Warsaw University of Technology
Faculty of Electronics and Information Technology
Specialization: Computer Science
Course: Introduction to Artificial Intelligence

PROJECT DOCUMENTATION

Genetic Algorithm

Author:
Monika Jung

Student ID:
331384

Project Description

The goal of this project is to implement a **genetic algorithm** that solves the classical **Traveling Salesman Problem (TSP)**, which involves finding the shortest route that visits each given city exactly once and returns to the starting point.

The genetic algorithm in this project uses the following operations:

- **Roulette Wheel Selection**,
- **Single-Point Crossover**,
- **Mutation**
- **Generational Succession**.

Project Structure

1. Cities Dictionary

This dictionary contains the coordinates (x, y) of 10 cities represented by keys ("A", "B", "C", etc.). The coordinates represent the position of each city in a 2D space.

```
cities = {
    "A": (0, 0), "B": (1, 3), "C": (2, 1), "D": (4, 6), "E": (5, 2),
    "F": (6, 5), "G": (8, 7), "H": (9, 4), "I": (10, 8), "J": (12, 3)
}
```

11. `dictionary_matrix`

In order to optimize the calculation of distances between cities, we can create a matrix that stores the distances between every pair of cities in the problem. This allows for constant time retrieval of distances, instead of recalculating them each time. The matrix is represented as a dictionary of dictionaries, where each key corresponds to a city, and the value is another dictionary containing the distances to all other cities.

2. `calculate_distance(city1, city2)`

This function calculates the distance between two cities, `city1` and `city2`, using their coordinates.

3. `total_distance(route)`

This function calculates the total distance of a given route, which is a list of cities. It iterates through the cities in the route, adding the distance between consecutive cities, including the distance between the last city and the first one to form a closed loop.

4. `fitness(route)`

The fitness function is used to evaluate how "good" a given route is. The fitness is calculated as the inverse of the total distance, meaning shorter routes have higher fitness values.

5. `initialize_population(size)`

This function initializes the population by generating `size` random routes (permutations of the cities). It returns a list of routes as the initial population for the genetic algorithm.

6. Selection Methods

Two selection mechanisms are implemented:

- **Roulette Wheel Selection**: Individuals are selected based on their proportional fitness values, ensuring that fitter individuals have a higher probability of being chosen.
- **Tournament Selection**: A subset of the population competes, and the best-performing individual is selected for reproduction.

7. `single_point_crossover(parent1, parent2)`

The single-point crossover function takes two parent routes and generates a child by combining parts of both parents. The child inherits a part of the first parent up to a random crossover point and the rest from the second parent, ensuring all cities are included.

8. `mutate(route, mutation_rate=0.1)`

This function introduces a mutation in a route. With a probability determined by `mutation_rate`, two cities in the route are selected randomly and swapped to create a new route.

9. `genetic_algorithm(population_size=100, generations=500, mutation_rate=0.1)`

This is the main function that runs the genetic algorithm. It initializes the population, performs crossover, mutation, and selection over multiple generations, and returns the best route found and its total distance.

How the Genetic Algorithm Works

1. **Initialization**: The population is initialized with random routes.
2. **Selection**: We use either the roulette wheel selection or the tournament selection to choose two parents.
3. **Crossover**: A single-point crossover is applied to generate a child route.
4. **Mutation**: A mutation is applied to the child route with a given mutation rate.
5. **Generational Succession**: The new population replaces the old one, and the best route is selected.
6. **Termination**: The algorithm terminates after a specified number of generations.

