OPTIMIZATION ALGORITHMS

PROF. LEONARDO VANNESCHI
PROF. DAVIDE FARINATI
PROF. LIAH ROSENFELD

ALEXANDRA PINTO - 20211599
FRANCISCO FARINHA -20211550
ILONA NACU - 20211602
JOÃO BARRADAS - 20211590
RAFAEL PROENÇA - 20211681

1. Introduction

This report's purpose is to present the findings of our study, which served as a useful evaluation of our expertise in optimization algorithms. We had the chance to demonstrate our research skills and problem-solving prowess while also applying the theoretical principles we had studied in both practical and theoretical lectures. Our project concentrated on the application and improvement of Genetic Algorithms (GAs), specifically investigating various Genetic Operators to resolve a specified optimization Problem (OP) associated with route planning for an event run by NOVA IMS.

2. Objective

This project's main goal was to improve our comprehension of optimization algorithms, with a focus on genetic algorithms in particular. We entered into this project with the following goals in mind:

a. Application of theoretical ideas: To implement Genetic Operators efficiently, use the knowledge we have obtained from our theoretical and practical classes on optimization algorithms.
b. Enhance our ability to solve complex optimization issues by using a variety of genetic operators and taking into account the limits imposed by the given OP.
c. Investigate the literature already written about Genetic Operators, pick out any pertinent approaches, and incorporate them into the planning and execution of our project.
d. Evaluation of model performance: Using a given data set and a thorough grid search that includes at least 15 runs, build a final model and evaluate its performance by contrasting it with models created by other project groups.
e. Use good project management techniques, abide by the project rules, and provide a well-structured project report that clearly describes our methodology, implementation specifics, and results as examples of excellent project management.

By achieving these goals, we aimed to improve our coding and implementation, demonstrate our ability to apply theoretical knowledge to practical situations and extend our understanding of optimization algorithms and genetic operators.

3. Genetic Algorithm

To find the best answers within a large search area, evolutionary computation techniques called genetic algorithms are applied.

Genetic Algorithms (GAs) use solutions that are created at random and repeatedly enhance their quality through processes like selection, variation (crossover and mutation), and survival choice. GAs imitate natural selection, reproduction, hereditaries, variety, and competition in the spirit of Darwin's theory of evolution. While genetic operators alter an individual's structure independent of fitness, selection algorithms act depending on fitness.

We noted that our problem was quite similar to the Travelling Salesman Problem, so we based ourselves on it to do our research. In the TSP we have a set of cities and the distances between each other. The objective is to find the shortest route in which all cities are visited and in the end return to the first city. It is quite obvious the similarities between both problems, while in the TSP there are cities in our problem we have NOVA IMS rooms, and while in the first problem, there are distances between the cities in our problem we have the focus loss and both problems would require a minimization algorithm. Another difference is that in our case we do not need to return to the first room To solve the presented problem, we researched the TSP and we used the slides lectured in class, to comprehend some concepts better.

3.1. Define an Individual

In genetic algorithms, an individual is a potential solution or candidate for a given problem. Each individual is part of the population and is encoded as a chromosome or a set of hyperparameters. An

individual represents a potential arrangement of genes or variables that can be evaluated and optimized. Its fitness, determined by its characteristics, influences its role in the evolutionary process.

In the problem we are working on, each individual is a list with a specific path representation. That path is represented by numbers that correspond to letters (0 =A, 1=B, ..., 7=H).

When creating the individual, `create_indiv()`, we need to make sure that the last value of the list is 7, since H is always the last room, and shuffle the first seven rooms, A to G (0 to 6).

```
individual = list(range(0, 7))
# Shuffle the first 7 rooms (Room H needs to be the last one, so we do not shuffle it)
random.shuffle(individual)
individual.append(7)  #Appending 7 (room H) at the end
```

Figure 1.

An important condition that we have to be sure happens is that if room B is seen right after room F, the C is always optional. So we created an if condition that is checking if room B is seen immediately after room F in a sequence represented by the list "individual." If that condition is satisfied, the code proceeds to set room C as optional by replacing the element with a value of 99, randomly. 99 is chosen arbitrarily, it seemed like an extreme enough number, but it could have been anything besides 0 to 7.

```
if individual.index(1) == individual.index(5)+1:
    # Set room C as optional
    if random.choice([True, False]):
        individual[individual.index(2)] = 99
```

Figure 2.

We also create a definition `valid_indiv()` where we analyze if the individual respects all the restrictions.

1st - Check if each room is visited only once:

```
for i in [0,1,3,4,5,6,7]:
    if i not in individual:
        return False
```

Figure 3.

2nd - There are no repeated individuals:

```
if len(set(individual)) != len(individual):
    return False
```

Figure 4.

3rd - The length of each individual must be equal to 8:

```
if len(individual) != 8:
    return False
```

Figure 5.

4th - Room A (0) cannot be placed before F:

```
if individual.index(0) > individual.index(5):
    return False
```

Figure 6.

5th - Check if Room H is the last room:

```
if individual[-1] != 7:
    return False
```

Figure 7.

Each individual must have fitness, in our case, the focus loss, and our goal is to minimize it, in other words, the less focus loss the better. In a simple way, the focus loss is calculated by subtracting the focus loss in another room from the room that was before. In this case, we need to be careful since when C is replaced by 99, this means that this room will be "ignored" and the calculation of the focus loss is between the room that is before 99 and after 99.

```
for room in range(len(individual) - 1):
    room1 = individual[room]       # Defining the f
    room2 = individual[room + 1]   # Getting the ne
    if room1 == 99:  # If room1 is 99, it means C i
        continue     # so we move to the next itera

    if room2 == 99:   # If the second room is 99, m
        if room + 2 < len(individual):    # Checkin
            room2 = individual[room + 2]  # Updatin
        else:
            break   # Breaking out of the loop beca

    fitness_score += focus_loss[room1][room2]   # S
```

Figure 8.

Also, if the individual is not valid (in `valid_indiv()` is returning False) then an absurd fitness is attributed. This is a way to store the individuals but since their fitness is so high the individual will not be selected in every case. We attributed an absurd value because this was one of the options discussed in the theoretical classes about genetic algorithm.

```
if not valid_indiv(individual):
    fitness_score = 150
```

Figure 9.

After setting the individual criteria we are now able to create the population and its fitness.

3.2. Creation of the Population

When creating a population we start by deciding the population size. Next, we randomly generate individuals that will each represent a potential solution to the problem. Then we evaluate the fitness of each individual and measure how well they perform, in order to rank the individuals. We repeat this process of generating individuals and evaluating their fitness until we are happy with the population size. By doing this we are able to create a diverse population with several potential solutions, while also identifying good candidates for future evolution.

In our GA, we created a def named `create_pop(population_size)`, where population_size is an integer, representing the size of the population. And start to check if the individual is created (calling `create_indiv()`) and also if each individual in the population is valid (calling `valid_indiv()`).

```
while len(population) < population_size:
    individual = create_indiv()
    if valid_indiv(individual):    # Confirm
        population.append(individual)
```

Figure 10.

Finally, we create `calculate_pop_fit(focus_loss)`, where the parameter focus_loss is a list with our data, that has the losses of focus from room to room and this def will calculate the fitness of each individual in the population. We decided to use nested functions to avoid passing focus loss as a parameter for the GA.

```
def inner_calculate_pop_fit(population):

    fitness_scores = []   # Initializing hte list that will store the fitness
    for individual in population:
        fitness_score = calculate_individual_fitness(individual, focus_loss)
        fitness_scores.append(fitness_score)   # Appending each score to the
    return fitness_scores
```

Figure 11.

3.3. Performing Roulette Wheel Selection

Roulette wheel selection is a method used to choose individuals for reproduction based on their fitness. Fitness values are normalized and converted into probabilities. A random number determines which individual is selected, with higher fitness values increasing the chances. This process is repeated to select multiple individuals. Roulette wheel selection balances favoring higher fitness while maintaining diversity in the population to prevent premature convergence. [1]

Our genetic algorithm was inspired by the “The Travelling Salesman Problem”, we found that the most used methods of selection in this problem are the Tournament Selection and the Roulette Wheel Selection, in our project we ended up deciding to use the Roulette Wheel Selection as it was the one that was most fit to our code. So we define a function that performs a roulette wheel selection named `roulette_selection(population, fits)` with the hyperparameters population and fits. The population is a matrix composed of individuals from a population and fits is a list with values of fitness from all individuals in the population.

```
sum_of_fits = sum(fits)

probabilities = [1 - fitness / sum_of_fits for fitness in fits]

return random.choices(population, weights=probabilities)[0]
```

Figure 12.

