Using Genetic Algorithms for Problem Solving

João Ferreira 20221912, Miguel Mendes 20221904, Rodrigo Maia 20221934

Nova Information Management School

Introduction

For this project, we aimed to enhance a genetic algorithm to solve a given optimization problem. Building on previous coursework, we researched and tested various genetic operators to identify the most effective ones.

Additionally, we developed a dataset generation function to create data similar to the proposed problem, adhering to the required constraints.

We conducted an extensive grid search to test different parameter combinations, identified the best-performing models, and further validated them on additional datasets to determine the final result.

Methodology

Before explaining our methodology, it's important to clarify how we classified the 10 different areas in the game. Rather than using initials as suggested in the project description, we numbered them sequentially from 1 to 10:

1. Dirtmouth
2. Forgotten Crossroads
3. Greenpath
4. Queen’s Station
5. Queen’s Garden
6. City Storerooms
7. King’s Station
8. Resting Grounds
9. Distant Village
10. Stag Nest

As mentioned, we started by building on the work completed throughout the semester. Some resources were reused, adhering to the imposed restrictions, while others were developed from our research specific to the problem at hand.

Below, we will list the various functions and explain their different uses and approaches.

Individual functions

- **generate_solution** – adapted from the class material. Generates a list with representative numbers of the 10 locations, starting and ending in the first city.

- **get_fitness** – adapted from class material. Here, some restrictions are imposed. To comply with the 4th and 8th restrictions, the function returns -1 if an infraction is committed. We chose -1 because most of the time the individual's fitness will be positive (the gain is much more often positive than negative), so a solution with fitness -1 will be very difficult to consider. Furthermore, the fact that our algorithm involves probabilistic strategies means that 0 cannot be the value returned in this situation.
  For the 3rd constraint, our strategy was to select the maximum value from one of these two approaches:
  1. Follow the normal path, even if it is unlikely to be chosen due to its complexity. In this case, we perform a straightforward cumulative sum.

2. Add the gains associated with each position, then subtract the gains linked to the sequence city X → 7 → city Y.

We adopted this approach because, although it was tiring and probably not the best choice, we wanted to keep a solution involving this sequence as much as possible, thereby preserving genetic variability.

Population functions:

- **create_population** – the same as *generate_solution*, but executed several times to create a population.

- **evaluate_ population** – the same as *get_fitness*, but executed numerous times to evaluate a population.

Selection functions:

- **tournament_selection** – adapted from class material. This selection strategy randomly chooses a predefined k number of individuals from the population and then selects the best individual from that pool of k individuals.

- **roulette_wheel_selection** – this method assigns each individual a slice of a roulette wheel proportional to its fitness, then "spinning the wheel" by selecting a random number and choosing the individual whose assigned slice contains that number. This ensures that individuals with higher fitness have a greater chance of being selected. Note: the abs() function is used to guarantee that the probability is higher than 0 and lower than 1, and to prevent the function from being ruined by negative paths when there are 2 elements with the same probability of being chosen. For example, for a cumulative sum of 0.5, if the next iteration has a negative fitness of -0.1, the abs() function will make it so that the cumulative sum will now be 0.6 instead of 0.4.

- **rank_selection** – this method assigns probabilities to individuals based on their rank in the population. Individuals are sorted by their fitness, assigned ranks, and then the probability of selection is proportional to the individual's rank. Then, the algorithm selects individuals based on these probabilities.

- **random_selection** – randomly select one individual from the population.

Crossover functions:

The functions presented below are the result of a search carried out to find crossover strategies that are typically good in the well-known traveling salesman problem (TSP), since this has some similarities with the proposed challenge.

- **cycle_crossover** – this crossover method involves identifying cycles of elements between the parents and using these cycles to create offspring. The find_cycle auxiliary function is used to identify a cycle of elements between the parents. It returns a list of indexes where the corresponding values will remain the same in the offspring solutions.

