A4: Optimization with Genetic Algorithms

Student: SEBASTIAN BUZDUGAN
Professors: SERGIO GÓMEZ JIMÉNEZ, JORDI DUCH GAVALDÀ

Table of Contents

Introduction

The Traveling Salesman Problem (TSP) stands as a fundamental challenge in the realms of combinatorial optimization and computational theory. It poses a seemingly simple question: What is the shortest possible route that visits a list of cities once and returns to the origin? Despite its straightforward premise, the TSP quickly scales in complexity as the number of cities increases, epitomizing the category of NP-hard problems in computational studies.

In this project, I delve into the application of genetic algorithms to approach the TSP. We aim to explore and demonstrate the adaptability and efficacy of genetic algorithms in finding near-optimal solutions for TSP instances.

For a comprehensive analysis, I've selected a diverse range of datasets, each varying in the number of cities, to test the performance of our algorithm:

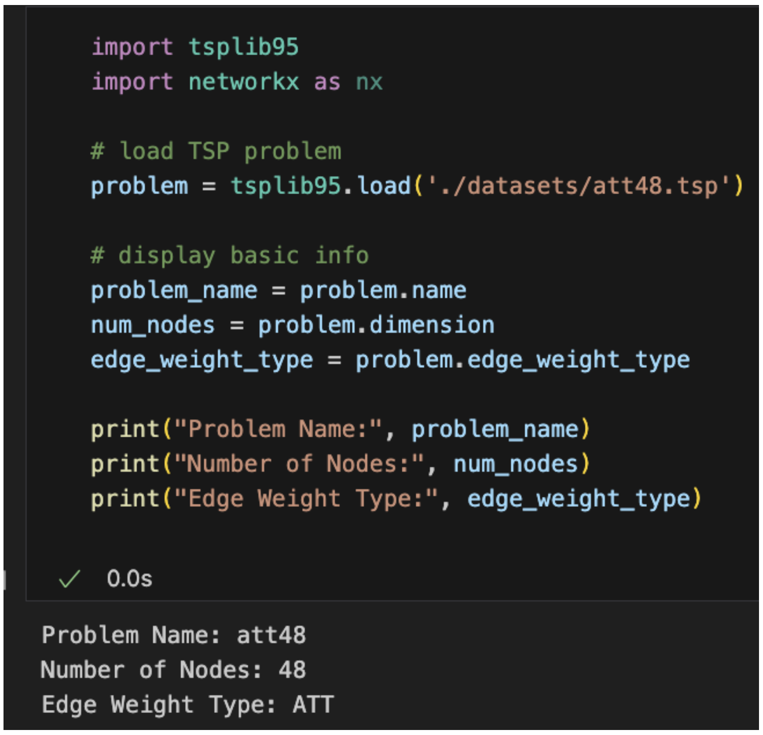
- **Five-City Problem (five.tsp):** A smaller dataset, ideal for understanding the basic behavior of the algorithm. (https://people.sc.fsu.edu/~jburkardt/datasets/tsp/five_d.txt)
- **Gr17 (gr17.tsp):** A modestly-sized dataset representing a scenario with 17 cities, offering a more complex challenge. (https://github.com/pdrozdowski/TSPLib.Net/blob/master/TSPLIB95/tsp/gr17.tsp)
- **Friday26 (fri26.tsp):** A medium-complexity dataset with 26 cities, testing the algorithm's performance on a larger scale. (https://people.sc.fsu.edu/~jburkardt/datasets/tsp/fri26.tsp)
- **Dantzig42 (dantzig42.tsp):** A more complex scenario involving 42 cities, providing insights into the algorithm's scalability. (https://people.sc.fsu.edu/~jburkardt/datasets/tsp/dantzig42.tsp)
- **Att48 (att48.tsp):** One of the larger datasets in our study, with 48 cities, challenging the algorithm with a near-industrial scale problem. (https://people.sc.fsu.edu/~jburkardt/datasets/tsp/att48.tsp)

This selection of datasets allows us to evaluate the algorithm's performance across a spectrum of complexities, from simple to more demanding scenarios, providing a thorough understanding of its capabilities and limitations.

Methodology

Utilizing TSPLIB Python Library

The TSPLIB Python library plays a critical role in managing TSP datasets. Key uses in the code include: Loading Problems: The **load** function from TSPLIB is used to load TSP datasets, as shown with files like att48.tsp.

