Tackling the Traveling Salesman Problem with Genetic Algorithms

Mauro Vázquez Chas
Dániel Mácsai

2024. 11. 10

Abstract

Abstract goes here.

Contents

1 Introduction

The Traveling Salesman Problem (TSP) is a classic combinatorial optimization problem that has been extensively studied in the field of mathematics and computer science. The problem is defined as follows: given a set of cities and the distances between them, the goal is to find the shortest possible route that visits each city exactly once and returns to the starting city. The TSP is an NP-hard problem, meaning that there is no known polynomial-time algorithm that can solve it exactly for all instances. In this sense, algorithms such us Genetic Algorithms (GAs) have been proposed as a way to find approximate solutions to the TSP.

1.1 Objectives

The primary objective of this research is to investigate the effectiveness of Genetic Algorithms (GAs) in solving the Traveling Salesman Problem (TSP) and to identify optimal parameter configurations that lead to improved performance. Specifically, we aim to:

1. **Implement** a GA framework to solve the TSP.
2. **Experiment** with different genetic operators (crossover and mutation) and parent selection methods.
3. **Tune** hyperparameters to optimize the GA's performance.
4. **Evaluate** the performance of the GA on a benchmark TSP dataset.
5. **Analyze** the impact of different parameter configurations on the solution quality and computational cost.

1.2 Scope

The scope of this research is limited to the application of GAs to the TSP. We will focus on the following aspects:

- **Problem Representation**: We will use a permutation-based representation to encode TSP solutions.
- **Genetic Operators**: We will implement and evaluate the performance of various crossover (e.g., Position-based Crossover, Order-based Crossover) and mutation (e.g., exchange mutation, insertion mutation, inversion mutation) operators.
- **Parent Selection**: We will explore different parent selection techniques, including random selection, roulette wheel selection, rank-based selection, and tournament selection.
- **Hyperparameter Tuning**: We will conduct experiments to determine the optimal values for hyperparameters such as population size, mutation rate, crossover rate, and number of generations.
- **Performance Evaluation**: We will evaluate the performance of the GA using the quality of the solutions obtained.

We will not delve into more complex variations of the TSP, such as the Vehicle Routing Problem or the Capacitated Vehicle Routing Problem. Additionally, we will not explore other metaheuristic algorithms or hybrid approaches that combine GAs with other techniques.

2 Methodology

2.1 Problem Representation

In this work we employed the Pittsburgh view, where each individual in the population represents a complete solution to the TSP. The representation of the problem is a permutation of the cities, where each city is visited exactly once. For example, for a set of cities $\{1, 2, 3, 4\}$, solution $\{1, 3, 2, 4\}$ would represent the path in figure 1.

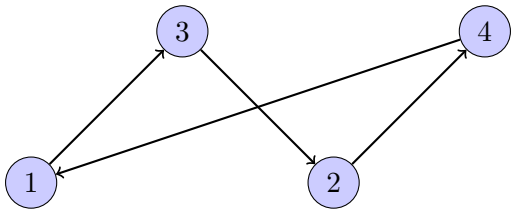


Figure 1: Path representation of a TSP solution with nodes 1, 3, 2, and 4.

We considered different approaches on the representation of each chromosome, either binary code or integer code. Using binary code, each solution would be a string of bits, with each city being a certain number of them. There are variations that use a different encoding mechanism, for example gray encoding. The integer code is a list of integers, where each city is represented by an integer. In the end, we decided to use integer code (called *path representation* in [4]), as it is more intuitive and in binary code, the number of bits used for each city could not be divided. This decision was also influenced by the research made in [4], where the authors explicitly state the following about binary code:

> This representation might be useful for small problem instances of the TSP. However, for larger problem instances the binary strings which represent the tours become unmanageably large. Another disadvantage of the binary representation is that the classical operators do not necessarily result in legal offspring tours; repair algorithms would be necessary.

