Republic of Algeria Democratic and Popular
Ministry of Higher Education and Scientific Research
University of Science and Technology Houari Boumediene

Faculty of Mathematics

Department of Operational Research

Rapport

Bachelor in Operational Research

Theme

Optimizing Multiple Team Formation Problem
using Meta-heuristics Algorithms

Presented by:
SEKKAI Meriem
MEKERBA Ali

Supervised by :
Mr.CHAIBLAINE Yacine

Academic year : 2022/2023

Contents

List of Figures

Chapter 0

Optimizing Multiple Team Formation Problem with Meta-heuristics Algorithms

Abstract

In today's fast-paced world, building effective teams for multiple projects is a challenging task. Allocating individuals with the right skills and social compatibility can have a significant impact on team productivity and success. To address this issue, we propose the Multiple Team Formation Problem (MTFP) as an optimization problem that takes into account the social interaction among team members. By using sociometric techniques and metaheuristic algorithms such as Simulated Annealing, Genetic Algorithms, and Particle Swarm Optimization, we aim to find the best possible teams for various projects.

Our approach is not only beneficial for project managers, but it's also a core algorithm for a startup called Aplox. The platform connects skilled people from different communities to identify real-world challenges and the skills required to solve them. Using our team-making algorithm, Aplox helps individuals form collaborative teams which cover all the required skills and minimizes the communication cost among team members. Our proposed method offers a systematic and effective way to build teams, taking into account social interaction among team members, which can significantly improve team performance and productivity in real-world scenarios.

In conclusion, our algorithm can be used as a decision-making tool for managers to build effective teams for multiple projects, and as a collaboration platform for strangers who want to work together as a team to solve real-world challenges.

Chapter 1

MTFP Problem

1.1 Introduction

The ability to form effective teams is essential for the success of any project. Traditionally, teams have been built based solely on technical skills. However, recent studies have emphasized the importance of social interaction and cohesion among team members [1]. The interdependence of skills, trust, integration, and technical abilities of each team member plays a significant role in the success of a team. Ineffective teams with a lack of cohesion and social interaction can lead to reduced motivation, productivity, and ultimately, impact project success.

The Multiple Team Formation Problem (MTFP) is a complex optimization problem that considers social interaction among team members while forming teams for multiple projects. Various approaches have been developed to solve the MTFP while minimizing computational costs. However, it remains an NP-hard problem that requires the use of metaheuristic algorithms to find good solutions. [2]

In this thesis, we propose a metaheuristic-based approach to solve the MTFP, which is a complex optimization problem that considers the social interaction among team members while forming teams for multiple projects. Our approach will employ various metaheuristic algorithms, such as Simulated Annealing, Genetic Algorithms, and Particle Swarm Optimization, to find near-optimal solutions. By incorporating sociometric techniques, we will formulate the problem as an optimization problem and use our proposed approach to provide insights for managers to form effective teams with high social cohesion. The ultimate goal is to improve team performance and productivity in real-world scenarios.

The remainder of this thesis is structured as follows: Section 2 provides a review of related work in team formation and metaheuristic algorithms. Section 3 describes the problem formulation and proposed algorithm in detail. Finally, we conclude the thesis with Section 4 and present the experimental setup and results obtained using our algorithm.

1.2 Methodology

In this section, we will discuss the methodology we used to solve the MTFP. Our goal was to assign individuals to multiple projects, considering both their technical skills and social compatibilities. To accomplish this, we planned to employ a metaheuristic algorithm, which can search for near-optimal solutions within a reasonable amount of computational time.

We are currently exploring several metaheuristic algorithms that can potentially solve MTFP, including Simulated Annealing (SA), Genetic Algorithms (GA), and Particle Swarm Optimization (PSO), BRAin Drain Optimization (BRADO) and Jaya Optimization [3]. We are also investigating the feasibility of implementing a state space reduction algorithm to further enhance the efficiency of the optimization process.

By using these algorithms, we can generate and evaluate candidate solutions iteratively in a large search space. This iterative process allows us to potentially find optimal or near-optimal solutions for MTFP.

To evaluate the social compatibility of individuals, we plan to define a set of attributes that includes personality traits, communication style preferences (online or offline), language proficiency, and work availability. We plan to use sociometric techniques to quantify the social interaction among team members based on these attributes.

Now, we aim to tackle MTFP by formulating it as an optimization problem. The objective of our optimization problem is to minimize both the communication cost and search space while forming a team that can effectively perform all the required tasks. Table 1 provides the necessary terms and mathematical notations for this problem.

Notation	***Meaning***
G	Graph of experts with number of skillsets
G'	Sub-graph of experts with required skillsets
S	Number of skills defined in social network
S'	A set of reduced number of skills in social network
s(x_i)	An expert xi having a skill
SP(s_k)	A set of professionals skilled in sk
T	Tasks to be fulfilled by experts of specific skills
TC	The sum of distance between every expert pair in the team
CC	Communication cost between two experts
X	Number of experts in social network

Table 1.1: List of notations

1.3 MTFP Formulation

In the team formation problem, we can view the process as a graph, G(X,S), where X = {x_1,...,x_m} represents the m number of experts and S = {s_1,...,s_n} represents the n number of skills. Each expert, xi, has a set of skills, s(x_i)⊆S, and the set of skilled experts with skills s_k is denoted by SP(s_k) ⊆X. The goal is to find a set of experts that can cover all or some of the skills in set S.

To model the problem mathematically, we need to consider:

1.3.1 Communication cost

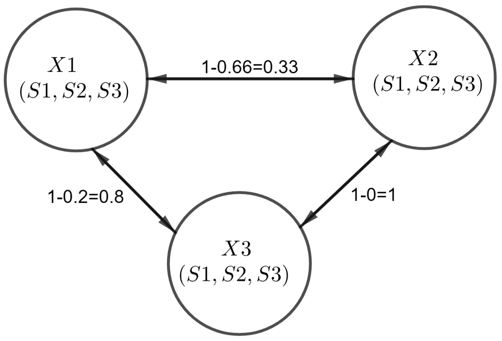
It is the measure of how closely related two experts are in the given social network based on their common skills [4] We can calculate the communication cost between two adjacent experts, x_i and x_j, using the Jaccard distance [5](the measure of similarity between two sets. It is defined as the ratio of the size of the intersection of the two sets to the size of the union of the two sets), as shown in Eq (1). On the other hand, the communication cost between non-adjacent experts is the sum of the shortest path between them.

$$CC(x_i, x_j) = 1 - \frac{s(x_i) \bigcap s(x_j)}{s(x_i) \bigcup s(x_j)} \tag{1.1}$$

Total Cost (TC) is the measure of the total distance between a Team of Experts (TE) with skills from graph G(X,S)

$$TC = \sum_i^n \sum_{j=i+1}^n d(TE_i, TE_j) \tag{1.2}$$

To illustrate this problem, let's consider a scenario where we have three experts, X = {x_1, x_2, x_3}, each with a different set of skills, S = {s_1, s_2, s_3, s_4, s_5}. We need to form teams that can cover all required skills while minimizing the communication cost between team members.



One possible team that can be formed is TE, consisting of experts with the required skills s_1, s_2, s_4. In addition to creating individual teams, we can also form social networks of teams, where experts can communicate with each other effectively. The figure shows an example of how we can create T1 = {x_1, x_3} and T2 = {x_2, x_3}, among other possible teams. This approach helps to optimize team formation, allowing them to perform tasks effectively while minimizing communication costs.

1.3.2 Search space reduction

We propose a method to reduce the search space in order to optimize the team formation process. This involves obtaining a sub-set of the original data that can effectively represent the entire set of data. We do this by reducing the data both horizontally and vertically.

Horizontal reduction involves selecting only the skills that are required for the given task, thus discarding any unnecessary skills. This helps to streamline the search process by narrowing down the focus to only the relevant skills.

Vertical reduction involves discarding any experts who do not have any of the required skills for the given task. This further reduces the search space and eliminates any irrelevant experts who would not contribute to the task.

The result of this reduction process is a reduced graph G', which contains only the reduced experts X' and reduced skills S'. By searching for optimal solutions within this reduced graph, we can significantly reduce the computational complexity and improve the efficiency of the team formation process. [6]

To ensure effective communication and minimize costs, we need to consider candidate preferences when forming teams. These preferences include factors such as language of communication, work mode, and communication style. For instance, if all members speak French, work part-time, and prefer remote work, we can use this information to guide the team formation process and create teams with good communication compatibility.