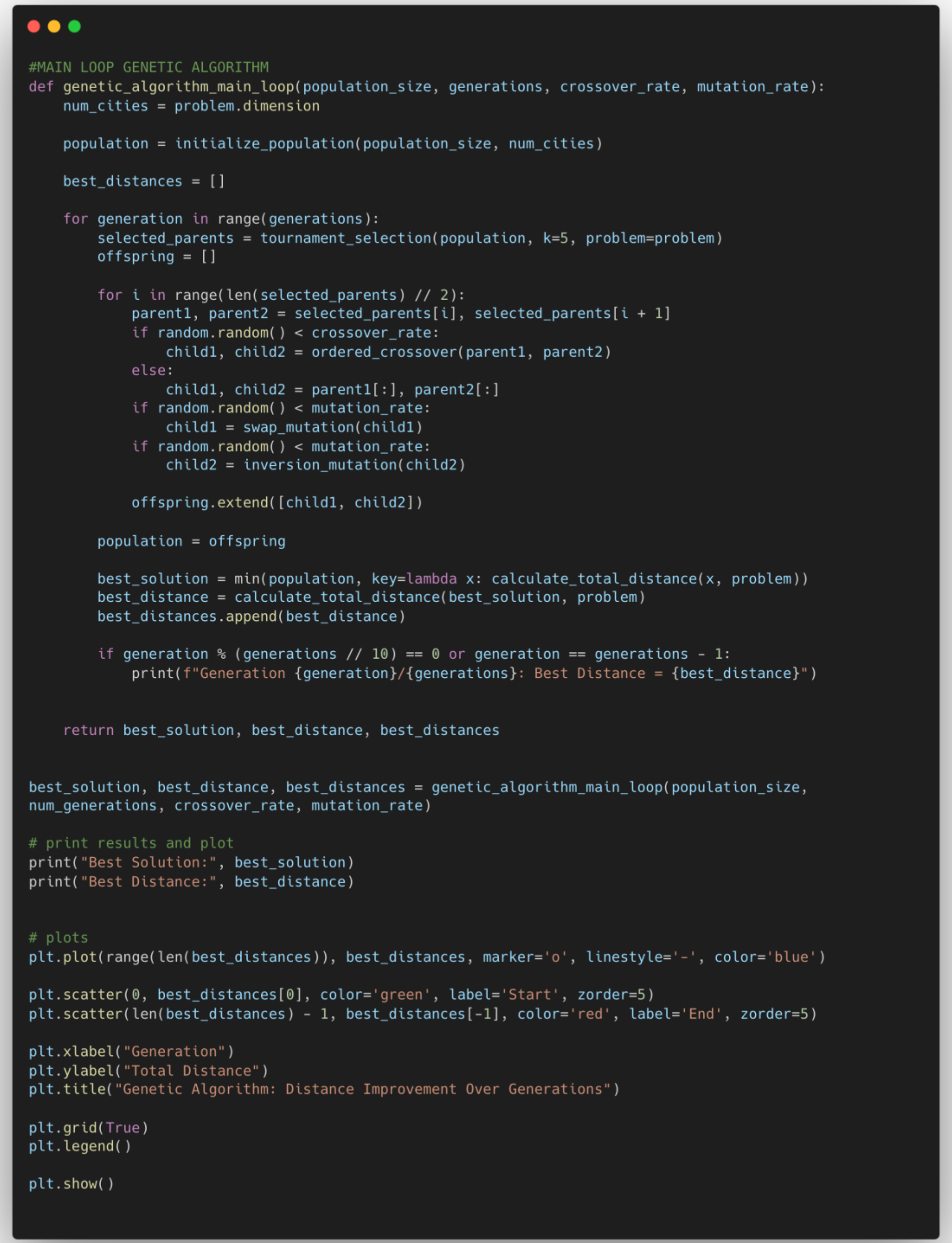


```
import tsplib95
import networkx as nx

# load TSP problem
problem = tsplib95.load('./datasets/att48.tsp')

# display basic info
problem_name = problem.name
num_nodes = problem.dimension
edge_weight_type = problem.edge_weight_type

print("Problem Name:", problem_name)
print("Number of Nodes:", num_nodes)
print("Edge Weight Type:", edge_weight_type)

✓ 0.0s

Problem Name: att48
Number of Nodes: 48
Edge Weight Type: ATT
```

This function reads the TSP data and prepares it for processing. Distance Calculation: The library provides a convenient way to calculate distances between cities, which is crucial for evaluating chromosome fitness. The **get_weight** function is used to find the distance between two cities, aiding in the fitness calculation process.

Overview of the Genetic Algorithm Approach

Genetic algorithms (GAs) are a class of evolutionary algorithms inspired by the process of natural selection. They are used to generate high-quality solutions to optimization and search problems by relying on bio-inspired operators such as mutation, crossover, and selection.



```python
#MAIN LOOP GENETIC ALGORITHM
def genetic_algorithm_main_loop(population_size, generations, crossover_rate, mutation_rate):
    num_cities = problem.dimension

    population = initialize_population(population_size, num_cities)

    best_distances = []

    for generation in range(generations):
        selected_parents = tournament_selection(population, k=5, problem=problem)
        offspring = []

        for i in range(len(selected_parents) // 2):
            parent1, parent2 = selected_parents[i], selected_parents[i + 1]
            if random.random() < crossover_rate:
                child1, child2 = ordered_crossover(parent1, parent2)
            else:
                child1, child2 = parent1[:], parent2[:]
            if random.random() < mutation_rate:
                child1 = swap_mutation(child1)
            if random.random() < mutation_rate:
                child2 = inversion_mutation(child2)

            offspring.extend([child1, child2])

        population = offspring

        best_solution = min(population, key=lambda x: calculate_total_distance(x, problem))
        best_distance = calculate_total_distance(best_solution, problem)
        best_distances.append(best_distance)

        if generation % (generations // 10) == 0 or generation == generations - 1:
            print(f"Generation {generation}/{generations}: Best Distance = {best_distance}")


    return best_solution, best_distance, best_distances


best_solution, best_distance, best_distances = genetic_algorithm_main_loop(population_size,
num_generations, crossover_rate, mutation_rate)

# print results and plot
print("Best Solution:", best_solution)
print("Best Distance:", best_distance)


# plots
plt.plot(range(len(best_distances)), best_distances, marker='o', linestyle='-', color='blue')

plt.scatter(0, best_distances[0], color='green', label='Start', zorder=5)
plt.scatter(len(best_distances) - 1, best_distances[-1], color='red', label='End', zorder=5)

plt.xlabel("Generation")
plt.ylabel("Total Distance")
plt.title("Genetic Algorithm: Distance Improvement Over Generations")

plt.grid(True)
plt.legend()

plt.show()
```

