Final Project

Optimization Algorithms

Dinis Gaspar nº 20221869

Dinis Fernandes nº 20221848

Luis Mendes nº 20221949

Luis Soeiro nº20211536

Table of Contents

Introduction

The following report will describe the methodologies used to obtain the ideal path the player should take in *Hollow Night.* It will go over the chosen and developed genetic operators with the goal of finding the ideal path using Genetic Algorithms.

Firstly, we'll start with a general overview of the problem at hand, The goal is to find the best path for a player to take in the videogame *Hollow Night,* this is the path that maximizes Geo (the in-game currency) gained, this currency is accumulated by navigating through the different areas within the game but can be lost if the player's character dies. The paths must start and end in the same town ("Dirtmouth") and go through all the cities. The only exception is if the path includes Distant Village right after Queen's Station, in this case King's Station doesn't need to be visited if it implies a lower Geo gains/losses value. There are also some situations that make a path unviable such as if City Storerooms is visited right after Queen's Garden's or if the Resting Grounds are present in the first half of the journey.

Methodology

To implement a Genetic algorithm (GA) is we need to know: how to represent a solution, how to calculate the value of a solution, apply selection, crossover, and mutation operations, know the best probability of crossover, size of the population, number of generations and satisfy specific constraints (unique for each problem). So, in order to solve the problem, that is to obtain the ideal path the player should take in *Hollow Night'* we search for all these factors and found what we consider the best parameters. In the following pages we will explain the whole process.

General Functions:

To solve this problem using Genetic algorithms, firstly we need to establish the way to represent the paths as individuals. In our implementation, we represent them as lists of upper-case strings where each string represents the initials of the area it represents. These lists will be 11 elements long as there are 10 areas to visit, but obviously Dirthmouth appears at the start and at the end. The information containing the Geo gains/losses from moving areas is a Matrix (list of lists) where a line corresponds to the Geo earning of moving from that area to all other areas and a column the Geo earning of moving from all other areas to that area.

1. Constraints:

To manage the conditions explained in the introductory section of this report we created three functions to check constraints: Firstly, to handle the feasibility of individuals (viability of a path) we created two functions to check the unviability of a path, used later in this implementation to differentiate between feasible and infeasible solutions, further explanations later. Then, to handle the possibility of skipping King's Station, if the required condition is met the upper case 'KS' is replaced by a lower case 'ks', which in our implementation means that the Geo earning will be calculated using and not using the 'KS' in the path, choosing the one that maximizes the calculation.

2. Initialization:

In any GA process the first step is to create a population of randomly generated individuals. To this effect, we created two functions:

- The function to generate individuals is done by simply selecting areas at random without replacement from the 9 possible areas (Dirtmouth is excluded) until all areas have been selected. In our implementation, the constraint to allow for King's Station to be skipped is applied during the generation of the individual, thus if the individual meets the criteria it will already contain the lower case 'ks' upon creation. The individuals are returned in the format described above.
- The function to generate a population simply repeats this process for the specified number of iterations (population size). The returned population is a list of lists.

3. Fitness:

To calculate the fitness (Geo earning) of an individual we create a correspondence between the areas and their corresponding index in the Matrix (using a dictionary), then if 'ks' is in the solution, the fitness will be the maximum value of including KS and not including it, if is not we proceed to calculate the usual fitness of the solution. The fitness calculation is then split into two cases:

- For infeasible solutions they are given a fitness corresponding to the lowest Geo loss in the Matrix times 10, this ensures that they have the lowest possible fitness, as it is not possible to have a path with a lower geo loss. Then to ensure that all the fitnesses are positive we then sum the absolute value of that number back + 1. This is the theoretical approach, in practice all the fitnesses for infeasible solutions are simply defined as 1.
- For feasible solutions, we iterate through the individual until the penultimate element then we use the correspondence dictionary to index the Matrix, the line is the current area, and the column is the area in the next position. Then we sum the absolute value of the lowest possible value described above + 1 to ensure that the range of the fitnesses remains unchanged.

Within the fitness calculation there is also an option to apply fitness sharing to the fitness of feasible solutions, this is done by multiplying the fitness of an individual by the mean of its Hamming distance (number of different bits). This allows us to increase the fitness of individuals, which are, on average, distant from the remaining population, thus increasing the chance of finding global optimums (in theory), by spreading out the population. This can't be applied to infeasible solutions because, in theory, they could end up with higher fitnesses than some feasible solutions, which isn't desirable, that´s why the fitness for unfeasible solutions is 1. There is also an implementation using sum of hamming distance, this can't be used with Roulette selection as it results in the probabilities of infeasible solutions being selected being practically 0 which violates the principles of selection algorithms.

4. Genetic Algorithm:

The Genetic Algorithm function is derived from the one developed during the practical classes with the following changes:

- The Matrix of Geo gains must be a parameter of the algorithm.
- Probability of mutation is removed as the operators required for this type of problem don't use predetermined probabilities.
- Before the crossover section the individuals are transformed to be in upper case, essentially undoing the lower case 'ks' transformation, this is done to ensure that there is no possibility of KS appearing twice in the same individual, in its upper case and lower-case forms.
- Before adding offspring to the offspring population, they are again checked for the possibility of a 'KS' skip and transformed if needed.

Selection:

For this project we selected and prepared four selection algorithms, which are prepared to handle maximization as well as minimization, as this problem is a maximization one, the following explanations focus on the maximization portion of the algorithm:

1. Fitness Proportional (Roulette):

This method is probability based and thus higher fitness individuals have a higher probability of being selected, the probability of an individual being selected is calculated by dividing its fitness by the sum of all fitnesses. This along with our fitness calculation ensures that all individuals have a chance of being selected.