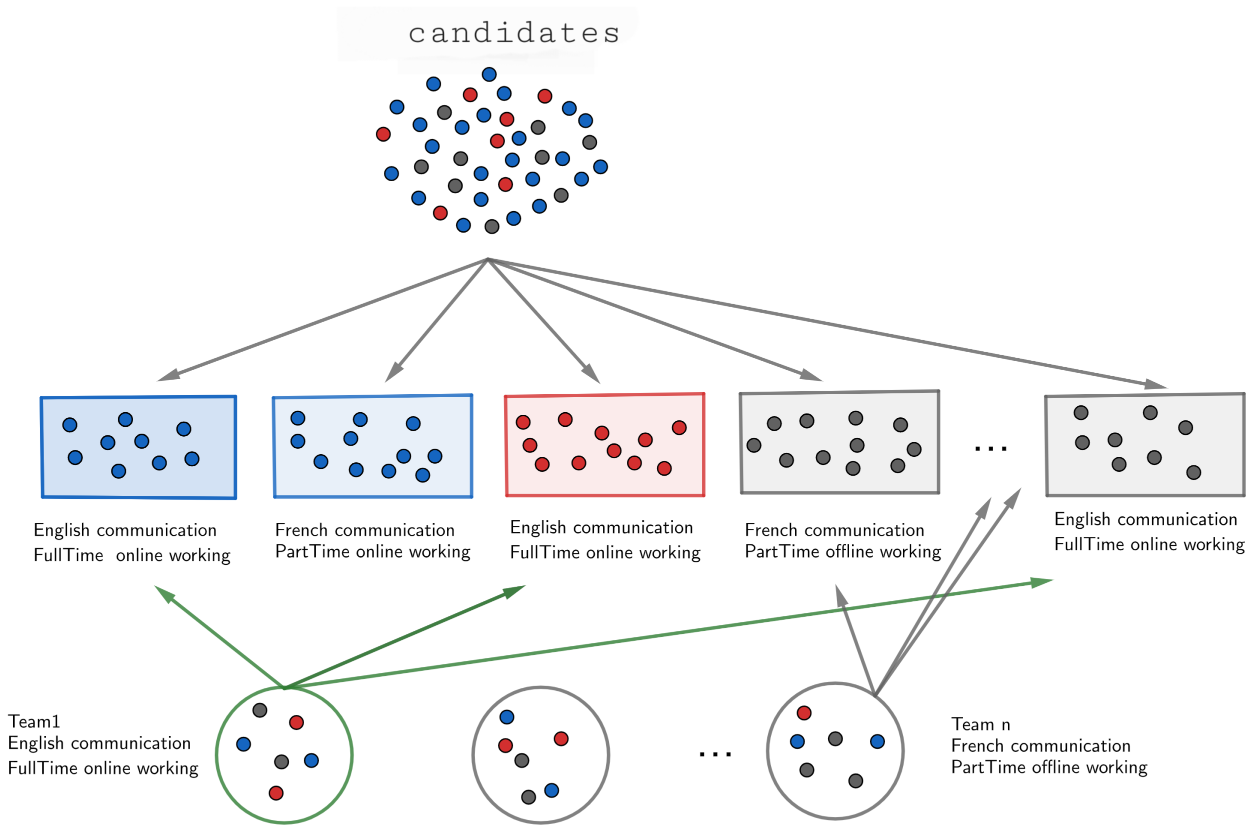

candidates
English communication FullTime online working
French communication PartTime online working
English communication FullTime online working
French communication PartTime offline working
...
English communication FullTime online working
Team1 English communication FullTime online working
...
Team n French communication PartTime offline working

Figure 1.1: A Visual Guide to the MTFP Algorithm

Chapter 2

Optimization Methods for MTFP

2.1 Optimization methods

Optimization methods are powerful problem-solving techniques that play a critical role in various fields, including Operations Research, engineering, economics, and finance. By finding the best solution to a problem while considering specific constraints, optimization methods can help organizations make better decisions, reduce costs, increase efficiency, and improve overall performance. However, solving complex optimization problems is not always straightforward, and traditional optimization methods may not always provide satisfactory results within a reasonable time frame. This is where alternative techniques, such as heuristics and metaheuristics, come into play, offering more flexible and efficient approaches to finding near-optimal solutions to complex problems.

Let's take a closer look at some of these optimization methods. Exact methods are often referred to as "brute force" methods, because they search through every possible solution in order to find the best one. While this can be effective for small problems, it quickly becomes impractical for larger ones. Imagine trying to find the optimal route for a delivery truck that needs to visit 100 different locations - the number of possible routes is so large that it would take years for a computer to search through them all!

That's where heuristics come in. These methods use "rules of thumb" to quickly eliminate poor solutions and focus on the most promising ones. For example, a heuristic for the traveling salesman problem might start by choosing the two closest cities and then repeatedly adding the next closest city until all cities have been visited. This approach won't always find the optimal solution, but it's usually good enough to get close.

Finally, we have metaheuristics, which take heuristic methods to the next level by combining multiple heuristics in a way that allows them to explore a wider range of solutions. One popular example is the genetic algorithm, which mimics the process of natural selection to gradually improve the quality of solutions over time.

All of these methods have their strengths and weaknesses, and the best approach will depend on the specific problem at hand. As the famous mathematician John von Neumann once said, "There's no sense in being precise when you don't even know what you're talking about." In other words, sometimes an approximate solution is better than no solution at all!

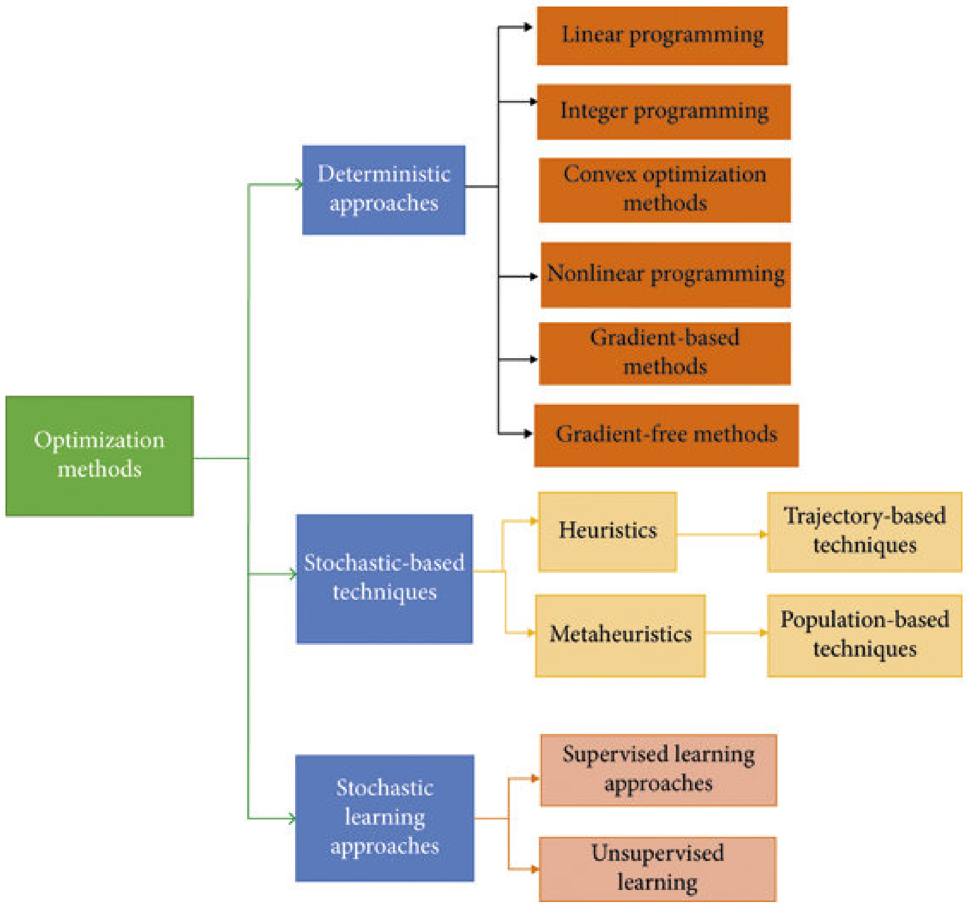


Figure 2.1: Classification of optimization techniques

2.1.1 Heuristics

First introduced in 1945 by G. Polya and later developed in the 70s, heuristics are basically designed to provide better computational performance compared to conventional optimisation techniques (such as exact methods).
At the expense of lower accuracy, heuristics work on complex optimisation problems by breaking them down into more manageable sub-problems that are solved using a combination of known methods.

Heuristics are a class of optimization methods that aim to provide quick and practical solutions to complex problems. Unlike exact methods that are designed to solve problems to optimality, heuristics are focused on finding near-optimal solutions within a reasonable amount of time.

The term "heuristics" derives from the Greek word "heuriskein," which means "to discover." The idea behind heuristics is to use a set of rules or principles to guide the search for solutions to complex problems. These rules are often based on intuition, past experience, or common sense.

One of the key advantages of heuristics is their ability to handle complex problems that are difficult to solve using exact methods. Heuristics work by breaking down a complex problem into smaller, more manageable sub-problems that can be solved using known techniques. This divide-and-conquer approach enables heuristics to find solutions that are close to the optimal solution, even if the problem itself is very difficult to solve.

Heuristics have been successfully applied in many fields, including computer science, engineering, and business. For example, in the field of computer science, heuristics have been used to develop algorithms for scheduling, routing, and network optimization. In the field of business, heuristics have been used to optimize supply chain management, inventory control, and production scheduling.

In conclusion, heuristics are a powerful class of optimization methods that can provide practical solutions to complex problems. While they may not always find the optimal solution, heuristics are able to find near-optimal solutions within a reasonable amount of time. With their ability to handle complex problems and their widespread applications, heuristics will continue to be an important tool in the field of optimization.