3.4. Performing Crossover

Crossover is an operator that combines genetic information from two parents, in order to create new offspring.

It works by selecting parents based on their fitness, choosing a crossover point along their chromosomes, exchanging genetic material beyond that point, and generating two offspring, training genetic material from both parents. These offspring can then replace some individuals in the existing population. As in mutation, there is a chance of crossover occurring or not, but while the chance of mutation happening is usually really low, the chance of crossover happening is usually as it is true in nature the chance of the sons to inherit its parent's traits is usually really big.

This operator helps explore new regions of the search space by combining traits from different parents, creating diversity, and improving the algorithm's ability to find optimal solutions.

After doing some research we decided to use three crossover methods that are commonly used for the “Traveling Salesman Problem” and that we figured would fit well in our code. These are the Improved Cycle Crossover, the Partially Mapped Crossover, and the Order Crossover. We will now describe these methods below. [2]

When room C is not entered, we substitute the value 2 which usually represents room C for the value 99, with this we found a problem when one parent had the value 2 and the other had 99. In order to solve this problem we created a function called `absent_room_case(p1)` where when the first parent has the value 2 and the second parent has the value 99, the value of the first parent will be replaced by 99. For example, if the first parent is equal to [2, 1, 3, 4, 6, 0, 5, 7] and this condition applies after this operation then it will be represented by [99, 1, 3, 4, 6, 0, 5, 7]. On the other hand, if the first parent has the value 99 and the second parent has the value 2, the value of the first parent will be replaced by 2. Following the same example, if the first parent is equal to [99, 1, 3, 4, 6, 0, 5, 7] and this condition applies after this operation then it will be represented by [2, 1, 3, 4, 6, 0, 5, 7].

```
for room in range(len(p1)):
    if p1[room] == 2:
        p1[p1.index(2)] = 99
    elif p1[room] == 99:
        p1[p1.index(99)] = 2
return p1
```

Figure 13.

In the project description one of the rules is that the last room to be visited must always be room H, in order to accomplish this when we generate new offspring in each crossover we append the room H, assigned as the number 7 in the end of the list. We can see this in Figure 14.

```
offspring1.append(7)
offspring2.append(7)
```

Figure 14.

3.4.1. Improved Cycle Crossover

Improved Cycle Crossover is an advanced variant in genetic algorithms for solving permutation-based optimization problems.

This operator preserves beneficial cycles from parents, accurately places remaining elements, and fosters exploration of the search space. It identifies cycles through element mapping, transfers them faithfully to offspring, and maps unmapped elements between parents. Improved Cycle Crossover overcomes limitations of Cycle Crossover and enhances the exchange of genetic traits. 3

In the example below, the improved cycle crossover is applied to parents [4, 6, 0, 5, 3, 2, 1] and [99, 0, 5, 1, 4, 6, 3]. The resulting offspring are as follows:

1. Offspring 1: [99, 6, 0, 5, 1, 3, 4, 7]
    a. The improved cycle crossover begins by identifying cycles between the parents. The first cycle starts with the value 4 from parent1 and follows the corresponding values in parent2: 4 -> 99 -> 4. This cycle is completed.
    b. The next cycle starts with the value 6 from parent1 and follows the corresponding values in parent2: 6 -> 0 -> 6. This cycle is completed.
    c. The remaining values are copied from parent2 to offspring1.
    d. The resulting offspring1 is [99, 6, 0, 5, 1, 3, 4, 7].
2. Offspring 2: [4, 3, 1, 5, 0, 6, 2, 7]
    a. The improved cycle crossover starts by finding cycles between the parents. The first cycle begins with the value 2 from parent1 and follows the corresponding values in parent2: 2 -> 4 -> 6 -> 2.s cycle is completed.
    b. The next cycle starts with the value 0 from parent1 and follows the corresponding values in parent2: 0 -> 5 -> 3 -> 0. This cycle is completed.
    c. The remaining values are copied from parent2 to offspring2.
    d. The resulting offspring2 is [4, 3, 1, 5, 0, 6, 2, 7].

```
Original parents [4, 6, 0, 5, 3, 2, 1] [99, 0, 5, 1, 4, 6, 3]
This is the parent1 in Improved Cycle Crossover: [4, 6, 0, 5, 3, 99, 1]
This is the parent2 in Improved Cycle Crossover: [99, 0, 5, 1, 4, 6, 3]
Original parents [99, 0, 5, 1, 4, 6, 3] [4, 6, 0, 5, 3, 2, 1]
This is the parent1 in Improved Cycle Crossover: [2, 0, 5, 1, 4, 6, 3]
This is the parent2 in Improved Cycle Crossover: [4, 6, 0, 5, 3, 2, 1]
The offspring of Improved Cycle Crossover are:  ([99, 6, 0, 5, 1, 3, 4, 7], [4, 3, 1, 5, 0, 6, 2, 7])
```

Figure 15.

3.4.2. Partially Mapped Crossover (PMX)

Partially Mapped Crossover (PMX) is a genetic algorithm technique that selects a crossover point and swaps positions between the crossover points based on the second parent. To maintain validity, cities in these positions are swapped, not overwritten. PMX can be generalized to n-point crossover. A new technique is introduced to handle permutation encoding, ensuring valid offspring without requiring special operator definitions. Conventional bit-string crossover and mutation operators are adequate for this robust representation. 4 5

Now we will proceed to explain the example of PMX, in Figure 16.

1. The parents are [4, 6, 0, 5, 3, 2, 1] and [99, 0, 5, 1, 4, 6, 3].
2. Parent 1 is modified based on the rule regarding room C, which was already explained, resulting in [4, 6, 0, 5, 3, 2, 1].
3. Crossover points are determined. In this case, the crossover starts and end points are at index 2 and 4, respectively.
4. The offspring is initialized replacing the values outside the crossover range, with “None”, the offspring becomes [None, None, 0, 5, None, None, None]. The remaining bits from Parent 2 are [99, 1, 4, 6, 3].
5. Parent 1 is modified again, following the rule about room C, resulting in [2, 0, 5, 1, 4, 6, 3].
6. Crossover points are determined once more, remaining the same.
7. The offspring is updated, becoming [None, None, 5, 1, None, None, None] and the remaining bits from Parent 2 are [4, 6, 0, 3, 2].
8. The offspring of the Partially Mapped Crossover are [6, 1, 0, 5, 99, 3, 4, 7] and [3, 2, 5, 1, 0, 4, 6, 7]. Number 7 is appended at the end of the list.

```
The parents are:  [4, 6, 0, 5, 3, 2, 1] [99, 0, 5, 1, 4, 6, 3]
Parent 1, after change: [4, 6, 0, 5, 3, 99, 1]
Crossover start, crossover end: 2 4
Offspring after copying values from parent1:  [None, None, 0, 5, None, None, None]
Remaining bits: [99, 1, 4, 6, 3]
Parent 1, after change: [2, 0, 5, 1, 4, 6, 3]
Crossover start, crossover end: 2 4
Offspring after copying values from parent1:  [None, None, 5, 1, None, None, None]
Remaining bits: [4, 6, 0, 3, 2]
The offspring of Partially Mapped Crossover are:  ([6, 1, 0, 5, 99, 3, 4, 7], [3, 2, 5, 1, 0, 4, 6, 7])
```

Figure 16.

3.4.3. Ordered Crossover

Ordered Crossover, a genetic operator, partitions parent chromosomes using two random crossover points. It preserves the order of elements in the left and right sections while determining the middle section based on the problem's specific ordering. This method is commonly used in problems that require maintaining a specific order, such as U-shaped assembly line balancing. 2 6

We will now explain an example of Order Crossover. It can be seen below the explanation.