Conclusions from the Plots

Effect of Population Size on Algorithm Performance

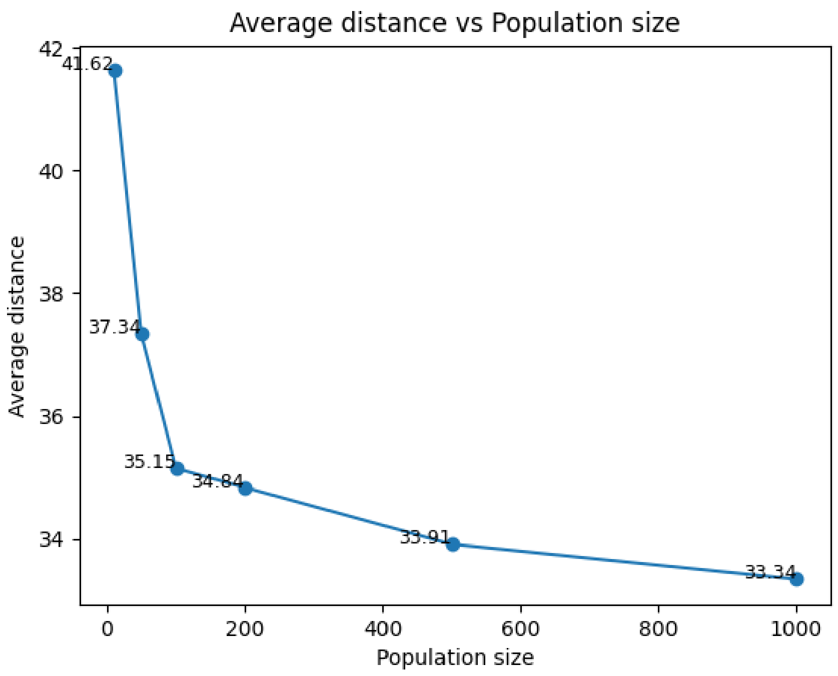


Figure 1: Effect of Population Size on Algorithm Performance.

This graph shows that larger population sizes improve performance, with decreasing average distances.
A population size of around 500-1000 gives the best results, while very small populations (e.g., 10 or 50) perform significantly worse.
However, the improvement slows down as the population increases beyond a certain point.

Effect of Generations on Algorithm Performance

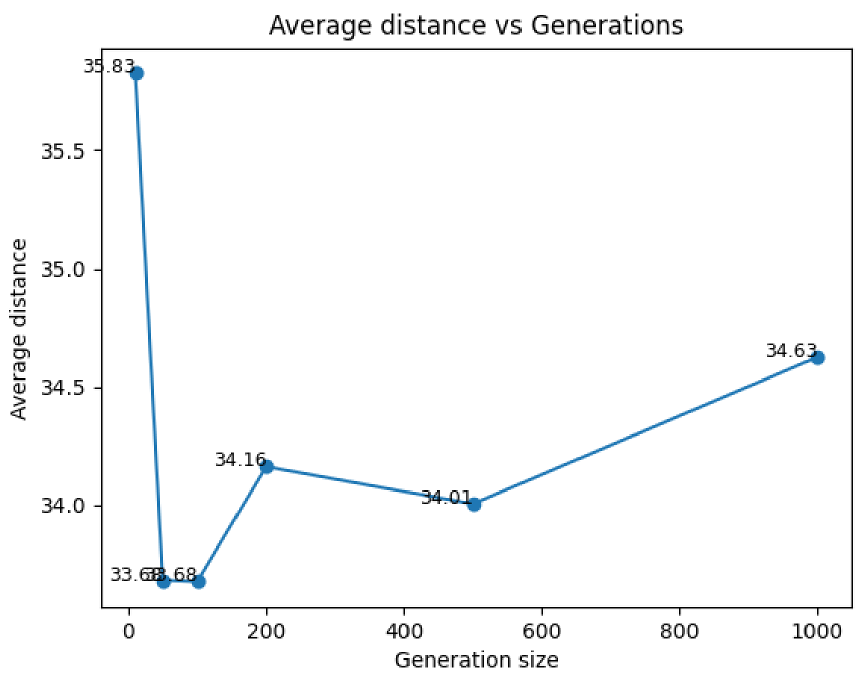


Figure 2: Effect of Population Size on Algorithm Performance.

The second graph suggests that more generations generally improve results, but only up to a certain threshold.
Some variations appear, possibly due to randomness, but the optimal number of generations seems to be around 50-100 for this problem.
There is diminishing improvement, meaning further generations do not significantly enhance the solution.