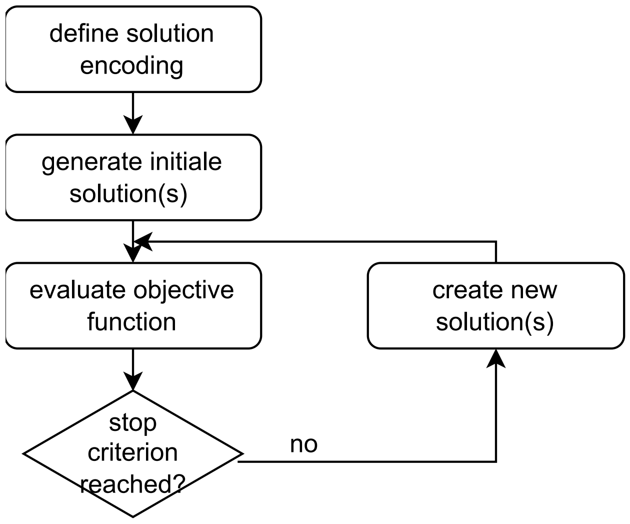


Figure 2.2: General principle of a heuristic method

2.1.2 Metaheuristics

Metaheuristics can be thought of as a "survival of the fittest" approach to problem-solving. Just like how organisms evolve and adapt to their environments over time, metaheuristic algorithms also evolve and adapt their search strategies to find better solutions over time.

One common metaphor used to explain metaheuristics is the concept of a traveler trying to find the best route to a destination. Imagine a traveler who has never been to a particular city before, and has no map or GPS to guide them. They must rely on trial and error to find the best route. They might try different roads, highways, shortcuts, and ask locals for directions. Each time they try a new route, they learn something new and adjust their strategy accordingly until they find the best route to their destination. Metaheuristics work in a similar way - they try different solutions, learn from their successes and failures, and continually refine their search strategy until they find the best solution to a problem.

Metaheuristics can be applied to a wide range of problems, including scheduling, routing, resource allocation, and machine learning. Some popular metaheuristic algorithms include simulated annealing, tabu search, genetic algorithms, and particle swarm optimization.

One of the benefits of metaheuristics is their ability to handle complex, real-world problems with large solution spaces that cannot be easily solved by traditional optimization methods. Additionally, metaheuristics can often find near-optimal solutions in a reasonable amount of time, making them a practical and effective tool for many practical applications.

In summary, metaheuristics are powerful problem-solving techniques that rely on generating and refining sets of potential solutions to find the best solution to a problem. They are adaptable, flexible, and can be applied to a wide range of real-world problems. So, next time you're lost in a new city, just remember that metaheuristics have got you covered!

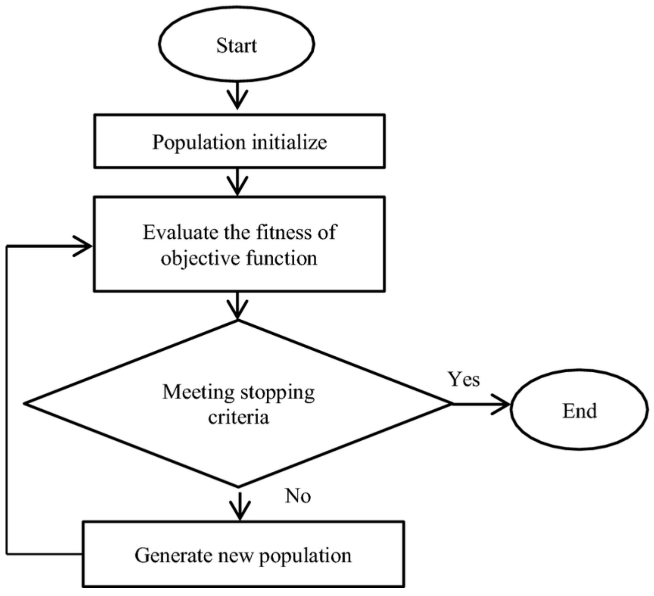

Start
Population initialize
Evaluate the fitness of objective function
Meeting stopping criteria
Yes
End
No
Generate new population

Figure 2.3: General principle of a metaheuristic method

2.2 Genetic algorithm

2.2.1 Introduction

Genetic algorithms are a fascinating optimization technique inspired by the principles of natural selection and genetics. Invented by John Holland in the 1970s, genetic algorithms have since become a widely-used method for solving complex problems in engineering, computer science, and beyond.

At their core, genetic algorithms mimic the process of natural selection to generate new solutions to a problem. Just as biological organisms evolve over time to adapt to their environment, genetic algorithms use selection, crossover, and mutation operators to generate and improve candidate solutions to a problem. By combining these operators with a fitness function that evaluates the quality of each solution, genetic algorithms can quickly explore a large search space and converge towards optimal solutions.

One of the benefits of genetic algorithms is their flexibility and adaptability. They can be applied to a wide range of problems, from optimizing complex engineering designs to training machine learning models. They also have the advantage of being relatively easy to implement and parallelize, making them well-suited to modern computing architectures.

Despite their mathematical and algorithmic nature, genetic algorithms have a certain charm and appeal that draws researchers and enthusiasts alike. They offer a window into the natural world, showing us how evolution and adaptation can be harnessed to solve some of the most challenging problems we face. Whether you're a biologist, computer scientist, or simply curious about the natural world, genetic algorithms offer a fascinating glimpse into the mysteries of life and learning.

The algorithm works by creating a population of individuals, where each individual represents a potential solution to the problem at hand, and each individual is represented by a string, inspired by the chromosome. Each individual (chromosome/solution) is composed of several genes.

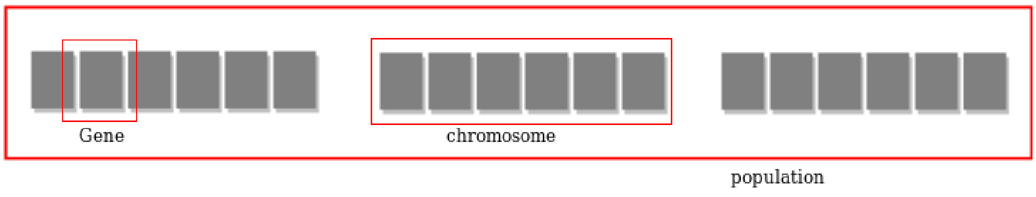


Figure 2.4: A description of the terminology used in genetic algorithms.

The algorithm starts with two individuals representing the parents, where each individual has a fitness score.

Fitness is one of the basic functions of the algorithm, which verifies the effectiveness of the potential solution compared to a defined condition, where the individuals with high fitness score are the more likely to pass to the next generation (or survive in the biological term) by the end of **the selection**.

Each individual (chromosome) is composed of several genes, during reproduction, the algorithm creates new individuals by exchanging genes of the selected chromosomes (solutions) using the genetic operation called **crossover**. and making some small changes in the individual genes, inspired by the natural genetic technique called **mutation**. To obtain a new generation that will again pass through the fitness function to check if it satisfies the termination conditions.

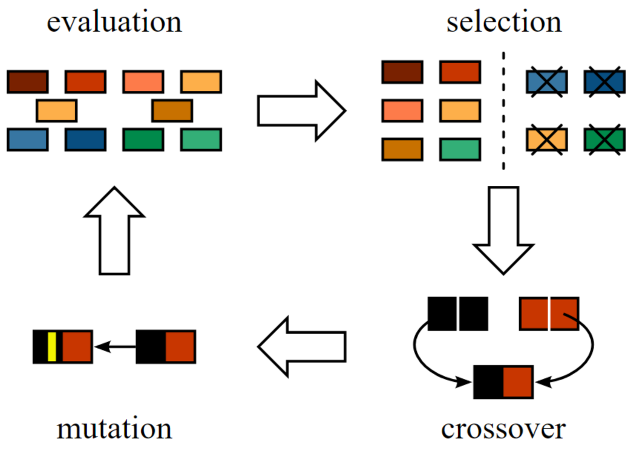


Figure 2.5: An explanation of the sequence of actions involved in implementing a genetic algorithm.

The cycle :*fitness evaluation, selection, crossover, mutation* is repeated until a satisfactory solution is found.

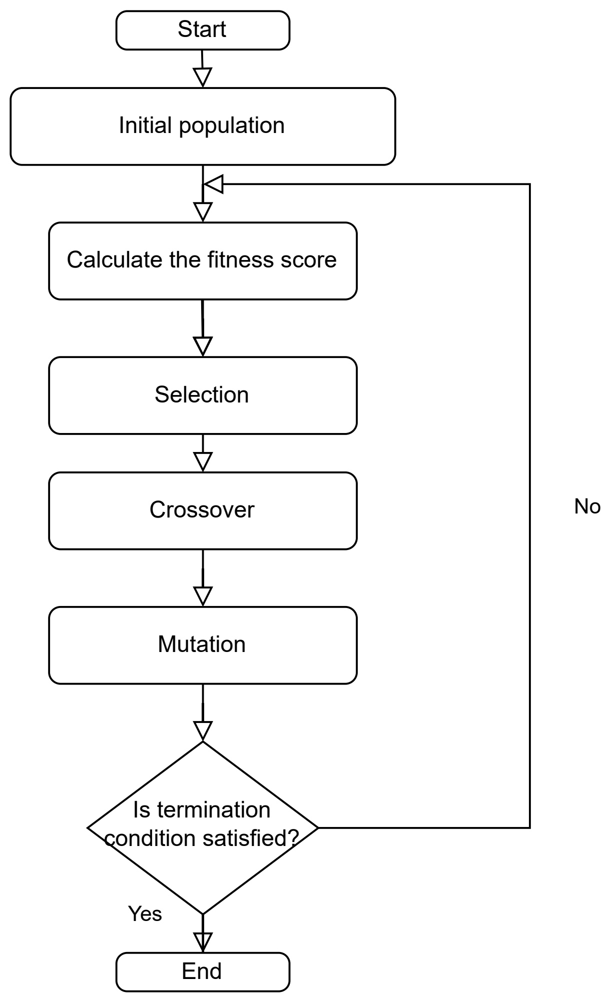

Start
Initial population
Calculate the fitness score
Selection
Crossover
Mutation
Is termination condition satisfied?
No
Yes
End

Figure 2.6: Genetic Algorithm Based on Natural Selection Theory for Optimization Problems

In conclusion, genetic algorithms are a powerful optimization technique inspired by the principles of natural selection and genetics. The algorithm involves creating a population of potential solutions, evaluating their fitness, selecting the fittest individuals, and breeding them to produce the next generation. This process continues until a satisfactory solution is reached.

