature is implemented in the explained above method, apart from fitness calculation. While iterating over the teams to extract the attributes of the students, the validity of the team in terms of the list of forbidden pairs is checked. For each student, when its attributes are placed in the temporary arrays, its ID is checked for existence among the keys in the hash with the forbidden teammates. If that is the case, all the IDs of students who are not allowed to be teamed up with that student are added to a temporary array. Also, each student that is to have its attributes extracted is checked whether its ID exists in the list of forbidden students for that team. In case of invalid student pairing, the method generating the initial random position is invoked, and the fitness calculation is restarted.

Chapter 7

Legal, Social, Ethical and Professional Issues

7.1 Public Orientation of the project

Apart from proposing a solution to a complicated computer science problem, there is another purpose of this project. The presented algorithm aims to provide benefits for a large-scale audience. Creating optimally functioning team formation algorithm can increase the educational results and student satisfaction for a substantial portion of the society. Furthermore, if the proposed in the project concepts succeed in providing a core for developing a more general algorithm, solving a broader set of problems, it can have an even more beneficial social impact. On the other hand, it aims to provide teachers and project supervisors with a product that is capable of raising their job expertise to an even higher level. To sum up, increasing both the quality of the teaching provided, and the learning outcomes for a significant portion of the students can have an extraordinary impact on the educational system in general.

7.2 Ethical Issues

This project did not require an ethical clearance since it did not rely on real people data for its development. For all tests, computer-generated data sets, representing real data distributions, were used. Furthermore, even when applied to a real problem, it does not require acquiring any personal data about students. It needs gender, ethnicity and grade. However, the privacy of each individual is highly respected, and the option of a student deciding not to specify those is

fully supported. Also, the student identifiers can be in literally any form. This representation allows for absolutely anonymous team allocation even by third parties

Chapter 8

Testing and Evaluation

8.1 Test Cases

The implementation testing was performed using the RSpec testing tool. All the tests(below called specifications) are located in the *spec* directory of the gem project.

The behaviour of each class has been tested to ensure it functions as intended. The specs were written before the functionalities implementation to ensure that all functionalities are following the pre-defined requirements. The tested functionalities are only the ones that are available to be invoked by the call of a public method. It is assumed that in a well-structured implementation if all publicly callable units are proven to have the required behaviour, the private methods they use are also working as intended.

8.1.1 Validation Class

Apart from a single method, all other ones inside the *Validation* class are available for standalone use by the user, apart from their hard-coded use in the default algorithm run. Each method was tested by supplying covering the full range of values for which the given method should approve the validity, as well as at least one specifications for each group of input that should be rejected, for example - instances of the required class, but with an invalid value, instances of other classes etc. In other words, each literal, forming the full condition passed to the exception raising method was tested. 8.1

```
RSpec.describe MBPSOTeamFormation::Validation do
  val = MBPSOTeamFormation::Validation.new
  describe "Validate Numbers" do
```

```
    it "Integer validator accepts only valid positive integers"
        do
      expect(val.validate_number(3, 'test', 'pos_int')).to eq(3)
      expect { val.validate_number(-2, 'test', 'pos_int') }.to
         raise_error(ArgumentError)
      expect { val.validate_number(0, 'test', 'pos_int') }.to
         raise_error(ArgumentError)
      expect { val.validate_number(3.1, 'test', 'pos_int') }.to
         raise_error(ArgumentError)
      expect { val.validate_number(-2.1, 'test', 'pos_int') }.to
         raise_error(ArgumentError)
      expect { val.validate_number([3, 5], 'test', 'pos_int') }.
         to raise_error(ArgumentError)
      expect { val.validate_number('should fail', 'test', '
         pos_int') }.to raise_error(ArgumentError)
    end
  end
end
\label{influences}
```

Listing 8.1: Test case example

```
MBPSOTeamFormation::Validation
  Validate Numbers
    Integer validator accepts only valid positive integers
    Non-negative number accepts only valid integers and floats
    Works only with 'pos_int' and 'nn_num' arguments
  Validate Survaival Number
    Accepts only integers between 2 and the  number of students
  Validate Control Parameters
    Accepts only valid 3-valued arrays with floats in [0:1]
    Works only with 'local' and 'personal' arguments
  Validate Skill Table
    Accepts only hashes that cover the whole [0:100] Integer range and are not ambiguous
    Accepts only integer keys
  Validate Boolean
    Accepts only logical/bolean values
  Validate Data Set
    Accepts only CSV::Table formats
    Accepts data sets only with valid headers and valid number of columns
    Declines data sets with duplicating IDs
    Declines data sets with gender values different from '-1', '0' and '1'
    Declines data sets with ethnicity values different from the Integers in the [-1:4] range
    Declines data sets with gender values different from '-1', '0' and 1
  Validate Forbidden Pair
    Accepts instances of Array or CSV::Table with two values per row
```

Figure 8.1: Validation class test cases

8.1.2 MVH Class

The full set of possible scenarios was tested, when it comes to detection and handling of missing values.8.2

```
MBPSOTeamFormation::MVH
  Detecting missing values
    Returning true if no values are missing
    Raise exception if missing values are found and the user doesn't want them handled
  Calculating needed statistics for the data set
    Calculates correctly the most frequent gender
    Calculates correctly the most frequent ethnicity
    Calculates correctly the mean of the grades
    Calculated correctly standard deviation
  Replacing the missing values
    Replacing missing gender
    Replacing missing ethnicity
    Replacing missing grade
```

Figure 8.2: Validation class test cases

8.1.3 Neighbourhood and Particle Classes

The testing of the *Neighbourhood* and *Particle* classes overlap to some extent. The reason for that is that the former's main purpose is to maintain and dictate the behaviour of the particles belonging to it. In other words, most of its functionality is meaningless without a *Particle* object being involved. The test cases for it are shown in, followed by the ones for the *Particle* class in 8.3 and 8.3.

```
MBPSOTeamFormation::Neighbourhood
  Initialisation
    Succesfully initialising and generating particles list of the correct size
    All particles in the list are successfully created
    Particles list manipulation
      Successfully adding particles to the neighbourhood
      Successfully maintaining particles list size when removing particles
      Returning particle only if thera are present particles in the list before removing
      Returning nil only if trying to remove particle from empty list
    Value update
      Accurately updating inertia
      Successfully iterate particles and accurately update local best fitness
```

Figure 8.3: Neighbourhood class test cases

```
MBPSOTeamFormation::Particle
  Initialising Particle
    Initial Position generated
    Every student is assigned to excatly one team after initial position generation
    Every team is composed of exactly 4 students after initial position generation
  Calculating particle parameters
    Accurately calculating position
    Accurately updating personal best fitness and position
    Maintaining valid position for a number of iterations
```

Figure 8.4: Particle class test cases

8.2 Testing Algorithm Performance

The nature of the current problem allows mostly practical ways of researching its specific characteristics. Otherwise speaking all behaviour patterns and insights were acquired via extensive testing. A substantial part of the algorithm development involved many repetitions of a four-step cycle.

