TSP & PIP

GENETIC ALGORITHMS

GROUP ESCAPING LOCAL OPTIMUM

ANA ST. AUBYN R2016713
ANA RITA MARQUES R2016700
DAVID SILVA R2016730
PEDRO ALVES R2016734T

CIFO
JANUARY 11TH 2020

CONTENTS

1. OBJECTIVE

This project aims to explore the optimization concepts of the Genetic Algorithm, applying them in the design and development of two combinatorial optimization techniques - Travelling Salesman Problem (TSP) and Portfolio Investment Problem (PIP).

The main objective for the TSP was to do the design thinking - encoding rules, fitness function, admissibility function, etc. - but also to apply and develop different approaches to reach an optimal solution (e.g. crossover, mutation, initialization and selection operators) and compare them, whereas in the PIP the focus was on designing the problem and implementing a simpler version of the algorithm.

2. TRAVELLING SALESMAN PROBLEM

TSP is a trivial optimization problem where the main goal is to minimize the distance traveled by a hypothetical salesperson between a set of cities that must all be visited, starting and finishing in the same city.

> "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?"

2.1. METHODOLOGY

We used the functions already created by the professor and created some new ones. The flowchart in the annexes shows how the algorithm navigates through these functions.

2.2. CONCEPTUALIZATION

A) ENCODING

Keeping in mind that we could receive a distance matrix with n cities, the encoding of the solution must be as general as possible. We considered that a solution is a fixed length string of cities, each identified by a different number. We found this to be the most appropriate encoding as we are dealing with a permutational problem which demands that each city should be visited exactly once. Furthermore, it allows us to consider the adjacency and order of the cities. Considering a search space of

permutations over n cities, it's easy to understand that there are n! possible solutions. For a better search space dimension understanding, if n=10 the dimension of it is of the order of magnitude of 10^6, but for n=100 cities it is in order of magnitude 10^{157}.

Given that an optimal tour exists and since it is a circular path, the start of the path is indifferent for the calculation of distances. We can then fix the city for the start of the path, reducing the complexity of the problem by n, from n! to (n-1)! possible solutions. It is not a very big reduction since for 100 cities we go from 10^{157} (=100!) to 10^{155}, but it is a free search space reduction without any loss in the quality of the solution and, since we are concerned about performance, it is a good gain, that scales with the increase in complexity of the problem.

B) ADMISSIBILITY

The admissibility function must cover what should not be allowed in our solutions, so we considered that an admissible solution complies with the following requisites:

- ✓ The length of the solution representation must be equal to the size of the encoding rule (number of cities - 1);
- ✓ The count of the unique values in the solution representation must be equal to the size of encoding rule;
- ✓ The values in the solution representation must belong to the set of city identifiers created in the encoding rule.

C) FITNESS FUNCTION

Since the objective of the TSP is to find the shortest path that goes through each of the cities once and then returns back to the initial one, we defined the fitness function for this problem as the total distance for the defined route including the departure point (city 0) and the final one (city 0). By minimizing this function, we will find the optimal tour.

2.3. MODIFICATIONS

A) COMPUTATIONAL EFFICIENCY

When computing Hill Climbing Initialization, we faced a problem that could reduce the chances of getting a better solution within the time and computational constraints. Steepest Ascent Hill Climbing by itself is a heavy computational algorithm that takes time to evolve to a better solution as it evaluates all the neighborhood. But adding to this, if

the algorithm is applied over lists, can take 10 times more than when applied over numpy arrays. Numpy arrays takes up less space, are faster than lists and, in terms of functionality, they have optimized functions. Therefore, we changed our initial neighborhood function to a more optimized one using numpy arrays.

B) CONFIDENCE INTERVAL

The way the script was provided to us, the confidence interval for the estimate of the average values of the fitness in each configuration had a confidence of 68%. A 68% confidence interval is a range of values that you can be 68% certain contains the true mean of the fitness of the configuration. We wanted to be surer than 68%, so we changed it to a 95% confidence interval by using the following formula:

$$CI_{95\%} = FitnessMean \ \pm 1.96 * \frac{FitnessStandardDeviation}{\sqrt{NumberOfRuns}}$$

C) SCRIPT DATA

In order to be able to make a grid search automatically, we created a dictionary with abbreviations for each method, and various lists for each parameter of the algorithm. We then make the algorithm go through grid search creating the filename automatically from the abbreviations. We can then use the file names for simpler further analysis, allows for better handling of the runs and distributed computing.

D) AWS INSTANCES

To overcome our resource constraints, we decided to use AWS instances to run our configurations and explore a myriad of alternatives. However, the instances that we used were covered by the free tier and weren't very powerful. For this reason, we didn't manage to use this service in a reliable way, but we learned how to use it for further opportunities.

2.4. VARIATIONS TABLE

To increase the performance of the algorithm, we developed and implemented different variations of the initialization algorithms and genetic operators that are summarized in the following table.

Approaches and Genetic Operators	Implemented Variations
Initialization Approach	1. Random Initialization
	2. Greedy Initialization
	3. Hill Climbing Initialization
	4. Multiple Initialization
Parent Selection Approach	1. Roulette Wheel Selection
	2. Rank Selection (Baseline)
	3. Tournament Selection (Baseline)
Crossover Operator	1. Partially Mapped Crossover (PMX)
	2. Cycle Crossover
	3. Order 1 Crossover
	4. Heuristic crossover
	5. Multiple Crossover
Mutation Operator	1. Swap Mutation
	2. Insert Mutation
	3. Invertion Mutation
	4. Scramble Mutation
	5. Greedy Swap Mutation
	6. Multiple Mutation
Replacement Approach	1. Standard Replacement (Baseline)
	2. Elitism Replacement (Baseline)

Table 1 – Implemented Variations Table

Besides the basic Initialization Approaches given, we developed 3 more:

2. **Greedy Initialization** is an algorithm created on our own. Instead of initializing the population randomly, the algorithm randomly selects the first city and until the length of the individual is achieved, it assigns the next city based on the smallest distance between the actual city and the still possible cities in the current city's correspondent row of the distance matrix, excluding the first city in the distance matrix.