Genetic algorithms have been successfully applied in various fields, including engineering, finance, and computer science, to solve optimization problems that would be difficult or impossible to solve using traditional methods.

By mimicking the process of natural selection, genetic algorithms can effectively explore the solution space and converge towards an optimal solution. They can handle both discrete and continuous optimization problems and can be adapted to various constraints and objectives.

In summary, genetic algorithms are a valuable tool for solving complex optimization problems. They provide a flexible and powerful approach that can handle a wide range of problems and produce high-quality solutions.

2.2.2 Genetic algorithm applied in MTFP:

Representation of the problem: The MTFP can be solved using a genetic algorithm. The problem can be represented by a database that contains information about the members' skills, language preferences, communication style preferences, work availability, and matching challenges.

To start the genetic algorithm, the first step is to filter the members for a given challenge based on their skills that match the challenge's requirements and their personal preferences. This step is achieved by calling the "filter_eligible_members" function, which filters the dataset to only include members who possess the required skills and have not been assigned to a team already.

Once the eligible members for a challenge are identified, the algorithm generates an initial population of teams. Each set of teams will verify the requirements of the challenge.

Using GA tools to improve the solution, the algorithm assigns a fitness score to each team, which is determined by how well the team members complement each other's skills and their preferences, such as language and communication style. The "weighted_score function" computes a weighted score for each member, taking into account missing skills, matching skills and availability, and assigns a score to the team based on the members' scores.

Based on their fitness scores, the algorithm then selects parents for the next generation. The selection process is implemented using a tournament selection strategy, which selects the best individuals from a random subset of the population. The offspring are generated through crossover and mutation of the team members. The crossover process involves exchanging genetic information between two parents, resulting in offspring with a combination of their traits. The mutation process involves randomly replacing members of the team with other matching members who have the required skills.
The offspring are evaluated using the fitness function, and the least fit individuals in the population are replaced by the new offspring. The process is repeated for a certain number of generations, defined by the input of the algorithm, until the termination condition is satisfied.

The output of the algorithm would be the best set of teams found with the highest fitness score. In other words, a set of teams that can solve all challenges and have compatible language and communication preferences.

The mutation function is called during the crossover and mutation process to ensure that eligible members with missing skills are added to the team. The function uses a weighted score to sort eligible members by descending order and adds them to the team until all required skills are filled or there are no more eligible members.

In order to implement the genetic algorithm for solving MTFP, we will use a pseudo code approach to provide a clear and concise representation of the algorithm. The pseudo code will outline the steps involved in the algorithm, from filtering eligible members to generating an initial population of teams, and then improving the solution through genetic operations such as crossover and mutation. Let's take a look at the pseudo code implementation below:

Before we dive into the pseudo code for creating teams with the required skills for a challenge, let's clarify a few things.
First, a team can only solve one challenge at a time.
Second, a team is considered formed when all the required skills for the challenge are present within the team.
Third, we want to minimize communication costs between team members to ensure effective collaboration.

Now, let's take a look at the pseudo code:

Algorithm 1 Genetic Algorithm Pseudocode for Multiple Team Formation Optimization

```
 1: Initialize empty list of teams
 2: for each challenge do
 3:     Initialize empty list of assigned members
 4:     while there are unassigned required skills for the challenge do
 5:         Initialize empty list of eligible members for the challenge
 6:         for each member in the dataset do
 7:             if the member possesses the required skill for the challenge and has not been assigned to the team then
 8:                 Append the member to the eligible members list
 9:             end if
10:         end for
11:         if the eligible members list is empty then
12:             Break the loop and move to the next challenge
13:         end if
14:         Initialize variables for current team's skills, language, communication style, and availability
15:         for each member in the current team do
16:             Add member's skills to current team's skills
17:             Add member's language to current team's language
18:             Add member's communication style to current team's communication style
19:             Take the intersection of member's availability and the current team's availability
20:         end for
21:         Compute the missing skills for the challenge by subtracting the current team's skills from the required skills
22:         Remove any eligible member from the list who has the same skill and level 3 as an existing member in the team
23:         for each member in the eligible members list do
24:             Compute the weighted score for the member based on their skills, level, availability, and missing skills
25:         end for
26:         Sort the eligible members list by descending order of their weighted score
27:         for each member in the eligible members list do
28:             if the member matches the current team's language, communication style, and availability, and has not been assigned to the team then
29:                 Add the member to the team
30:                 Add the member to the assigned members list
31:                 Update the current team's skills, language, communication style, and availability
32:                 Remove the assigned member's skill from the missing skills list for the challenge
33:                 if there are no more missing skills for the challenge then
34:                     Break the loop and move to the next challenge
35:                 end if
36:             end if
37:         end for
38:         Add the current team to the list of teams
39:     end while
40: end for
```

If you're wondering how long it will take for the algorithm to assign members to teams, fear not! Our trusty time complexity analysis is here to save the day.

The complexity can be expressed as O(max_iterations * (n + population_size * team_size)), where max_iterations is the maximum number of iterations the algorithm will run for, n is the number of members in the dataset, population_size is the number of teams in the population, and team_size is the number of members in each team.

In other words, the time it takes to form teams is directly proportional to the number of challenges, members in the dataset, and the number of teams and members per team. It's not rocket science, but it does take a bit of computational power.

Thankfully, the time complexity is reasonable and should work efficiently for most practical use cases

The application of genetic algorithms in solving the MTFP problem presents a promising solution to an otherwise complex problem. The use of evolutionary principles in generating solutions is a novel approach that has shown great success in solving optimization problems. In particular, the genetic algorithm used in this study has been shown to be effective in generating high-quality team compositions that meet the requirements of the given challenge while minimizing communication costs between team members.

One of the key strengths of the genetic algorithm is its ability to explore the solution space efficiently. This is achieved through the use of selection, crossover, and mutation operations that simulate natural selection, reproduction, and genetic variation. These operations enable the algorithm to converge towards high-quality solutions quickly, making it ideal for solving complex problems such as the MTFP.

Another strength of the genetic algorithm is its ability to handle constraints effectively. The MTFP problem is a constraint satisfaction problem, where the solution must satisfy multiple constraints such as the required skills, communication styles, and availability of team members. The genetic algorithm used in this study incorporates these constraints as fitness functions, ensuring that only solutions that meet the requirements are selected.

The results of our application demonstrate that the genetic algorithm is a viable approach for solving the MTFP problem. The algorithm was able to generate high-quality team compositions that met the requirements of the given challenge while minimizing communication costs between team members. Moreover, the algorithm was able to handle constraints effectively, producing solutions that satisfied the required skills, communication styles, and availability of team members.

In conclusion, the application of genetic algorithms in solving the MTFP problem is a promising area of research that holds great potential for solving complex optimization problems. The use of evolutionary principles in generating solutions provides an efficient and effective approach that can handle constraints and explore the solution space quickly.

Chapter 3

MTFP Application

3.1 Python language:

In the previous chapter we explained how the genetic algorithm works in the MTFP problem and the efficacy of the solution it would give. In this chapter we will implement it in the Python language.

3.1.1 History

In December 1989, **Guido van Rossum**, a Dutch programmer at **the Centrum Wiskunde Informatica (CWI)**in the Netherlands, created Python as a successor to *the ABC programming language*[7], a language designed to teach programming concepts, and wanted to create a similar language that was easy to read, write and understand.

Python was first released it in 1991 as *Python 0.9.0*, next Python 2.0 was released in 2000.
Python 3.0, released in 2008, was a major revision not completely backward-compatible with earlier versions.[7]

Python is a high-level, general-purpose programming language, which means it's easy to read and use compared to other popular codes like Java and C language. The simplicity of the syntax and the wide availability of libraries and tools that can be used for many purposes are the reasons for Python's popularity.

3.1.2 Why python?

libraries: Defined as a powerful language for data analysis and optimisation tasks, Python has several tools such as Num**Py, SciPy, Pandas, Matplotlib** and other scientific computing and data analysis libraries that can help and make it easy to manipulate.

Readability: Python is designed to use the English language, making it easy to read. With a set of strict rules that are in place, Python makes it easy to create code that anyone can follow.

Community: The popularity of Python ensures a large community of developers with lots of networking and on/offline workshops to ask, learn and ensure that bugs are fixed quickly and new features are added regularly.

Open-source-language: Another advantage of Python is that it is an open source language, which gives free access to the codes, encourages collaborative development, which can lead to better code quality and faster development cycles, and continuous improvement of the programming language.

Overall, Python's ease of use, flexibility, versatility, and powerful libraries make it a great language for a wide range of applications, including web development, data analysis, scientific computing, and artificial intelligence and optimisation tasks.

3.2 MTFP application using Python

Building effective teams for multiple projects is difficult these days, but the impact of having the right skills and social skills among team members is proven to have an impact on team productivity and project success.
In this purpose, we propose an algorithm to solve the MTFP that uses metaheuristic algorithms aiming to improve the solutions by creating the best performing teams.