- **alternating_position_crossover** – in this crossover strategy, genes are alternately selected from the parents to create offspring, ensuring that each gene appears only once in each offspring. Also, under one of the restrictions imposed, this process starts and ends with specified values (in this case, city number 1: Dirtmouth), preserving that characteristic of the parents.

- **partial_mapped_crossover** – this process involves selecting two random crossover points, and then copying the segment between these points from one parent to the corresponding offspring. After that, mappings are created based on the copied segments to ensure unique genes in the offspring. Finally, the remaining genes are filled in by resolving the mappings between the parents.

- **position_based_crossover** – in this strategy, genes are assigned indices within the solution. This process involves randomly selecting a subset of positions from one parent and filling in the remaining offspring positions with genes from the other parent.

- **sequential constructive crossover (SCX)** – this crossover constructs offspring by sequentially selecting the next place from two parents based on a predefined order, ensuring that each city is included exactly once. It starts with the first city, then alternates between parents to choose the next place, avoiding places already included, until a complete individual is formed.

Mutation functions:

- **swap_mutation** – randomly swaps two distinct genes of an individual with a given probability. If a randomly generated number is less than the mutation probability

previously defined, two distinct indices of the individual are randomly selected, and the genes at those indices are swapped.

- **inversion_mutation** – reverses a segment of genes of an individual with a given probability. If a randomly generated number is less than the mutation probability previously defined, two distinct indices are randomly selected to define the inversion range. Then, the genes within this range are reversed, including the ones randomly chosen.

- **scramble_mutation** – shuffles a segment of genes of an individual with a given probability. If a randomly generated number is less than the mutation probability previously defined, two distinct indices are randomly selected to define the scramble range. Then, the genes within this range are shuffled randomly.

- **insertion_mutation** – inserts a gene from one position to another with a given probability. If a randomly generated number is less than the mutation probability, a random gene is selected from its original position and reinserted at another random position in the individual.

- **two-opt mutation** - randomly selects two points in an individual, excluding the first and last elements (Dirtmouth), and reverses the order of the places between them, the two points selected are not inversed only the ones in the middle .

Dataset generation function

We created a function to generate a dataset with a similar structure to the one proposed, while also respecting the constraints. A certain range in which numbers are generated is part of the function parameters.

Besides that, we decided to create a parameter that allowed us to choose the percentage of negative and positive values that form the matrix, apart from the diagonal elements.

Grid Search

During the grid search phase, we conducted two separate grid searches:

The **first grid search** tested:

- **population size**: [50, 75, 100]
- **probability of mutation**: [0.05, 0.15, 0.25]

- **probability of crossover**: [0.7, 0.8, 0.9]
- **number of generations**: [100, 150, 200]
- **selector**: roulette wheel selection, random selection, rank selection and tournament selection (with 2 different tournament sizes)
- **mutator**: swap mutation, inversion mutation, scramble mutation and insertion mutation
- **elitism**: [True, False]
- **elite function**: with different numbers of elite individuals (1, 2 and 3)
- **crossover**: cycle crossover, alternating position crossover, partial mapped crossover, position-based crossover

We evaluated the parameter combinations by calculating the median performance from 3 runs on each of the 5 datasets, and then summing these 5 medians.

While this method may introduce minor errors in identifying the best combination, alternative evaluation methods have their own drawbacks. For example, comparing medians across datasets often results in favouring the same dataset repeatedly.

Using the mean of medians could be more sensitive to outliers, and ranking might not adequately reward combinations that perform exceptionally well on certain datasets but slightly worse on others.

The datasets in this initial grid search had values ranging from -100 to 1000 (excluding 0), making the sum of medians a suitable metric for highlighting combinations that perform significantly better in one dataset while remaining competitive in others.