3. **Hill Climbing Initialization** was, at first, a simple Hill Climbing algorithm fed by random initialization. However, random solutions tend to have very low fitness, and Hill Climbing with steepest ascent for a one swap neighborhood is computationally expensive and takes several iterations to converge to a local optimum. On the other hand, Greedy Initialization is very light computationally and when applied to Hill Climbing, adjacency is not a problem, therefore, we fed Hill Climbing with the greedy initialization, decreasing the number of iterations and getting a better solution at the beginning of the algorithm. We observed a

10-fold decrease in the average iterations of hill climbing for every individual, from almost 100 iterations every time to a range of 8 to 20 when fed with greedy initialization.

4. **Multiple Initialization** combines the previous three initializations on the table. As we already said, Greedy Initialization produces similar individuals and, consequently, we believe it is not the best method to initialize the whole population since the only difference between the solutions produced is a lateral shift based on the first city selected. As various crossovers and mutations for permutations are focused on keeping part of the adjacency, initializing all the population with this method will produce a homogeneous population. On the other hand, Hill Climbing is a very good initialization but comes with a high computational cost if one uses as neighborhood all the solutions that can be obtained by swapping two cities. Still, we were concerned with possible loss of diversity, therefore, we thought about a method that combines diversity and good solutions from the beginning. For populations with size equal to 10 or larger, five solutions are initialized using the Greedy Algorithm, other five using Hill Climbing and the rest using Random Initialization.

Based on two scientific articles (Ferreira & Karnick, n.d.; Puljic & Manger, 2013), we explored the crossover possibilities to increase our performance and ended up implementing two crossovers not covered during the lessons:

4. The **Heuristic Crossover** is one of the best crossovers in terms of performance in both papers. This crossover, as it is explained in detail on the second paper, creates a child cycle by choosing from each allele the cheaper of the two respective parent arcs, this means that after choosing a random city to start (city 1), the crossover finds the city next to the city 1 on both parents and chooses the one with less cost (the closest city). If both cities on the parents are already in the child, it chooses the next city randomly from the missing cities.

5. The **Multiple Crossover** is a mix of the three crossovers that provided better results in the analysis executed in phase 2. We considered this crossover because "*After a while, single crossovers methods start lacking variety in their reproductions (...)*". Besides that, the reason why we considered only the best crossovers and not all is due to the fact that "(...) *multiple crossovers technique works better with only the most efficient methods"*, and due to the maximum generation constraints.

Considering the mutation approaches, we also developed two different ones:

5. The **Greedy Swap Mutation** chooses a random mutation point and finds the closest city of the mutation point swapping the closest city with the city that was already next to the mutation point.

6. Once again and in order to diversify the genetic operators and get a variety of reproductions, we developed a **Multiple Mutation** that mixes the three best mutations according to the analysis executed in phase 2.

2.5. METHODS COMPARISON

For the methods comparison for each phase, we used Power BI to visualize all the run configurations. This application is published online and we can reach it through this link. In it, we can find line charts where each line represents the evolution of a different configuration. Each sheet is focused on a certain topic (e.g. crossover operators) and it can be interactively explored, for further analysis. When navigating through the sheets, the user must keep in mind that they may take some time to load, as each one has heavy information from more than 2000 runs.

2.5.1. PHASE ONE

After the baseline implementation composed by Random Initialization, Roulette Wheel Selection, Cycle Crossover, Swap, Insert and Inversion Mutations, we tested several crossover and mutation probabilities - 0.05, 0.1, 0.25, 0.5, 0.75, 0.9, 0.95 - to establish the appropriate probabilities for the next phase. As in the next phase we will perform all the possible configurations with all the initializations, selections, operators and replacements and, since our computational power and time is limited, it is imperative to shorten the crossover and mutation probabilities to speed up the process without compromising the algorithms comparison. This choice was based on the consistently better probabilities along all the run configurations.

When comparing the mutation probabilities, 0.95 and 0.9 seem to be the best ones alongside with 0.75, indicating that highest mutation probabilities should be better for the algorithm.

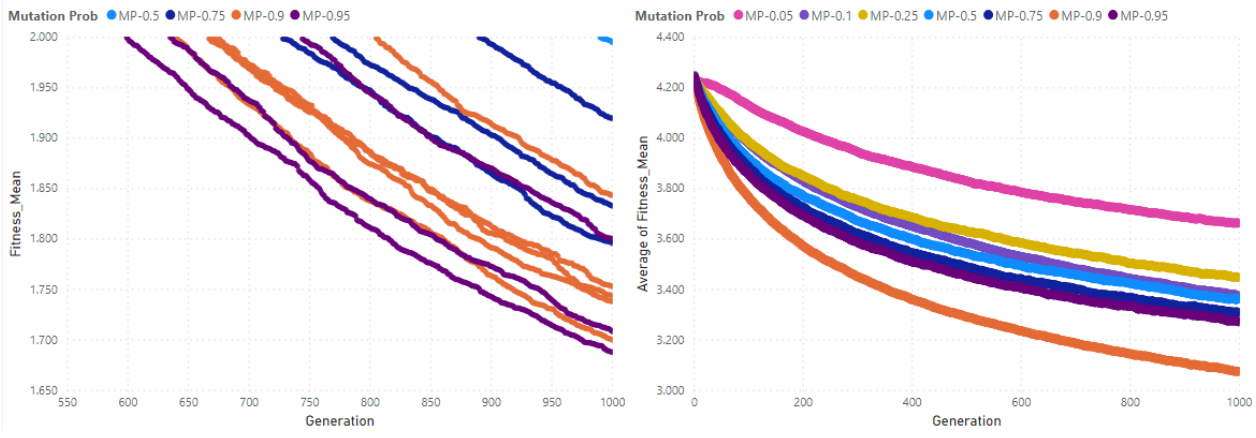


Figure 1 - Mutation and Crossover Probabilities for Phase One

Regarding the crossover probabilities, the opposite configuration reveals to be a better approach, meaning that small crossover probabilities - 0.05 and 0.1 - should be better in achieving the goal.