1. Perform research and develop a hypothesis
2. Implement the needed functionality, allowing observations on factors related to the hypothesis
3. Validate and study the results the implemented functionality provides
4. Accept/Reject the hypothesis and incorporate the new knowledge.

This development methodology led to fundamental changes in the building blocks of the algorithm. Even some key concepts such as velocity and position update were entirely reworked while analysing different existing approaches, their pros, cons and new ways to combine them. However, each iteration of the cycle represented a critical lesson learned and led to a deeper understanding of the topic.

Most of the conceptual conclusions were already described above in the project. Additionally, below is presented a table, summarising the concluded impact some of the main parameters have on algorithm performance. Those conclusions were used as a guide towards tuning the default parameters. Their impact on the algorithm performance is described in terms of their impact on exploration(XLR), exploitation(XLT), convergence towards the global

maximum(CGM) and algorithm convergence(AC). where L[1,2,3] and P[1,2,3] are the 3-valued

Parameter	Value	XLR	XLT	GCM	AC
Initial Inertia	High(> 1)	⇑		⇓	⇑
	Low(approaching 0)	⇓		⇑	⇑
Final Inertia	High(> 1)	⇑	⇓		⇓
	Low(approaching 1)	⇓	⇑		⇑
P[1]	High(> 1)			⇑	⇑
	Low(approaching 1)			⇓	⇓
P[2]	High(> 1)	⇑	⇑	⇑	⇑
	Low(approaching 1)	⇓	⇓	⇓	⇓
P[3]	High(> 1)	⇑	⇓	⇓	⇑
	Low(approaching 1)	⇓	⇑	⇑	⇓
L[1]	High(> 1)	⇑	⇓	⇓	⇑
	Low(approaching 1)	⇓	⇑	⇑	⇓
L[2]	High[¿ 1]			⇓	⇑
	Low (approaching 1)			⇑	⇓
L[3]	High(> 1)	⇓	⇑	⇑	⇑
	Low(approaching 1)	⇑	⇓	⇓	⇓

Table 8.1: Caption

control parameters.And an arrow pointing up means the following things:

- Increased initial exploration, when in the XLR column
- Increased final exploitation, when in the XLT column
- The average particles fitness being closer to the global best towards the end of the run, when in the GCM column
- Increased number of iterations performed, when in the AC column

To visualise a run, figure 8.5 shows the average fitness of the particles of a run with the default parameters. The first thing we see is that the average fitness follows a steady increase. However, there are more things that we can conclude from this image:

1. The algorithm starts with wide fluctuations in the beginning, covering a large range of fitness values. That indicates that the particles are actively exploring the search space.
2. For the first roughly 1800-200 iterations, we can barely observe a common fitness change. Around the 2000th iteration, the survival number and the inertia weight are already at $\frac{2}{3}$ of their way towards their final, exploitation oriented values. A similar statement can be given for the neighbourhoods, as well. We can easily see that around this iteration, the average fitness start slowly climbing while increasing the rate at which it increases

its fitness (goes toward the global maximum) around iteration 3000(when both survival number and inertia weight have already reached their final values), which confirms the exploration/exploitation balance that was a goal to be achieved.

3. The upper bound of the graph is the global best fitness.This is how 10.4 prints the graphs, the upper bound is always the global best, so we can get an idea of how close to the global best the fitness gets. As we see in the graphics, the algorithm provides also converges with fitnesses very close to the global best. The fluctuations at the average fitness are significantly reduced already at the middle of the run and furtherly decay towards the algorithm termination.

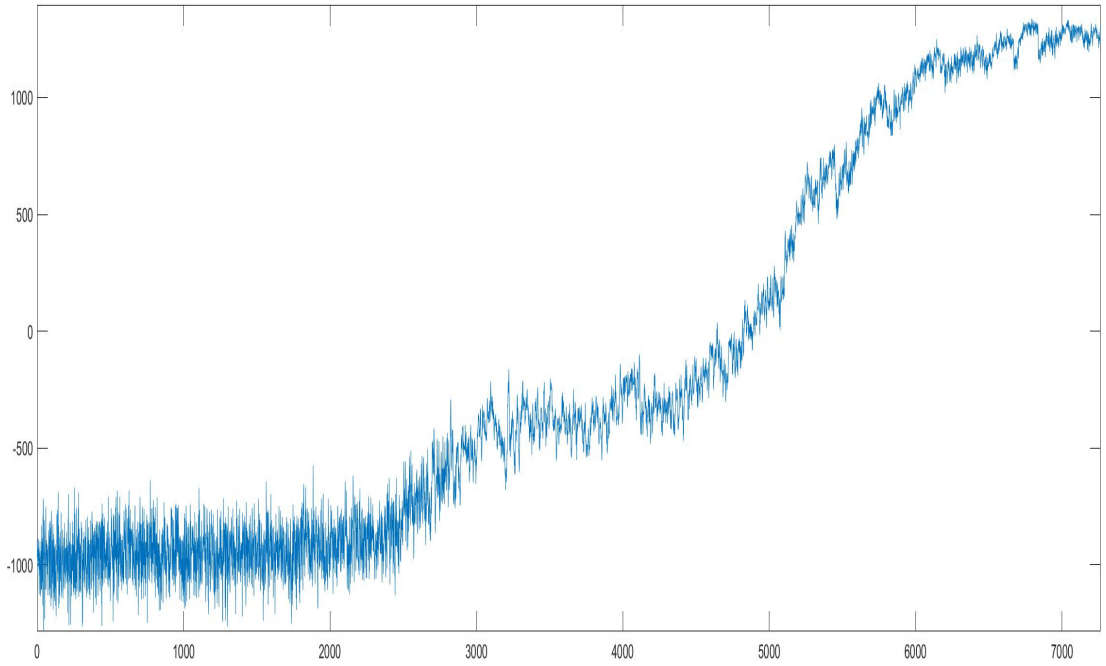

Figure 8.5: *Particles average fitness over a run with default parameters*

After getting a general idea of what information the visualisations give, we can take a look at the full visualisation of the run generated by 10.4.