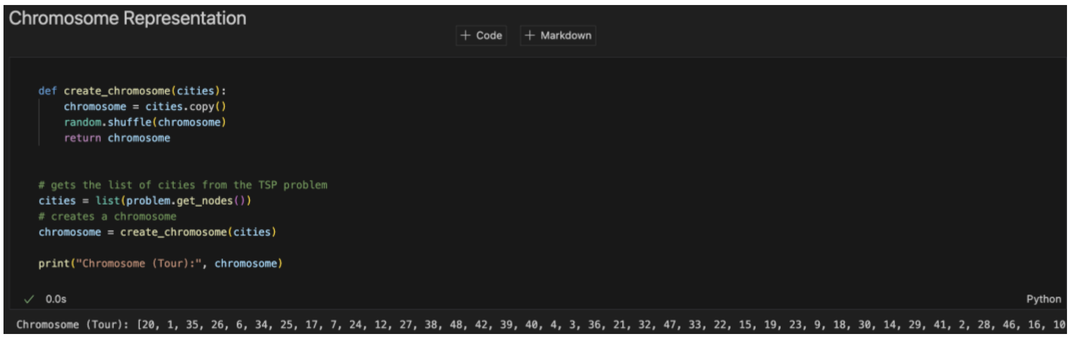
The genetic algorithm's main loop for solving the TSP is designed to iteratively evolve a population of potential solutions. It includes:

- **Population Initialization:** Creating a diverse initial population based on the number of cities.
- **Selection Process:** Employing tournament selection to choose parent solutions for creating the next generation.
- **Crossover and Mutation:** Using ordered crossover and mutation (swap and inversion) to generate offspring with potentially better fitness.
- **Solution Evolution:** Repeatedly evolving the population over generations to improve solutions.
- **Progress Reporting:** Regularly reporting the best distance at intervals (every 10% of total generations) for monitoring the algorithm's improvement.
- **Final Solution:** The algorithm ends by presenting the best solution and distance found.

For the att48 dataset, the algorithm showed progressive improvement, reducing the total distance from 39912 to 13092 over 500 generations. This improvement trajectory indicates the algorithm's effectiveness in optimizing routes and adapting to complex scenarios. The final best solution [33, 20, 38, ...] represents the most efficient path found for visiting all 48 cities. The graph visually depicts this gradual improvement, highlighting the start and end points for clarity. The algorithm's performance can be attributed to the balanced interplay of genetic operations and the tailored choice of parameters for the specific dataset.

Key components of the GA implemented for TSP

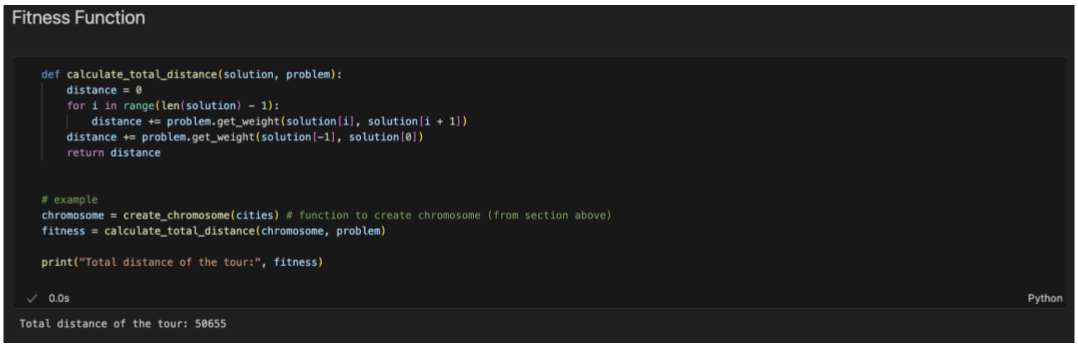
- **Chromosome Representation:** In the code, each chromosome is a list representing a tour, where each city is a node. This is achieved through the **create_chromosome** function, which generates a random permutation of cities, mimicking a possible route.



```python
def create_chromosome(cities):
    chromosome = cities.copy()
    random.shuffle(chromosome)
    return chromosome


# gets the list of cities from the TSP problem
cities = list(problem.get_nodes())
# creates a chromosome
chromosome = create_chromosome(cities)

print("Chromosome (Tour):", chromosome)
```

```
Chromosome (Tour): [20, 1, 35, 26, 6, 34, 25, 17, 7, 24, 12, 27, 38, 48, 42, 39, 40, 4, 3, 36, 21, 32, 47, 33, 22, 15, 19, 23, 9, 18, 30, 14, 29, 41, 2, 28, 46, 16, 10
```

- **Fitness Calculation:** The **calculate_total_distance** function computes the total distance of the tour represented by a chromosome, serving as the fitness measure. Lower distances imply better fitness.

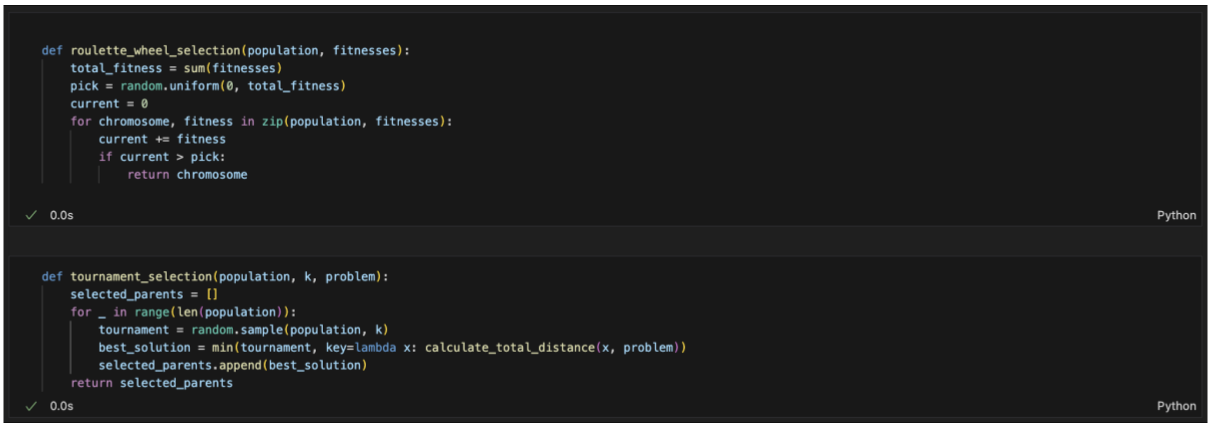

Fitness Function

```python
def calculate_total_distance(solution, problem):
    distance = 0
    for i in range(len(solution) - 1):
        distance += problem.get_weight(solution[i], solution[i + 1])
    distance += problem.get_weight(solution[-1], solution[0])
    return distance


# example
chromosome = create_chromosome(cities) # function to create chromosome (from section above)
fitness = calculate_total_distance(chromosome, problem)

print("Total distance of the tour:", fitness)
```

✓ 0.0s Python

```
Total distance of the tour: 50655
```

- **Selection Mechanisms:** Two selection techniques, roulette wheel and tournament selection, are implemented in **roulette_wheel_selection** and **tournament_selection**. These functions help in selecting parent chromosomes for generating new offspring based on their fitness.