Despite the clear results, in theory, we typically expect that combining high crossover probabilities and small mutation probabilities would help us to achieve the optimal solution. As the previous results were based on a small sample of mutations and just one crossover, we believe that this hypothesis should also be explored. Therefore, the mutation and crossover probabilities considered for phase 2 was 0.9 and 0.1 for both set.

In this phase, we could also exclude the Standard Replacement from the operators to take into account, because, as we can see clearly from the graph below, Standard Replacement is giving poorest results when compared to Elitism Replacement, creating a non-evolving pattern.

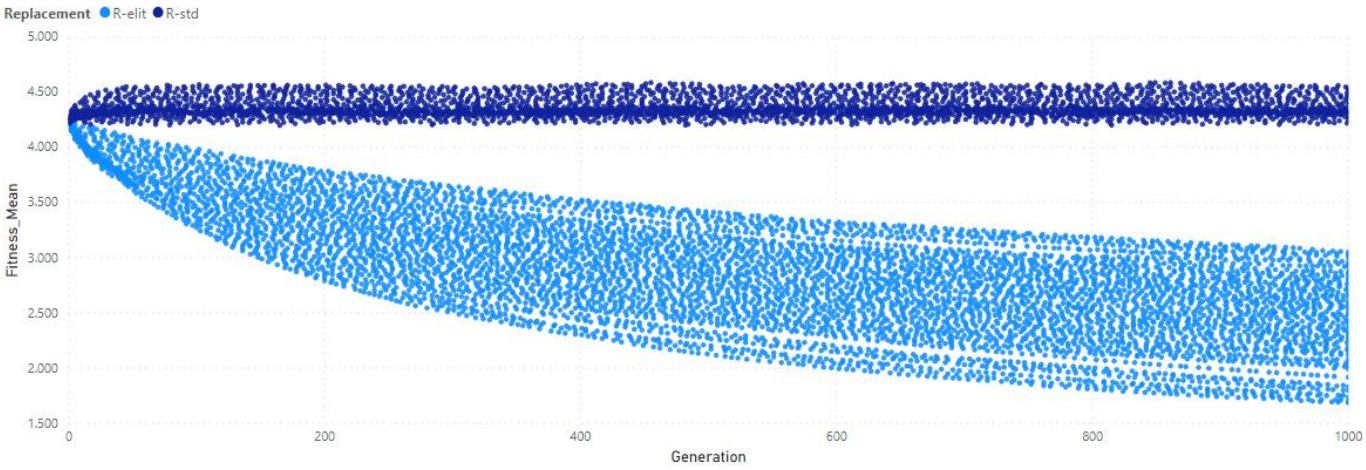


Figure 2 - Elitism vs Standard Replacement

2.5.2. PHASE TWO

Phase two is exploratory and the main goal is to compare the performance of all the configurations with the previously defined probabilities, in order to discover the best crossovers and mutations operators and then define the ones that should be in the multiple crossover and mutation, allowing us to also capture the performance of each configuration to get the best operators to invest in the next phase.

Running all the possible configurations with our operators, the best solution obtained has a fitness of 763 units of distance.

Regarding the mutation's performance, we can clearly observe that the best one is Invert Mutation, followed by Scramble and Greedy. However, Invert Mutation provides much better solutions than the other two and, therefore, for the multiple mutation we

decided to give a higher probability to Invert Mutation (0.8) while the other two were left with a probability of 0.1.

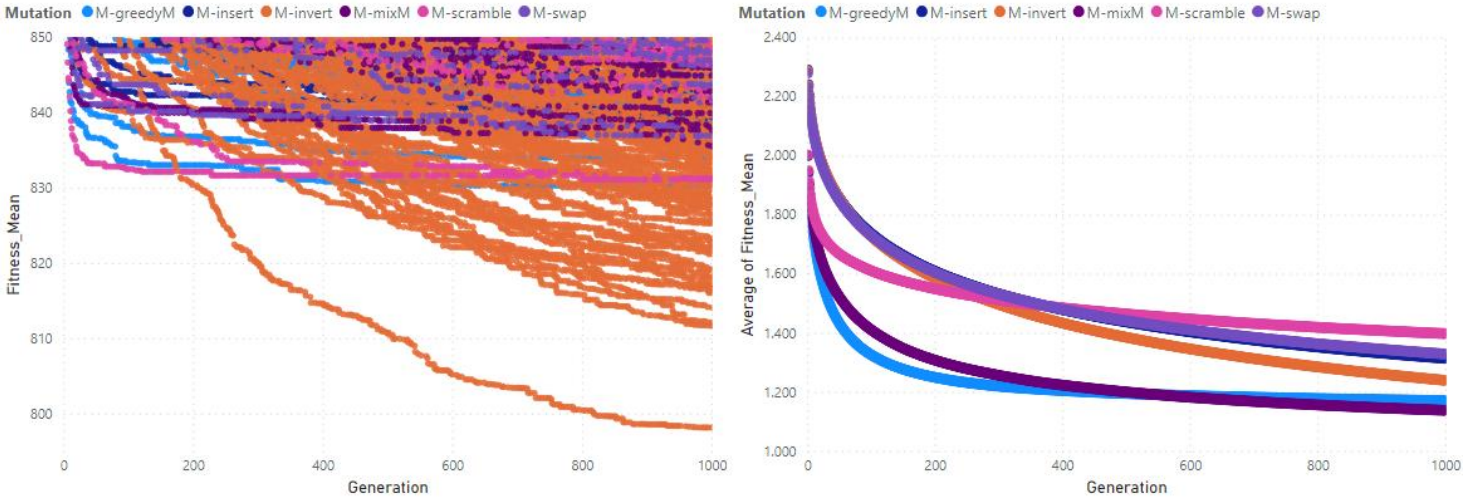


Figure 3 - Mutation Approaches Comparison (average on the right)

Once again, there is one crossover that is much better than the others - Heuristic Crossover - and it is followed by PMX, Multiple and Order1 crossovers. The three crossovers chosen to perform in Multiple Crossover were exactly the ones that gave better results in this phase. Therefore, the only thing that we changed was the probabilities to choose the crossovers with a probability of 0.8 to heuristic and 0.1 to PMX and Order1.