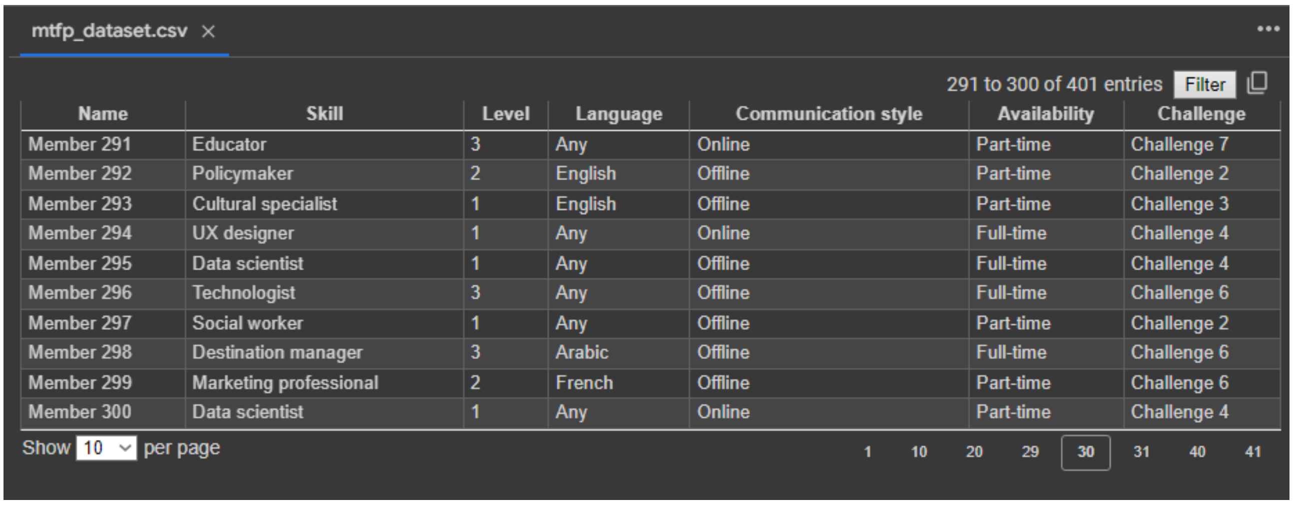
In this chapter, we will explain how our algorithm works on this team formation problem, which can be used as a decision-making tool to build effective teams to solve several real-world challenges.

In the first part, we will explain the problem formulation (data), the function used, the main code using genetic algorithm, and then an output example given by this algorithm.

3.2.1 Data

As defined in the problem description, we want to match individuals with a certain skill level in data science, computer engineer or marketing skills... and a set of different language and communication style preferences to a range of challenges that require a set of these skills. To be able to implement each member's information in the application, we need to structure it in a data set that contains:

1. The name of the member.
2. The skills and their level.
3. Their language preference (Arabic, French, English or "any" if the member can communicate in all 3).
4. Her communication style (online or offline).
5. Availability (full-time or part-time).
6. and a **randomly matched challenge** to their skill, as shown in FIGURE 3.1.


mtfp_dataset.csv

291 to 300 of 401 entries

Name	Skill	Level	Language	Communication style	Availability	Challenge
Member 291	Educator	3	Any	Online	Part-time	Challenge 7
Member 292	Policymaker	2	English	Offline	Part-time	Challenge 2
Member 293	Cultural specialist	1	English	Offline	Part-time	Challenge 3
Member 294	UX designer	1	Any	Online	Full-time	Challenge 4
Member 295	Data scientist	1	Any	Offline	Full-time	Challenge 4
Member 296	Technologist	3	Any	Offline	Full-time	Challenge 6
Member 297	Social worker	1	Any	Offline	Part-time	Challenge 2
Member 298	Destination manager	3	Arabic	Offline	Full-time	Challenge 6
Member 299	Marketing professional	2	French	Offline	Part-time	Challenge 6
Member 300	Data scientist	1	Any	Online	Part-time	Challenge 4

Show 10 per page

Figure 3.1: An instance of the data used in the study.

3.2.2 Functions

Load the dataset from the CSV file This code is basically creating a function that loads data from a CSV file called "mtfp_dataset.csv". The file contains information about members of a team, including their name, skill, level, language, communication style, availability, and challenges.

To read the data from the file, the code uses the 'csv' module in Python. It then reads each row of data from the file and creates a dictionary (member) that contains the information about each member. This makes it easier to access and manipulate the data later on.

The function returns the dataset, which is a list of all the members and their information. This information can then be used for various purposes, such as analyzing the team's strengths and weaknesses, identifying potential communication issues, or optimizing the team's performance.

In short, this code is all about loading data from a CSV file and structuring it in a way that makes it easy to work with. It's a simple but important step in many data-related tasks and can help you make sense of the information you have.

```python
def load_dataset():
    dataset = []
    with open('mtfp_dataset.csv', 'r') as file:
        reader = csv.DictReader(file)
        for row in reader:
            member = {}
            member["name"] = row["Name"]
            member["skill"] = row["Skill"]
            member["level"] = int(row["Level"])
            member["language"] = row["Language"]
            member["communication_style"] = row["Communication style"]
            member["availability"] = row["Availability"]
            member["challenge"] = row["Challenge"]
            dataset.append(member)
    return dataset
# Load the dataset
dataset = load_dataset()
```

Define the challenges and their required skills plus the languages, the communication styles and work availability.

This code defines two dictionaries that are used to store information about different challenges and their corresponding titles, as well as the language and communication style preferences and work availability of potential team members.

The first dictionary, 'challenges', includes a list of potential professions that may be relevant to each challenge. For example, for Challenge 1, the professions listed are "Data scientist", "Computer engineer", "Researcher", "Business analyst", "Policymaker", and "Educator".

The second dictionary, **'challenges_with_titles'**, maps each challenge to a specific title that describes the challenge. For instance, Challenge 1 is given the title "Lack of investment in advanced artificial intelligence".

The **'language_preferences'** list specifies the preferred languages of team members, which include English, Arabic, French, and any other language. Meanwhile, **'communication_style_preferences'** lists whether the team members prefer to communicate online or offline. Finally, the **'work_availability'** list indicates whether potential team members are available for part-time or full-time work.

```
challenges = {
    "Challenge 1": ["Data scientist", "Computer engineer", "Researcher", "Business analyst", "Policymaker", "Educator"],
    "Challenge 2": ["Technologist", "Mental health professional", "Social worker", "Psychologist", "Policymaker"],
    "Challenge 3": ["Anthropologist", "Community leader", "Cultural specialist", "Researcher", "Educator"],
    "Challenge 4": ["Cybersecurity expert", "Ethicist", "Data scientist", "Policymaker", "UX designer"],
    "Challenge 5": ["Environmental scientist", "Engineer", "Policymaker", "Educator", "Urban planner", "Business leader" ],
    "Challenge 6" :[ "Tourism expert", "Marketing professional", "Destination manager", "Tour operator", "Technologist", "Researcher"],
    "Challenge 7": ["Engineer", "Designer", "Entrepreneur", "Educator","Policymaker"]
}


challenges_with_titles = {
    "Challenge 1": "Lack of investment in advanced artificial intelligence",
    "Challenge 2": "Limited access to mental health services",
    "Challenge 3": "The decline of traditional cultures",
    "Challenge 4": "Inadequate protection of personal data and privacy",
    "Challenge 5": "High levels of pollution in urbanareas",
    "Challenge 6": "Limited access to affordable and accessible tourism services for solo travelers",
    "Challenge 7": "Restricted availability of virtual and augmented reality technology",
}

# Define the language and communication style preferences

language_preferences = ["English", "Arabic", "French", "Any"]
communication_style_preferences = ["Online", "Offline"]
work_availability = ["Part-time", "Full-time"]
```

Filter eligible members: This function filters out eligible members from a dataset based on certain criteria. It takes three arguments: dataset, which is a list of members with their relevant information such as skill, level, language, communication style, availability, and challenge; challenge, which is a string representing the specific challenge that needs to be addressed; and **assigned_members**, which is a list of names of members who have already been assigned to this challenge.

First, the function retrieves the list of required skills for the challenge from the challenges dictionary using the challenge argument. Then, it filters the dataset list to only include members who possess the required skills and have not been assigned already. It does this using a list comprehension that iterates over each member in the dataset, and checks if the member's challenge attribute matches the challenge argument and their name attribute is not in the assigned_members list.

Finally, the function returns two values: **eligible_members**, which is the filtered list of members who meet the criteria, and **required_skills**, which is the list of skills required for the challenge.

```python
def filter_eligible_members(dataset, challenge, assigned_members):
    # Define the required skills for the challenge
    required_skills = challenges[challenge]

    # Filter the dataset to only include members who possess the required
   skills and have not been assigned already
    eligible_members = [member for member in dataset if member["challenge"] ==
   challenge and member["name"] not in assigned_members]

    return eligible_members, required_skills
```

Initialize communication cost: This function is used to initialize the communication costs between team members based on their language preferences. The communication cost is represented as a dictionary, where the keys are tuples of languages and the values are the costs of communication between members who speak those languages.

The function loops through all possible pairs of languages, except for cases where both languages are the same. For each pair of languages, it sets the communication cost to 1, unless at least one of the languages is "Any", in which case the cost is set to 0.5.