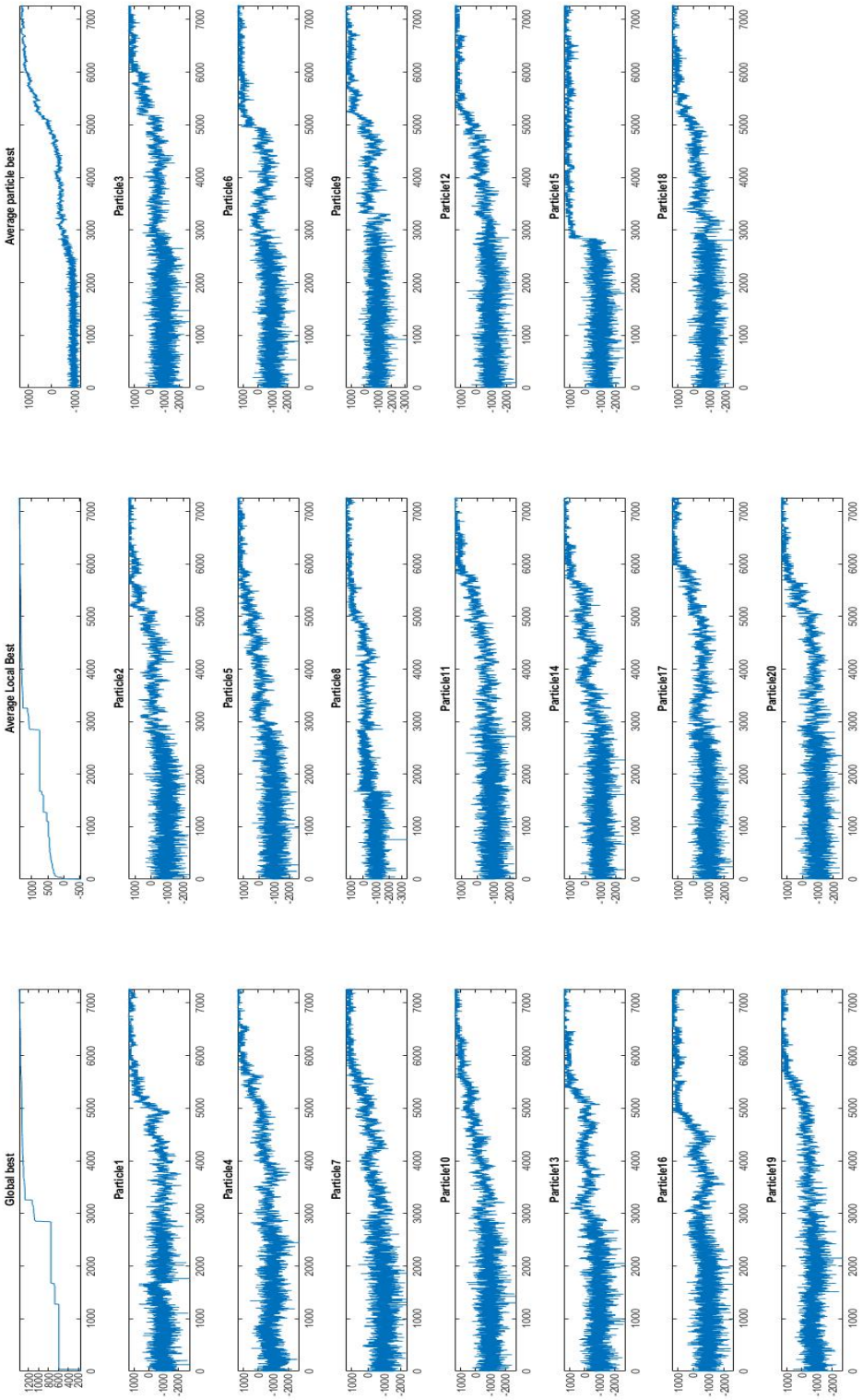

Figure 8.6: *Full Visualisation of run with default parameters*

Above we can observe the explained concepts, on a broader set of data. Also, the behaviour of each particle can be seen, rather than an approximation for the whole population. This furtherly proves the capability of the algorithm to explore and collectively converge towards a global best solution, with the right parameters.

Another example run is illustrated in 8.7. It uses the default parameters. However, it has the third value in the local control parameter set to 0.3. With some parameters, changes in their values have a tremendous effect on algorithm performance, as we can observe in that figure. On the plots for the particles, the fully explorative behaviour is obvious. That makes the algorithm to stagnate in a local maximum with without the particles being able to move towards the global maximum. Which furtherly illustrates the importance of the choice of the parameters.

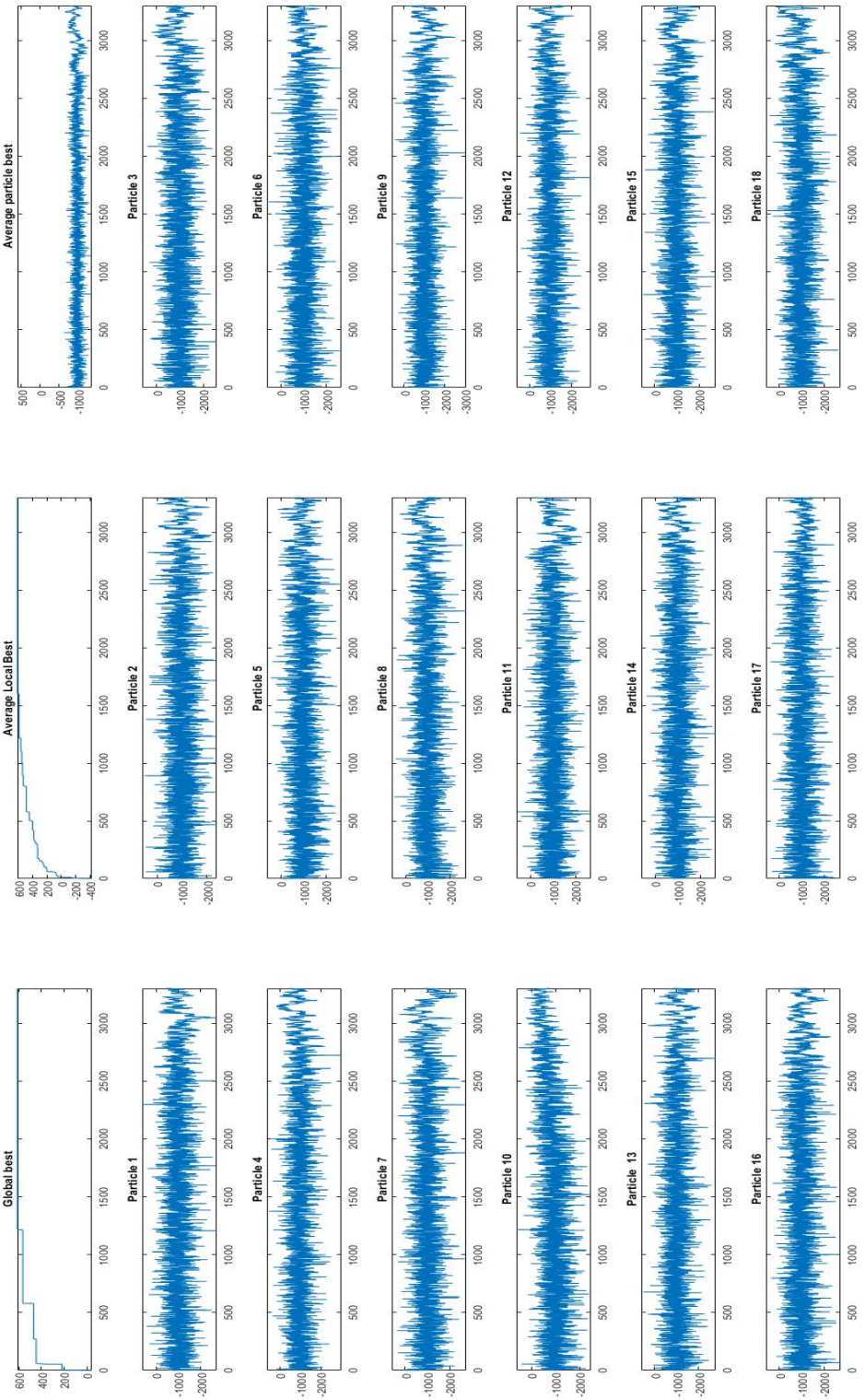

Figure 8.7: *Run with decreased* $control_param_local[2] = 0.3$

Figure 8.8 shows another interesting example of the way parameters change algorithm behaviour. At first glance, the plot suggests decent performance. We have the exploration; it turns into exploitation; we have increasing fitnesses. However, at the point where the algorithm terminates, we can see how far is the average best, as well most of the particles from the global bests. Therefore, apart from balancing global and local searches, there are other factors which require precise parameters tuning. The illustrated run uses $control\_param\_personal : [0.2, 0.4, 0.55]$ and $final\_survival\_number : 8$.