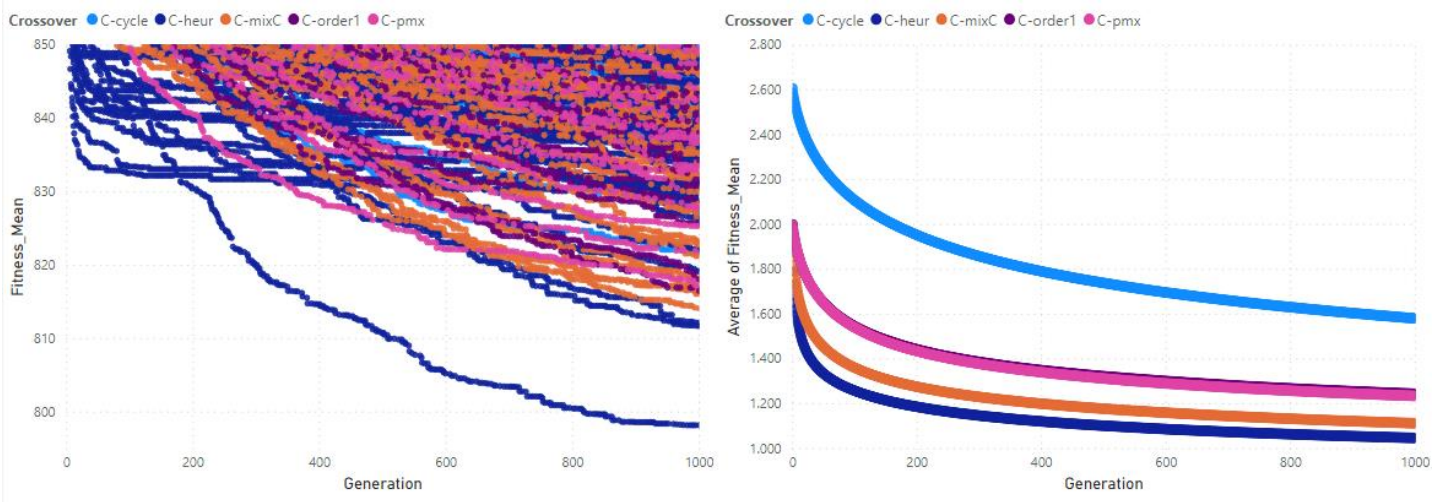


Figure 4 - Crossover Approaches Comparison (average on the right)

Comparing the parent selection approaches, Tournament Selection is clearly the best one mostly with 10 contenders but also with when 5 contend. Roulette Wheel is not even plotted which means that this selection is very bad compared to the other two and, despite the fact that Rank Selection is better than Roulette Wheel, Tournament Selection takes over the visualization. Thus, the choice is simple, we will proceed only

with Tournament Selection with tournament sizes of 10 and we will try with 15, keeping aside the tournament size of 5 because of time constraints.

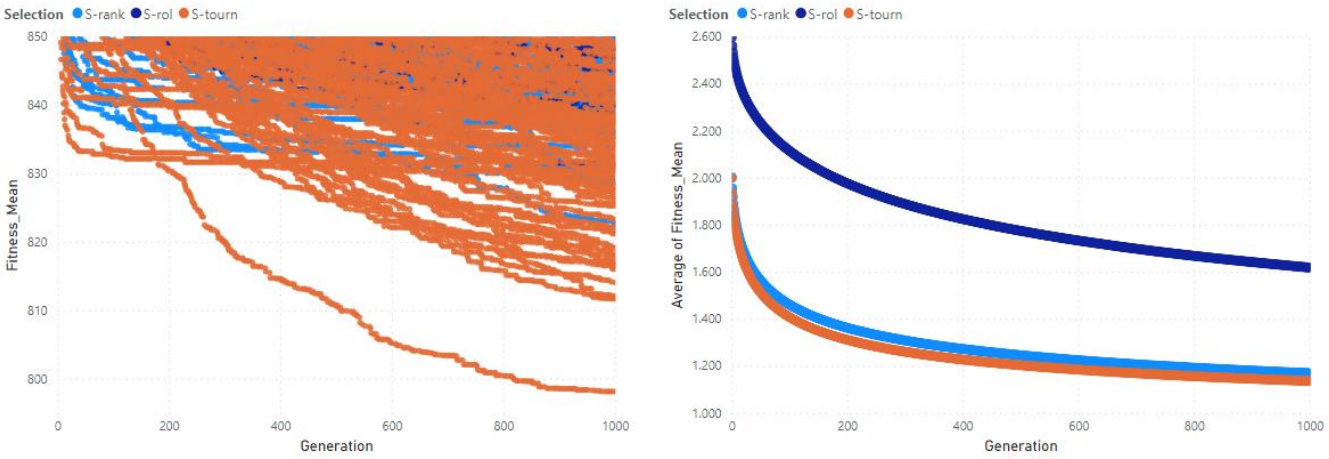


Figure 5 - Parent Selection Approaches Comparison (average on the right)

When it comes to the initializations, the best solutions were outputted by Hill Climbing, fed by the Greedy algorithm and Multiple Initialization as a close second. We then thought about the composition of the Multiple Initialization since, at the beginning, we were initializing populations where two solutions were Greedy, other two Hill Climbing and the rest Random. However, after seeing the results, and since Hill Climbing provides the best solutions, it makes more sense to initialize more solutions with this algorithm and with the Greedy one. Therefore, we changed the composition in order for it to initialize five solutions with Hill Climbing, other five with Greedy and the rest with Random Initialization.