1. In this example the parents are [4, 6, 0, 5, 3, 2, 1] and [99, 0, 5, 1, 4, 6, 3].
2. In this example parent1 has the value 2 and parent2 has the value 99, both values correspond to the letter C so when this happens we have to replace the value 2 by 99 or the value 99 by 2. With this, parent1 is modified.
3. The crossover points retrieved from the first parent are randomly assigned.
4. The crossover points defined in point 3. form a section that will be directly inserted in offspring.
5. Now we’ll get the bits of the other parent that do not appear in offspring. Those bits will be ordered from left to right but starting in the index next to the most-right crossover point index.
6. With this, the bits of 5. are inserted on the right side of offspring, respecting its length, so the bits that are left to be inserted are added to the left side of offspring.
7. This process is repeated one more time, but with parents in switched places, creating the second offspring
8. Both offspring are retrieved.

```
Parent 1 and 2 are : [4, 6, 0, 5, 3, 2, 1] [99, 0, 5, 1, 4, 6, 3]
Parent 1: [4, 6, 0, 5, 3, 99, 1]
Parent 2: [99, 0, 5, 1, 4, 6, 3]
```

Figure 17.

3.4.4. Slide Crossover

Now we will pass to the “original” crossovers. An important note to Slide Crossover and Fog crossover is that we cannot have guarantees that these crossovers are completely new and original, so far, we did not find any crossover with this code.

Let’s divide the Slide Crossover into some steps:

1. Select a random breaking point, K, between 0 and the second-to-last index.
2. Slide the first K numbers of the first parent(parent1) to the end, maintaining their relative order.
3. The resulting segment becomes the initial segment of the offspring.
4. Look at the numbers that are not in the offspring and determine their order in the second parent (parent2).
5. Replace the corresponding positions of these numbers in the offspring.
6. The last number (H) is always fixed and remains unchanged in the offspring

In summary, the Slide Crossover involves sliding a portion of the first parent and combining it with the order of the numbers from the second parent that are not present in the initial segment while preserving the last number of sequences.

We can see an example in the following piece of code:

```
Original parents [4, 6, 0, 5, 3, 2, 1] [99, 0, 5, 1, 4, 6, 3]
This is the parent1 in Slide Crossover: [4, 6, 0, 5, 3, 99, 1]
This is the parent2 in Slide Crossover: [99, 0, 5, 1, 4, 6, 3]
Break point: 2
Offspring: [None, None, None, None, None, 4, 6]
Remaining bits of parent2 in Slide Crossover: [99, 0, 5, 1, 3]
Original parents [99, 0, 5, 1, 4, 6, 3] [4, 6, 0, 5, 3, 2, 1]
This is the parent1 in Slide Crossover: [2, 0, 5, 1, 4, 6, 3]
This is the parent2 in Slide Crossover: [4, 6, 0, 5, 3, 2, 1]
Break point: 2
Offspring: [None, None, None, None, None, 2, 0]
Remaining bits of parent2 in Slide Crossover: [4, 6, 5, 3, 1]
```

Figure 18.

Final Offspring resulting from the parent1 and parent 2:

```
The offspring of Slide Crossover are:  ([99, 0, 5, 1, 3, 4, 6, 7], [4, 6, 5, 3, 1, 2, 0, 7])
```

Figure 19.

3.4.5. Fog Crossover

The second crossover created by us is named Fog Crossover. And works in the following steps:

1. Select a breaking point, K, in the range of 0 to the second-to-last index.
2. Erase the first K numbers from Parent 1.
3. Slide the remaining numbers one index to the left.
4. Determine the order of the numbers not present in the erased segment based on Parent 2.
5. Place these missing numbers in the offspring according to their order in Parent 2.

In summary, the Fog Crossover involves erasing a segment from Parent 1, sliding the remaining numbers, and determining the order of missing numbers based on Parent 2. And the same is appearing for the other offspring, erasing a segment from parent2, sliding the remaining numbers one to the left, and determining the order of missing numbers based on parent1.

The following output gives us an example of how Fog Crossover works:

```
Original parents [4, 6, 0, 5, 3, 2, 1] [99, 0, 5, 1, 4, 6, 3]
This is the parent1 in Fog Crossover: [4, 6, 0, 5, 3, 99, 1]
This is the parent2 in Fog Crossover: [99, 0, 5, 1, 4, 6, 3]
Break point: 6
Offspring: [None, None, None, None, None, None, None]
Remaining bits of parent2 in Fog Crossover: [99, 0, 5, 1, 4, 6, 3]
Original parents [99, 0, 5, 1, 4, 6, 3] [4, 6, 0, 5, 3, 2, 1]
This is the parent1 in Fog Crossover: [2, 0, 5, 1, 4, 6, 3]
This is the parent2 in Fog Crossover: [4, 6, 0, 5, 3, 2, 1]
Break point: 6
Offspring: [None, None, None, None, None, None, None]
Remaining bits of parent2 in Fog Crossover: [4, 6, 0, 5, 3, 2, 1]
```

Figure 20.

Final Offspring resulting from the parent1 and parent2:

```
The offspring of Fog Crossover are:  ([99, 0, 5, 1, 4, 6, 3, 7], [4, 6, 0, 5, 3, 2, 1, 7])
```

Figure 21.

3.5. Performing Mutation

Mutation in genetic algorithms is a process that introduces random modifications to individual chromosomes.

It involves selecting chromosomes at random and altering one or more genes within them. This introduces genetic diversity, which aids in avoiding local optima and discovering fresh solutions. The mutation rate is a parameter that defines the chance of occurring a mutation, usually, it is best to define a low mutation rate, as even in the animal world it is not common to occur a mutation.

Similarly to what has been done in crossovers the number 7 is not included in the mutation procedures and only appends to the final individual at the end of the procedure.

The balance between preserving favorable solutions and venturing into unexplored territory, mutation improves the effectiveness of genetic algorithms. Modified individuals may either replace or coexist with the original population.

3.5.1. Twors Mutation

Twors Mutation is a mutation operator that randomly selects two genes within an individual's chromosome and exchanges their positions. Twors Mutation prevents premature convergence and potentially uncovers improved solutions. [7]

In the example below (Figure 22.) we do the following:

1. Select two random genes for each parent (2 and 6 for parent1 | 2 and 4 for parent2);
2. Exchange the values of those genes (6 <> 2 | 4 <> 2);
3. Retrieve the final individual ([4, 6, 1, 5, 3, 2, 0] for parent1 | [99, 0, 4, 1, 5, 6, 3] for parent2).

```
The parents of the twors mutation are:
 [4, 6, 0, 5, 3, 2, 1]
 [99, 0, 5, 1, 4, 6, 3]
Position 1, Position 2:  0 5
Individual after swap: [2, 6, 0, 5, 3, 4, 1]
[2, 6, 0, 5, 3, 4, 1, 7]
Position 1, Position 2:  0 4
Individual after swap: [4, 0, 5, 1, 99, 6, 3]
[4, 0, 5, 1, 99, 6, 3, 7]
```

Figure 22.

3.5.2. Reverse Sequence Mutation

Reverse Sequence Mutation is a mutation operator in genetic algorithms. It randomly selects a section of an individual's chromosome and reverses the order of elements within that section. This results in random changes in a localized region, promoting diversity and exploring different permutations. It helps prevent premature convergence and with the discovery of new solutions. [7]

In Figure 23. we can see how this is applied in our project code:

1. We start by choosing two random genes (start and end position)(2 and 6 for parent1 | 5 and 6 for parent2);
2. Select the substring using start and end position ([0, 5, 3, 2, 1] for parent1 | [6, 3] for parent2);
3. Proceed to reverse the order of the selected substring ([1, 2, 3, 5, 0] for parent1 | [3, 6] for parent2);
4. Retrieve the final individual ([4, 6, 1, 2, 3, 5, 0] for parent1 | [99, 0, 5, 1, 4, 3, 6] for parent2).

```
The parents of the reverse sequence mutation are:
 [4, 6, 0, 5, 3, 2, 1]
 [99, 0, 5, 1, 4, 6, 3]
Start position, end position:  2 6
Reversed substring: [1, 2, 3, 5, 0]
Individual after reversing substring: [4, 6, 1, 2, 3, 5, 0]
[4, 6, 1, 2, 3, 5, 0, 7]
Start position, end position:  5 6
Reversed substring: [3, 6]
Individual after reversing substring: [99, 0, 5, 1, 4, 3, 6]
[99, 0, 5, 1, 4, 3, 6, 7]
```

Figure 23.