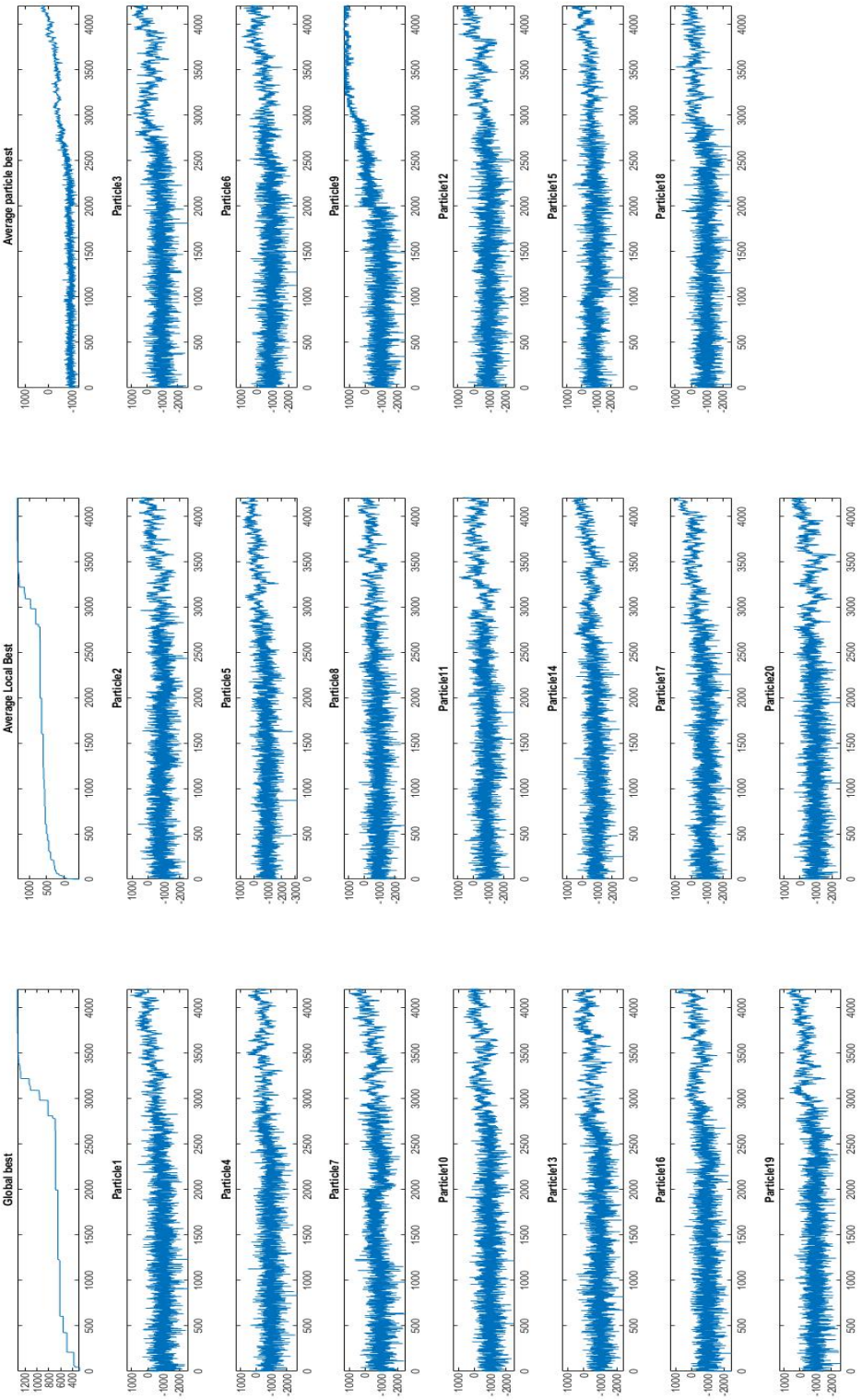

Figure 8.8: *Particles average fitness over a run with default parameters*

This outlines the ways the algorithm was tuned, however similar tests and analyses were performed countless times. Due to the complicated way in which the control parameters influence behaviour shown in table 8.1, it is impossible to describe all tuning tests exhaustively. *Note: By default values is meant the values left as default in table 6.1.*

Generating Test Data No real data was used for the tests. The experimental data sets were generated by a script provided by Dr Keppens10.6. However, there are two slight changes to the original version of the script he provided. First of all, the table headers were changed to be accepted by the validation method for the data set, also when opening the target file before writing into it, *newline* =″, argument was added because the original version was printing empty rows between the entries.

8.3 Evaluation

Requirements Analysis Due to the scope of the project and the goal of researching, designing and producing an algorithm, rather than a standalone software product, the possibilities for setting functional requirements in the early stages of the project were limited. However, two sets of user and system requirements were outlined and used as a guide to what functionalities should the implementation provide. Some of the requirements overlap, and in such cases, they are mentioned together. The fulfilment of most of the requirements is explained thoroughly in the report.

UR1: Installing gem from remote and local sources *Overlaps with SR2*

The gem is publicly available on RubyGems.com and can be installed by simply running **gem install MBPSO_Team_Formation** from the terminal. That will install the gem in the current directory. Furthermore, the gemfile, containing the full functionality of the program, is provided, along with the current report, as well as the source code from which the gem can be built. That gives a way of remote installation, by writing a single command in the terminal, as well as two ways of local installation of the gem.

UR2: Validate the data set and all other input before giving them as input to the algorithm *Overlaps with SR5 and SR6*

In the design and implementation chapters, the mechanisms for validating user input are discussed and tested earlier in this section. All user input is available for validation both via separate calls to the $Validation$ class and automatically by the program.

UR3 Easily run the algorithm for the software engineering project problem by providing only the data set *Overlaps with SR3 and SR4*

The basic functionality and primary purpose of the project is proposing optimal team allocation via smooth interaction with the user. The minimal input it requires is a data set, and via a simple call to the *run* method, it provides a suggested allocation.

UR4 Separately handle missing values in the data set *Overlaps with SR7*

As discussed in the design chapter, there are chances and valid reasons for the data set having missing values. Therefore missing value checks were implemented and tested, along with the automatic way of handling them, which is set to be executed by default. However, the project supervisor and any other user are provided with the functionality of only checking about missing data. In a well-designed system, if a missing value is present, most likely there will be a specific reason for that. Hence, the user may want to handle the situation in a custom way.