In this sense, we are not doing what it is often called canonical or simple genetic algorithm (SGA) as it is described in [3], where the representation is binary. It would be more accurate to say we are using Non-Canonical GAs with integer code representation for each permutation, as they are described in Chapter 17 of [1]. We think that this approach is more suitable for the TSP, as it allows for a more direct representation of the problem and avoids the need for additional repair mechanisms.

2.2 An overview on Genetic Algorithms

Genetic Algorithms are, perhaps, the most known evolutionary algorithm. They were pioneered by John Holland in the 1970s, and have since undergone extensive research and numerous adaptations.

The traditional genetic algorithm, according to [3], uses bit-string representation and works in the following way: given a population of μ individuals, we perform the following steps for a certain number of generations:

1. Select μ parents from the population, allowing duplicates
2. Shuffle the intermediary population and apply crossover and mutation operators to create new individuals
3. The intermediary population replaces the old population.

The selection of the parents is often performed using a selection operator that often requires a fitness function evaluating the quality of the individuals. Classic operators for population modification include bit-flip mutation and one-point crossover.

2.3 Parent Selection Methods

Parent selection is a crucial step in the genetic algorithm, as it determines the genetic material that will be passed on to the next generation. In our case, we employed some of the most common mechanisms, according to the material proposed in the lectures [2], and adapting them to the usage of the distance function instead of the fitness function (slightly different as we want to minimize it). The methods used were:

- Random selection: parents are selected randomly from the population
- Roulette wheel selection: the probability of selecting an individual is proportional to its fitness (in our case, inversely proportional to the distance function)
- Rank roulette wheel selection: similar to the previous one, but the probability of selection is proportional to the rank of the individual (in our case, inversely proportional to the rank of the distance function)
- Tournament selection: a random subset of the population is selected, and the best individual from this subset is chosen as a parent. We used subsets of size 3 and 5.

2.4 Crossover Techniques

In genetic algorithms, crossover operators play a crucial role in combining the genetic information of two parent solutions to produce offspring. These techniques simulate the natural process of reproduction, where the genetic material of two parents is mixed to create offspring that inherit traits from both parents. The goal of the crossover operator is to generate new solutions that inherit the best features of the parents while maintaining diversity in the population. In this section, we describe two widely used crossover operators: the **Position-based Crossover (POS)** and **Order-based Crossover (OX1)** operators as these had the best result in [4].

2.4.1 Position-based Crossover (POS)

The **Position-based Crossover** (POS) operator is particularly useful when dealing with permutation-based problems, such as the **Traveling Salesman Problem (TSP)** or other problems where the solution involves a sequence of items or locations. In POS, the crossover is performed by selecting specific positions (or indices) from the parent chromosomes and transferring them to the offspring.

Implementation of POS Crossover:

1. **Input:** Two parent solutions, `Parent1` and `Parent2`, both represented as permutations of the same set of elements (e.g., cities in the TSP).
2. **Process:**
    - Randomly select a subset of positions (indices) from `Parent1`. These positions will be inherited by the offspring.
    - Copy the elements from `Parent1` at the selected positions to the corresponding positions in the offspring.
    - For the remaining positions, fill them with the elements from `Parent2`, ensuring that no element is repeated. This is done by copying the elements of `Parent2` that do not already exist in the offspring.
    - We then repeat this process for the second offspring by swapping the roles of `Parent1` and `Parent2`.

Example: Suppose we have the following parents:

$$\text{Parent1: } [4, 1, 3, 5, 2] \quad \text{and} \quad \text{Parent2: } [2, 4, 1, 3, 5]$$

A possible crossover operation could be:

1. Randomly select a subset of positions from `Parent1`. For example, positions 1 and 3 (the corresponding elements are [4, 3]).
2. Copy the elements at the selected positions from `Parent1` to the offspring. Now the partially filled offspring looks like this: [4, _, 3, _, _].
3. Fill the remaining positions with the elements from `Parent2` that are not already present in the offspring.
4. Final `Offspring 1:` [4, 2, 3, 1, 5].
5. For the second offspring, we use the same positions, the corresponing elements are [2, 1], so the offspring will look like this: [2, _, 1, _, _].
6. After filling it in from `Parent 1`, we end up with the final `Offspring 2:` [2, 4, 1, 3, 5].