3.5.3. Partially Shuffle Mutation

Partially Shuffle Mutation is an operator commonly used in genetic algorithms. It works by randomly selecting a subset of chromosomal components from an individual and then shuffling them while leaving the others in the original place. The main objective is to discover different configurations and to avoid becoming stuck in local optima. [7]

In this example (Figure 24.) we do the following:

1. Randomly choose two genes (start and end position) like in Reverse Sequence Mutation(1 and 5 for parent1 | 0 and 4 for parent2);

2. Select the substring using the start and end position ([6, 0, 5, 3, 2] for parent1 | [99, 0, 5, 1, 4] for parent2);
3. Shuffle the substring, in other words randomly reorder the substring genes ([0, 2, 5, 6, 3] for parent1 | [0, 99, 5, 1, 4] for parent2);
4. Retrieve the final individual ([4, 0, 2, 5, 6, 3, 1] for parent1 | [0, 99, 5, 1, 4, 6, 3] for parent2).

```
The parents of the partial shuffle mutation are:
 [4, 6, 0, 5, 3, 2, 1]
 [99, 0, 5, 1, 4, 6, 3]
Start position, end position:  1 5
Shuffled substring: [0, 2, 5, 6, 3]
Individual after mutation: [4, 0, 2, 5, 6, 3, 1]
[4, 0, 2, 5, 6, 3, 1, 7]
Start position, end position:  0 4
Shuffled substring: [0, 99, 5, 1, 4]
Individual after mutation: [0, 99, 5, 1, 4, 6, 3]
[0, 99, 5, 1, 4, 6, 3, 7]
```

Figure 24.

3.5.4. Inverted Exchange Mutation

Mutation Inverted X-change is a mutation operator utilized in genetic algorithms. It randomly selects two positions within an individual's chromosome and exchanges the elements between those positions in an inverted order. The primary aim is to prevent premature convergence and potentially uncover enhanced solutions. [8]

Now we will explain this mutation with an example that can be seen below:

1. Randomly choose two genes (start and end position) like in the last two mutation operators(0 and 5 for parent1 | 5 and 6 for parent2);
2. Select the substring using the start and end position ([4, 6, 0, 5, 3, 2] for parent1 | [6, 3] for parent2);
3. Proceed to invert the substring ([2, 3, 5, 0, 6, 4] for parent1 | [3, 6] for parent2);
4. Randomly select a gene (room) of the inverted substring (2 for parent1 | 3 for parent2);
5. Define the genes outside the inverted substring ([1] for parent1 | [99, 0, 5, 1, 3] for parent2);
6. If genes outside exist, select at random one of them (1 for parent1 | 5 for parent2);
7. Exchange selected genes in points 4. and 6. (2 <> 1 | 3 <> 5);
8. Retrieve the final individual ([1, 3, 5, 0, 6, 4, 2] for parent1 | [99, 0, 3, 1, 4, 5, 6] for parent2).

```
The parents of the inverted exchange mutation are:
 [4, 6, 0, 5, 3, 2, 1]
 [99, 0, 5, 1, 4, 6, 3]
Start position, end position:  0 5
Inverted substring: [2, 3, 5, 0, 6, 4]
Selected room inside the substring: 2
Rooms outside substring: [1]
Selected room outside substring: 1
Individual after mutation: [1, 3, 5, 0, 6, 4, 2]
[1, 3, 5, 0, 6, 4, 2, 7]
Start position, end position:  5 6
Inverted substring: [3, 6]
Selected room inside the substring: 3
Rooms outside substring: [99, 0, 5, 1, 4]
Selected room outside substring: 5
Individual after mutation: [99, 0, 3, 1, 4, 5, 6]
[99, 0, 3, 1, 4, 5, 6, 7]
```

Figure 25.

3.6. Algorithm

For the actual algorithm that performs all the necessary steps, we performed minor changes to the one given in the practical classes.

Hyperparameters and description	
`create_population`	Function that will create a population.
`evaluate_population`	That will return the fitness of a population.
`maximization`	Boolean value to indicate if it's a maximization (True) or minimization (False) problem.
`gens`	Number of generations for the algorithm.
`pop_size`	Number indicating the size of a population.
`selector`	Function to select individuals from the population.
`mutator`	Function that performs mutation on an individual.
`crossover_operator`	Function that performs crossover on two parents.
`p_c`	Number between 0 and 1 that indicates the probability of crossover happening.
`p_m`	Number between 0 and 1 that indicates the probability of crossover happening.
`elitism`	Boolean that indicates if there should be elitism in the algorithm.
`verbose`	Indicates if there should be information printed (True), or not (False), on each generation and final solution.
`log`	Indicates if the values at each generation should be stored in a CSV file.

`path`	Path of the location of the CSV file that will store fitness values of each generation.
`seed=None`	Default: None. If indicated, it will help set the random.seed value, so that results can be replicated.
`return_convergence=False`	Default: False. Indicates if fitness values at each iteration should be returned. They can later be used for convergence plots (more about them in 3.7. Grid Search).

The algorithm starts by creating an initial population and evaluating its fitness. Then begins a loop that will be repeated for all generations: create an empty population (that will be the offspring population), and until the new population has the same size as the old one, we repeatedly choose individuals from the population (two parents) and apply crossover operators and mutation operators, then insert them in the new population. After we have a new population of the same size as the old one, we replace the old one with the new one. When elitism is set to true, and in our case it always is, because we do not want to lose the best individuals from each generation, we can store them to the next one, guaranteeing that we keep the best individuals at each iteration.

We also see, if verbose is set to True, what the minimum values of fitness are at each generation and the final best solution.

We changed a few things in the algorithm, mainly adding a seed, an option to return fitness values at each iteration (return_convergence), and another print, at the end, that sums up what the final solution was, including the individual itself. The seed was for us to be able to replicate the results if needed, especially when performing Grid Search. The return_convergence option was added for us to be able to make convergence plots and be able to compare which operator combination worked best. As for the added print, it was in order to easily see, each time the algorithm ran, what the final solution was.

3.7. Grid Search

Grid Search is a method that, when provided with a set of hyperparameters, searches exhaustively for the best combination, according to some criteria. It is clear why this is useful for our problem because testing each combination manually is not practical and we, most likely, would not remember to test every possible one.

We performed 3 grid searches, each for different purposes. Only one of them was the official, proper, Grid Search of 15 runs. The other two were variations created for plotting purposes.

3.7.1 Simple Grid Search

First, we executed a Grid Search of 15 runs, with the most appropriate hyperparameters. Even though most articles perform 30, due to time constraints. In terms of what criteria we used for it to give us the best solution, we chose fitness values and execution time, as, due to the reduced size of our problem, most combinations would likely reach the desired fitness, but some will do it faster than others. For each run of the grid search, we provided the algorithm with a new dataset, to make sure we got robust enough results. At each run we stored the best solution in a dictionary and at the end we compared all the results, in order to see which hyperparameters appeared more often.

These are the hyperparameters we used in figure 26.

```
pop_size_values = [50, 100]
p_m_values = [0.1, 0.2, 0.35, 0.8, 0.9]
p_c_values = [0.1, 0.2, 0.35, 0.8, 0.9]
crossover_operators = [pmx_crossover, improved_cycle_crossover, ordered_crossover, fog_crossover, slide_crossover]
mutation_operators = [inverted_exchange_mutation, inversion_mutation, reverse_sequence_mutation,
                      partial_shuffle_mutation]
gens_values = [100, 150, 200]
```

Figure 26.

We did not add all the values we wanted for the p_m and p_c, because otherwise it would have taken too long and unfortunately we could not risk it.