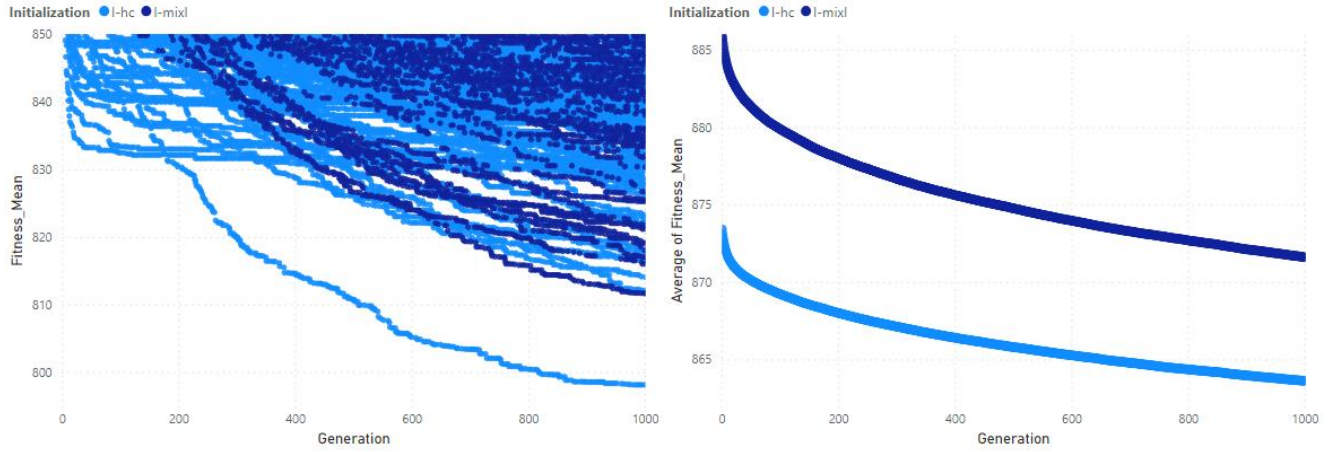


Figure 6 - Initialization Approaches Comparison (average on the right)

Finally, and bearing in mind that our time is limited, for our final set of configurations we wanted to explore more mutation and crossover probabilities. We had two options: we could either explore the probabilities' middle range (i.e. 0.30, 0.50, 0.60) or explore the probabilities' extremes range (i.e. 0.9, 0.1, 0.05, 0.95). We decided to explore the

extremes because as we concluded from phase 1, extreme probabilities tend to give better results.

Hence, the range of possibilities that we will explore for the next phase is the following:

- ✓ **Initializations**: Hill Climbing and New Multiple Initialization
- ✓ **Parent Selection Approach:** Tournament
- ✓ **Tournament Size**: 10, 15
- ✓ **Crossover Operators:** Heuristic and New Multiple Crossover
- ✓ **Mutation Operators:** Invert and New Multiple Mutation
- ✓ **Crossover and Mutation Probabilities:** 0.05, 0.1, 0.9, 0.95

2.5.3. PHASE THREE

Since in the previous phase we chose the best parametrization, in this phase, we will run the algorithm using it and check which combination of parameters gives us the best solution throughout 50 runs. The comparison between configurations for this phase are on the sixth page forward of the Power BI application mentioned before.

The best solution found has a fitness of 755.46 distance units and is produced using the following parameters: Multiple Initialization, Heuristic Crossover, Multiple Mutation, Tournament Selection, tournament size of 10, crossover probability of 0.1 and mutation probability of 0.9, Elitism Replacement.

The top ten best solutions and the respective configurations obtained are described in the following table.

Initialization	Tournament Size	Crossover	Crossover Prob	Mutation	Mutation Prob	Fitness
Multiple	10	Heuristic	0.1	Multiple	0.9	755.46
Hill Climbing	15	Heuristic	0.05	Multiple	0.9	758.29
Hill Climbing	15	Multiple	0.05	Multiple	0.95	758.93
Multiple	10	Multiple	0.1	Multiple	0.95	759.10
Multiple	10	Heuristic	0.1	Multiple	0.95	759.50
Hill Climbing	15	Heuristic	0.1	Multiple	0.9	760.08
Multiple	15	Heuristic	0.1	Multiple	0.9	760.34
Hill Climbing	15	Heuristic	0.05	Multiple	0.95	760.51
Multiple	10	Heuristic	0.05	Multiple	0.95	760.58
Multiple	15	Multiple	0.05	Multiple	0.95	761.49

Table 2 - Top 10 Results

We plotted the average performance for each parameter of interest.

In the charts below, we can notice that **Multiple Initialization** has a higher fitness average than the Hill Climbing for the first generations but finishes with a lower average. For **tournament size**, as we concluded before, **the higher it is, the better results it provides**.