2. Rank Selection:

This method is also probability based and assigns a rank to all individuals based on their fitness then calculates the probabilities based on the ranks of individuals, where higher fitness individuals have higher ranks (larger numbers) leading to higher probabilities of being selected. This is essentially a variation of roulette selection where only the rank matters and not the actual fitness, this can be beneficial to a wider range of selections in cases where a small number of individuals have disproportionately high fitness when compared to the remaining population.

3. Tournament selection:

This method selects a certain number (pool size) of individuals (a pool) completely at random with replacement and then selects the one with the highest fitness. This ensures that all individuals can be selected because it is possible (although unlikely) that the entire pool is just copies of a single individual (i.e. the one with the worst fitness).

4. Self-Adaptive Tournament Selection:

This method is a variation of Tournament selection which considers the (normalized) standard deviation of the fitnesses within the population to manipulate the predetermined pool size; therefore, populations with higher standard deviations in fitness values will have larger pools and vice versa.

Crossovers:

Note: In our implementation, the necessary precautions have been taken to ensure that the first and last elements of an individual remain unchanged throughout the crossover process, thus those precautions won't be mentioned in the explanations. More technical explanations with examples can be found in the bibliography. The following explanations are quoted directly from 'Genetic algorithms for the traveling salesman problem' to ensure maximum clarity.

1. Partially Mapped Crossover (PMX):

This operator first randomly selects two cut points on both parents. In order to create an offspring, the substring between the two cut points in the first parent replaces the corresponding substring in the second parent. Then, the positions outside of the cut points are filled with the exact value in the parent if they are not in the cut point already, if they are, in order to eliminate duplicates and recover all cities, the value is the correspondence of the parent in the cut point. Example:

parent 1	:	**1**	**2**	|	**5**	**6**	**4**	|	**3**	**8**	**7**
parent 2	:	1	4	|	2	3	6	|	5	7	8
offspring											
(step 1)	:	1	4		**5**	**6**	**4**		5	7	8
(step 2)	:	1	3		**5**	**6**	**4**		2	7	8

2. Cycle Crossover:

The cycle crossover focuses on subsets of cities that occupy the same subset of positions in both parents. Then, these cities are copied from the first parent to the offspring (at the same positions), and the remaining positions are filled with the cities of the second parent. In this way, the position of each city is inherited from one of the two parents. However, many edges can be broken in the process because the initial subset of cities is not necessarily located at consecutive positions in the parent tours. Example:

parent 1	:	**1**	**3**	**5**	**6**	**4**	**2**	**8**	**7**
parent 2	:	1	4	2	3	6	5	7	8
offspring	:	1	**3**	2	**6**	**4**	5	7	8

3. Order-based crossover (OBX):

This crossover also focuses on the relative order of the cities on the parent chromosomes. First, a subset of cities is selected in the first parent. In the offspring, these cities appear in the same order as in the first parent, but at positions taken from the second parent. Then, the remaining positions are filled with the cities of the second parent. Example:

parent 1	:	**1**	**2**	**5**	**6**	**4**	**3**	**8**	**7**
parent 2	:	1	4	2	3	6	5	7	8
offspring	:	1	**5**	2	**4**	6	**3**	7	8

4. Position-based crossover (PBX):

Here, a subset of positions is selected in the first parent. Then, the cities found at these positions are copied to the offspring (at the same positions). The other positions are filled with the remaining cities, in the same order as in the second parent ensuring no repetition. Example:

parent 1	:	**1**	**2**	**5**	**6**	**4**	**3**	**8**	**7**
parent 2	:	1	4	2	3	6	5	7	8
offspring	:	1	2	**5**	6	**4**	**3**	7	8

5. Order crossover (OX):

In order to create an offspring, the string between the two cut points in the first parent is first copied to the offspring. Then, the remaining positions are filled by considering the sequence of cities in the second parent ensuring no repetition, starting after the second cut point (when the end of the chromosome is reached, the sequence continues at position 1). Example:

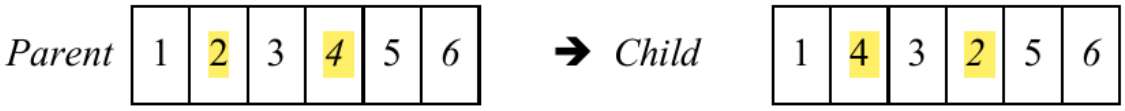
parent 1	:	**1**	**2**	**|**	**5**	**6**	**4**	**|**	**3**	**8**	**7**
parent 2	:	1	4	|	2	3	6	|	5	7	8
offspring											
(step 1)	:	–	–		**5**	**6**	**4**		–	–	–
(step 2)	:	2	3		**5**	**6**	**4**		7	8	1

Mutators:

Note: In our implementation, the necessary precautions have been taken to ensure that the first and last elements of an individual remain unchanged throughout the mutation process, thus those precautions won't be mentioned in the explanations. More technical explanations with examples can be found in the bibliography.