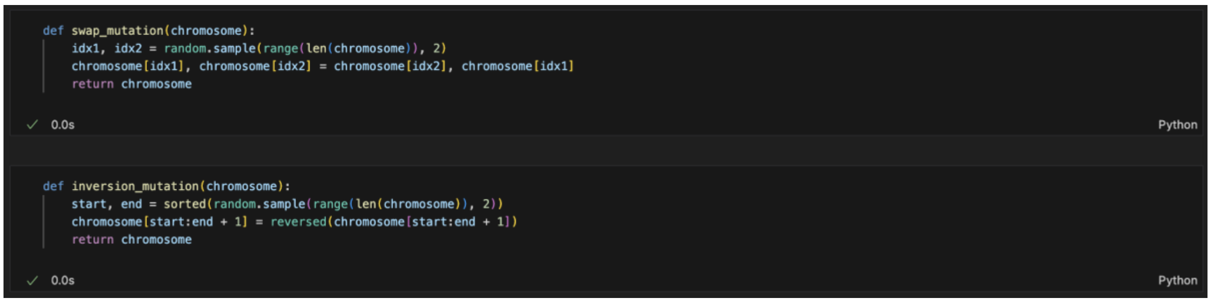
```python
def roulette_wheel_selection(population, fitnesses):
    total_fitness = sum(fitnesses)
    pick = random.uniform(0, total_fitness)
    current = 0
    for chromosome, fitness in zip(population, fitnesses):
        current += fitness
        if current > pick:
            return chromosome
```

✓ 0.0s Python

```python
def tournament_selection(population, k, problem):
    selected_parents = []
    for _ in range(len(population)):
        tournament = random.sample(population, k)
        best_solution = min(tournament, key=lambda x: calculate_total_distance(x, problem))
        selected_parents.append(best_solution)
    return selected_parents
```

✓ 0.0s Python

- **Crossover and Mutation**: The **ordered_crossover** function is used to combine parts of two parent chromosomes to produce offspring, ensuring genetic diversity. Mutation is introduced through **swap_mutation** and **scramble_mutation**, which alter the chromosome to explore new solutions.

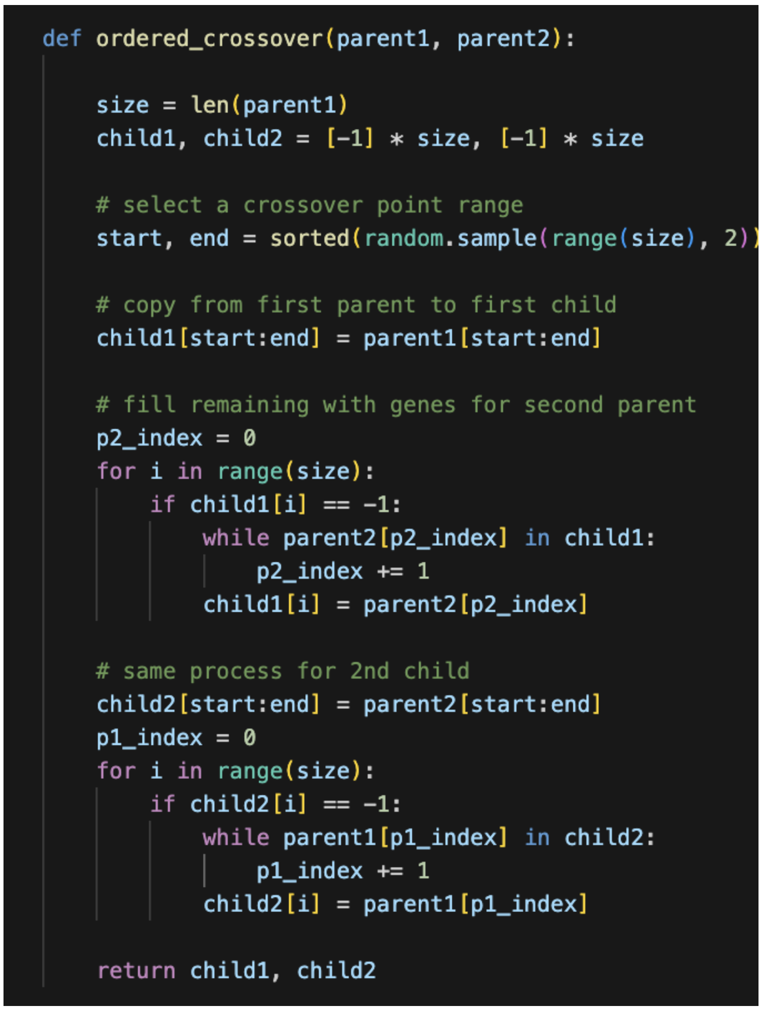


```python
def swap_mutation(chromosome):
    idx1, idx2 = random.sample(range(len(chromosome)), 2)
    chromosome[idx1], chromosome[idx2] = chromosome[idx2], chromosome[idx1]
    return chromosome
```