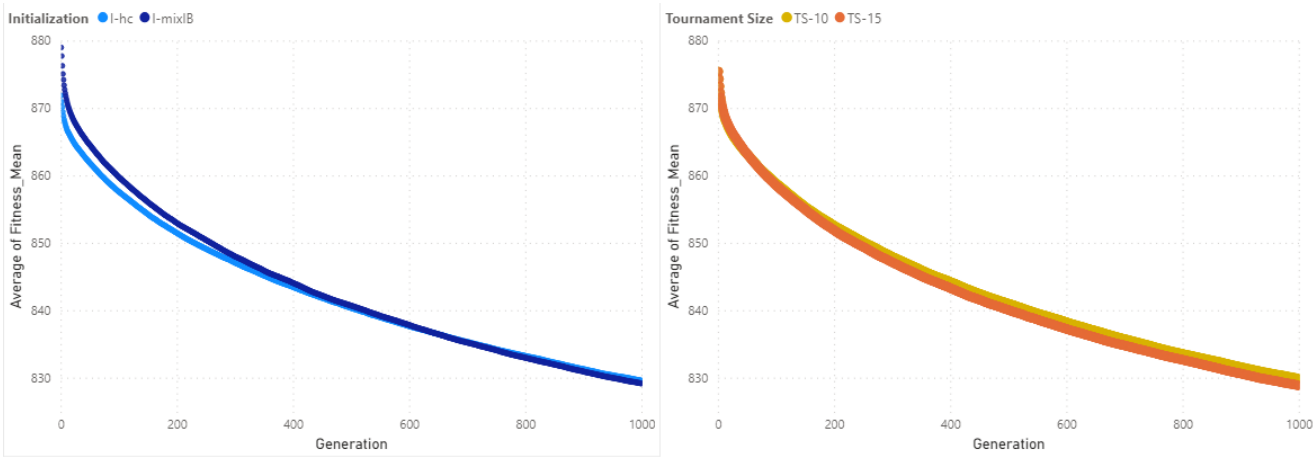
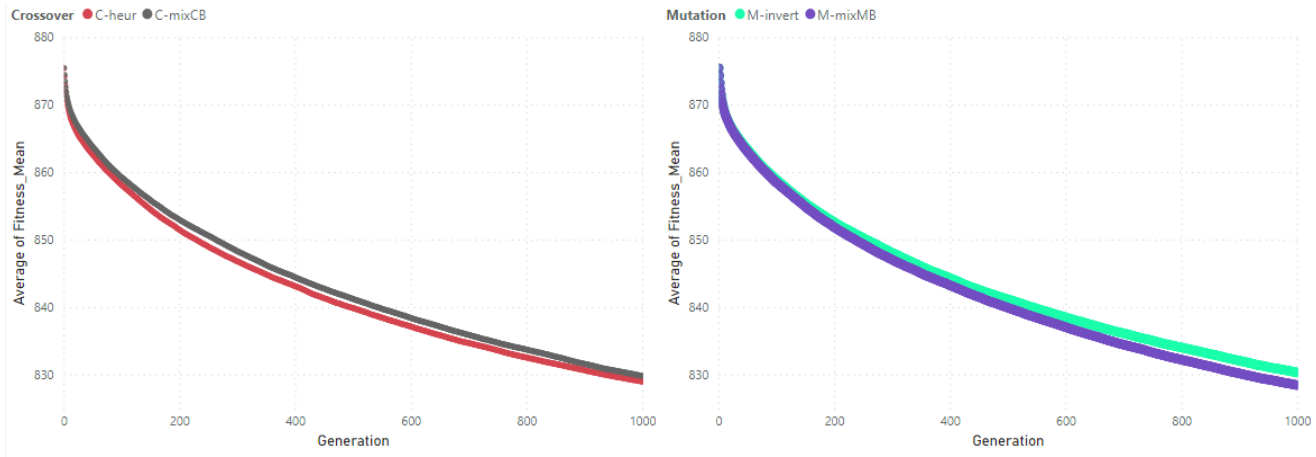


Figure 7 - Average Performance of Initialization Approaches and Tournament Size

Even though the best result was obtained with Multiple Initialization, and it is the best on average from all the final runs, the best configuration on average was achieved by using Hill Climbing Initialization.

On average, by looking at the chart below, we can also conclude that the **Heuristic Crossover** provides better results than the Multiple Crossover and, when it comes to



mutations, the **Multiple Mutation** is the one better on average.

Figure 8 - Average Performance of Crossover and Mutation Approaches

Finally, from the chart below, we can clearly see that **lower crossover probabilities** tend to be better than higher ones. **Mutation probabilities** work the other way around, as lower probabilities are worse than **higher** ones.