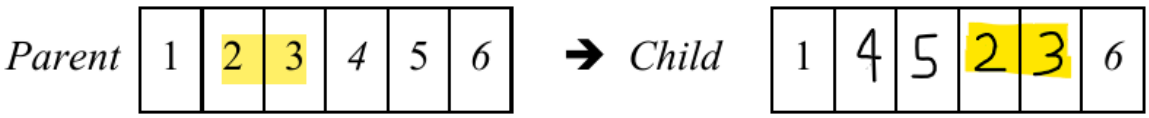
1. **Swap:**

This is the simplest operator we implemented, it randomly selects two positions within the individual and swaps the elements in those positions. Example:



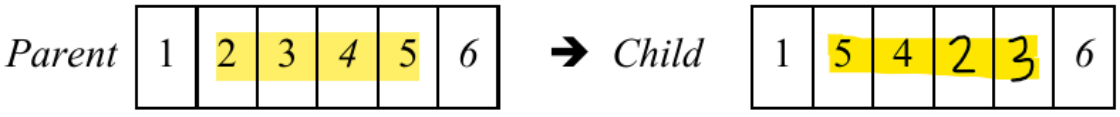
2. **Displacement:**

This operator selects a portion of the individual and moves it to a different position within it. In our implementation, it is possible to select the size of the section to be moved. Example: displacement of size 2.



3. **Scramble:**

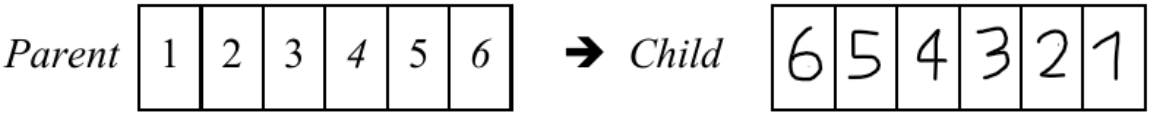
This operator randomly shuffles the elements within a certain section of the individual. In our implementation the size and location of this section are random. Example:



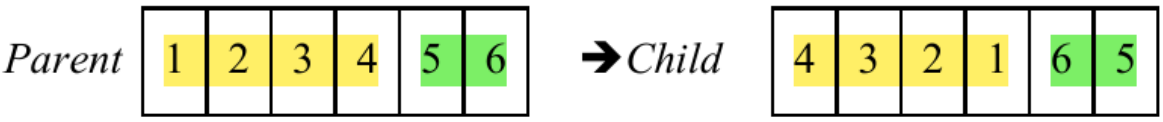
4. **Invert:**

This operator simply inverts the order of the elements in the individual. We have implemented two variants of this operator:

- **0 Point inversion:** This variant simply inverts the entire individual. Example:

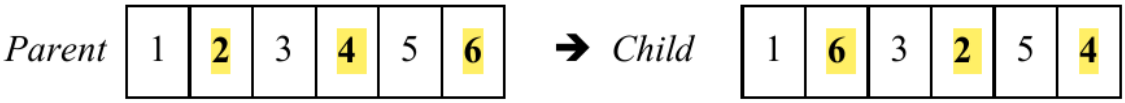


- **1 Point inversion:** This variant selects a random point to split the individual into two sections and then inverts those sections independently. Example:



5. **Thrors:**

This operator randomly selects a few positions within the individual and then "shifts" the elements in those positions to the next selected position (in the case of the last selected position the next position is the first position). In our implementation, it is possible to specify the number of positions to select. Example: Thrors of 3 positions.



Datasets:

When testing the best combination of parameters we will use two datasets, both respecting the condition that moving from Greenpath to Forgotten Crossroads (indexes 2,1 on the matrix) must be at least 3.2 % less than the minimum between all the other positive Geo gains.

- One is a fully random dataset, adapted to respect the condition mentioned above.
- The second dataset contains actual values from playing the game, gotten from a player, and tested to verify the condition mentioned above.

Searching for the optimal parameters:

Running a full grid search would take far too long (years to run), as we were considering 69.3 million combinations and because of this we ran our tests in a sequence where the best parameters from a test would be implemented into the evaluation of the next element. This process was done across a minimum of 15 different runs per unique combination to ensure robustness of the results.

So, what was done was: find the best parameters separately from less dependent to more dependent, reducing the complexity exponentially. Starting with choosing between using Elitism or not, then defining the population size and number of generations, after that we found the probability of crossover, then the best combination of fitness function and selection algorithm, and finally the combination of crossover and mutation operator. Using in each step the findings of the previous tests.

- Elitism:

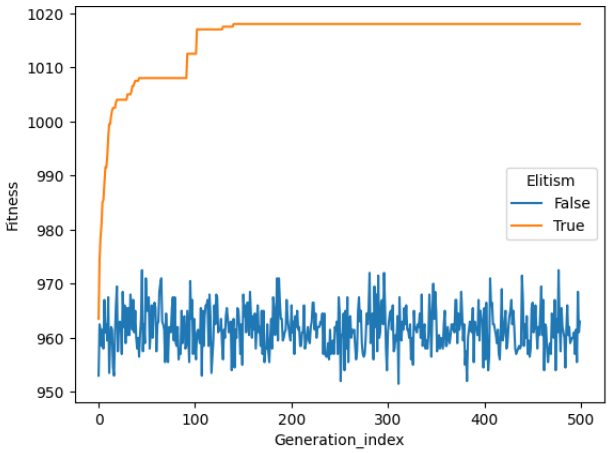


Figure 1 Testing Elitism

Conclusion: Using elitism is better. So, we will use it.