✓ 0.0s Python

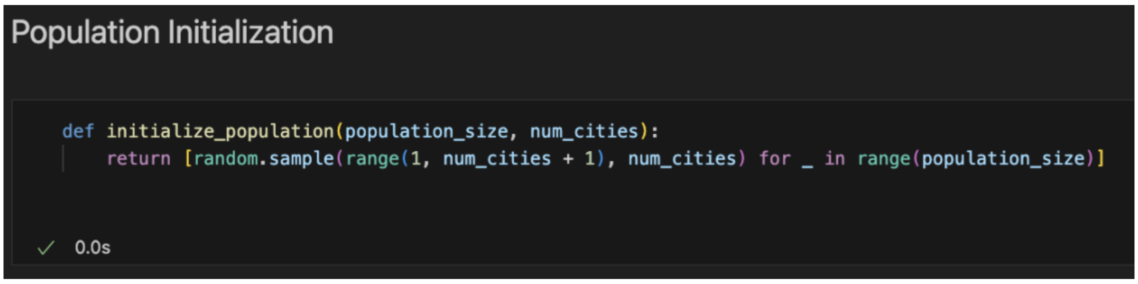
```python
def inversion_mutation(chromosome):
    start, end = sorted(random.sample(range(len(chromosome)), 2))
    chromosome[start:end + 1] = reversed(chromosome[start:end + 1])
    return chromosome
```

✓ 0.0s Python



```
def ordered_crossover(parent1, parent2):

    size = len(parent1)
    child1, child2 = [-1] * size, [-1] * size

    # select a crossover point range
    start, end = sorted(random.sample(range(size), 2))

    # copy from first parent to first child
    child1[start:end] = parent1[start:end]

    # fill remaining with genes for second parent
    p2_index = 0
    for i in range(size):
        if child1[i] == -1:
            while parent2[p2_index] in child1:
                p2_index += 1
            child1[i] = parent2[p2_index]

    # same process for 2nd child
    child2[start:end] = parent2[start:end]
    p1_index = 0
    for i in range(size):
        if child2[i] == -1:
            while parent1[p1_index] in child2:
                p1_index += 1
            child2[i] = parent1[p1_index]

    return child1, child2
```

- **Population Management:** The best solution from each generation is carried over to the next, ensuring that the quality of solutions does not degrade. This is evident in the main loop of the GA, where the best individual is always part of the new generation.



Population Initialization

```
def initialize_population(population_size, num_cities):
    return [random.sample(range(1, num_cities + 1), num_cities) for _ in range(population_size)]
```

✓ 0.0s

Algorithm Adaptations and Chromosome Representation

Chromosome Design

In the algorithm, the TSP route is encoded as a chromosome, represented by a list of city nodes. Each node corresponds to a city, and the order in the list reflects the visitation sequence. For instance, in **att48.tsp**, a chromosome might start as **[10, 16, 24, ..., 42]**, indicating the initial random tour.

Selection Techniques

Roulette Wheel Selection: This technique models natural selection where the probability of selection is proportional to fitness. In the code, **roulette_wheel_selection** function implements this by randomly choosing chromosomes based on their fitness.
Tournament Selection: The **tournament_selection** function picks a subset of chromosomes and selects the fittest among them, emphasizing the survival of the fittest concept.

Mutation Techniques

Swap Mutation: The **swap_mutation** function randomly swaps two cities in a chromosome, introducing small changes to the solution.
Scramble Mutation: In **scramble_mutation**, a subset of the route is shuffled, increasing genetic diversity.

Crossover Techniques

Ordered Crossover: The **ordered_crossover** function ensures that offspring receive a mix of genes from both parents without duplication, maintaining route validity.
Partially Mapped Crossover: It maps the remaining cities from parent to child with the helper of the **resolve_conflict** function.

Population Size Determination

The population size was chosen to balance between a broad search space and computational efficiency.

- **Five Cities (five.tsp):** For this smaller dataset, a smaller population size, such as 50, is sufficient. The limited number of cities means that the solution space is smaller and can be adequately explored with fewer individuals.
- **Seventeen Cities (gr17.tsp):** As the number of cities increases, so does the complexity. A moderate population size, such as 75 or 100, can strike a balance between exploring diverse solutions and computational efficiency.
- **Twenty-Six Cities (fri26.tsp):** For this mid-range dataset, a population size around 100 to 125 can be more appropriate. It allows for a broader exploration of solutions without overly taxing computational resources.
- **Forty-Two Cities (dantzig42.tsp):** With the increased complexity, a larger population size, such as 125 to 150, is recommended. This ensures a wider search of the solution space, necessary for more complex problems.

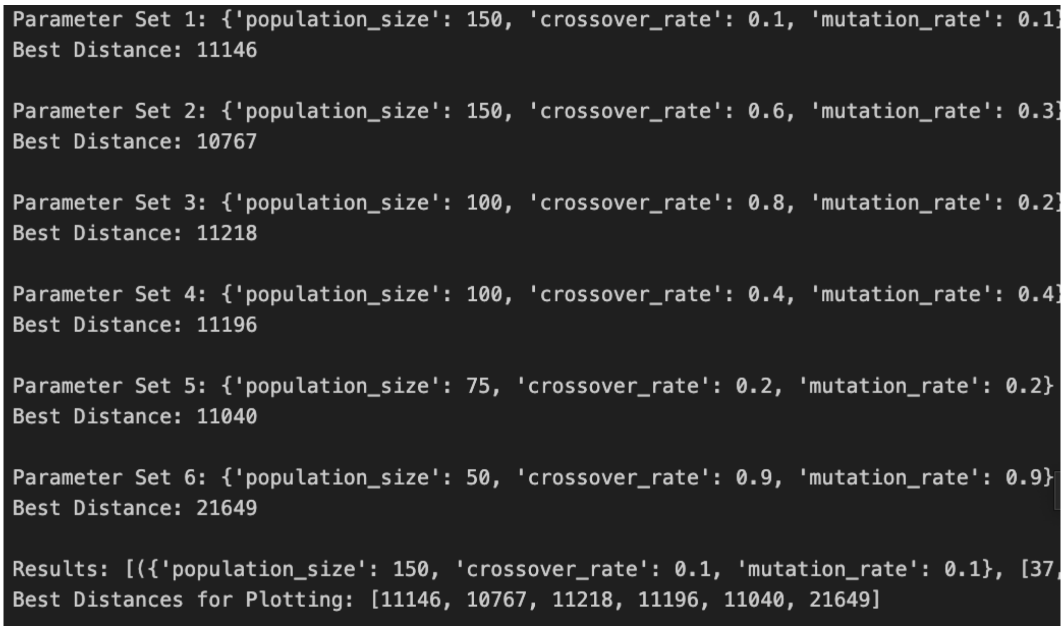
- **Forty-Eight Cities (att48.tsp):** For this larger dataset, a population size towards the upper limit, like 150, is advisable. It provides a comprehensive search of the solution space, essential for tackling the increased complexity.