Role in Genetic Algorithm: POS crossover ensures that the offspring inherits a specific order of cities from `Parent1` while incorporating diversity by filling the remaining positions from `Parent2`. This technique is particularly effective for permutation-based problems as it preserves the relative ordering of elements while allowing diversity in the offspring population.

2.4.2 Order-based Crossover (OX1)

The **Order-based Crossover (OX1)** operator is another permutation-based crossover technique often used for solving combinatorial optimization problems. It is designed to preserve the relative ordering of elements from the parents while preventing duplicates in the offspring. OX1 works by selecting a subsequence of elements from one parent and then copying the remaining elements from the other parent in the same order they appear.

Implementation of OX1 Crossover:

1. **Input:** Two parent solutions, `Parent1` and `Parent2`, both represented as permutations.
2. **Process:**
   - Randomly select a continuous subsequence of elements from `Parent1`. This subsequence will be retained in the offspring.
   - For the remaining positions, copy the elements from `Parent2` in the order they appear, while ensuring that no element is duplicated in the offspring. This can be done by skipping over any element that is already present in the offspring.
   - The second offspring is generated by swapping the roles of `Parent1` and `Parent2`, i.e., selecting a subsequence from `Parent2` and filling in the remaining positions with elements from `Parent1`.

Example: Given the parents:

$$\text{Parent1: } [4, 1, 3, 5, 2] \quad \text{and} \quad \text{Parent2: } [2, 4, 1, 3, 5]$$

Suppose we randomly select a subsequence from `Parent1`, such as positions 2 to 4 (i.e., [1, 3, 5]). Now, the offspring will look like this:

1. `Offspring 1(partially filled):` [_, 1, 3, 5, _] (copy positions 2 to 4 from `Parent1`).

2. Copy the remaining elements from `Parent2` that are not yet in the offspring. The remaining elements from `Parent2` are [2, 4], which will fill the empty slots.

3. Final `Offspring 1:` [2, 1, 3, 5, 4].

4. For the second offspring, we select the same positions as before, the corresponding elements are [4, 1, 3], so the offspring will look like this: [_, 4, 1, 3, _], and after filling it, the final `Offspring 2` is [5, 4, 1, 3, 2].

Role in Genetic Algorithm: The **OX1 crossover** is particularly effective in maintaining the order of cities in the offspring, which is critical for problems like the TSP where the relative order of cities is important. By ensuring no duplicates and preserving order, OX1 allows for the effective exploration of the solution space while retaining useful genetic material from both parents. This leads to higher-quality offspring in the evolutionary process.

2.4.3 Summary

Both the **Position-based Crossover (POS)** and **Order-based Crossover (OX1)** operators are great tools in solving permutation-based optimization problems like the TSP. These operators ensure that the offspring inherit useful traits from their parents while maintaining diversity and preventing duplicates. The role of these crossover techniques in a genetic algorithm is to generate high-quality solutions by combining the best features of the parent solutions and driving the algorithm towards the optimal solution. We see that basically the POS operator is a more general version of the OX1 operator, as it allows for more flexibility in the selection of the subsequence to be inherited from the first parent. At this point it is not clear if allowing more generality will result in better solutions or not, we will have to test it.

2.5 Mutation Operators

The mutation operator is a crucial part of the genetic algorithm, as it introduces diversity into the population. For further details, we referred to [4]. In the end, in our implementation, we used the following mutation operators:

- **Exchange mutation**: two random cities are arbitrarily swapped in the chromosome
- **Insertion mutation**: a random city is removed and reinserted at another random position in the chromosom
- **Inversion Mutation (IVM)**: A subset of the chromosome (i.e., a segment of the tour) is reversed. This mutation is effective in localizing improvements by reversing part of the tour, which may reduce the total travel distance. The **Inversion Mutation Algorithm (IVM)** follows these steps:

  1. **Select the random subset**: Two indices, i and j, are selected randomly such that $i < j$. These indices define the boundaries of the subset to be inverted. We now remove this subset, and reverse it.
  2. **Replace the original segment**: The reversed segment is placed back into the chromosome into a randomly chosen position.