The optimal parameter combination from this grid search was:

- **population size**: 50
- **probability of mutation**: 0.15
- **probability of crossover**: 0.9
- **number of generations**: 150
- **selector**: tournament selection (with tournament size = 2)
- **mutator**: scramble mutation
- **elitism**: True
- **elite function**: numbers of elite individuals = 3
- **crossover**: position-based crossover

In the second grid search, we reduced the number of parameters to address the numerous ties observed in the first grid search. This refined search aimed to narrow down the parameter space, enabling a more focused comparison with the results from the initial grid search.

The **second grid search** tested:

- **population size**: [75, 100]
- **probability of mutation**: [0.15, 0.25]
- **probability of crossover**: [0.8, 0.9]
- **number of generations**: [150, 200]
- **selector**: roulette wheel selection, random selection, rank selection and tournament selection (with 2 different tournament sizes)
- **mutator**: swap mutation, inversion mutation, scramble mutation and insertion mutation
- **elitism**: [True, False]
- **elite function**: with different numbers of elite individuals (1, 2 and 3)
- **crossover**: cycle crossover, alternating position crossover, partial mapped crossover, position-based crossover

We continued to use the sum of medians as the evaluation metric, but this time, the datasets had a wider range of values from -10000 to 10000 to minimize ties. The optimal parameter combination from this grid search was :

- **population size**: 75
- **probability of mutation**: 0.25
- **probability of crossover**: 0.9
- **number of generations**: 200
- **selector**: tournament selection (with tournament size = 10)
- **mutator**: swap mutation
- **elitism**: False
- **elite function**: numbers of elite individuals = 1
- **crossover**: position-based crossover

Upon completing both grid searches, we compared the optimal combinations by running each combination on 15 different datasets. We then plotted and analyzed the results to determine the overall best-performing combination, which was the one from the **second grid search**.

Visualizations and Conclusions

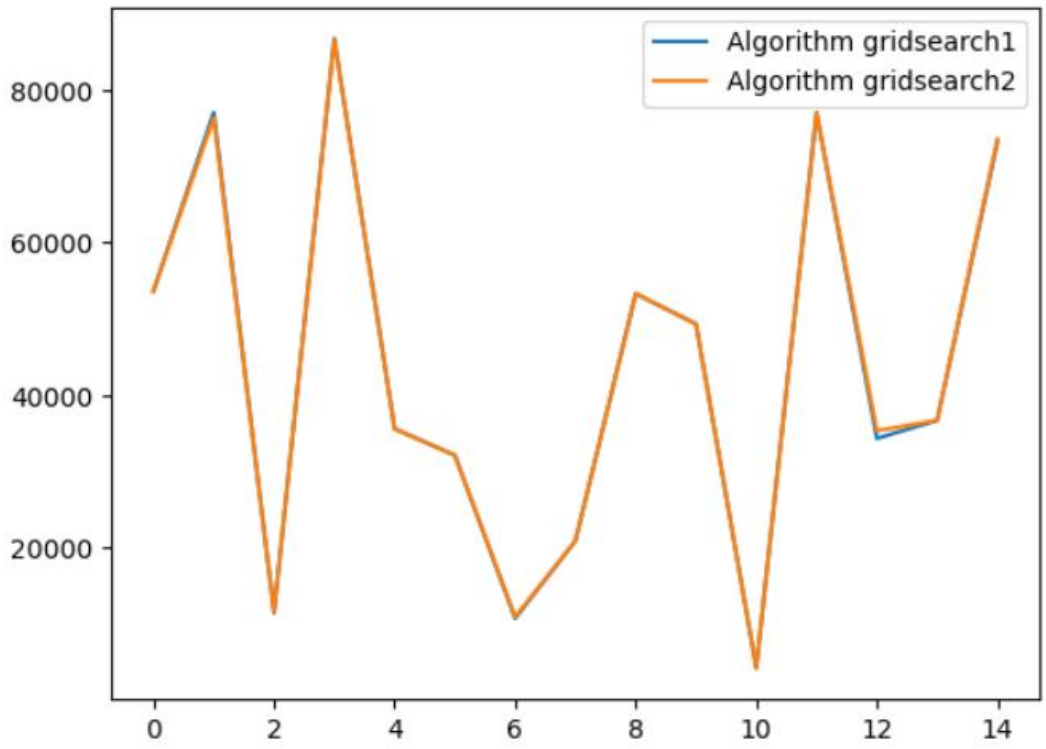

Figure 1. Plot with the best parameter combinations for both the 1st and 2nd gridsearch.