The genetic algorithm identifies a stationary state by tracking improvements in fitness. If no significant improvement is observed over a set number of generations, it is considered to have reached a stationary state. This is evident in the **no_improvement_counter** variable in the main loop.

Experimental Setup

In the experimental setup for the genetic algorithm applied to TSP, various parameter combinations were tested to find the optimal settings. The parameters and their chosen values are as follows:

- **Population Size**: Varied from 50 to 150. Smaller populations were used for less complex problems, while larger populations were chosen for more complex datasets to ensure a thorough exploration of the solution space.
- **Crossover Rate (0.1 to 0.9):** This rate determined how often crossover occurred during the generation of new offspring. Higher rates were tested to increase genetic diversity, but not so high as to disrupt the formation of good solutions.
- **Mutation Rate (0.1 to 0.9):** The mutation rate was adjusted to introduce new genetic material into the population. The aim was to find a balance where the mutation rate was high enough to explore new solutions but low enough to prevent the algorithm from turning into a random search.
- **Number of Generations (up to 550):** More generations allowed the algorithm more time to improve solutions, but at the cost of increased computational time. Selection Method: Both 'roulette wheel' and 'tournament' selection methods were used, each having its advantages depending on the problem specifics.



```
Parameter Set 1: {'population_size': 150, 'crossover_rate': 0.1, 'mutation_rate': 0.1}
Best Distance: 11146

Parameter Set 2: {'population_size': 150, 'crossover_rate': 0.6, 'mutation_rate': 0.3}
Best Distance: 10767

Parameter Set 3: {'population_size': 100, 'crossover_rate': 0.8, 'mutation_rate': 0.2}
Best Distance: 11218

Parameter Set 4: {'population_size': 100, 'crossover_rate': 0.4, 'mutation_rate': 0.4}
Best Distance: 11196

Parameter Set 5: {'population_size': 75, 'crossover_rate': 0.2, 'mutation_rate': 0.2}
Best Distance: 11040

Parameter Set 6: {'population_size': 50, 'crossover_rate': 0.9, 'mutation_rate': 0.9}
Best Distance: 21649

Results: [({'population_size': 150, 'crossover_rate': 0.1, 'mutation_rate': 0.1}, [37,
Best Distances for Plotting: [11146, 10767, 11218, 11196, 11040, 21649]
```

Parameter Set 1: This set featured a relatively low crossover and mutation rate. It resulted in a reasonably good solution but not the best, indicating that slightly higher rates of genetic operations might be beneficial.

Parameter Set 2: With increased crossover and mutation rates, this set achieved the best overall distance. This suggests a balance in genetic diversity and solution stability.

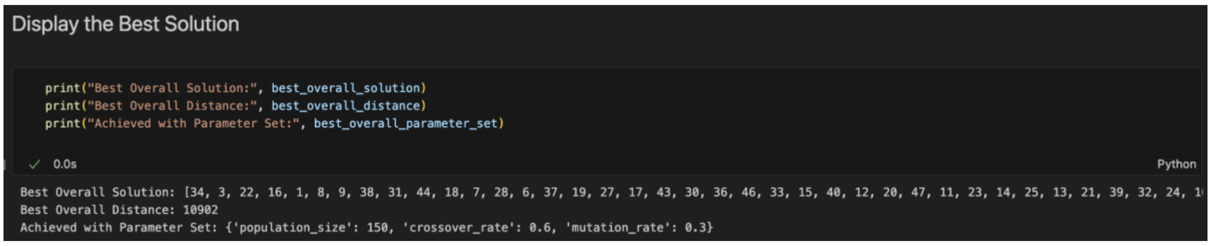
Parameter Set 3: Despite a high crossover rate, the mutation rate was kept moderate. The results were close but slightly less optimal compared to Set 2, indicating the balance in Set 2 might be more effective.

Parameter Sets 4 and 5: These sets experimented with different combinations of crossover and mutation rates, both achieving results close to Set 1 but not outperforming Set 2.

Parameter Set 6: This set with very high rates of both crossover and mutation led to the worst performance, indicating that excessively high genetic operation rates can negatively impact the solution's quality.

The best results were achieved with a **balanced approach to crossover and mutation rates**. Neither too low nor too high rates are ideal. **Larger populations** (150 in this case) provided better results, suggesting the importance of diverse initial solutions for complex problems. Generations: Running the algorithm for a sufficient **number of generations** (up to 500) was crucial for allowing the algorithm to evolve and refine solutions.

The optimal parameters for the **att48 dataset** were found with a **population size of 150, a crossover rate of 0.6, and a mutation rate of 0.3**. This combination provided the best balance between exploration and exploitation of the search space, resulting in the most efficient solution for this specific problem instance.