Finally, the communication cost dictionary is returned by the function.

```python
def initialize_communication_costs():
    # Initialize the communication costs dictionary
    communication_costs = {}
    for language1 in language_preferences:
        for language2 in language_preferences:
            if language1 != language2:
                cost = 1
                if language1 == "Any" or language2 == "Any":
                    cost = 0.5
                communication_costs[(language1, language2)] = cost

    return communication_costs
```

initialize population: This function generates an initial population of teams, based on the parameters specified. Imagine you have a pool of eligible members with different skills, communication styles, availability, and language preferences. The initialize population function selects the best members for the job and combines them into a team of specified size.The function first selects a random subset of eligible members that meet the challenge's required skills. It then transforms each member into a tuple and eliminates any duplicates. Finally, it selects the top performers for the job and calculates their fitness function based on the communication costs and required skills.

```python
def initialize_population(population_size, eligible_members, team_size,
   communication_costs, required_skills):
    # Generate an initial population of teams
    population = []
    for i in range(population_size):
        team = random.sample(eligible_members, min(team_size,
   len(eligible_members)))
        team = [tuple(member.items()) for member in team]
        team = [dict(t) for t in set(team)]
        team = team[:team_size]
        fitness = fitness_function(team, communication_costs, required_skills)
        population.append({"team": team, "fitness": fitness})

    return population
```

In this function, we're selecting the parents that will be used to create the next generation of teams. To do this, we use the selection function, which chooses the best individuals from the population based on their fitness scores. We only want to select half the population as parents, so we divide the population_size parameter by 2.

By selecting the best individuals as parents, we're ensuring that the next generation of teams will be even better than the previous one. It's like choosing the most talented athletes to be on the national team - they're more likely to win the championship!

So, in summary, the **select_parents function** chooses the cream of the crop from the current population to be the parents of the next generation of teams.

```python
def select_parents(population, population_size):
    # Select the parents for the next generation
    parents = selection(population, population_size // 2)

    return parents
```

Have you ever wondered how nature creates new species through the combination of the genes of their parents? Well, this function does something similar but with teams of humans!
Given a set of parents, eligible members, required skills, assigned members, and communication costs, it generates offspring through the process of crossover and mutation.

First, it selects two parents from the given set of parents, and applies crossover to their team members. Crossover is like mixing the genes of the parents to create a new individual, but in this case, it's about mixing the skills and attributes of team members to create a new team.

Then, it applies mutation to the offspring. Mutation is like introducing random changes in the genes of an individual, but here, it's about changing the composition of the team by adding or removing team members randomly.

The function then evaluates the fitness of each offspring, which is a measure of how well the team meets the required skills and minimizes communication costs.

Finally, it updates the assigned members set with the names of the members in the offspring, so that they won't be assigned again in the future.

With this process, the function generates a new set of offspring that may be stronger than their parents and can contribute to the evolution of better teams.

```python
def generate_offspring(parents, eligible_members, required_skills, assigned_members, communication_costs):
    # Generate offspring through crossover and mutation
    offspring = []
    for i in range(0, len(parents), 2):
        parent1 = parents[i]
        parent2 = parents[i + 1] if i + 1 < len(parents) else parent1
        if parent1["team"] != parent2["team"]:
            offspring1, offspring2 = crossover(parent1["team"], parent2["team"])
            offspring1 = mutation(offspring1, eligible_members, required_skills, assigned_members)
            offspring2 = mutation(offspring2, eligible_members, required_skills, assigned_members)
            offspring.append({"team": offspring1, "fitness": fitness_function(offspring1, communication_costs,required_skills)})
            offspring.append({"team": offspring2, "fitness": fitness_function(offspring2, communication_costs, required_skills)})

            # Update the assigned_members set with the names of the members in the offspring
            assigned_members.append(member["name"] for member in offspring1 + offspring2)

    return offspring
```

At this point in the genetic algorithm, we've selected parents from the current population and generated offspring through crossover and mutation. Now, it's time to replace the least fit individuals in the population with the new offspring.

The replace_least_fit_individuals function takes in the current population, the desired population size, and the offspring that were just generated. It removes the least fit individuals in the current population (which is the first half of the population sorted by fitness) and replaces them with the offspring. Then, it sorts the entire population by fitness to ensure that the fittest individuals are at the end of the list.

By doing this, we are prioritizing the survival of the fittest individuals in the population while also introducing new genetic diversity through the offspring. This helps prevent the population from converging too quickly to a suboptimal solution and allows the algorithm to explore a wider range of possible solutions.

```python
def replace_least_fit_individuals(population, population_size, offspring):
    # Replace the least fit individuals in the population with the new offspring
    population = population[:population_size // 2] + offspring
    population = sorted(population, key=lambda x: x["fitness"])

    return population
```

The update_best_individual function plays a crucial role in our genetic algorithm as it helps us track the best solution we have found so far.

In the genetic algorithm, we maintain a population of candidate solutions to our problem. At each iteration of the algorithm, we generate new candidate solutions (offspring) by selecting the best-performing individuals (parents) from the current population and applying genetic operators such as crossover and mutation. We then evaluate the fitness of these new candidate solutions and replace the least fit individuals in the population with the new offspring.

Throughout this process, we want to keep track of the best-performing individual we have found so far, as this is the solution that we will ultimately return. The update_best_individual function takes in the current population and the current best individual and updates the best individual if necessary. Specifically, if the fitness of the best individual in the current population is less than the fitness of the current best individual, we update the current best individual to be the best individual in the current population.

By updating the best individual at each iteration of the algorithm, we ensure that we always have access to the best solution we have found so far. This is important because the genetic algorithm is a heuristic search algorithm, which means that it may not find the optimal solution to our problem. However, by keeping track of the best solution we have found so far, we can ensure that we return a high-quality solution even if it is not the optimal solution.

```python

def update_best_individual(population, best_individual):
    # Update the best individual if necessary
    if population[0]["fitness"] < best_individual["fitness"]:
        best_individual = population[0]

    return best_individual
```

Imagine you're on a quest to assemble the best team possible to accomplish a difficult challenge. You've spent countless hours scouring the land to find the most skilled and talented individuals who possess the required skills for your mission. Finally, you've gathered a group of people who seem to have what it takes to tackle the challenge. But how do you know if they're the best team possible?

This is where get_highest_skill_levels() comes in. It helps you identify the highest skill levels of each required skill for your team. It's like having a crystal ball that shows you how well your team will perform in each aspect of the challenge.

You take each team member and evaluate their skill level for each required skill. If a member is an expert in a skill, you set the skill level to 3 and mark that there is an expert in that skill. If a member has a skill level of 2 and there isn't yet an expert in that skill, you set the skill level to 2. Finally, you return a dictionary of the highest skill levels for each required skill.

With this information, you can assess whether your team is well-rounded and capable of tackling every aspect of the challenge. And if there are any gaps in your team's skills, you'll know exactly what you need to focus on to strengthen your team's abilities.

```python
    def get_highest_skill_levels(team, required_skills):
    skill_levels = {skill: 0 for skill in required_skills}
    has_expert = {skill: False for skill in required_skills}
    for member in team:
        if member["skill"] in required_skills:
            if member["level"] == 3:
                skill_levels[member["skill"]] = member["level"]
                has_expert[member["skill"]] = True
            elif member["level"] == 2 and skill_levels[member["skill"]] <= 1
   and not has_expert[member["skill"]]:
                skill_levels[member["skill"]] = member["level"]
    return skill_levels
```

Now you have a pool of potential team members, each with their own skills, communication styles, and languages spoken. You know that communication is key to the success of the project, and you need to find a way to calculate the communication cost of each potential team.

To do this, you write a function called calculate_communication_cost(). This function takes in a team, represented as a list of team members, and a dictionary of communication costs. The communication costs dictionary is a mapping of language pairs to costs. If two team members speak different languages, it will cost more to communicate with each other.

The function starts by checking whether all team members speak the same language. If they don't, it adds a high cost to the communication cost. If any team member speaks the language "any", it assumes that the team can communicate in any language and does not add any extra cost.

Next, the function loops through all pairs of team members and calculates the cost of communication between them. It does this by looking up the cost in the communication costs dictionary based on the language pair of the two team members. If the team members have different communication styles, the cost is doubled.

Finally, the function returns the total communication cost of the team. With this information, you can make an informed decision about which team to choose for your project.

```python
def calculate_communication_cost(team, communication_costs):
    all_same_language = len(set([member['language'] for member in team])) == 1
    any_language = 'any' in [member['language'] for member in team]
    communication_cost = 0
    for i in range(len(team)):
        for j in range(i+1, len(team)):
            member1 = team[i]
            member2 = team[j]
            communication_style1 = member1["communication_style"]
            communication_style2 = member2["communication_style"]
            language1 = member1["language"]
            language2 = member2["language"]
            cost = communication_costs.get((language1, language2), 1)  # 
   default to 1 if key not found
            if all_same_language and language1 != language2:
                communication_cost += 10
            elif not any_language and language1 == 'any' or language2 == 'any':
                communication_cost += 1
            else:
                if communication_style1 != communication_style2:
                    cost *= 2
                communication_cost += cost
    return communication_cost
```

Calculating the skill cost is an important step in evaluating the fitness of a team. This cost is determined by looking at the skill levels of each member in the team, and measuring how much they fall short of the required level. The higher the skill cost, the less fit the team is considered to be. Think of it like filling out a job application where you need to meet certain skill requirements. If you don't meet those requirements, your application won't be considered, and the same applies to a team's fitness for a challenge. The skill cost is the measure of how much work the team needs to do to meet the required skill levels and be considered a strong candidate for the challenge

```python
def calculate_skill_cost(skill_levels):
    return sum([max(0, 3 - skill_levels[skill]) for skill in skill_levels])
```

The fitness_function is responsible for evaluating the fitness of a given team based on two important factors: communication cost and skill cost.