Run	Pop_size	P_m	P_c	Crossover	Mutation	Gen_values	Fitness	Execution Time
1	50	0.1	0.35	Slide	Reverse Sequence	100	16.7	0.1411
2	50	0.1	0.8	Slide	Reverse Sequence	100	14.0	0.1409
3	50	0.2	0.35	Slide	Partial Shuffle	100	26.1	0.1405
4	50	0.8	0.8	Slide	Reverse Sequence	100	28.9	0.1096
5	50	0.1	0.9	Improved Cycle	Reverse Sequence	100	27.9	0.0812
6	50	0.2	0.1	Slide	Partial Shuffle	100	33.8	0.0941
7	50	0.1	0.9	Fog	Reverse Sequence	100	14.2	0.0931
8	50	0.1	0.35	Slide	Partial Shuffle	100	22.9	0.0942
9	50	0.1	0.9	Fog	Inverted Exchange	100	27.0	0.0942
10	50	0.1	0.35	Improved Cycle	Twors	100	30.5	0.0932
11	50	0.1	0.9	Fog	Inverted Exchange	100	28.4	0.0786
12	50	0.1	0.1	Ordered	Partial Shuffle	100	26.6	0.0940
13	50	0.1	0.35	Slide	Reverse Sequence	100	23.7	0.0940
14	50	0.1	0.2	Slide	Partial Shuffle	100	31.6	0.0912
15	50	0.2	0.9	Slide	Reverse Sequence	100	19.3	0.0940

Analyzing the results of the Grid Search we can easily conclude that the best hyperparameters are the Slide Crossover that is the method of crossover that appears the most, a total amount of nine appearances. The best mutation method is the Reverse Sequence Mutation, which appears seven times.

However, there are two combinations that appear to be the best, the combination of Slide Crossover with Partial Shuffle Mutation and the combination of Slide Crossover and Reverse Sequence Mutation. The first combination appears four times and the second five times. In order to decide which combination is the best we will use the Boxplots and the Convergence Plots.

3.7.2. Boxplots

After obtaining the solutions of the first grid search, we moved on to plots, to confirm our results. So, we fixed the number of population size, generations, and probability of crossovers and mutators, and only tested the combination of operators. We did that with boxplots and convergence plots, to see which

provided the more consistent good results and converged first, and confirmed if the results aligned with the first Grid Search performed.

We chose this approach in three steps because performing a grid search of 15 runs and plotting everything for every combination would take too long and the results would be very hard to interpret, as the plots would be too full of information.

The boxplots were produced using a loop similar to the first grid search, but with some fixed hyperparameters, and at the end it stored each combination of hyperparameters and their respective result at each iteration (Figure 27). And the execution time was not used either, as what we want to explore is fitness and how consistent are the results of each combination.

```
param_combination = (pop_size, p_m, p_c, crossover_op, mutation_op, gens)
if param_combination not in fitness_scores_dict:   # Checking if the combination is already in the dictionary
    fitness_scores_dict[param_combination] = []
fitness_scores_dict[param_combination].append(min_fitness)  # Appending the values to the dictionary
```

Figure 27.

After the loop concluded, for the 15 datasets, the median value for each combination of hyperparameters was used for the boxplot. We did not use the mean, as it is too susceptible for outliers.

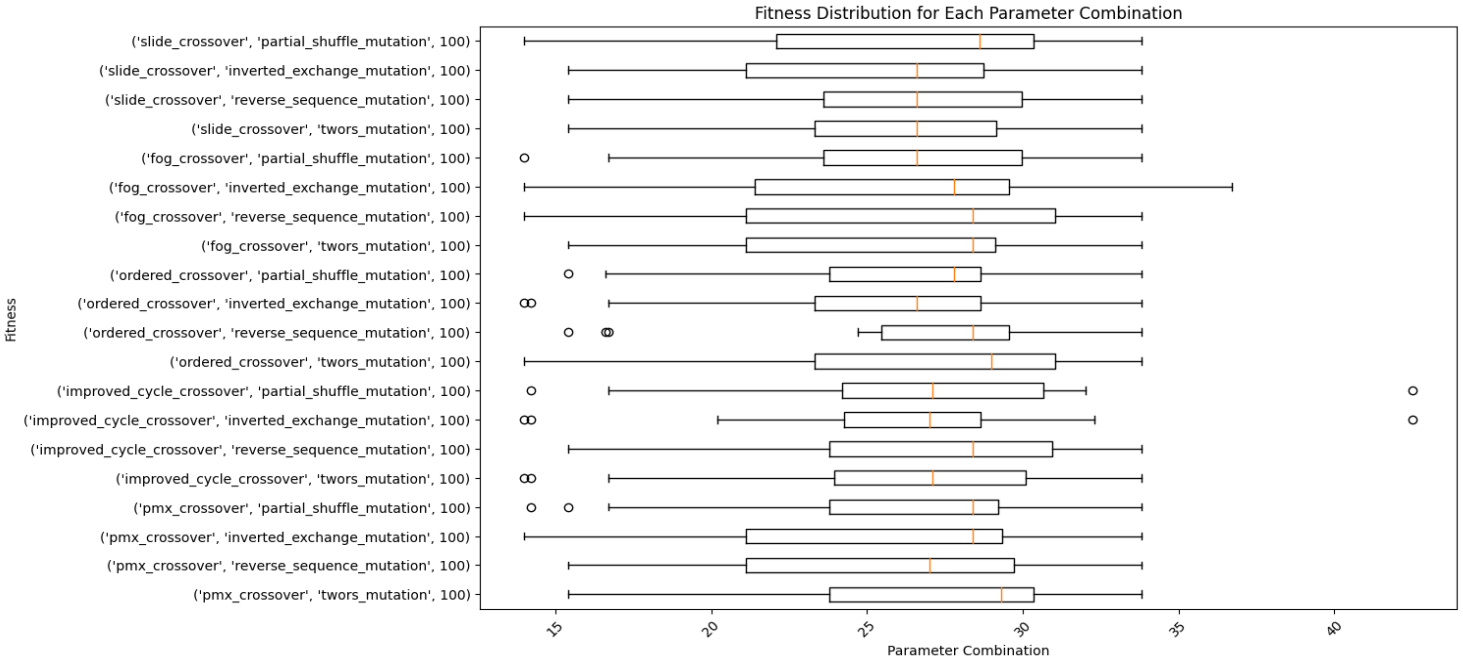


Figure 28.

We did Boxplots for one hundred generations as suggested in Grid Search, but we saw that there existed lots of outliers and we weren't able to get too low fitnesses. So we decided to try out 200 generations, which gave us many more consistent results. (Figure 29.)