In Figure 1, we observe two fitness landscapes exhibiting similar behaviour. Both these landscapes have a lot of peaks (local optima / optimal solutions) and valleys (poor solutions), which indicates that the landscapes are rugged.

However, despite these similarities, the fitness values from the second grid search more often than not surpass those from the first grid search. This is evident in the plot, where the orange color representing the second grid search can be seen above the blue color representing

the first grid search more times than below. This indicates that the second grid search generally yielded better fitness values, suggesting it was more effective in optimizing the parameters.

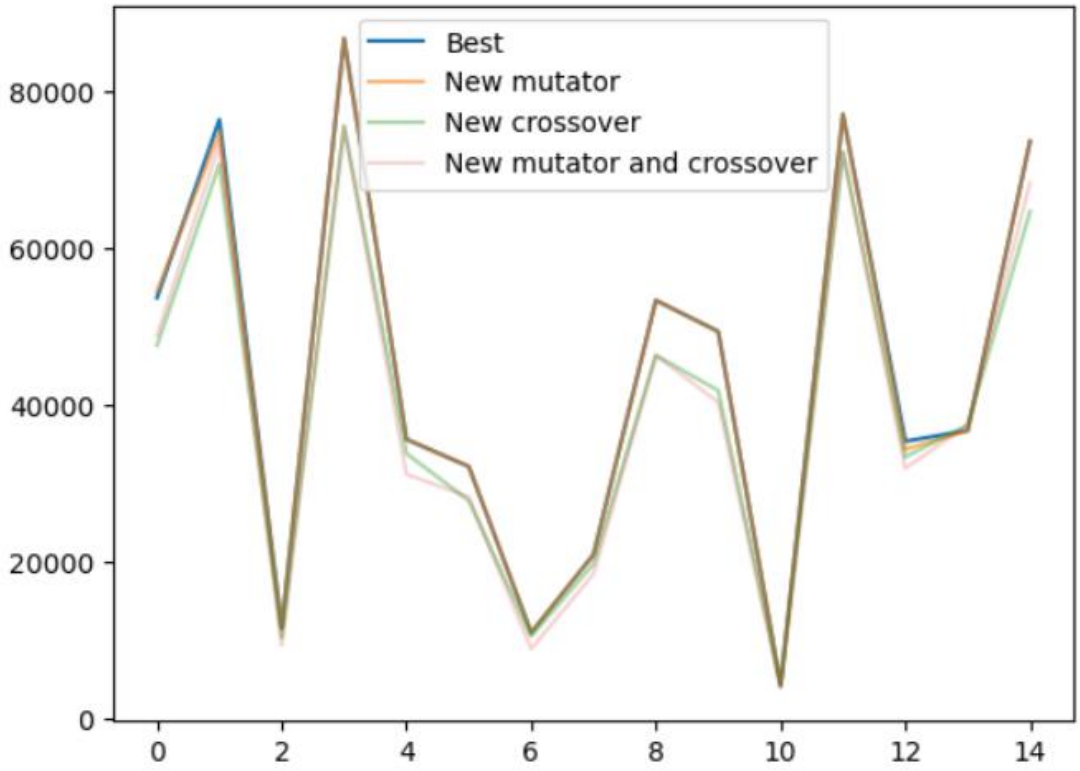

Figure 2. Plot with original best solution and possible best solutions (with new parameters)

We developed an additional crossover (sequential constructive crossover (scx)) and a mutator (two-opt mutation). However, due to time constraints, we couldn't run a grid search that included these new parameters. Instead, we opted to test whether replacing the existing mutator/crossover in our final model with these new ones would yield improvements.