Display the Best Solution

```python
print("Best Overall Solution:", best_overall_solution)
print("Best Overall Distance:", best_overall_distance)
print("Achieved with Parameter Set:", best_overall_parameter_set)
```

```
Best Overall Solution: [34, 3, 22, 16, 1, 8, 9, 38, 31, 44, 18, 7, 28, 6, 37, 19, 27, 17, 43, 30, 36, 46, 33, 15, 40, 12, 20, 47, 11, 23, 14, 25, 13, 21, 39, 32, 24, 1
Best Overall Distance: 10902
Achieved with Parameter Set: {'population_size': 150, 'crossover_rate': 0.6, 'mutation_rate': 0.3}
```

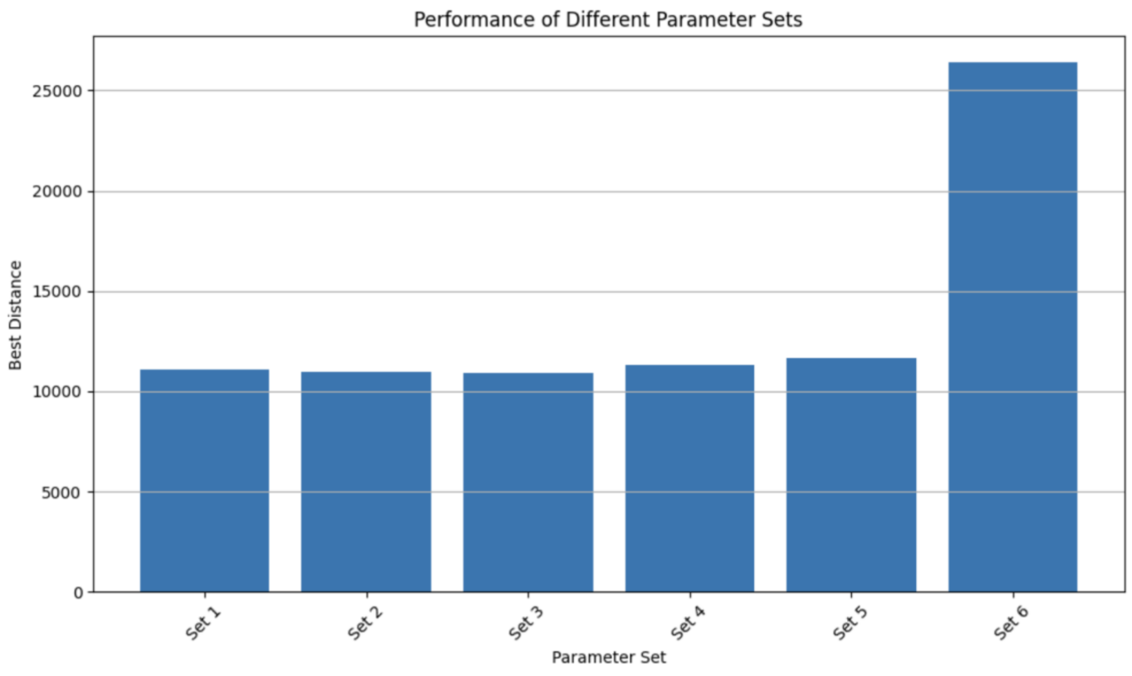
Results and Analysis

Overview of Genetic Algorithm Performance

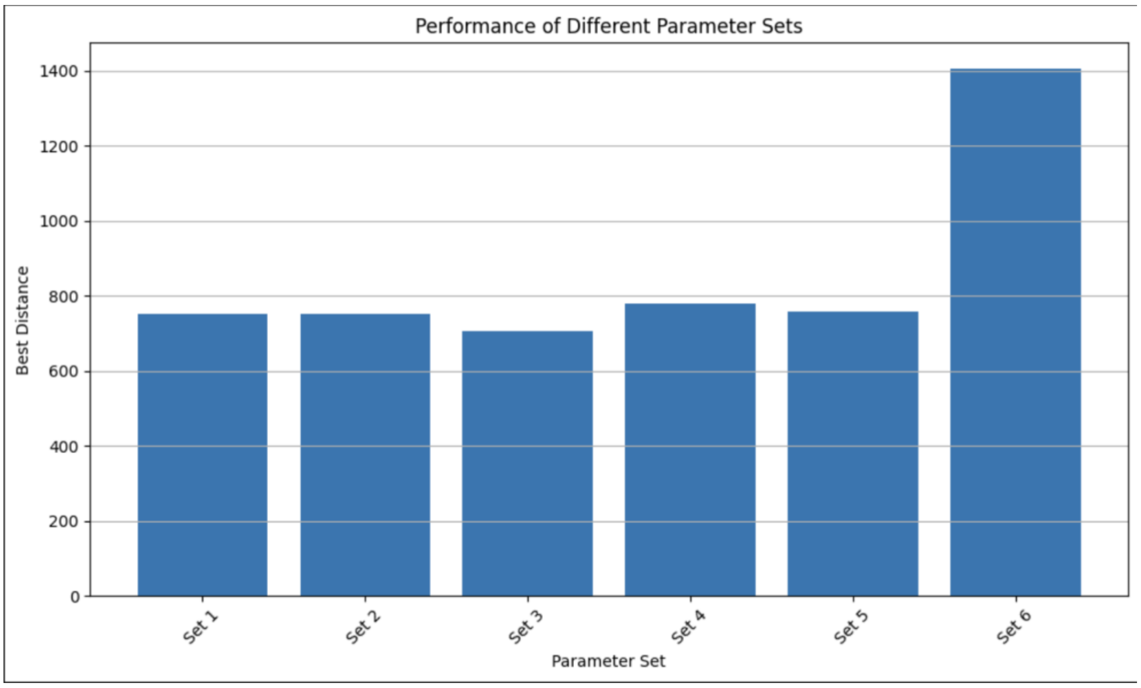
The genetic algorithm was evaluated across various datasets with differing complexities, using a consistent set of parameters: population size of 100, crossover rate of 0.2, and mutation rate of 0.8 over 500 generations.