The communication cost is calculated based on the communication network within the team, and the skill cost is calculated based on the skill levels of team members in the required skills.

To calculate the communication cost, the function calls calculate_communication_cost, which determines how well team members can communicate with each other based on their assigned roles and responsibilities.

To calculate the skill cost, the function first calls get_highest_skill_levels to determine the highest skill level in each required skill area among team members. Then, it calculates the cost of each skill by subtracting the highest skill level from 3 (the maximum skill level), and adding up all the resulting costs.

Finally, the fitness_function combines the communication cost and skill cost to create an overall fitness score for the team. The lower the score, the better the fitness of the team.

```python
def fitness_function(team, communication_costs, required_skills):
    skill_levels = get_highest_skill_levels(team, required_skills)
    communication_cost = calculate_communication_cost(team, communication_costs)
    skill_cost = calculate_skill_cost(skill_levels)
    fitness = communication_cost + skill_cost
    return fitness
```

This function is all about selection, but not just any kind of selection, it's tournament selection! It's like a battle royale for individuals in the population, where only the fittest survive. The tournament size is set to 3, which means that each round of the tournament pits three individuals against each other.

The goal is to select the k fittest individuals from the population to be the parents of the next generation. So, in each round of the tournament, we randomly select three individuals from the population and have them compete against each other. The winner, aka the fittest individual, is then selected to move on to the next round.

We repeat this process until we have selected k individuals to be the parents. By doing this, we ensure that the parents for the next generation are not just the fittest individuals, but also diverse in terms of their characteristics. This diversity is important for the genetic algorithm to explore the solution space and find the best possible solution.

So, let the tournament begin! May the fittest individuals survive and pave the way for the next generation.

```python
def selection(population, k):
    # Select k individuals from the population using tournament selection
    tournament_size = 3
    selected = []
    for i in range(k):
        tournament = random.sample(population, tournament_size)
        winner = min(tournament, key=lambda x: x["fitness"])
        selected.append(winner)
    return selected
```

Suppose you have two parents who are both great at some things, but could be better if they shared their talents. Well, that's where crossover comes in. We take a random point in their "genetic code" and swap the parts after that point, essentially creating two "offspring" that inherit some of the strengths of each parent.

```python
def crossover(parent1, parent2):
    # Perform single-point crossover and return two offspring
    crossover_point = random.randint(1, len(parent1) - 1)
    offspring1 = parent1[:crossover_point] + parent2[crossover_point:]
    offspring2 = parent2[:crossover_point] + parent1[crossover_point:]
    return offspring1, offspring2
```

The weighted_score function takes into account three important factors: the number of missing skills, the overlap of current skills, and the overlap of current availability. It assigns a score to each potential team member based on these factors, giving you a more objective way to evaluate their suitability for the project.

First, the function calculates a missing score that ranges from 0 to 1. This score represents how many of the required skills are missing from the potential team member's skill set. If a member has all the required skills, their missing score will be 1, meaning they are a perfect fit for the project.

Next, the function looks at how many of the member's skills overlap with the current team's skills. This factor is important because you want team members who can collaborate effectively and build on each other's strengths. The function calculates the overlap score by taking the number of common skills between the member and the current team, and dividing it by the number of missing skills. The higher the overlap score, the more likely the member will be a valuable addition to the team.

Finally, the function takes into account the member's availability. After all, having the right skills is one thing, but you also need team members who can commit to the project and meet the necessary deadlines. The function calculates the availability score by taking the number of common available time slots between the member and the current team, and dividing it by the member's total available time slots. The higher the availability score, the more likely the member will be able to work with the team effectively.

By using the weighted scoring system provided by this function, you can more objectively evaluate potential team members based on their skills, availability, and compatibility with the existing team.

```python
def weighted_score(member, current_levels, current_skills, required_skills_fraction, current_availability, missing_skills):
    missing_count = len(missing_skills)
    if missing_count == 0:
        missing_score = 1
    else:
        missing_score = 1 / (1 + missing_count)
    return (
        missing_score,
        len(set(member['skill']).intersection(current_skills)) / missing_count,
        len(set(member['availability']).intersection(current_availability)) / len(member['availability'])
    )
```

So you have a team, but you feel like it needs a little something extra. That's where mutation comes in - it's like adding a pinch of spice to your team recipe.

First, we take a look at the team's current skills, levels, languages, communication styles, and availability. Then we figure out which required skills are missing - no one likes a team with holes in it.

Next, we sift through eligible members, excluding those who have the same skill and level 3 as an existing team member (because we don't want too much of a good thing). We filter eligible members by their missing skills, language, communication style, availability, and whether they've already been assigned to the team.

Once we have our eligible members, we score them based on how well their skills, availability, and communication style match the team's needs. We sort them in descending order by their score, and then add them to the team one by one until all required skills are filled or we run out of eligible members.

```python
def mutation(team, eligible_members, required_skills, assigned_members):
    # Check if there are eligible members to add to the team
    if not eligible_members:
        return team

    # Get the current skills and levels in the team
    current_skills = {member['skill'] for member in team if member['level'] < 3}
    current_levels = [member['level'] for member in team]
    current_languages = {member['language'] for member in team}
    current_communication_styles = {member['communication_style'] for member in team}
    current_availability = set.intersection(*[set(member['availability']) for member in team])

    # Find the required skills that are missing in the team
    missing_skills = set(required_skills) - current_skills

    # Compute the fraction of required skills that match the member's skills
    required_skills_fraction = len(missing_skills) / len(required_skills)

    # Exclude eligible members who have the same skill and level 3 as an existing member in the team
    eligible_members = [member for member in eligible_members if not (member['skill'] in current_skills and member['level'] == 3)]

    # Filter eligible members to those who have at least one missing skill and who match the team's language,
    # communication style, and availability, and who have not already been assigned to the team
    eligible_members = [member for member in eligible_members if set(member['skill']).intersection(missing_skills) and
                          member['language'] in current_languages and member['communication_style'] in current_communication_styles and
                          set(member['availability']).issubset(current_availability) and member not in assigned_members]

    # Check if there are any eligible members with missing skills
    if not eligible_members:
        return team

    # Sort eligible members by descending order of their weighted score
    sorted_members = sorted(eligible_members, key=lambda member: weighted_score(member, current_levels, current_skills, required_skills_fraction, current_availability), reverse=True)

    # Add members to the team until all required skills are filled or there are no more eligible members
    for member in sorted_members:
        assigned_members.append(member)
        team.append(member)

        current_skills.add(member['skill'])
        missing_skills = set(required_skills) - current_skills
        if not missing_skills:
            break

    return team
```

```python
# Define the maximum number of teams to display for each challenge
max_display_teams = 3

def get_eligible_members(challenge, assigned_members, dataset):
    return [member for member in dataset if member["challenge"] == challenge
    and member["name"] not in assigned_members]
```

3.2.3 Main code:

Genetic algorithm function: Welcome to the genetic algorithm, where we will use evolution to find the best team for your challenge. First, we filter out the eligible members based on the challenge and those already assigned. We then initialize the communication costs and create the best individual with an empty team and infinite fitness.

Next, we generate an initial population with a specified size, and iterate over generations to improve the fitness of the teams. We select the parents for the next generation, generate offspring through crossover and mutation, and replace the least fit individuals in the population with the new offspring. We also update the best individual if necessary.

But, what's the fitness function? We use the required skills and communication costs to calculate the fitness of a team. The goal is to minimize the communication costs and maximize the number of required skills covered by the team.

Finally, we stop early if we have found a perfect solution, meaning all required skills are covered without any communication costs.

```python
def genetic_algorithm(challenge, team_size, population_size, max_iterations,
   assigned_members=set()):
    # Filter eligible members based on challenge and assigned members
    eligible_members, required_skills = filter_eligible_members(dataset,
   challenge, assigned_members)

    # Initialize communication costs
    communication_costs = initialize_communication_costs()

    # Initialize the best individual
    best_individual = {"team": [], "fitness": float("inf")}

    # Initialize the population
    population = initialize_population(population_size, eligible_members,
   team_size, communication_costs, required_skills)

    # Iterate over generations
    for i in range(max_iterations):
        # Select parents for the next generation
        parents = select_parents(population, population_size)

        # Generate offspring through crossover and mutation
        offspring = generate_offspring(parents, eligible_members,
   required_skills, assigned_members, communication_costs)

        # Replace the least fit individuals in the population with the new
   offspring
```

```
        population = replace_least_fit_individuals(population, population_size, offspring)

        # Update the best individual if necessary
        best_individual = update_best_individual(population, best_individual)

        # Stop early if we have found a perfect solution
        if best_individual["fitness"] == 7:
            break

    return best_individual
```

3.2.4 Output

In this new subchapter, we will be discussing the output of the team formation algorithm. Specifically, we will be looking at how the formed teams are displayed to the user. The output of the algorithm will provide a list of teams with their respective members, skills, and levels. We will also be discussing the different parameters used in the code, such as population size and max iterations, which impact the efficiency and accuracy of the algorithm.