  This mutation operator is effective in the **Travelling Salesman Problem (TSP)** because it can significantly alter the tour by reversing part of the path, which could potentially reduce the total travel distance. It is particularly useful when the cities in the selected subset are in an order that is suboptimal, as reversing them can yield better results.

Each of these mutation techniques introduces diversity into the population, ensuring that the genetic algorithm does not get stuck in local optima and can continue to evolve towards better solutions.

2.6 Elitism

Elitism is a mechanism that ensures that the best individuals from the current population are passed on to the next generation. This helps maintain the best solutions found so far and prevents the degradation of chromosome quality. In our implementation, we considered GA configurations without elitism and with elitism, both keeping the best individual and the three best individuals from the previous generation.

2.7 Other Considered Approaches

2.7.1 Fitness Function

As we already noted, we employed the total distance of the path as the value to minimize. However, the common approach would be to use a fitness function, where better chromosomes have higher values. In this sense, we could have used any monotonic decreasing function of the distance, for example, the reciprocal of the distance or a decaying exponential function. This would have allowed us to use the classic selection operators without modifications. However, we decided to use the distance directly, as it is more intuitive and straightforward.

2.7.2 Population Replacement

In this work we used a generational approach, where the entire population is replaced at each generation (μ, λ), with $\mu = \lambda$. However, we also considered a steady-state approach where only a subset of the population is replaced at each generation. Ultimately we decided not to do this, as it would have required a more complex implementation and in words of [3]: "It can also lead to premature convergece as the population tends to rapidly focus on the fittest member".

A useful follow-up of this work would be to implement a steady-state approach by inverse tournament selection as it doesn't remove as much genetic diversity, due to its innate randomness.

Even though we described our implementation as $(\mu, \lambda), \mu = \lambda$, it could be seen as a $(\mu, \lambda), \mu > \lambda$ approach when we make use of elitism, as we keep the best individuals from the previous generation: $\mu = \lambda + \text{elitism}$.

2.7.3 Population Initialization

We evaluated different ways to initialize the population, including random initialization (the one we opted for) and heuristic initialization. This last one, refers to the usage of a heuristic algorithm to generate the initial population. This could be useful for large instances of the TSP, as it would provide a good starting point for the genetic algorithm. The practice is commented in Chapter 13 of [1], where the authors also explore the hybridization of evolutionary strategies with domain specific knowledge and algorithms, highlighting the good results of the available literature.

However, we decided not to implement this, as it would require the usage of a heuristic algorithm and other techniques that are out of the scope of this work. Having said this, it would be interesting to further explore this.

2.7.4 Other Operators

We also considered multitud of other mutation, crossover and parent selection techniques, but in the end we didn't implement them to keep the grid search manageable.

3 Implementation

3.1 Dataset

In this work, we consider a dataset of 29 cities from Bavaria, Germany. The dataset contains the coordinates of each city, as well as the distances between them (provided in a distance matrix). The dataset (called *bays29.tsp*) is available at the following link: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp/

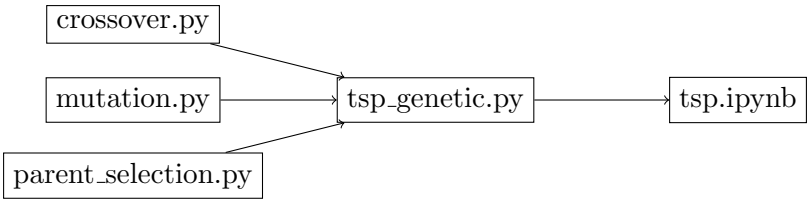


Figure 2: Code Structure Diagram

3.2 Code Structure

One of our main goals, was to create a flexible and modular implementation that allows for easy experimentation with different configurations and operators. To achieve this, we used Python as the main programming language, and we implemented the parent-selection methods, crossover operators, and mutation operators as separate modules. This design allows us to easily swap out different operators and configurations without changing the core logic of the genetic algorithm. Aside from this, the class that contains the genetic algorithm is also modular, allowing for easy extension and modification. All of these files, can be found in the *src* folder of the project.