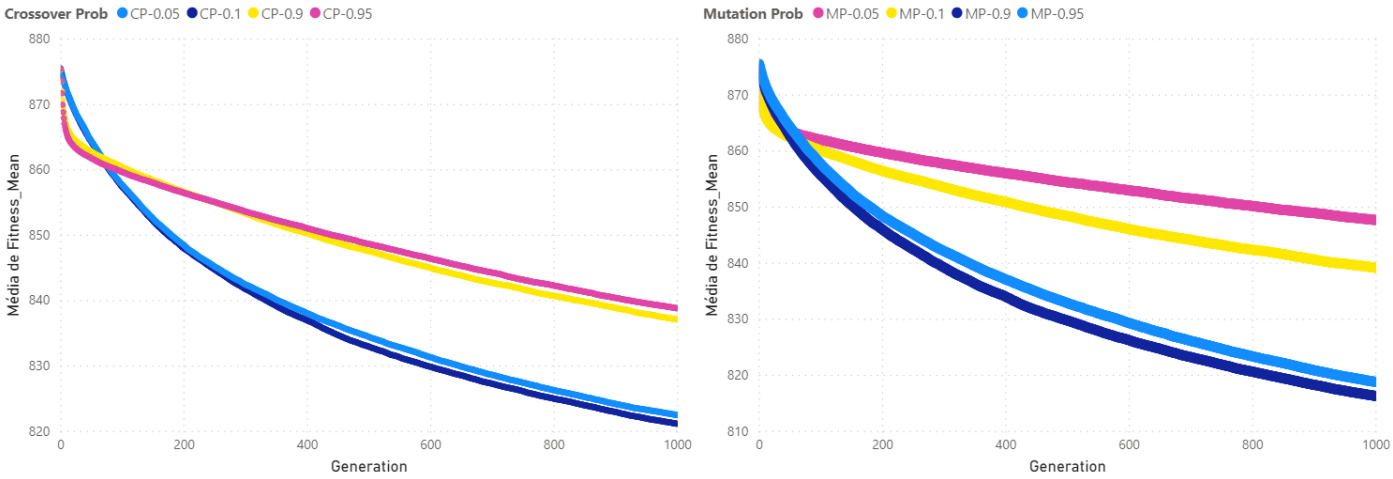


Figure 9 - Average Performance of Crossover and Mutation Probabilities

For the four configurations that gave us the best final solutions, we plotted the evolution of their fitness mean throughout generations and included the confidence interval of 95%. Since there is not a configuration where this interval does not intercept the interval of any of the others, we cannot conclude that one is statistically better. We will then stick to the one that gave us the best result.

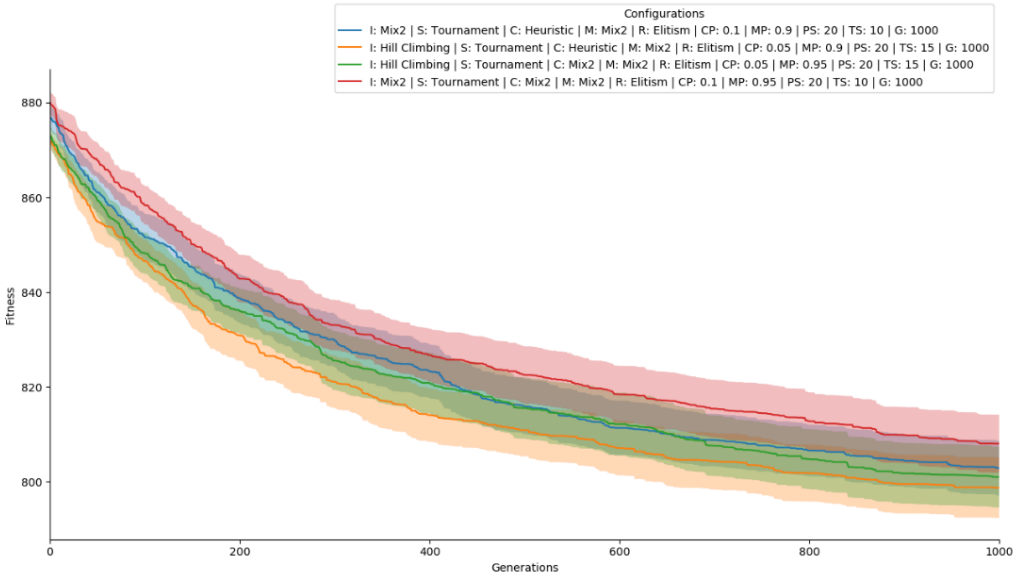


Figure 10 - Comparison of 4 Best Configurations

We chose the best solution, the second best, the worst and an intermediate one and we plotted the route between cities to see the evolution of our algorithm. The red dot represents the starting city, whilst the yellow one represents the second city visited so that we know the direction of the path, as the distances are asymmetric. The worst solution represented has a fitness of 4361.33 distance units; the intermediate one has a

fitness of 867.1183 and the second best solution has a fitness of 758.29 compared to our overall best of 755.46.

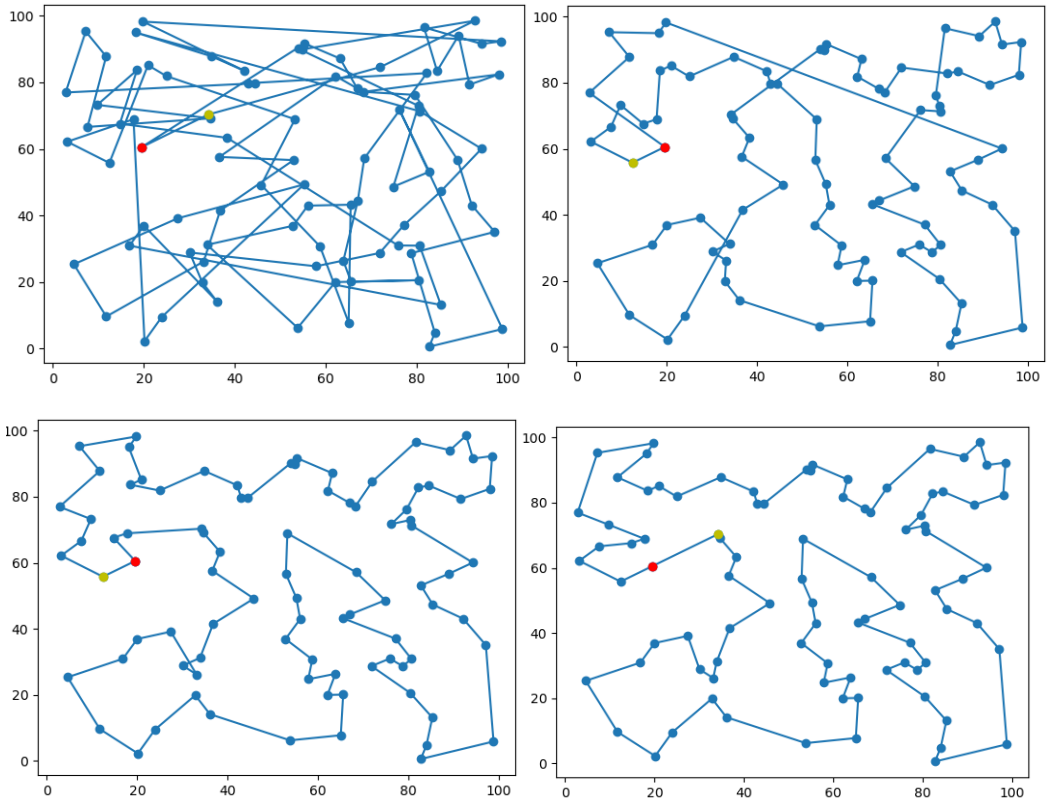


Figure 11 - Differences Between Paths for Different Fitnesses

2.6. CONCLUSION

We believe we achieved a good solution given the time and algorithm size constraints that drove us to further investigate and deploy ever better solutions and learned a lot while doing so. The fact that the best solution is achieved by using a mixed approach on every aspect, shows that we can incorporate diversity creation in the algorithm itself, by allowing the algorithm to follow different paths of evolution concurrently. Furthermore, our best algorithm is mutation driven and not as more usually crossover driven.