UR5 Choose if the missing values should be automatically handled *Overlaps with SR8*

As explained in the paragraph above, providing the functionality does not obligate the user to use it. A flag can be passed to the gem constructor or to the $fill\_missing\_values$ method to indicate the user's decision on whether he/she wants the values automatically handled.

• **UR6 Provide a custom grade to skill mapping** Even though a default way of turning grade into skill is proposed, the project was developed with scalability in mind which implies different uses. Furthermore, a grade value is very irrelevant on its own. Same grades can mean different things in different domains. Also, there is supposedly no person who knows better his/her students and can decide more adequately what is a reasonable way of mapping grades to skill values. Hence, the user has full control over the grade to skill transfer.

UR7 Fully customise the algorithm behaviour *Overlaps with SR9 and SR9*

In addition to implementing a piece of software that assigns students to teams for their software engineering group project, the task of creating a flexible algorithm with reliable conceptual base and scaling capabilities was taken just as seriously. While developing features that are related to the specific problem, Every method and variable mechanics were designed and tested to take different shapes of data and allow for further expansion.

UR8 Request statistics to facilitate optimisation *Overlaps with SR9 and SR10*

The algorithm implementation would be useless if it tries to provide adaptable software without allowing for internal changes by the user. Furthermore, due to the need of statistical tests to required in order to obtain any information about the algorithm behaviour, easy way of exporting that data used for its initial implementation is provided for facilitating further project work.

UR9 Adapt the algorithm for other team allocation problems by optimising the parameters The algorithm provides custom functionality and ready mechanics for solving any team problem for which a solution can be represented in the same way a particle is represented. However, for different situations - team sizes, grades, ethnicities and, the current default parameters are likely to fail. Therefore, due to the scalable way it was implemented and the means of optimising the parameters, it is applicable to similar problems.

UR11 Solution validation according to a reliable heurist Developing a way of validation the algorithm performance appeared to be a highly complicated task. Variants using extremely explorative runs with many particles over long periods, as well as ways of estimating a possible optimal value by statistical analysis of the data set were tested. However, none of them was even close to a reliable solution to the requirement. Even exhaustive search approaches were very challenging to develop for small inputs. This requirement remains an open challenge for further work on the topic.

UR12 Provide a solid base for future development into general team formation problem solver The algorithm is backed by solid research in the team formation approaches used so far, as well as the specifics of PSO and BPSO. Solutions were suggested for many of the addressed issues in the papers suggesting different versions of the PSO algorithms or team formation ones. It provides a highly customisable novel BPSO version. However, it is currently limited to problems, for which a solution can be represented and evaluated as a two-dimensional binary array. Project expansions and ways of reducing problems, so they become solvable by the current algorithm also remain an open challenge for further improvements.

8.4 Results

The resulting implementation closely follows and confirms the validity of the new concepts of the proposed algorithm. The algorithm provides valid and diverse teams, without any unintended

behaviour or breaking any hard constraints. The mechanisms provided for controlling and migrating between different types of search proved themselves working.

When it comes to run time, all the test were performed using data sets containing around 260 students. They were ran on a computer running Windows 10 with Intel Core-i7 4720HQ, quad-core CPU running on basic frequency of 2.60GHz and 16GB RAM. During tests, up to 10 algorithm runs were performed simultaneously, whilst still allowing the computer to software such as RubyMine and Matlab. The average time a configuration with 20 particles was assigning a class of 260 students to teams of 4 was an average around an hour. Having in mind the main scope of the project, which is a year project, that allows forcing the algorithm to perform longer explorative search with a larger number of particles, as it only needs to produce one successfull allocation and the rest of the year is sorted.

Regarding its performance, the big question still remains what exactly is the quality of the produced solutions, what is the fitness of the optimal solution and how that be calculated for evaluating the algorithm performance. Even though it satisfies the requirements for its functionality and implements all aspects of the proposed algoorithm, a reliable heuristic needs to be designed for confirming the validity of the algorithm.

Chapter 9

Conclusion and Future Work

9.1 Final Thoughts

Stochastic metaheuristic algorithms provide very promising approaches for solving problems which are still perceived as great challenges in computer science. As one of them, PSO has already proven its capabilities and efficiency. Furthermore, the current projects suggest that it is highly capable of adapting new techniques and expanding its applicability. Even though working on this project was challenging, precious insights in the field of stochastic optimisation were acquired, which will be beneficial knowledge given the current development of the field. Also the produced algorithm provides the capability of further work on plenty of interesting topics. After designing the algorithm and applying it to a highly complicated problem, no clear answer whether the algorithm provided the optimal solution was found. However, one thing is for sure, even if the solution is not very close to being optimal, the algorithm is highly capable of further improvements of its performance and functionality. This is a strong motivation for looking into any of the following below topics in the near future.

One way to extend the algorithm is either by changing its capabilities, so it can be addressed to a broader range of problems or by finding ways via which highly complicated problems can be simplified and modified to work with the MBPSO. Looking for ways to achieve those two things can be the goal of a highly challenging, but rewarding research project.

Another way of extending the project arises from the problem of whether its performance is optimal. Which turns the optimisation problem into an optimisation problem on its own. Therefore another interesting topic to work on is developing an algorithm for optimising the parameters and performance of MBPSO. Even though it takes time to perform a run of the

algorithm, with computational capabilities nowadays, many instances of the problem can be executed simultaneously. So the development of optimisation algorithm that controls and analyses the performance of parallelly running instances of the problem to be optimised is a deep subject to dive in, but can also result in the creation of two compelling new optimisation algorithms.

Chapter 10

User Guide

10.1 Installing the Gem

The gem can be installed both remotely and locally. The easiest way for installing the gem is by navigating to the desired directory in the terminal and running:

gem install MBPSO_Team_Formation

That should produce the following output:

```
C:\Users\Anton Pashov\Desktop>gem install MBPSO_Team_Formation
Successfully installed MBPSO_Team_Formation-0.1.0
Parsing documentation for MBPSO_Team_Formation-0.1.0
Done installing documentation for MBPSO_Team_Formation after 0 seconds
1 gem installed

C:\Users\Anton Pashov\Desktop>_
```

Figure 10.1: Remote installation of the gem

Running the **gem install** command first searches online and if it cannot find it, looks in the current folder. The gem is published on RubyGems.com, and it will always be installed from there. However, it can be forced to be installed locally, by running:

gem install –local MBPSO_Team_Formation-0.1.0.gem

which should produce the same output. Note that when using local installation the gemfile should be in the current directory(or the path to the gem should be specified) in it. Also, should be used(gemname-version.gem), unlike remote installation where only the gem name is needed.

10.2 Using the Gem

10.2.1 Using Default Values

Using the default gem functionality is very straightforward, once the gem is installed. All you need is installing it in the folder, where the data set and the ruby file/project that will use the gem are located. The code below is an example of using the gem.

```
# Importing the package that will read the .csv file
require 'csv'
# Importing the gem
require 'MBPSO_Team_Formation'


# Reading the data set
table = CSV.parse(File.read('data.csv'), headers: true)
# Creating object of the gem
mbpso = MBPSOTeamFormation::MBPSO.new(table)
# Running the algorithm
mbpso.run
```

Listing 10.1: SImple gem use

If all goes well, the program should not print anything initially. The only reason for initial output is exception/warning, which are all self explanatory and also explained in chapter 6. The expected final output is mentioned in 6.1.4.