Another clear target of ours, was to code (almost) everything from scratch. For this reason, we only used the following libraries: numpy, random, matplotlib, typing, logging, json, datetime, and pandas.

To use all of these modules, we set up a jupyter notebook file *tsp.ipynb* that contains the data reading function, the grid search function and most graph plotting cells (see 2). In this notebook, we also wrote a *Usage Example* section, where we show how to use the genetic algorithm class and all of its parameters, although if more insight is needed, all the functions and classes are documented in their respective docstrings.

4 Discussion of Results

4.1 Experimental Setup

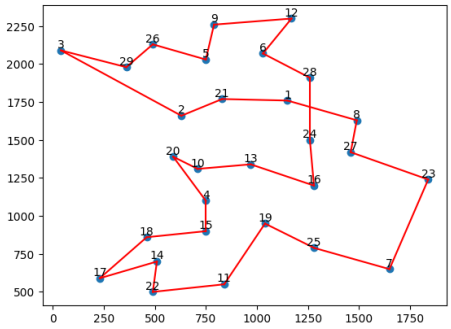
In order to evaluate the influence of each hyperparameter on the performance of the genetic algorithm, we conducted a grid search over a range of values for each parameter. The hyperparameters considered in the grid search were (3240 total configurations):

- **population_size**: [50, 200], **elitism**: [0, 1, 3], **generations**: [100, 200]
- **m_rate**: [0.01, 0.05, 0.1], **c_rate**: [0.6, 0.8, 1]
- **select_parents**: {tournament, roulette, rank_roulette, random}
- **crossover**: {POS, OX1}, **mutation**: {exchange, insertion, IVM}
- **tournament_size**: [3, 5]

4.2 Best Configuration

The best distance achieved was **2055.0**, using parameters:

- **population size**: 200
- **elitism**: 1, **generations**: 200
- **mutation rate**: 0.1, **crossover rate**: 1
- **tournament selection** with a **tournament size**: 5
- **crossover**: POS, **mutation**: insertion