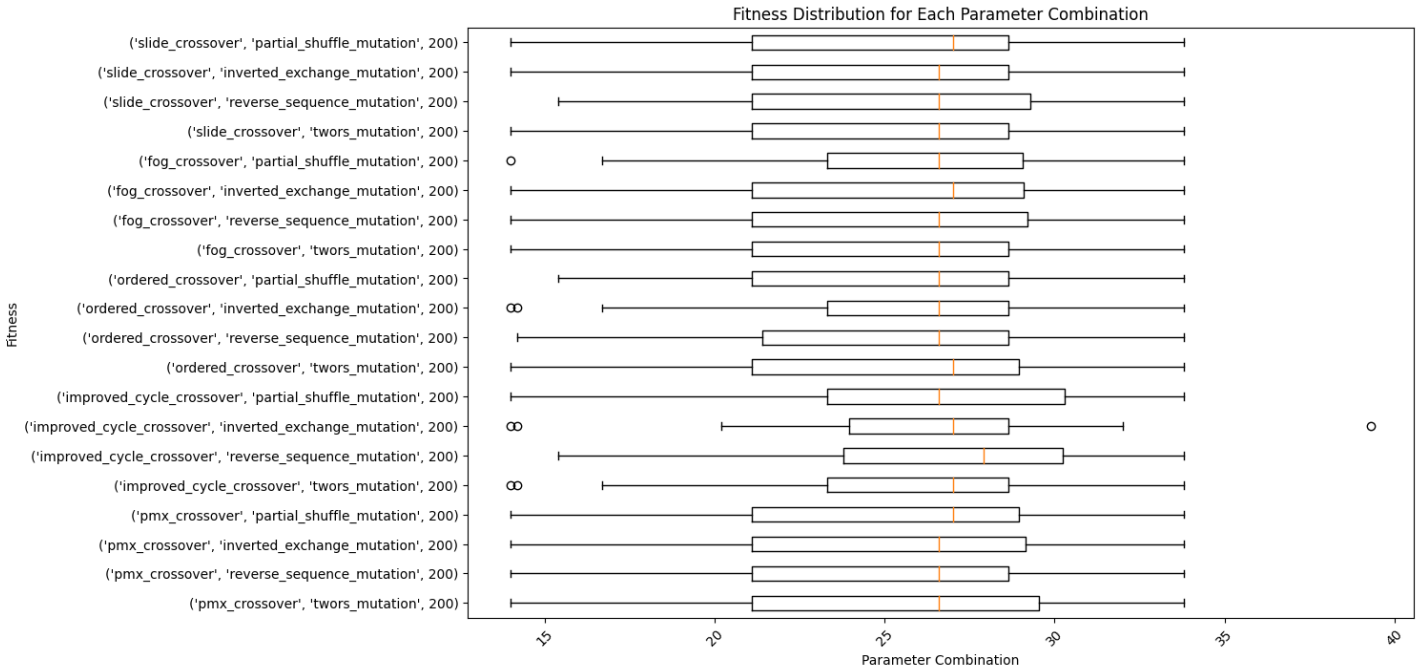


Figure 29.

To identify the most consistent pairs, it is important to note that we exclude combinations that exhibit outliers, with the lowest range of fitness and also those that have a higher range in boxplots besides the 1st quartile and the 3rd quartile being, if the boxplot is a big one. By disregarding these outliers, we focus on pairs that demonstrate greater overall consistency in performance. Based on the BoxPlots in Figure 29., the most consistent pairs (Crossover, Mutation) are:

- Slide Crossover and Partial Shuffle Mutation;
- Slide Crossover and Inverted Exchange Mutation;
- Slide Crossover and Twors Mutation;
- Fog Crossover and Twors Mutation;
- Orderer Crossover and Twors Mutation;
- Partially Mapped Crossover (PMX) and Reverse Sequence Mutation.

3.7.3. Convergence Plots

As for convergence plots, our approach was slightly different. Instead of using 15 datasets, we only used one, but to make sure that our results took into account randomness, we still performed each combination of hyperparameters 15 times, but we removed the seed. We are aware that this means that the results cannot be replicated, but in this case, it made more sense to have some randomness to it, so that we got results that are more likely to reflect the actual behavior. Only one dataset was used in this case, so we are conscient that the results could be different for different ones, but these convergence plots are more for exploration of the crossover and mutation operators we chose, and confirming results that we obtained by the other exhaustive searches.

To be able to obtain these plots, we used a loop, where we stored each convergence list in convergence_list, for each run, and then, we stored that list of lists in a variable called list_for_plots. This variable is composed of lists, but it is easier to comprehend its format with its shape: [20, 15, 200].

So, with a pointer, increased at each iteration, we were able to access all the convergence values obtained, for all 15 runs, and perform the median for each generation, after transposing the corresponding

lists for each combination. We needed to transpose them because what we want is the median for each generation, not the median for each run of the algorithm.

```
transposed_lists = zip(*list_for_plots[pointer])

medians = [statistics.median(elements) for elements in transposed_lists]
```

Figure 30.

Although we could have done everything in the first two loops since this is a more confusing line of thought, we preferred to have this separated into two parts, the first to store the values, and the second one to plot, as it helped us understand how the code reached its desired results. The end results are 20 convergence plots. Figures 30 to 36 are the combinations that are also present in the first grid search (3.7.1. Simple Grid Search).

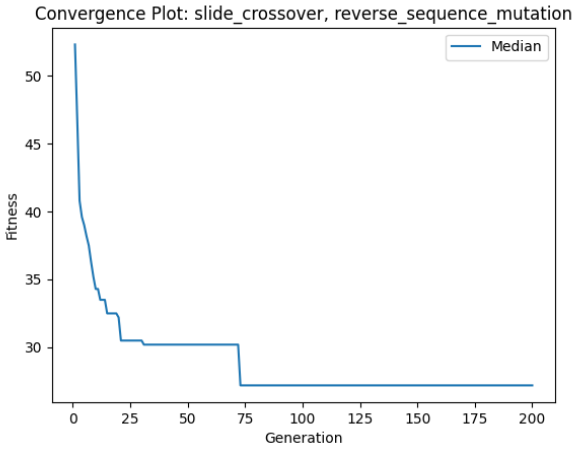


Figure 31.

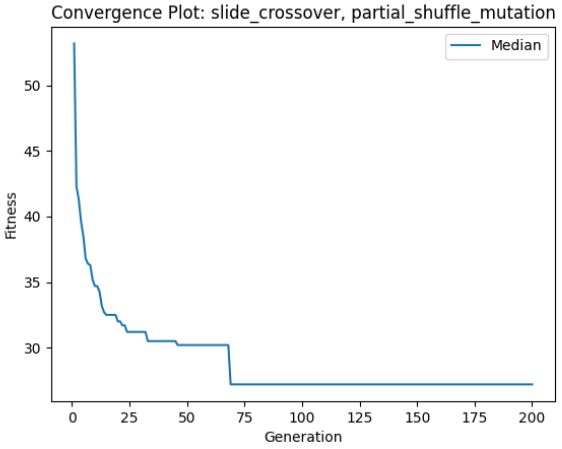


Figure 32.

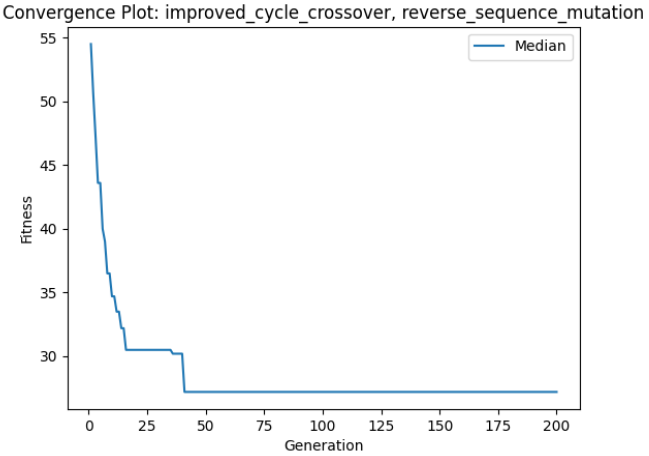


Figure 33.

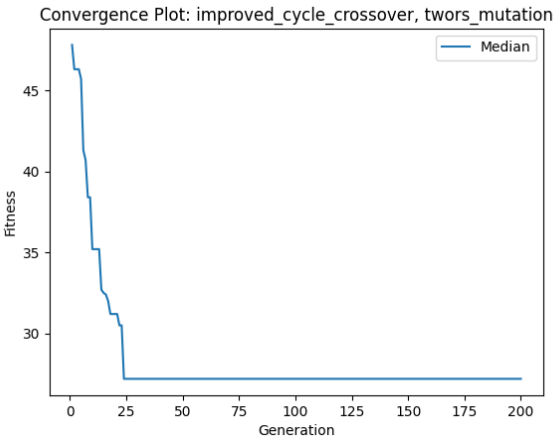


Figure 34.

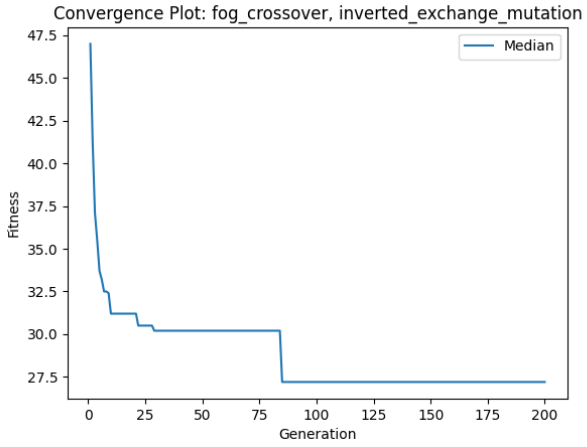


Figure 35.