10.2.2 Customising Algorithm

To use a customised version of the algorithm, one or more of the optional parameters need to be specified. In Ruby passing named parameters happens by adding an argument in the form **parameter_name: parameter_value** when creating an object of the given class. For example to change the number of particles, the algorithm should be used as follows:

```
mbpso = MBPSOTeamFormation::MBPSO.new(table, num_particles: 8)
mbpso.run
```

Listing 10.2: Optional parameter example

Full list of the optional parameters, their meaning and default values is listed in table 6.1.

10.2.3 Exporting Run Data

To export statistics about the algorithm run, the *output_stats* flag should be set to true. Additionally, the name of the variable where the program will export the data can be specified using *output_stats_name*. For example:

```
mbpso = MBPSOTeamFormation::MBPSO.new(table, output_stats: true,
    output_stats_name: 'test.csv')
mbpso.run
```

Listing 10.3: Exporting run data

Full list of the optional parameters, their meaning and default values is listed in table 6.1.

10.2.4 Visualising Statistics

Easiest way to analyse the exported stats is by visualising them. Plotting the data easily provides insights including exploration/exploitation behaviour(shown by the big changes in particles fitness), convergence, fitness of the solution and more. All the data can be visualised in Matlab, using the following script:

```
% MATLAB SCRIPT

%replace the filename
varname = 'filename.csv';
data = csvread(varname);
g_bests = data(1,:);
avg_l_best = data(2,:);
particles = data(3:size(data,1), :);
num_particles = size(data, 1) - 2;
average_particle_fitness = mean(particles);
iterations = [1:size(data,2)];
rows = fix(num_particles/3)+1;
if  rem(num_particles, 3) ~= 0
    rows = rows +1;
end

% This line requires Matblab 2019b or above
```

```
tiledlayout(rows,3)


nexttile
plot( iterations, g_bests)
title('Global best')
axis([0 size(iterations, 2) min(g_bests) max(g_bests)])


nexttile
plot( iterations, avg_l_best)
title('Average Local Best')
axis([0 size(iterations, 2) min(avg_l_best) max(g_bests)])


nexttile
plot(iterations, average_particle_fitness)
title('Average particle best')
axis([0 size(iterations, 2) min(average_particle_fitness) max(
    g_bests)])


for i = 1:num_particles
nexttile
plot(iterations, data(i+2,:))
title(strcat('Particle ',string(i)))
axis([0 size(iterations, 2) min(data(i+2,:)) max(g_bests)])


end
```

Listing 10.4: Matlab script for statistics visualisation

The visualisation should look like:

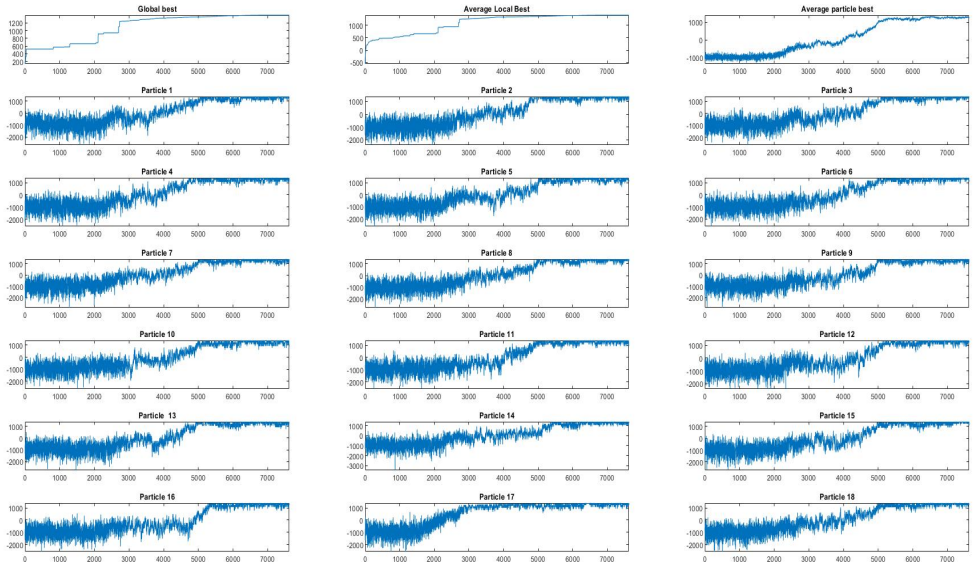

Figure 10.2: Visualised Data

10.2.5 Optimising Algorithm

To furtherly optimise the algorithm for the current or new problem are needed extensive empirical experiments. Even though it is time consuming, the easiest way for performing the tests is to create a list of run configurations and make the algorithm perform one or more runs for each of the configurations. Example of an automated script, recording statistics for different run configurations is given below:

```
require 'csv'
# Don't forget to install the gem in the directory
# Inside which is the script
require 'MBPSO_Team_Formation'

# Change 'data.csv' to the specific data set name
table = CSV.parse(File.read('data.csv'), headers: true)

# List of forbidden pairs, if present
forbidden_pairs = CSV.parse(File.read('forbidden_pairs.csv'),
    headers: false)
```

```
# CSV file containing the run configurations.
# The columns need to be named with the names of the optional
    parameters,
# while the rows need to contain the values for those parameters
    for each run
run_configuration = CSV.parse(File.read('run_configuration.csv'),
     headers: true)


# Iterate for each row in the configurations file
run_configuration.each do |x|
  # Generating file name, containg the used parameters values
  # This can be customised according to the user's preference
  string = "num_p=#{x['num_particles'].to_i},init_in=#{x['
      initial_inertia'].to_f},fin_in=#{x['final_inertia'].to_f}.
      csv"

  # Instantiate an object with the needed parameters
  # Note, only an example set is used to visualise the
      functionality
  mbpso = MBPSO_Team_Formation::MBPSO.new(table, num_particles: x
      ['num_particles'].to_i, initial_inertia: x['initial_inertia'
      ].to_f, final_inertia: x['final_inertia'].to_f,
      control_param_personal: [x['control_param_personal_0'].to_f,
       x['control_param_personal_1'].to_f, x['
      control_param_personal_2'].to_f], control_param_local: [x['
      control_param_local_0'].to_f, x['control_param_local_1'].
      to_f, x['control_param_local_2'].to_f], output_stats: true,
      output_stats_name: string, forbidden_pairs: forbidden_pairs)
  mbpso.run
end
puts "Success"
```

```
# Repeated runs for each configurations is suggested
# for confirming the results, due to the random characteristics
   of the algorithm
```