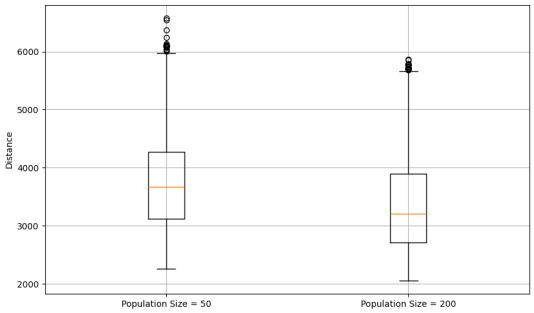


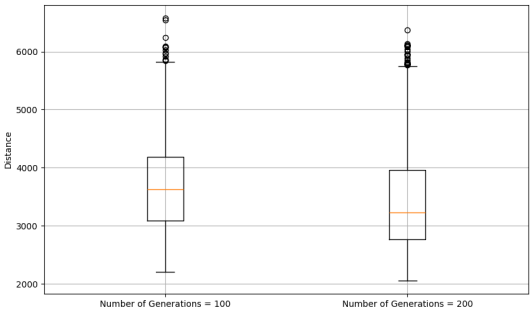
4.3 Performance Analysis

To be able to measure the objective performance of the GA while tackling TSP, we reasoned that it would be useful to compare it to the optimal solution. For this, the brute force solution is certainly not feasible, so we searched for already existing implementations of algorithms. We found that the Concorde TSP solver is one of the best tools for this task. We used the Python wrapper *concorde* to solve the TSP for the same dataset, but the runtime for the algorithm was too long, so we had to stop it. We also tried other methods and different LP solvers to get the exact optimal result, but unfortunately we did not succeed. This shows the difficulty of TSP and the need for approximate solutions: the GA was able to find a solution with a distance of 2055. We checked by hand that by changing two edges in the path we can improve this by 21 and by searching online (since this is a well known dataset) we found that the optimal distance is 2020. This means that the GA found a solution that is 1.73% worse than the optimal solution. This is a good result, as the GA is a heuristic algorithm and it is not guaranteed to find the optimal solution.

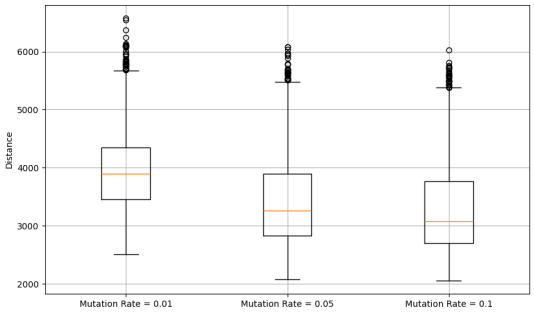
4.4 Hyperparameter Analysis

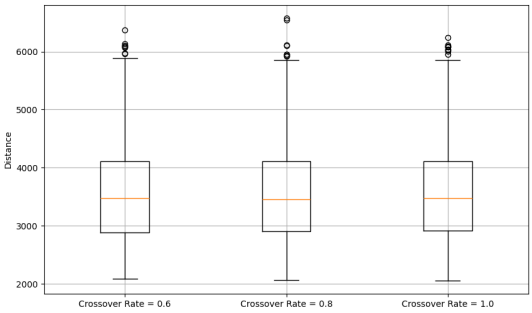
In this section, we will analyze the boxplots of the results obtained from the grid search. Each graph will present the results of every considered configuration segmenting by hyperparameter possibility.



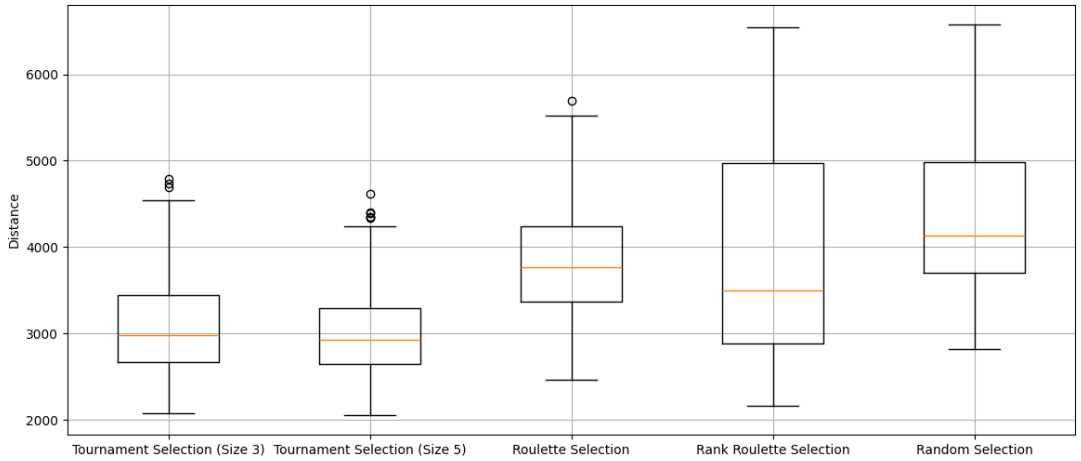


On the first graph, we can see pretty intuitive results: the larger the population size and the more amount of generations, the better the results. This fact is also supported by the best configuration found and the literature consulted. In the case were we apply elitism, this would be evident as the metrics for each generation can only improve.