- Population Size and Number of Generations:

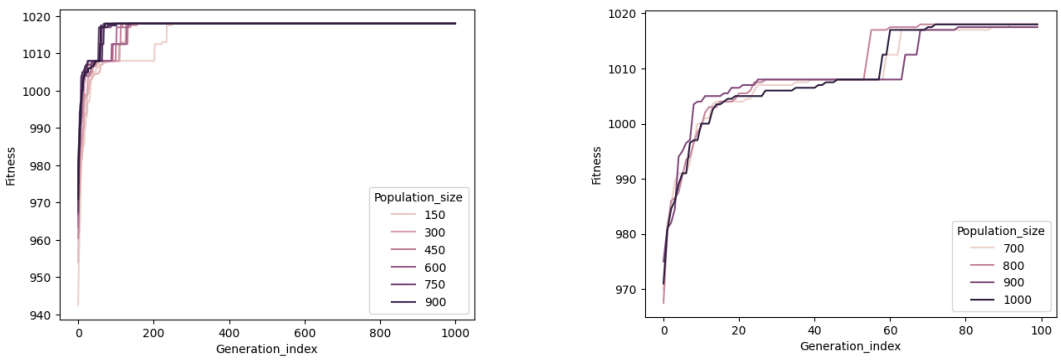


Figure 2 Testing Population and Generation Size

Conclusion:

Population Size: The larger the population size, the faster the maximum is reached, and the more solutions are tested. We will use population size = 800, because it gives good results and is as large as 900 or 1000 are.

Number of Generations: in 300 generations fitness stops improving (exploration phase), but we will use 500 generations to add some margin of error, since the convergence of the fitness will depend on the matrix of distance used, ensuring the existence of an exploitation phase.

- Probability of Crossover:

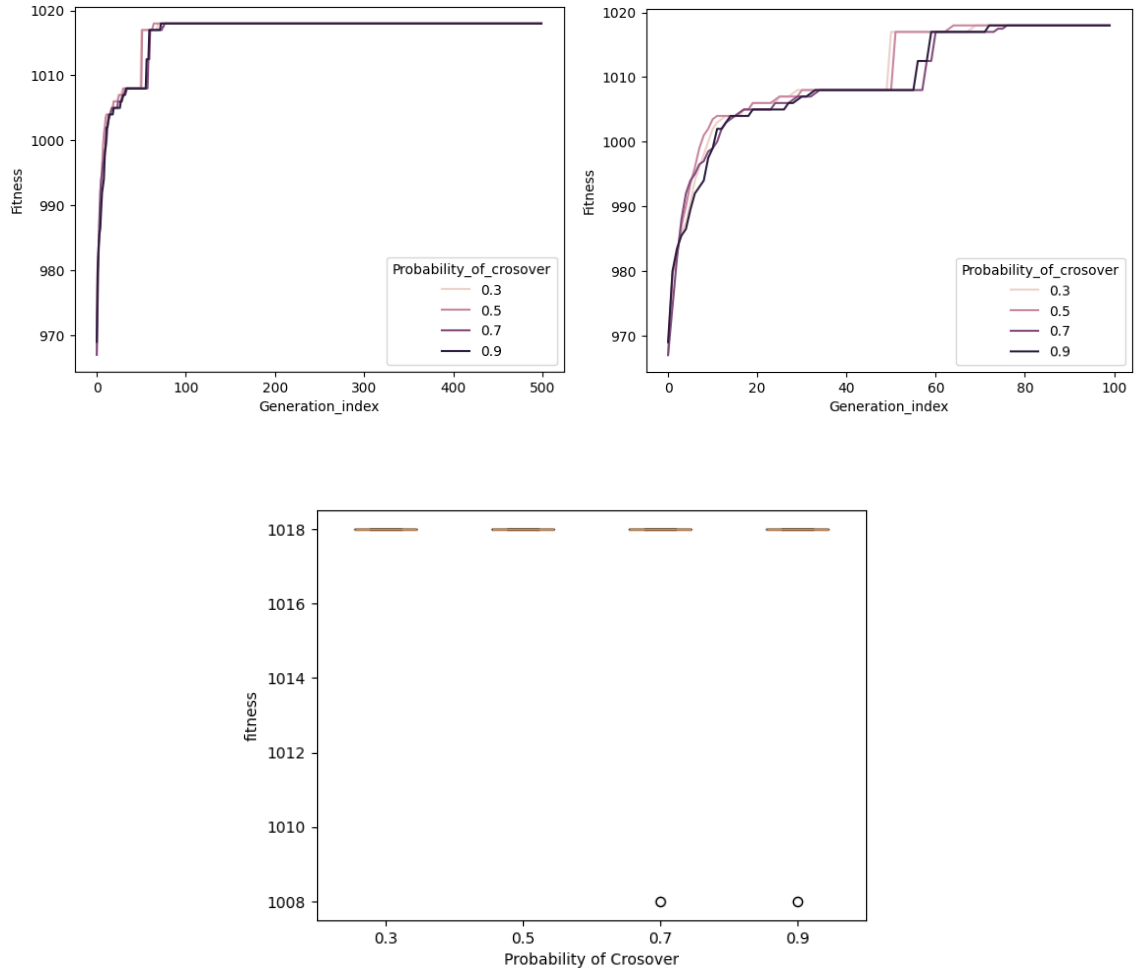


Figure 3 Testing Probability of Crossover

Conclusion: All results are good, but because the 0.7 and 0.9 had outliers in some generations with bad results we will use probability of crossover = 0.5, giving a 50% chance of doing crossover. Since mutation is always applied, we still have variability of the individuals in the offspring population.