Listing 10.5: Script for automated algorithm use

Furthermore, to make sure the algorotihm performace is real, the algorithm needs to be tested on a variety of data sets. To generate a data set, the following script can be used. Credits to Dr. Jeroen Keppens for providing it, as mentioned in the testing chapter.

```
import csv
from random import gauss, random

student_count = 260
mark_avg = 65
mark_std = 14
gender_prop_male = 0.638
gender_prop_female = 0.186
ethnicity_prop_white = 0.348
ethnicity_prop_mixed = 0.019
ethnicity_prop_asian = 0.324
ethnicity_prop_black = 0.014
ethnicity_prop_other = 0.033

cumulative_prop_white = ethnicity_prop_white
cumulative_prop_mixed = cumulative_prop_white +
   ethnicity_prop_mixed
cumulative_prop_asian = cumulative_prop_mixed +
   ethnicity_prop_asian
cumulative_prop_black = cumulative_prop_asian +
   ethnicity_prop_black
cumulative_prop_other = cumulative_prop_black +
   ethnicity_prop_other

data = [["id","Gender","Ethnicity","Grade"]]
```

```
for i in range(1,student_count+1):
    mark = int(max(min(round(gauss(mark_avg,mark_std),0),100),0))
    rand = random()
    if rand < gender_prop_male:
        gender = 0
    elif rand < gender_prop_male + gender_prop_female:
        gender = 1
    else:
        gender = -1
    rand = random()
    if rand < cumulative_prop_white:
        ethnicity = 0
    elif rand < cumulative_prop_mixed:
        ethinicity = 1
    elif rand < cumulative_prop_asian:
        ethnicity = 2
    elif rand < cumulative_prop_black:
        ethnicity = 3
    elif rand < cumulative_prop_other:
        ethnicity = 4
    else:
        ethnicity = -1
    row = [i,gender,ethnicity,mark]
    data.append(row)

with open('data.csv', 'w', newline='') as file:
    writer = csv.writer(file)
    writer.writerows(data)
file.close()
```

Listing 10.6: Generating experimental data sets

10.2.6 Separately Validating Input

To validate any desired possible input for the algorithm, especially data sets and skill maps, the user should instantiate the *Validation* class and use its methods. The methods in the class are described in 6.2 and the allowed formats for each input variable is listed in 6.1. Sample code, validating a data set is shown below:

```
require 'csv'
require 'MBPSO_Team_Formation'

validation = MBPSO_Team_Formation::Validation.new
data = CSV.parse(File.read('data.csv'), headers: true)

puts "Successfull" if validation.validate_dataset(data).class ==
   CSV::Table
```

Listing 10.7: Validating data set

Full list of the validation methods and their arguments:

```
validation.validate_number(var, 'name', 'pos_int')
validation.validate_number(var, 'name', 'nn_num')
validation.validate_survival_number(var, num_students)
validation.validate_control_parameters(var, 'personal')
validation.validate__parameters(var, 'local')
validation.validate_skill_table(var)
validation.validate_bool(var, 'name')
validation.validate_dataset(var)
validation.validate_forbidden_pairs(var)
```

Listing 10.8: Validation methods

10.2.7 Checking Data Set for Missing Values and Filling them

To check or check and replace the missing values in the data set, the MVH class needs to be instantiated. To differentiate only checking and checking with replacing, the $tolerate_missing_values$ flag should be used. To only check for missing values:

```
require 'csv'
```

```
require 'MBPSO_Team_Formation'


mvh = MBPSO_Team_Formation::MVH.new
data = CSV.parse(File.read('data.csv'), headers: true)


mvh.fill_missing_values(data, false)
```

Listing 10.9: Validating data set

Unless an exception is thrown, there are no missing values in the data set. To check and replace values, the same code can be used, but the second argument should be *true*. No output would mean there are no missing values, otherwise warning will be printed.

References

[1] Saman M Abdulrahman. Using swarm intelligence for solving np-hard problems. *Academic Journal of Nawroz University*, 6(3):46–50, 2017.

[2] Farzaneh Afshinmanesh, Alireza Marandi, and Ashkan Rahimi-Kian. A novel binary particle swarm optimization method using artificial immune system. In *EUROCON 2005-The International Conference on" Computer as a Tool"*, volume 1, pages 217–220. IEEE, 2005.

[3] Alireza Alfi. Particle swarm optimization algorithm with dynamic inertia weight for online parameter identification applied to lorenz chaotic system. *International Journal of Innovative Computing, Information and Control*, 8(2):1191–1203, 2012.

[4] Habes Alkhraisat and Hasan Rashaideh. Dynamic inertia weight particle swarm optimization for solving nonogram puzzles. *International Journal of Advanced Computer Science and Applications (IJACSA)*, 7(10):277–280, 2016.

[5] Aris Anagnostopoulos, Luca Becchetti, Carlos Castillo, Aristides Gionis, and Stefano Leonardi. Online team formation in social networks. In *Proceedings of the 21st international conference on World Wide Web*, pages 839–848, 2012.

[6] Kenneth R Baker and Stephen G Powell. Methods for assigning students to groups: A study of alternative objective functions. *Journal of the Operational Research Society*, 53(4):397–404, 2002.

[7] Adil Baykasoglu, Turkay Dereli, and Sena Das. Project team selection using fuzzy optimization approach. *Cybernetics and Systems: An International Journal*, 38(2):155–185, 2007.

[8] Mehdi Beheshtian-Ardekani and Mo A Mahmood. Education development and validation of a tool for assigning students to groups for class projects. *Decision Sciences*, 17(1):92–113, 1986.

[9] Jack Brimberg, Nenad Mladenović, and Dragan Urošević. Solving the maximally diverse grouping problem by skewed general variable neighborhood search. *Information Sciences*, 295:650–675, 2015.

[10] Ai-ling Chen, Gen-ke Yang, and Zhi-ming Wu. Hybrid discrete particle swarm optimization algorithm for capacitated vehicle routing problem. *Journal of Zhejiang University-Science A*, 7(4):607–614, 2006.

[11] Shi-Jie Chen and Li Lin. Modeling team member characteristics for the formation of a multifunctional team in concurrent engineering. *IEEE transactions on Engineering Management*, 51(2):111–124, 2004.

[12] Shi Cheng, Yuhui Shi, and Quande Qin. Population diversity based study on search information propagation in particle swarm optimization. In *2012 IEEE Congress on Evolutionary Computation*, pages 1–8. IEEE, 2012.

[13] James S Dyer and John M Mulvey. An integrated optimization/information system for academic departmental planning. *Management Science*, 22(12):1332–1341, 1976.

[14] Walaa H El-Ashmawi, Ahmed F Ali, and Mohamed A Tawhid. An improved particle swarm optimization with a new swap operator for team formation problem. *Journal of Industrial Engineering International*, 15(1):53–71, 2019.

[15] Andries P Engelbrecht. *Computational intelligence: an introduction*. John Wiley & Sons, 2007.

[16] ZP Fan, Y Chen, Jian Ma, and S Zeng. Erratum: A hybrid genetic algorithmic approach to the maximally diverse grouping problem. *Journal of the Operational Research Society*, 62(7):1423–1430, 2011.

[17] Erin L Fitzpatrick and Ronald G Askin. Forming effective worker teams with multi-functional skill requirements. *Computers & Industrial Engineering*, 48(3):593–608, 2005.