Effect of Mutation Rate on Algorithm Performance

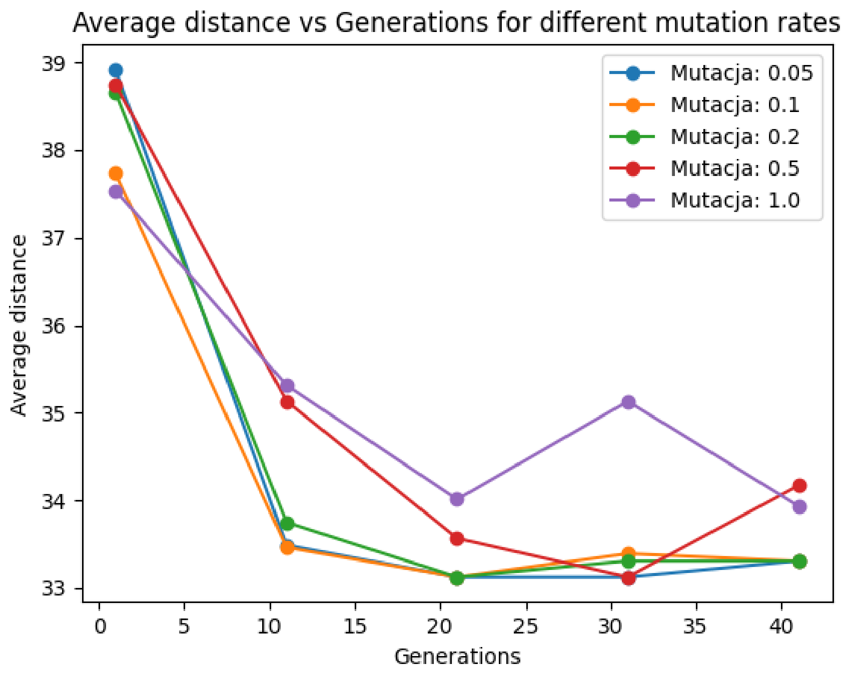
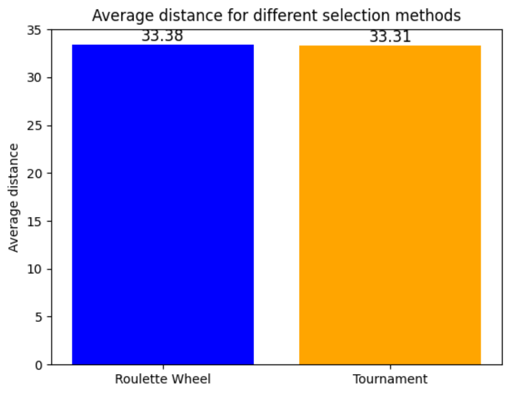
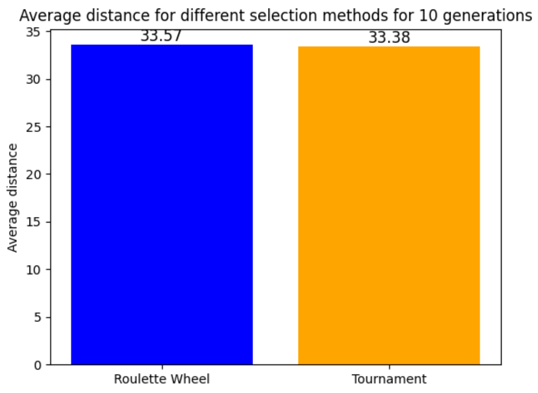


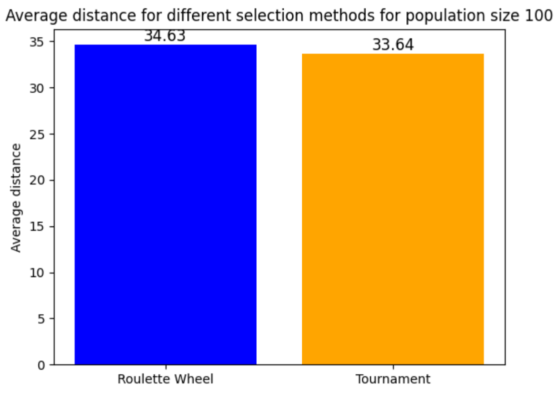
Figure 3: Effect of Population Size on Algorithm Performance.

The third graph shows how different mutation rates impact the average distance over generations.
Lower mutation rates (0.05, 0.1, 0.2) perform better, leading to faster convergence and better final results.
Higher mutation rates (0.5, 1.0) cause instability, making the solution fluctuate more and preventing proper convergence.

Comparison of Selection methods







The bar charts compare the average distances achieved using two different selection methods—Roulette Wheel and Tournament Selection—across different experimental conditions.

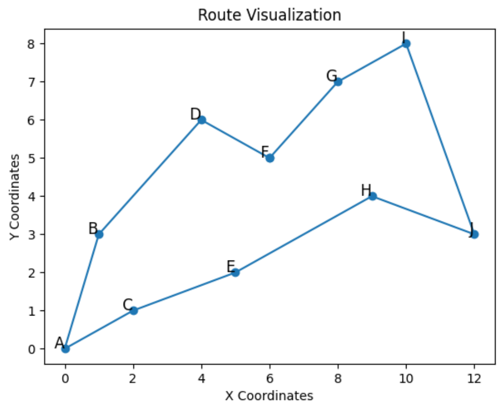
The key observations are:

- General Comparison (population = 500, generations = 50): Across all cases, Tournament Selection consistently results in a marginally lower average distance than the Roulette Wheel method, suggesting it may be a more effective selection approach for this problem.

- For 10 Generations: The average distance for both methods is very similar (33.57 for Roulette Wheel and 33.38 for Tournament Selection). Tournament Selection performs slightly better.

- For Population Size 100: The difference is more noticeable, with the Roulette Wheel selection yielding an average distance of 34.63, whereas Tournament Selection achieves 33.64. This suggests that Tournament Selection performs slightly better in smaller populations.

Conclusion



This genetic algorithm efficiently finds a near-optimal solution to the Traveling Salesman Problem by utilizing selection, crossover, and mutation operations. By experimenting with population size, mutation rate, and number of generations, the algorithm can be tuned to produce good solutions for various instances of the problem.