- Fitness Function and Selection:

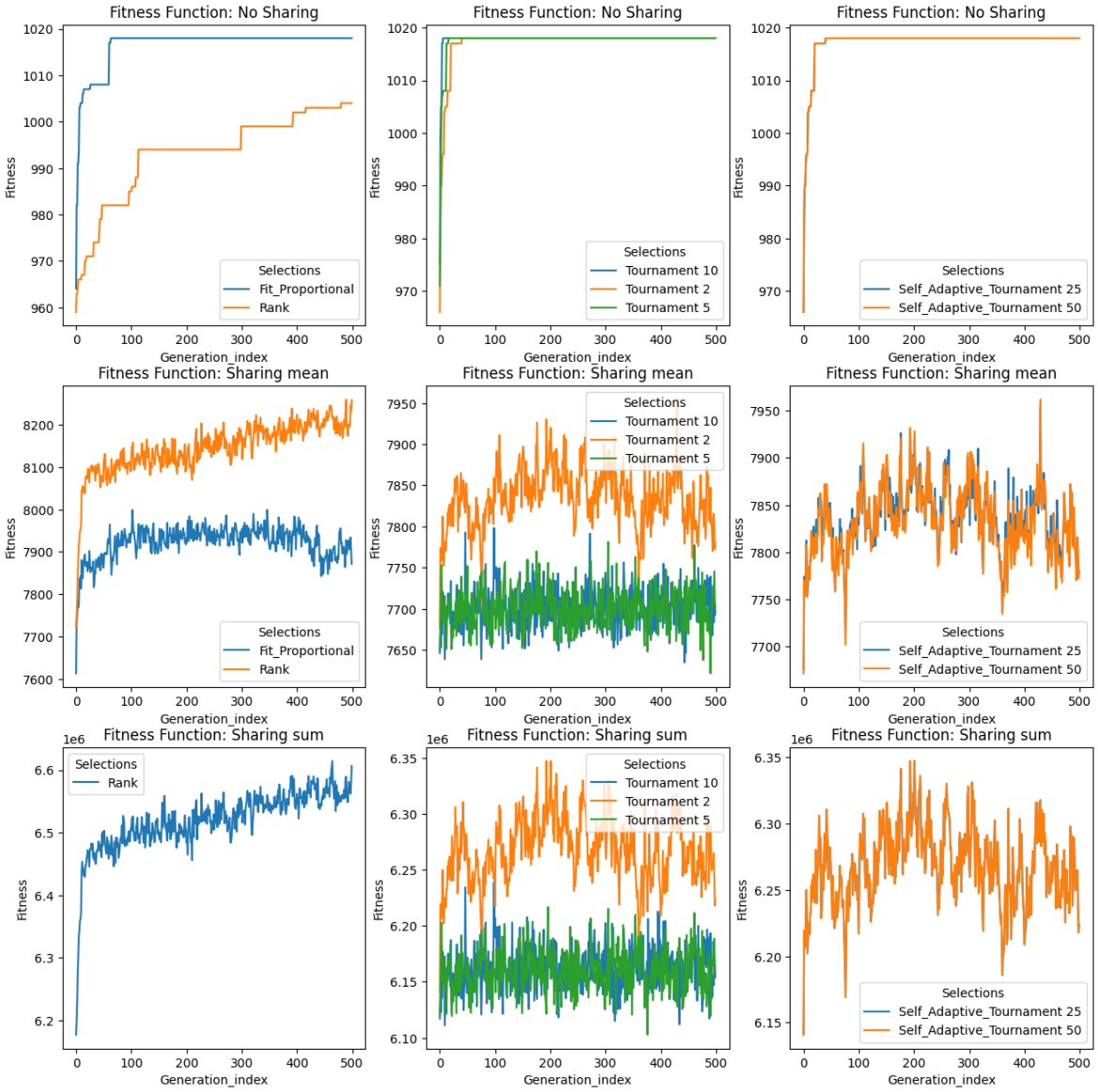


Figure 4 Comparison Selection and Fitness Function

Above we have the results with their respective fitness function, we can see that using the sharing fitnesses enables the algorithm to escape local optimums, but we need to compare if the final result is really the best local optimal, so we need to transform the solutions to their normal fitness value and compare with the same scale.