[18] Micael Gallego, Manuel Laguna, Rafael Martí, and Abraham Duarte. Tabu search with strategic oscillation for the maximally diverse grouping problem. *Journal of the Operational Research Society*, 64(5):724–734, 2013.

[19] Frank Heppner and Ulf Grenander. A stochastic nonlinear model for coordinated bird flocks. *The ubiquity of chaos*, 233:238, 1990.

[20] Tsu-Feng Ho, Shyong Jian Shyu, Feng-Hsu Wang, and Chris Tian-Jen Li. Composing high-heterogeneous and high-interaction groups in collaborative learning with particle swarm optimization. In *2009 WRI World congress on computer science and information engineering*, volume 4, pages 607–611. IEEE, 2009.

[21] Bassem Jarboui, Najeh Damak, Patrick Siarry, and A Rebai. A combinatorial particle swarm optimization for solving multi-mode resource-constrained project scheduling problems. *Applied Mathematics and Computation*, 195(1):299–308, 2008.

[22] Mehdi Kargar, Morteza Zihayat, and Aijun An. Finding affordable and collaborative teams from a network of experts. In *Proceedings of the 2013 SIAM International Conference on Data Mining*, pages 587–595. SIAM, 2013.

[23] James Kennedy and Russell Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95-International Conference on Neural Networks*, volume 4, pages 1942–1948. IEEE, 1995.

[24] James Kennedy and Russell C Eberhart. A discrete binary version of the particle swarm algorithm. In *1997 IEEE International conference on systems, man, and cybernetics. Computational cybernetics and simulation*, volume 5, pages 4104–4108. IEEE, 1997.

[25] Theodoros Lappas, Kun Liu, and Evimaria Terzi. Finding a team of experts in social networks. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 467–476, 2009.

[26] Yen-Ting Lin, Yueh-Min Huang, and Shu-Chen Cheng. An automatic group composition system for composing collaborative learning groups using enhanced particle swarm optimization. *Computers & Education*, 55(4):1483–1493, 2010.

[27] Yiping Lou, Philip C Abrami, and John C Spence. Effects of within-class grouping on student achievement: An exploratory model. *The Journal of Educational Research*, 94(2):101–112, 2000.

[28] Hossein Nezamabadi-pour, M Rostami-Shahrbabaki, and Malihe Maghfoori-Farsangi. Binary particle swarm optimization: challenges and new solutions. *CSI J Comput Sci Eng*, 6(1):21–32, 2008.

[29] Kaveh Pashaei, Fattaneh Taghiyareh, and Kambiz Badie. A recursive genetic framework for evolutionary decision-making in problems with high dynamism. *International Journal of Systems Science*, 46(15):2715–2731, 2015.

[30] Sadhana Puntambekar, Kris Nagel, Roland Hübscher, Mark Guzdial, and Janet L Kolodner. Intra-group and intergroup: An exploration of learning with complementary collaboration tools. In *Proc. 2nd International Conference on Computer Supported Collaborative Learning (CSCL'97)*, pages 207–214, 1997.

[31] Craig W Reynolds. Flocks, herds and schools: A distributed behavioral model. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 25–34, 1987.

[32] Francisco J Rodriguez, Manuel Lozano, Carlos García-Martínez, and Jonathan D GonzáLez-Barrera. An artificial bee colony algorithm for the maximally diverse grouping problem. *Information Sciences*, 230:183–196, 2013.

[33] Dominick Sanchez. Ds-pso: Particle swarm optimization with dynamic and static topologies. 2017.

[34] Davoud Sedighizadeh and Ellips Masehian. Particle swarm optimization methods, taxonomy and applications. *International journal of computer theory and engineering*, 1(5):486, 2009.

[35] DY Sha and Cheng-Yu Hsu. A hybrid particle swarm optimization for job shop scheduling problem. *Computers & Industrial Engineering*, 51(4):791–808, 2006.

[36] Yuhui Shi and Russell C Eberhart. Empirical study of particle swarm optimization. In *Proceedings of the 1999 congress on evolutionary computation-CEC99 (Cat. No. 99TH8406)*, volume 3, pages 1945–1950. IEEE, 1999.

[37] Jason Tillett, Raghuveer Rao, and Ferat Sahin. Cluster-head identification in ad hoc sensor networks using particle swarm optimization. In *2002 IEEE International Conference on Personal Wireless Communications*, pages 201–205. IEEE, 2002.

[38] Lev Semenovich Vygotsky. *Mind in society: The development of higher psychological processes*. Harvard university press, 1980.

[39] Kang-Ping Wang, Lan Huang, Chun-Guang Zhou, and Wei Pang. Particle swarm optimization for traveling salesman problem. In *Proceedings of the 2003 international conference on machine learning and cybernetics (IEEE cat. no. 03ex693)*, volume 3, pages 1583–1585. IEEE, 2003.

[40] RR Weitz and S Lakshminarayanan. An empirical comparison of heuristic and graph theoretic methods for creating maximally diverse groups, vlsi design, and exam scheduling. *Omega*, 25(4):473–482, 1997.

[41] RR Weitz and S Lakshminarayanan. An empirical comparison of heuristic methods for creating maximally diverse groups. *Journal of the operational Research Society*, 49(6):635–646, 1998.

[42] Wikipedia contributors. Ruby (programming language) — Wikipedia, the free encyclopedia, 2020. [Online; accessed 9-April-2020].

[43] Stephen JH Yang. Context aware ubiquitous learning environments for peer-to-peer collaborative learning. *Journal of Educational Technology & Society*, 9(1):188–201, 2006.

[44] Peng-Yeng Yin, Kuang-Cheng Chang, Gwo-Jen Hwang, Gwo-Haur Hwang, and Ying Chan. A particle swarm optimization approach to composing serial test sheets for multiple assessment criteria. *Journal of Educational Technology & Society*, 9(3):3–15, 2006.

[45] ARMEN Zzkarian and Andrew Kusiak. Forming teams: an analytical approach. *IIE transactions*, 31(1):85–97, 1999.

Appendix A

Source Code

A.1 Instructions

Complete source code listings must be submitted as an appendix to the report. The project source codes are usually spread out over several files/units. You should try to help the reader to navigate through your source code by providing a "table of contents" (titles of these files/units and one line descriptions). The first page of the program listings folder must contain the following statement certifying the work as your own: "I verify that I am the sole author of the programs contained in this folder, except where explicitly stated to the contrary". Your (typed) signature and the date should follow this statement.

All work on programs must stop once the code is submitted to KEATS. You are required to keep safely several copies of this version of the program and you must use one of these copies in the project examination. Your examiners may ask to see the last-modified dates of your program files, and may ask you to demonstrate that the program files you use in the project examination are identical to the program files you have uploaded to KEATS. Any attempt to demonstrate code that is not included in your submitted source listings is an attempt to cheat; any such attempt will be reported to the KCL Misconduct Committee.

You may find it easier to firstly generate a PDF of your source code using a text editor and then merge it to the end of your report. There are many free tools available that allow you to merge PDF files.