We acknowledge that this approach may not yield the optimal results for the new mutator/crossover, as they might perform better with a different combination of parameters (Ex: the new mutator/crossover could perform better with a different population size compared to population size of the final solution).

Nevertheless, we decided to use them as an attempt to enhance the final model. If the results improved, we retained the new, better-performing model; otherwise, we reverted to the original results obtained from the grid search.

In Figure 2, we have a total of 4 possible solutions (parameter combinations):

- **Best**: the original best combination of parameters (2nd grid search output).

- **New mutator**: using the same parameters as in Best, and only changing the mutator from ‘swap_mutation’ to ‘two-opt mutation’.

- **New crossover**: using the same parameters as in Best, and only changing the crossover from ‘position_based_crossover’ to ‘scx’.

- **New mutator and crossover**: using the same parameters as in Best, and only changing the mutator and crossover from ‘swap_mutation’ and ‘position_based_crossover’ to ‘two-opt mutation’ and ‘scx’, respectively.

Observing Figure 2, we see a total of four fitness landscapes with relatively similar behaviors. However, clear differences emerge among them: the fitness values for the ‘New crossover’ and ‘New mutator and crossover’ are significantly lower compared to ‘Best’ and ‘New mutator’.

The latter two landscapes have very similar values, so much so that the plot shows a new color (brown) resulting from the overlap of blue (representing ‘Best’) and orange (representing ‘New mutator’). In some regions, the blue color is more prominent, indicating that the fitness values for the ‘Best’ landscape are slightly higher.

This suggests that the original parameter combinations (‘Best’) yield better fitness values overall compared to the remaining parameter combinations, and therefore represent the most optimal solution.

NOTE: In the annex, we included two additional plots. Notably, in *Figure 4*, we can observe a significant drop in fitness occurring at seed 10.