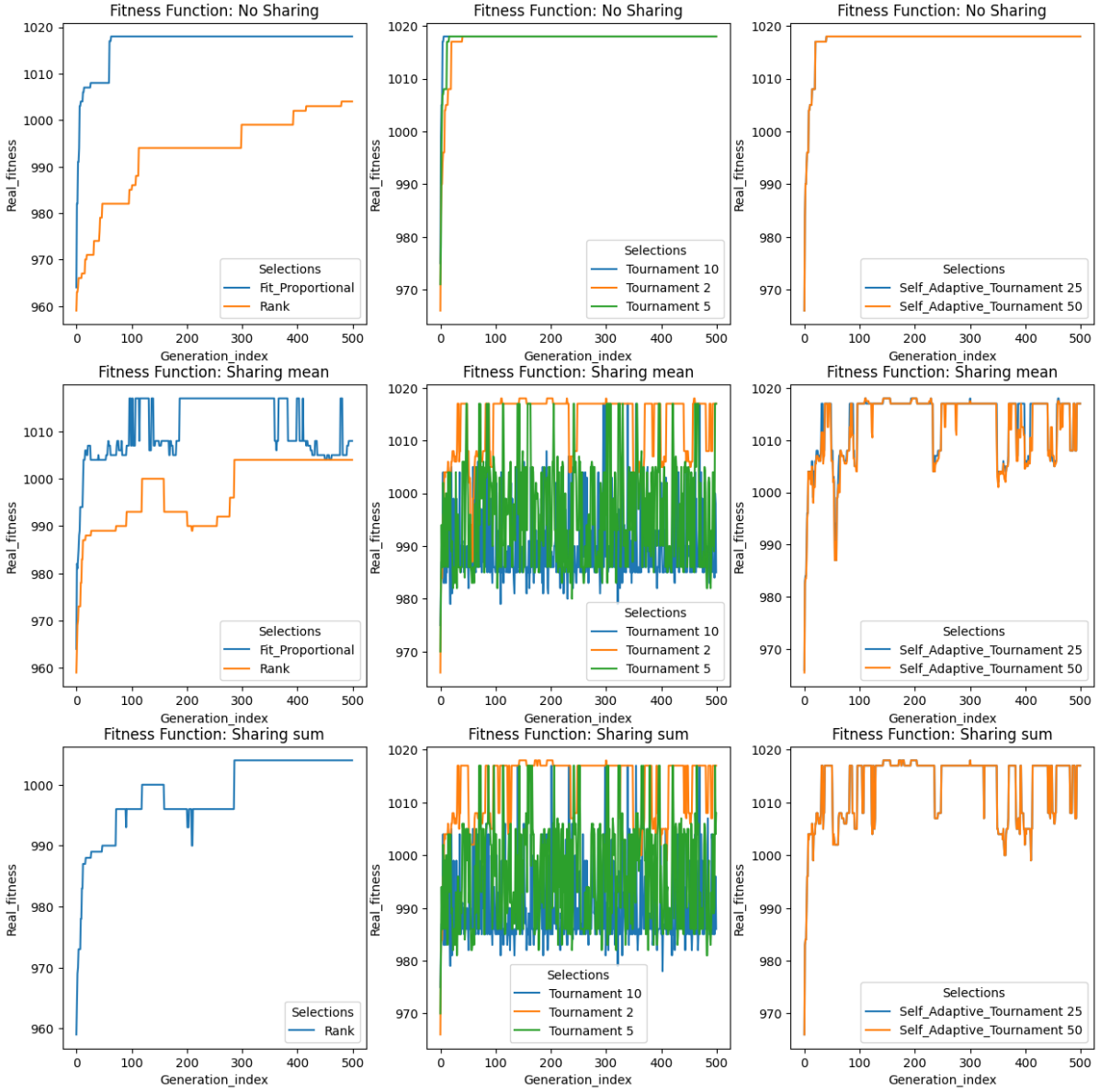


Figure 5 Comparison Selection and Fitness Function using the same scale.

After looking at the results of the fitness functions it's clear that using sharing fitness will allow us to escape local optimums, but doesn't guarantee that the last result, the one that will be delivered, is the best. This can be explained by the fact that a solution with a moderate fitness which is very distant from the rest of the popupaltion will have a very high fitness with sharing and will appear to be the best one when in reality there are other solutions with higher fitnesses. For that reason, we will use the fitness no sharing function.

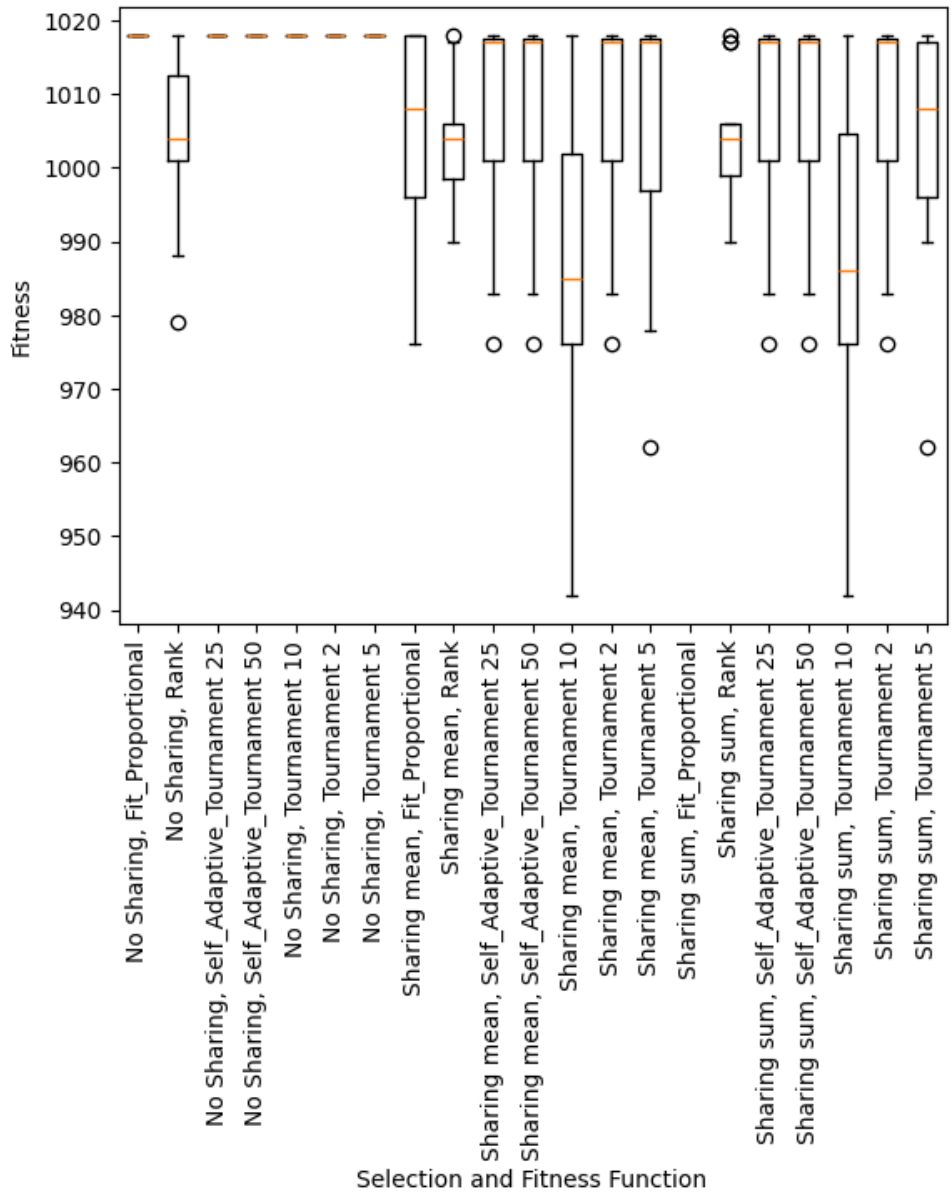


Figure 6 Testing Selection and Fitness Function

Considering only the distributions of the fitness function with no sharing we see that the results for fitness proportional, tournament selection and self-adaptive tournament selection are almost the same. We discard fitness proportional for being sensitive to the scale and extreme fitnesses and decide to choose the Self-adaptive tournament 25 because it considers the distribution of the fitnesses, applying bigger selection pressure when the fitnesses are different and less selection pressure when the fitnesses are close together; we don't consider Self-adaptive tournament 50 because it has the same results as 25.