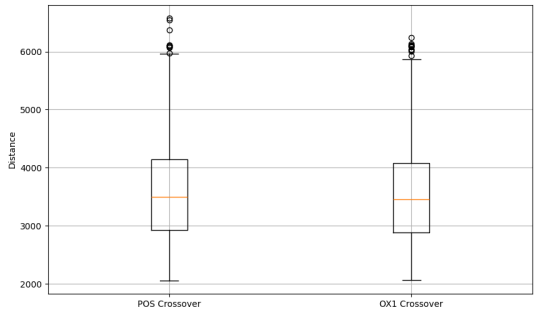


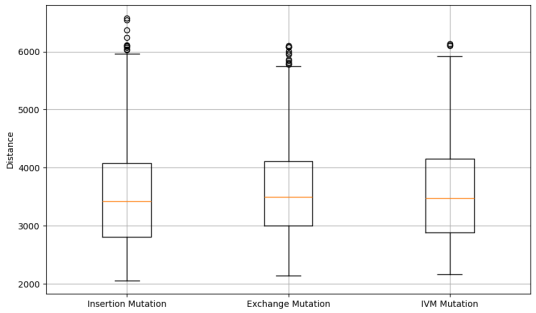


The results shown on the second graph are not so conclusive. Based on the options provided, we observe (slightly) that the larger the mutation rate, the better the results. This showcases the importance or the diversity provided by the mutation operator. It would be interesting to see if an even larger value would be worse.



In the case of the crossover rate, the results are even less decisive. All the options provide similar results, although the one present in the best configuration is crossover rate 1. This is not what we expected, as in the GA section of [3] it states that a common recommendation is to use a crossover rate between 0.6 and 0.8. We could assess this to the fact that we are not performing a canonical GA, but either way, the difference is not dramatic.

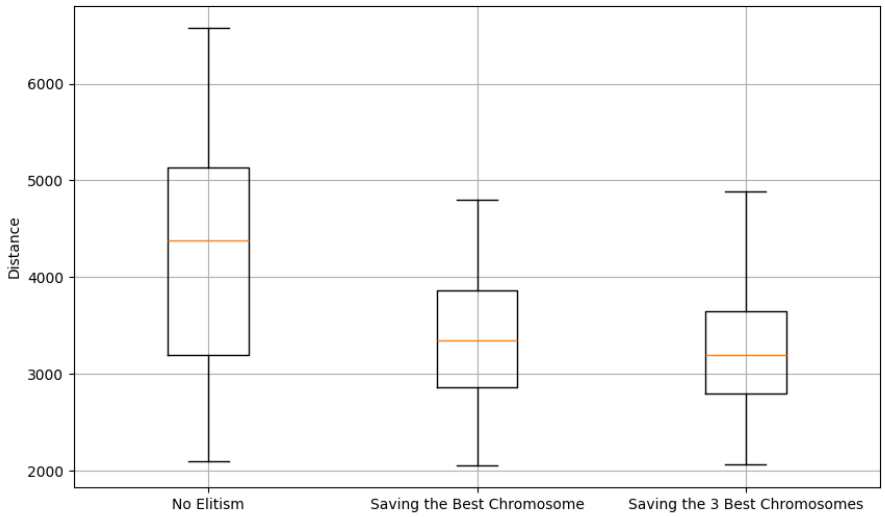




The graph regarding parent selection methods is the one with most options and offers a lot of insight. In the first place, we see a big gap in performance between the random selection method and the other three, as expected. On the other hand, we see a non-negligible difference between rank-roulette and normal roulette selection. The better performance of roulette selection could be due to it being more sensible to distance variation than rank-roulette selection. Finally, we notice that both tournament selection set-ups are better, in terms of median and standard deviation, than the other methods, having even better results with tournament size 5 instead of 3. This surprised us at first, and we still don't have a clear explanation for it. Tournament selection also mimics in a more faithful way the real world as it follows some sort of survival of the fittest.

4.5 Discussion

Interpret the results, discussing their implications and any observed patterns.