Annex

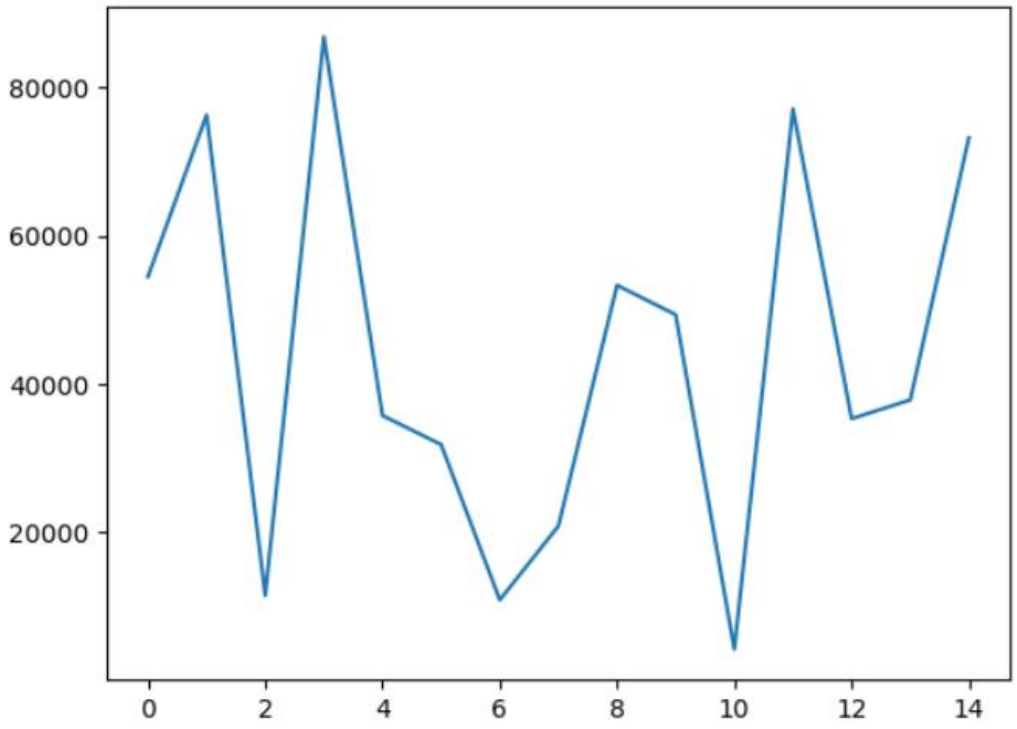

Figure 3: we ran the best solution for 15 different datasets.

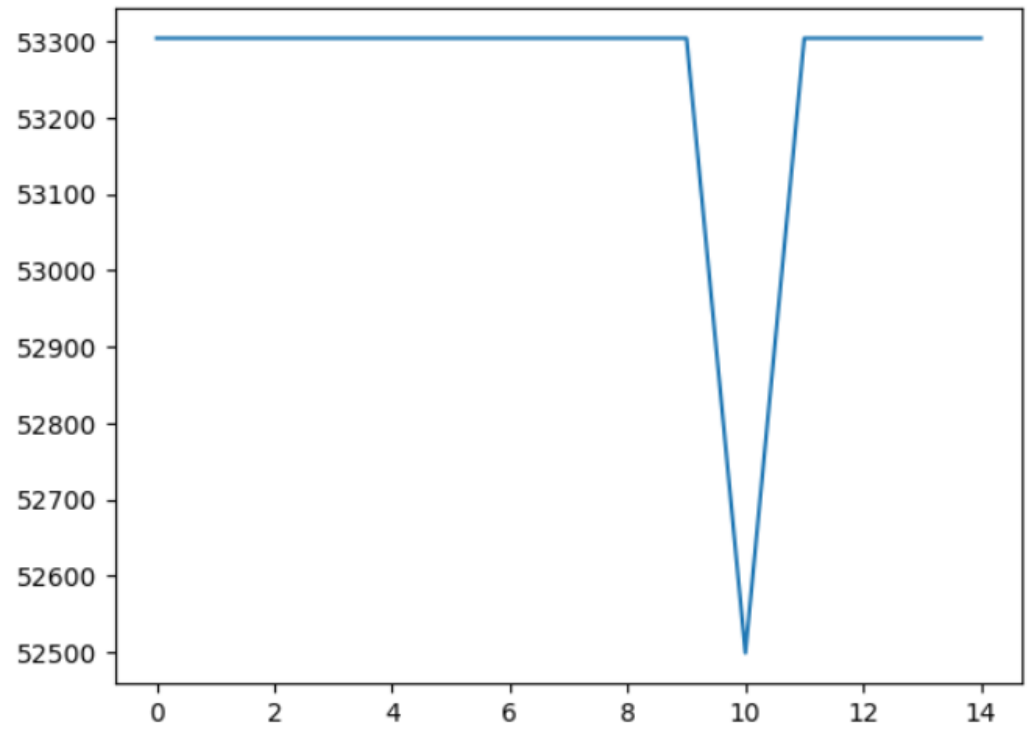

Figure 4: we ran the best solution in 1 dataset, for 15 different seeds.

References

1. https://www.hindawi.com/journals/cin/2017/7430125/
2. https://elearning.novaims.unl.pt/pluginfile.php/220325/mod_resource/content/1/03_Genetic-Algorithms.pdf
3. https://elearning.novaims.unl.pt/pluginfile.php/220330/mod_resource/content/1/Vanneschi-Silva-Book.pdf
4. https://www.researchgate.net/publication/236026740_AN_EFFICIENT_CROSSOVER_OPERATOR_FOR_TRAVELING_SALESMAN_PROBLEM
5. https://arxiv.org/pdf/1203.3099