Conclusion: We will use fitness function = No sharing, and selection = Self-adaptive tournament 25

- Crossover and Mutation:

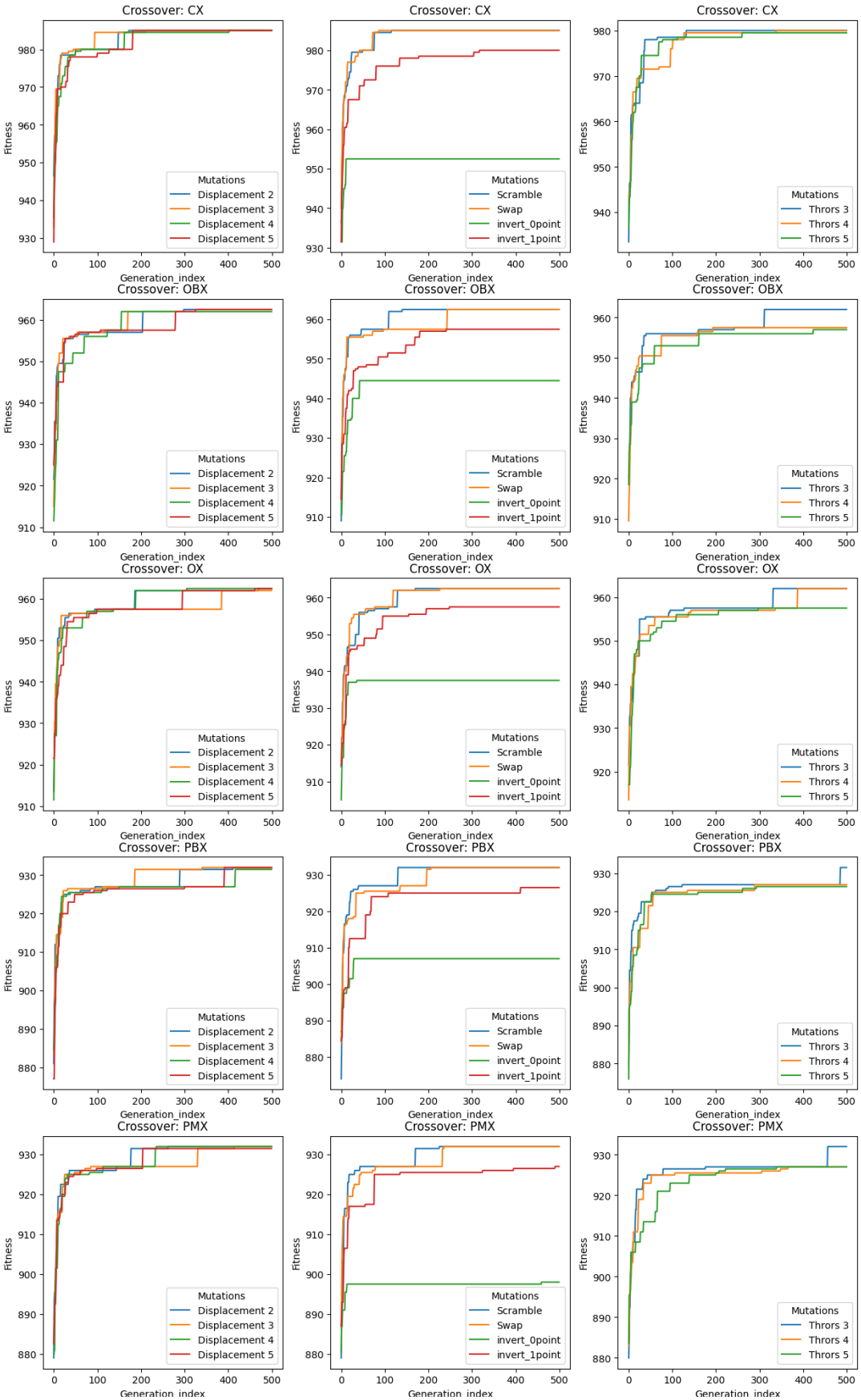


Figure 7 Comparison Crossover and Mutation

We see that overall, the Crossover that manages to achieve the best results is the cycle (CX) crossover, so we will analyze it deeply.

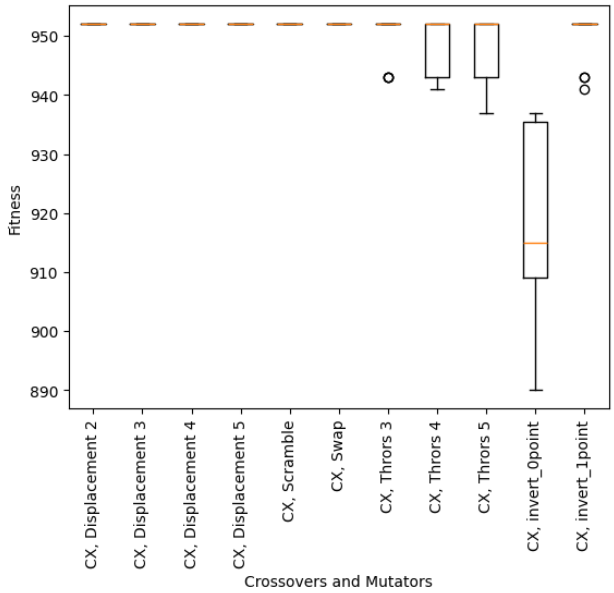


Figure 8 Testing Crossover and Mutation

Looking at the distribution of the last generation of the cycle crossover with the different mutators we see that Displacement 2,3,4,5, Swap and Scramble are good candidates to be the best mutator operator since they all achieve excellent results in the last generation.