5 Conclusion

5.1 Summary

Summarize the key findings and contributions of the project.

5.2 Future Work

Suggest potential areas for future research or improvements.

5.3 Notes on the usage of Generative AI

Here we describe our usage of Generative AI during the project:

- While coding, we used Copilot to generate the docsstrings for the functions. This was useful to have a structured documentation and to keep track of the parameters and return values of each function.
- When writing the report, we used GPT o1 to generate the latex template and later to check for grammatical mistakes.

References

[1] Thomas Bäck, David Fogel, and Zbigniew Michalewicz. *Evolutionary Computation 2—Advanced Algorithms and Operators*. Jan. 2000. DOI: 10.1201/9781420034349.

[2] Lluís A. Belanche. *Lecture slides in Computational Intelligence: Introduction to Genetic Algorithms*. 2024.

[3] A. Eiben and Jim Smith. *Introduction To Evolutionary Computing*. Vol. 45. Jan. 2003. ISBN: 978-3-642-07285-7. DOI: 10.1007/978-3-662-05094-1.

[4] Pedro Larranaga et al. "Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators". In: *Artificial Intelligence Review* 13 (Jan. 1999), pp. 129–170. DOI: 10.1023/A:1006529012972.

A Appendix: Class Documentation

A.1 Genetic Algorithm Class

The core of our implementation is the *tsp_genetic.py* module, which contains the *TSP_Genetic* class. This class is responsible for running the genetic algorithm, given a set of parameters and operators. The class contains the following methods:

- **__init__**: Initializes the genetic algorithm with the given hyperparameters:
  - **generations**: Number of generations to simulate.
  - **print_results**: Whether to print algorithm results.
  - **print_rate**: Frequency of progress printing.
  - **m_rate**: Mutation rate, probability of mutation in offspring.
  - **c_rate**: Crossover rate, probability of crossover between parent chromosomes..
  - **select_parents**: Parent selection method.
    - **random_selection**
    - **roulette_selection**
    - **rank_roulette_selection**
    - **tournament_selection**
  - **tournament_size**: Tournament size (just for tournament selection).
  - **crossover**: Crossover method for parent chromosomes.
    - **POS**
    - **OX1**
  - **crossover_call**: Arguments for the crossover method.
  - **mutation**: Mutation method for offspring chromosomes.
    - **exchange**
    - **insertion**
    - **IVM**
  - **elitism**: Number of best chromosomes retained in the next generation.
  - **results_path**: Path to save path_distance statistics.
  - **save_results**: Save path_distance statistics to a file.
- **run**: Runs the genetic algorithm for the specified initial population, cities and matrix of distances.
  - **population**: The initial population of chromosomes (list of lists).
  - **cities**: The list of positions of each city. Each row represents a city, with the first element as its x coordinate and the second as its y coordinate.
  - **distances**: A matrix representing the distances between each pair of cities.

- **_path_distance**: Given a chromosome provides the total path distance.
- **path_distance_population**: Given a population, returns an array with the total path distance of each chromosome.
- **plot_route**: Given a chromosome, plots its route.

A.2 Crossover Class

The *crossover.py* module contains the *Crossover* class, which is responsible for implementing the crossover operators. The class contains the following methods:

- **POS**
- **OX1**
  - **first_cut** (int, optional): The starting index for the crossover. If None, it uses the default value.
  - **second_cut** (int, optional): The ending index for the crossover. If None, it uses the default value.

The OX1 method is designed to be adjustable through hyperparameters, which can be specified in the class's *__init__* method using the *crossover_call* parameter. For example, if we intend to use the "OX1" operator, setting its first and second cut value to the 4th positions counting from each end, we would introduce in the *TSP_Genetic* instantiation: *crossover_call = "(4, cities.shape[0] - 4)"*.

A.3 Mutation Class

The *mutation.py* module contains the *Mutation* class, which is responsible for implementing the mutation operators. The class contains the following methods:

- **exchange**
- **insertion**
- **IVM**