3. PORTFOLIO INVESTMENT

PIP is a real-world problem where the major challenge is the design thinking where we only needed to outperform the baseline version of the genetic algorithm. The objective is to achieve a portfolio of investments that maximizes the expected return.

At first, we thought about simply maximizing returns given the constraint of Sharpe Ratio being above 1, this value guarantees that an increase in risk is rewarded by a higher

return in expected return. This warrants a good portfolio but also a riskier one and may not be the optimal solution when it comes to invest our money. Given the portfolio theory, we know that the maximum Sharpe Ratio intercepts the optimum portfolio (i.e. the one that produces the highest return given a marginal increase in risk) and guarantees the best Capital Allocation Line (CAL). So, maximizing the Sharpe Ratio could be a better solution for our problem, than simply maximizing returns without guaranteeing the best ratio of risk and return, as maximizing only the returns can produce riskier portfolios

To maximize the Sharpe Ratio, one must be able to find the weights of the stocks that make up the optimum portfolio, so a first approach would be to encode each stock with weights that vary between 0 and 1. This raised a red flag since it probably generates a huge search space and would result in irrelevant and really low weights for a lot of the stocks, meaning that we would invest in almost every stock, what may not make sense depending on the stocks provided. In fact, if the option was binary, include or not the stock in the portfolio, the search space would be 2^500 (if 500 stocks were provided) - this would be an impossible search space to search in. Worst is that, in theory, the weight of a portfolio could be any real number between 0 and 1, and, since there are infinite numbers between 0 and 1, the search space would actually be ∞^500. In practice, this is limited by the maximum precision of a float variable, and in Python, we replace ∞ by 1.7976931348623157e+308. It is an improvement, but still a very, very large number. Therefore, there are two parameters we must try to optimize in order to get to a decent search space. Starting with the exponent, we can realize that we do not need 500 stocks to build a good portfolio. We believe this can be left to the user to specify depending on computing power and time available. However, given the maximum number of stocks to use, we will introduce an algorithm that automatically chooses the best candidates for portfolio creation. We can then reduce the number of the exponent to a more friendly one: 10, 20 or even 50.

Now, considering the base, we know that it is of no interest to have a very small proportion of the budget in a stock, since trading commissions would deny any profit for trades of tens of dollars. Let's consider a much smaller size number, 0.001. Even if we allowed the weights to change in steps of 0.001, in our test portfolio it would represent only 100$ for a given stock, and many stocks are priced above 100$. So, a better precision would not benefit greatly the solution, and we prefer to trade a good enough portfolio that can be achieved relatively quickly than a perfect portfolio that takes days to calculate for a marginal gain, and that provides results when the assumptions (stock prices) have already changed.

Given a step of 0.001, we can now obtain a search space that is reduced to 1000^n where n can be a small integer number, much better than the initial ∞^{500}.

However, since we now know the step, another improvement of runtime can be done. Instead of using floats and the inherent problems of using an incomplete representation , we can always multiply the weights and the constraint by the inverse of the precision, e.g., for 0.001 we do 1/0.001=1000, so our weights will now vary from 0 to 1000, and the constraint is also multiplied so the sum of the weights must equal 1000. Given this, the algorithm can run on integer weights. Return to the original weight format for formula application is as simple as dividing all the weights by the multiplier.

At this point and given an option to build our portfolio from 500 different stocks, one must wonder which stocks to use. Again, we turn to domain knowledge to understand the best choice. We do know that the Sharpe ratio is the line that intercepts the axis at the rate of the "zero risk" investment and is tangent to the efficient frontier of our portfolio, determining the optimum portfolio for the given efficient frontier. So, our approach is the known one of selecting the best frontier, given a portfolio of stocks. We know that the frontier approaches the risk-free axis and moves up on return, if we include negatively correlated stocks, since the matrix is dominated by the correlations. So, since we have time constraints, we implemented a greedy selector, based loosely on a pareto selection. We order the stocks by the ratio between expected return and risk, we select the best ones and we go to the correlation matrix, selecting the stock with minimum correlation (closest to -1). We proceed with pair selection, until we find the desired number of stocks. If the number provided is odd, the last selection will only include the first stock, and not the correlated one.

We developed a basic crossover and we used some TSP mutations that work for this kind of problem but that are not optimized for the problem at hand and also some geometrical to deal with the real numbers problem before realizing we could change this problem into an integer weights problem. We feel we could explore the problem further, but that was not the objective. Overall we believe we created a good starting point and a working one that allows for a faster and better full resolution of the problem and that produces solutions with Sharpe ratio above 3, and expected returns above the baseline, for the limited amount of runs we produced.

4. REFERENCES

Ferreira, P. R. L., & Karnick, P. (n.d.). Genetic Algorithm with Multiple Crossovers on the Travelling Salesman Problem. 9.

Puljic, K., & Manger, R. (2013). Comparison of eight evolutionary crossover operators for the vehicle routing problem. 17.

Pinho, C. S., & Tavares, S. V. (2012). Análise Financeira e Mercados, Lisboa, Áreas Editora

5. ANNEXES

METHODOLOGY DIAGRAM