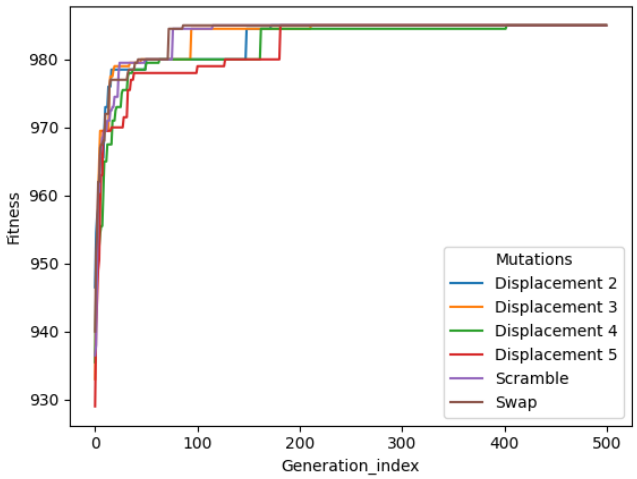


Figure 9 Comparison of Mutators using Cycle Crossover

We see that Displacement 3, Swap and Scramble are the fastest to achieve optimal results so we will choose between those 3.

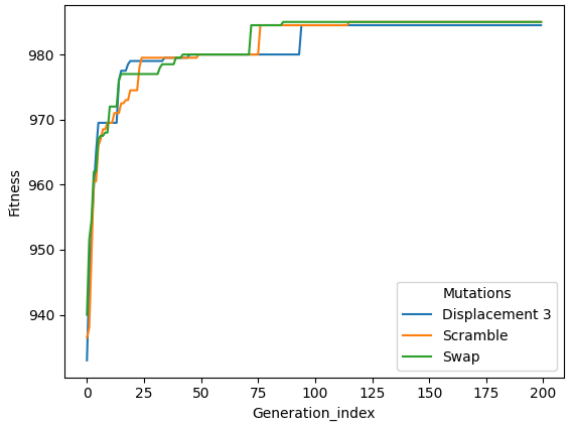


Figure 10 Comparison between best Mutators using Cycle Crossover

Conclusion: The 3 options are great; we will use the swap mutator operator with the cycle crossover because it's the one that achieves faster and better results and is the least complex mutator operator to implement.

Final Genetic Algorithm:

Best parameters:

- Elitism = True
- Population size = 800
- Number of generations = 500
- Fitness function = "No sharing"
- Selection function = "Self-adaptive tournament 25"
- Crossover operator = "Cycle"
- Mutation operator = "Swap"

The results using our datasets and doing 15 runs with these best parameters:

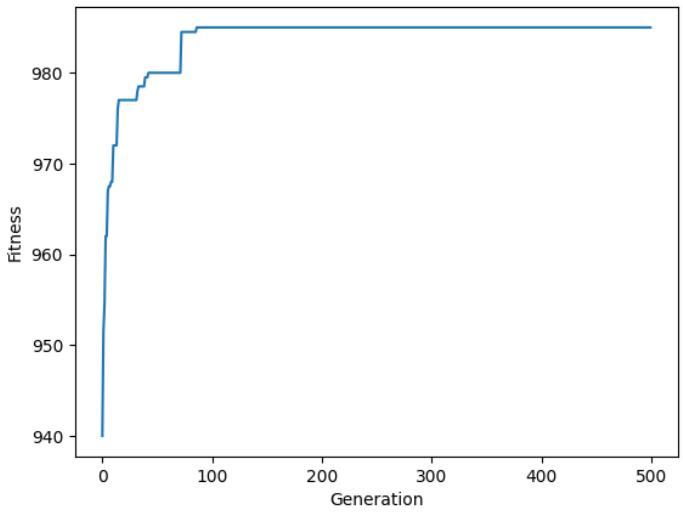


Figure 11 Performance of the best parameters for the Genetic Algorithm

Conclusion:

Given any matrix of geo earn/loss you can know the best path for you to use in the game if you apply the Genetic Algorithm that was found.

Bibliography:

Mutator operators:

- https://arxiv.org/abs/1203.3099 , "Analyzing the Performance of Mutation Operators to Solve the Travelling Salesman Problem" by Otman ABDOUN, Jaafar ABOUCHABAKA, Chakir TAJANI.

Selection Operators:

- inspiration of https://github.com/Mobink980/Learning/blob/2565a591c67207f8936304da7eed0eef2da26fff/genetics/selections.ipynb "Selections in genetic algorithms" for the Self-Adaptive Tournament Selection.
- "Lectures on Intelligent Systems", L. Vanneschi and S. Silva, for Rank, Fitness proportional and Tournament selection.

Crossover operators:

- https://link.springer.com/article/10.1007/BF02125403 , "Genetic algorithms for the traveling salesman problem" by Jean-Yves Potvin.

Fitness function:

- "Lectures on Intelligent Systems", L. Vanneschi and S. Silva.