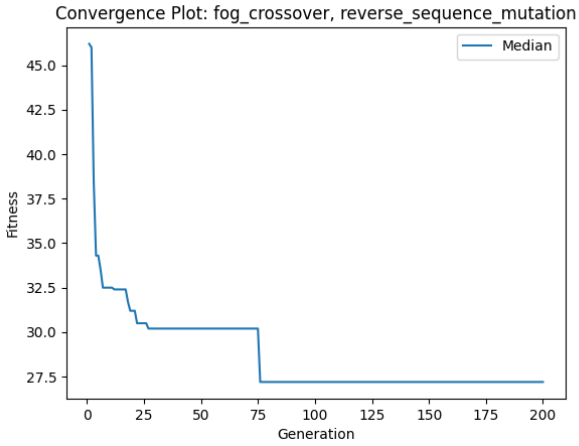


Figure 36.

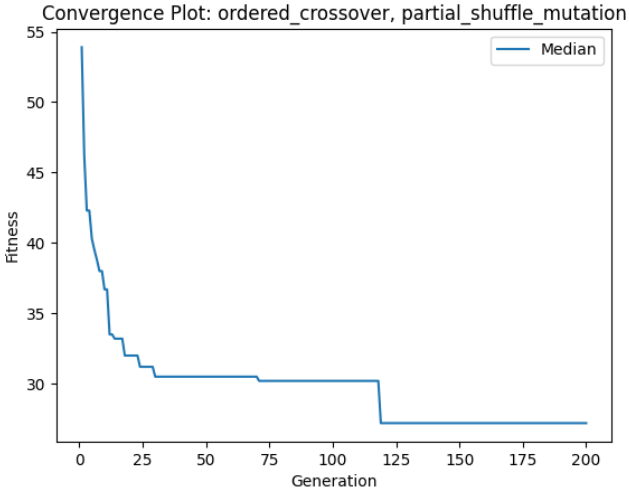


Figure 37.

In the convergence plots, we opted to use 200 generations instead of 100. This decision was based on the observation that when using 200 generations in the box plot analysis, we obtained more consistent results. We also tested in the main file, with the dataset provided in the project description file, the combinations with 100 and 200 generations, and we obtained better results. Therefore, to maintain consistency, we decided to utilize 200 generations in the convergence plots as well.

Upon analyzing the convergence plots displayed above (only considering the ones that showcased the highest performances in the first grid search, but the rest are provided in the annex), it is evident that the combination of the Improved_Cycle_Crossover and Twors_Mutation operators exhibits the most promising performance for 200 generations. This particular combination demonstrates the ability to achieve the optimal solution stability at a rapid pace, typically around the 25th generation, showcasing the quickest decay to the lowest value.

3.7.4. Best Hyperparameters

The *pop_size*, *p_m* and *p_c* were decided through the results presented in 3.7.1. Grid Search. Based on the conclusions mentioned in 3.7.2. the *gen_values* is set to 200. Considering 200 *gen_values* then the pair (Improved_Cycle_Crossover, Twors_Mutation) is the best one, by taking into account the 3.7.1. Grid Search and 3.7.3 Convergence Plots.

After interpreting the results, our main file uses the following hyperparameters:

	Pop_size	P_m	P_c*	Crossover	Mutation	Gen_values
Final	50	0.1	0.9	Improved_Cycle_Croer	Twors_Mutation	200

P_c*- After the grid search there was a tie between the values 0.9 and 0.35 for the probability of crossover, both appeared four times. Even though we ended up deciding tep a crossover rate = 0.9. We decided to keep this value as in most articles and in the classes it is usually recommended to have a crossover rate close to one and a mutation rate close to zero, so we figured it would make the most sense to keep a crossover rate equal to 0.9.

According to the dataset given in class the solution for our problem is [A, E, C, F, B, G, D, H].

```
Final solution: [0, 4, 2, 5, 1, 6, 3, 7]
Best fitness: 27.2
```

Figure 38.

4. Conclusion

In this project, our objective was to create an efficient genetic algorithm that could reach the best possible solutions. During the development of this work, we had the opportunity to improve our knowledge and skills in the creation and manipulation of genetic algorithms. In order to create new methods of crossover and mutation we also had to be creative and think outside the box. It was also a project that allowed us to improve our research abilities and teamwork skills.

To accomplish this project, it was essential to create a genetic algorithm that could explore multiple individuals across several generations to discover the best possible solution, minimizing focus loss. The use of various crossover and mutation methods was also crucial, allowing for the exploration of a wide diversity of individuals, which was a key factor in finding a good solution.

5. References

1- Johnson, Becky. “Genetic Algorithms: The Travelling Salesman Problem | by Becky Johnson.” *Medium*, 15 November 2017, https://medium.com/@becmjo/genetic-algorithms-and-the-travelling-salesman-problem-d10d1daf96a1. Accessed 26 May 2023.

2- Malmedal, Wenche. “Genetic Algorithm for Traveling Salesman Problem with Modified Cycle Crossover Operator.” *Hindawi*, https://www.hindawi.com/journals/cin/2017/7430125/. Accessed 26 May 2023.

3- “An Improved Genetic Algorithm Crossover Operator for Traveling Salesman Problem.” *DergiPark*, https://dergipark.org.tr/tr/download/article-file/534290. Accessed 26 May 2023.

4- *Genetic Algorithm Solution of the TSP Avoiding Special Crossover and Mutation*, https://user.ceng.metu.edu.tr/~ucoluk/research/publications/tspnew.pdf. Accessed 26 May 2023.

5- “Partially Mapped Crossover in Genetic Algorithms by Deeba Kannan.” *YouTube*, 24 July 2022, https://www.youtube.com/watch?v=EZg-l2FF-JM&t=205s. Accessed 26 May 2023.

6- “Ordered Crossover in Genetic Algorithm by Deeba Kannan.” *YouTube*, 24 July 2022, https://www.youtube.com/watch?v=7hDZyH2E4Yw. Accessed 26 May 2023.

7- Otman, Abdoun. “(PDF) Analyzing the Performance of Mutation Operators to Solve the Travelling Salesman Problem.” *ResearchGate*, https://www.researchgate.net/publication/221700559_Analyzing_the_Performance_of_Mutation_Operators_to_Solve_the_TravellingSalesman_Problem. Accessed 26 May 2023.

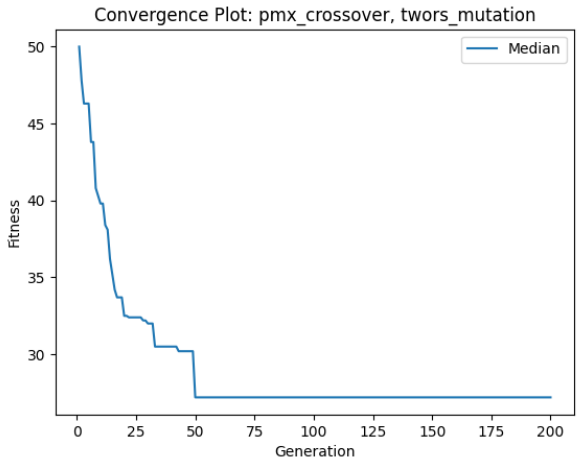
8- Darwin, Charles. “Combined Mutation Operators of Genetic Algorithm for the Travelling Salesman problem.” *Redalyc*, https://www.redalyc.org/pdf/2652/265219635002.pdf. Accessed 26 May 2023.