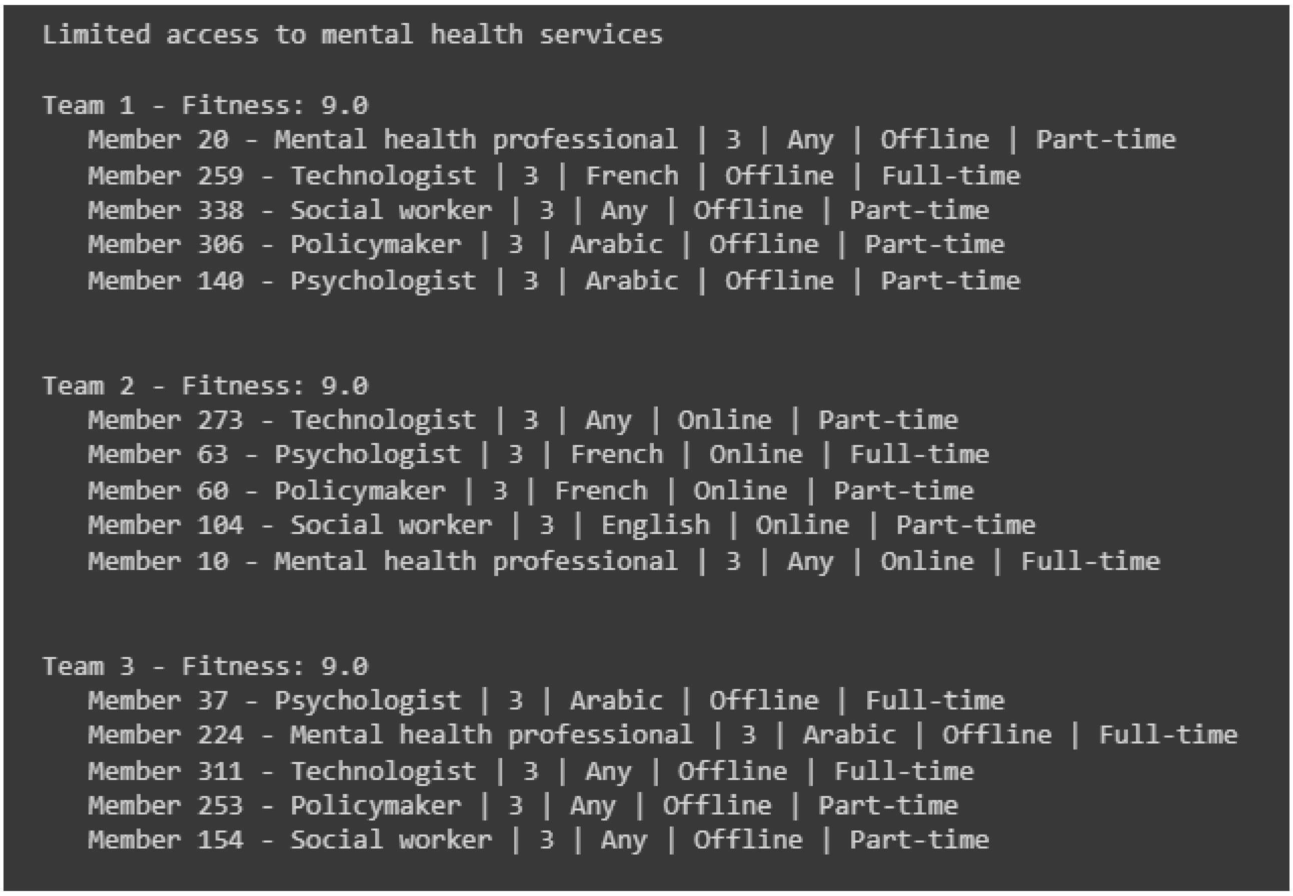
The main function in the output subchapter is "print_formed_teams", which takes the formed teams and the challenge as inputs and prints out the formed teams' details. Additionally, we will be discussing how the output of the algorithm can be improved further, such as incorporating graphical representations of the formed teams.

Before diving into the code and discussing the output function, we will first provide an overview of the team formation algorithm and the steps involved in it. After discussing the code, we will analyze the complexity of the algorithm and its implications. Finally, we will conclude with a discussion on the effectiveness of the algorithm and how it can be further improved.

With this introduction in mind, let's now take a closer look at the code block provided for the output subchapter. The code is responsible for generating teams for a given challenge and displaying the top 'max_display_teams' teams formed. The algorithm used to generate these teams is a genetic algorithm, which we will discuss in detail in the subsequent sections.

```python



def print_formed_teams(challenge, formed_teams):
    print(challenges_with_titles[challenge])
    for i in range(min(max_display_teams, len(formed_teams))):
        team = formed_teams[i]["team"]
        fitness = formed_teams[i]["fitness"]
        print("\nTeam", i+1, "- Fitness:", fitness)
        for member in team:
            print("  ", member["name"], "-", member["skill"], "|", member["level"], "|", member["language"], "|", member["communication_style"], "|", member["availability"])
        print()

if __name__ == ’__main__’:
    population_size = 92
    max_iterations = 512

    # Initialize assigned members
    assigned_members = []

    # Iterate over each challenge
    for challenge in challenges:
        # Get the required skills for the challenge
        required_skills = challenges[challenge]
        min_team_size = len(required_skills)
        max_team_size = len(required_skills)+1
        team_size = random.randint(min_team_size, max_team_size)

        # Get the eligible members for this challenge
        eligible_members = get_eligible_members(challenge, assigned_members, dataset)

        # Initialize a list to store the formed teams for this challenge
        formed_teams = []

        # Iterate over a range of iterations
        for i in range(max_display_teams):
            # Generate a team using the genetic algorithm
            team = genetic_algorithm(challenge, team_size, population_size, max_iterations, assigned_members)

            # Add the team to the list of formed teams
            formed_teams.append(team)

            # Add the names of the assigned members to the assigned_members list
            team_members = team[’team’]
            for member in team_members:
                assigned_members.append(member[’name’])

        # Sort the formed teams by fitness
        formed_teams = sorted(formed_teams, key=lambda x: x["fitness"])

        # Print the top ’max_display_teams’ formed teams for this challenge
        print_formed_teams(challenge, formed_teams)
```

The output shows the formed teams for each challenge, along with the fitness score for each team. Each team is listed with its members' names, skills, language, communication style, and availability. The teams are ranked in order of their fitness score, with the fittest team appearing first. In this case, the fitness score is a measure of how well the team members' skills match the required skills for the given challenge. The output also indicates whether each member is available online or offline, and whether they work full-time or part-time. The number of displayed teams is limited to a specified maximum number, which is set to six in this code. Overall, the output provides valuable information for team formation and can help identify the best team composition for each challenge.



```
Limited access to mental health services

Team 1 - Fitness: 9.0
   Member 20 - Mental health professional | 3 | Any | Offline | Part-time
   Member 259 - Technologist | 3 | French | Offline | Full-time
   Member 338 - Social worker | 3 | Any | Offline | Part-time
   Member 306 - Policymaker | 3 | Arabic | Offline | Part-time
   Member 140 - Psychologist | 3 | Arabic | Offline | Part-time


Team 2 - Fitness: 9.0
   Member 273 - Technologist | 3 | Any | Online | Part-time
   Member 63 - Psychologist | 3 | French | Online | Full-time
   Member 60 - Policymaker | 3 | French | Online | Part-time
   Member 104 - Social worker | 3 | English | Online | Part-time
   Member 10 - Mental health professional | 3 | Any | Online | Full-time


Team 3 - Fitness: 9.0
   Member 37 - Psychologist | 3 | Arabic | Offline | Full-time
   Member 224 - Mental health professional | 3 | Arabic | Offline | Full-time
   Member 311 - Technologist | 3 | Any | Offline | Full-time
   Member 253 - Policymaker | 3 | Any | Offline | Part-time
   Member 154 - Social worker | 3 | Any | Offline | Part-time
```

Figure 3.2:

Chapter 4

Conclusion

Throughout our graduation thesis, we have explored seven real-world challenges, including the limited access to mental health services. We applied a genetic algorithm approach to solve the Multidimensional Team Formation Problem (MTFP) in order to create teams of experts that can effectively address these challenges.

Our algorithm identified the optimal combination of team members based on their skills, availability, and language preferences. We used real data from Aplox, an innovative platform that connects skilled individuals from different communities and investors to solve real-world challenges by creating promising startups. By using Aplox's mobile app, users can easily identify the skills required to solve specific challenges and crowdsource people willing to solve those challenges, using our team-making algorithm to create successful teams.

In addition to team formation, Aplox mentors also provide valuable expertise and knowledge to help startups succeed, and investors who put their money into promising startups have access to the knowledge they have accumulated to develop their own business. Our target audience is people who want to create a startup but are struggling to find the right skills and investors who are looking to put their money into promising startups. Aplox is the ideal solution to create successful startups in a short time and with a dream team.

Overall, our genetic algorithm approach successfully solved the MTFP and created teams that can tackle real-world challenges. We believe that Aplox has the potential to revolutionize the startup world by connecting skilled individuals with investors and providing the necessary resources to succeed. With Aplox, anyone can turn their innovative ideas into successful businesses.

Bibliography

[1] Department of Psychology Steve W.J. Kozlowski. Enhancing the effectiveness of work groups and teams. 2006.

[2] Pablo Ballesteros-Pérez Jimmy H. Gutiérrez. The multiple team formation problem using sociometry. 2016.

[3] Aziz Abdul Izzatdin Son Tung Ngo, Jafreezal Jaafar. Some metaheuristic algorithms for solving multiple cross-functional team selection problems. 2021.

[4] Alireza Farasat A.G. Nikolaev. Social structure optimization in team formation. 2016.

[5] Yang Yu Jing Yang. Analysis on communication cost and team performance in team formation problem. 2018.

[6] Eduard Tulp Laurent Siklóssy. The space reduction method: a method to reduce the size of search spaces. 2002.

[7] Guido Van Rossum. The history of python :"a brief timeline of python". 20 January 2009.