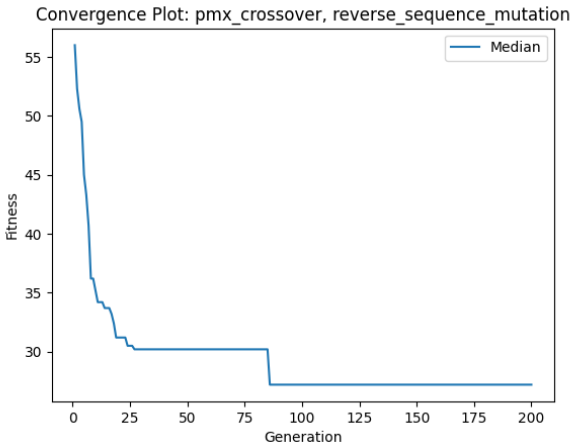
6. Annex

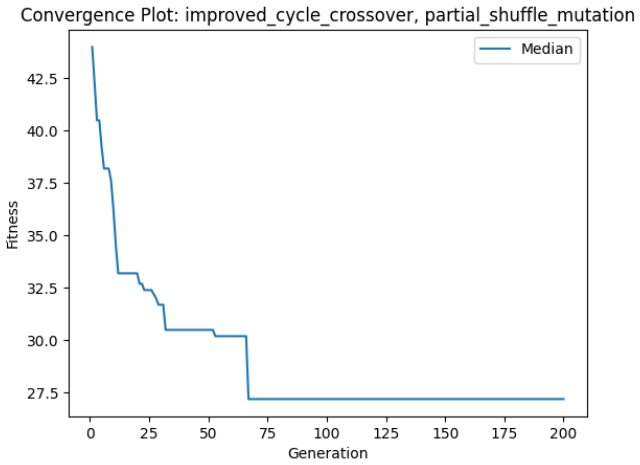
Grid Search output from 3.7.1. Simple Grid Search:

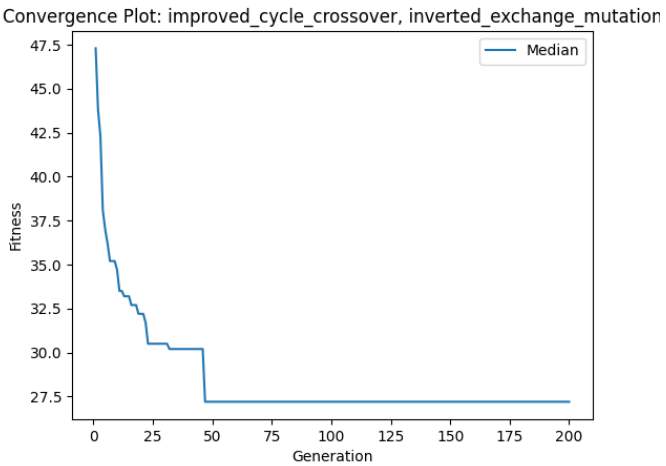
```
'''
Result of our Grid Search
exec_dict_final =
          {1: [(50, 0.1, 0.35, slide_crossover, partial_shuffle_mutation, 100), 16.7, [0, 1, 4, 6, 2, 3, 5, 7], 0.14105820655822754],
           2: [(50, 0.1, 0.8, slide_crossover, reverse_sequence_mutation, 100), 14.0, [4, 1, 0, 6, 5, 3, 2, 7], 0.14087557792663574],
           3: [(50, 0.2, 0.35, slide_crossover, reverse_sequence_mutation, 100), 26.1, [3, 4, 0, 1, 6, 5, 2, 7], 0.1405487060546875],
           4: [(50, 0.8, 0.8, slide_crossover, reverse_sequence_mutation, 100), 28.9, [1, 0, 3, 4, 6, 2, 5, 7], 0.10961651802062988],
           5: [(50, 0.1, 0.9, improved_cycle_crossover, reverse_sequence_mutation, 100), 27.9, [1, 0, 4, 2, 6, 5, 3, 7], 0.08118581771850586],
           6: [(50, 0.2, 0.1, slide_crossover, partial_shuffle_mutation, 100), 33.8, [0, 5, 1, 3, 2, 6, 4, 7], 0.09414649009704590],
           7: [(50, 0.1, 0.9, fog_crossover, reverse_sequence_mutation, 100), 14.2, [0, 4, 2, 5, 6, 1, 3, 7], 0.09306764602661133],
           8: [(50, 0.1, 0.35, slide_crossover, partial_shuffle_mutation, 100), 22.9, [1, 2, 6, 3, 0, 5, 4, 7], 0.09416723251342773],
           9: [(50, 0.1, 0.9, fog_crossover, inverted_exchange_mutation, 100), 27.0, [3, 0, 5, 1, 2, 6, 4, 7], 0.09417605400085449],
           10: [(50, 0.1, 0.35, improved_cycle_crossover, inversion_mutation, 100), 30.5, [0, 4, 5, 1, 2, 3, 6, 7], 0.09316897392272949],
           11: [(50, 0.1, 0.9, fog_crossover, inverted_exchange_mutation, 100), 28.4, [2, 6, 1, 4, 0, 3, 5, 7], 0.07859778404235840],
           12: [(50, 0.1, 0.1, ordered_crossover, partial_shuffle_mutation, 100), 26.6, [0, 4, 1, 3, 2, 6, 5, 7], 0.09399580955505371],
           13: [(50, 0.1, 0.35, slide_crossover, reverse_sequence_mutation, 100), 23.7, [3, 4, 1, 2, 6, 0, 5, 7], 0.09403085708618164],
           14: [(50, 0.1, 0.2, slide_crossover, partial_shuffle_mutation, 100), 31.6, [0, 6, 3, 2, 5, 1, 4, 7], 0.09116125106811523],
           15: [(50, 0.2, 0.9, slide_crossover, reverse_sequence_mutation, 100), 19.3, [2, 3, 1, 6, 0, 5, 4, 7], 0.09403753280639648]}
'''
```

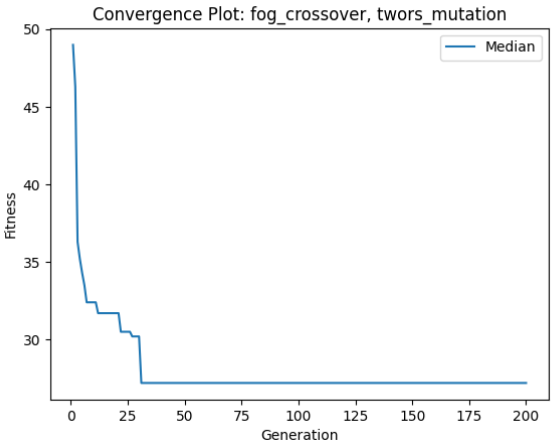
Convergence plots:

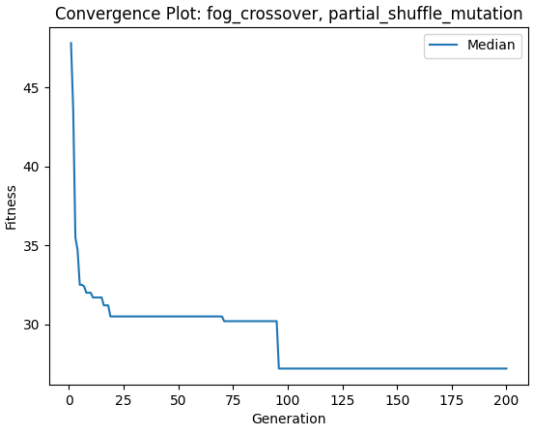


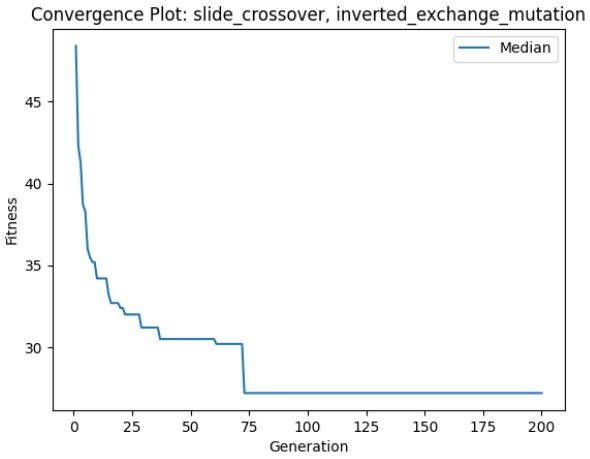







Convergence Plot: fog_crossover, twors_mutation
Median
Fitness
Generation
50
45
40
35
30
0
25
50
75
100
125
150
175
200


Convergence Plot: fog_crossover, partial_shuffle_mutation
Median
Fitness
Generation
45
40
35
30
0
25
50
75
100
125
150
175
200


Convergence Plot: slide_crossover, inverted_exchange_mutation
Median
Fitness
Generation
45
40
35
30
0
25
50
75
100
125
150
175
200