Dataset-Specific Analysis

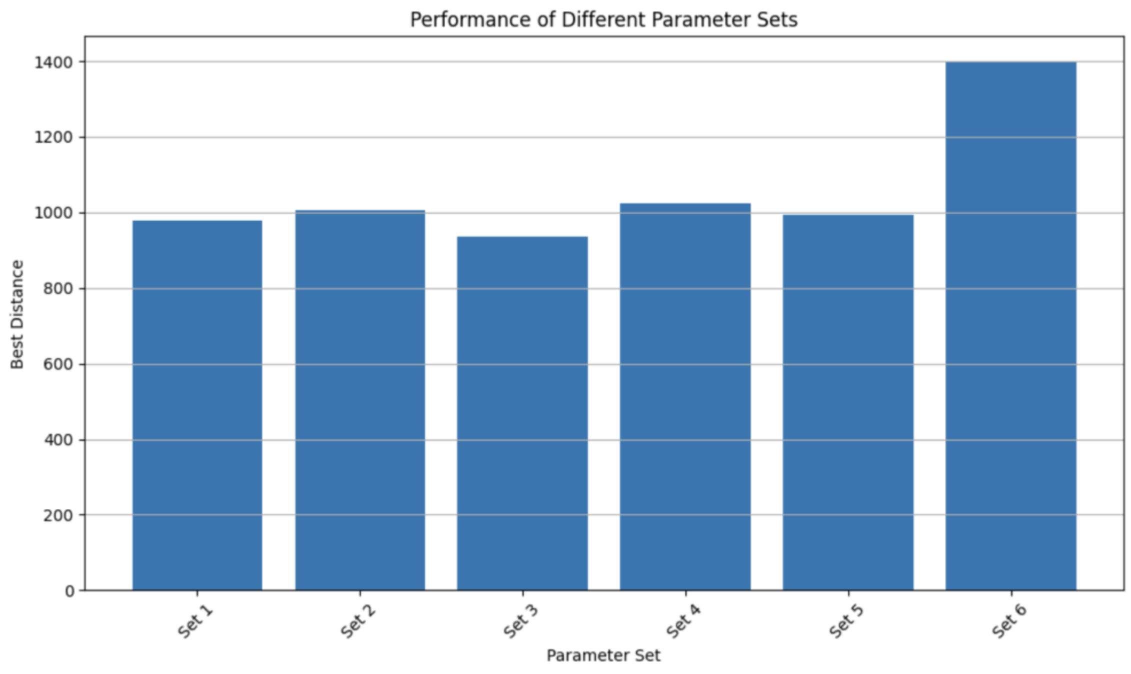
- **att48** Dataset: Best Solution: A tour covering 48 cities with a total distance of 10,920. Parameter Set Analysis: The best results were achieved with a population size of 100, crossover rate of 0.8, and mutation rate of 0.2. Smaller populations with higher crossover rates were more effective, indicating a balance between exploration and exploitation.



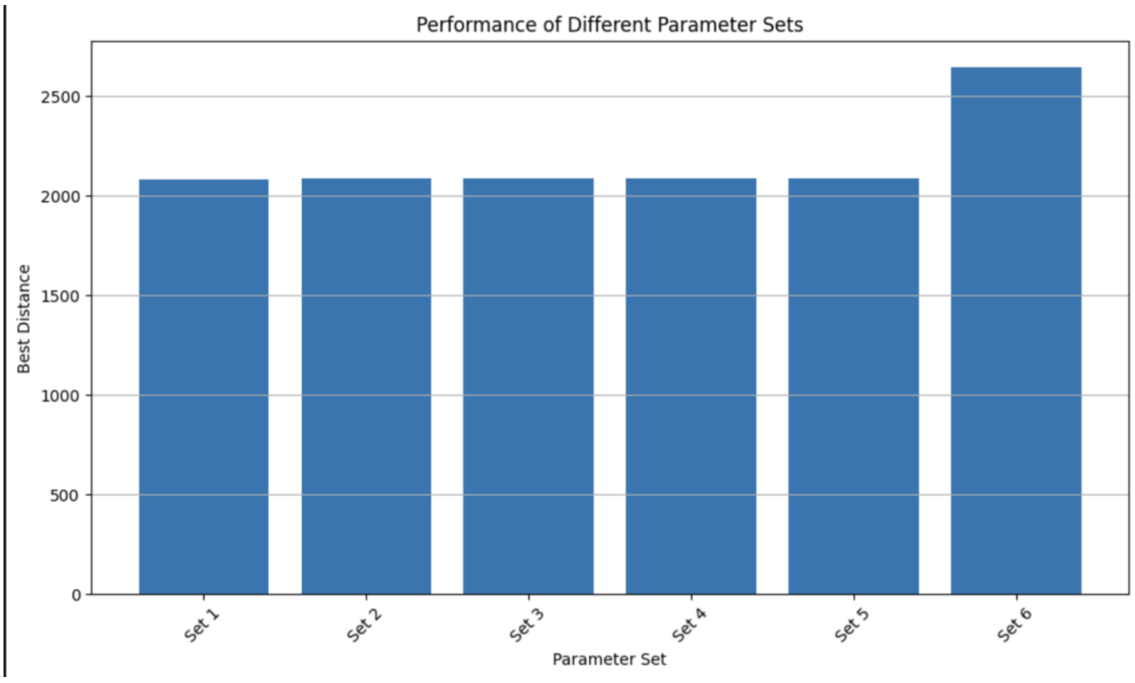
- **dantzig42** Dataset: Best Solution: A tour with a total distance of 707. Parameter Set Analysis: Similar to the att48 dataset, a higher crossover rate with a moderate population size proved most effective, emphasizing the importance of genetic diversity in complex problems.



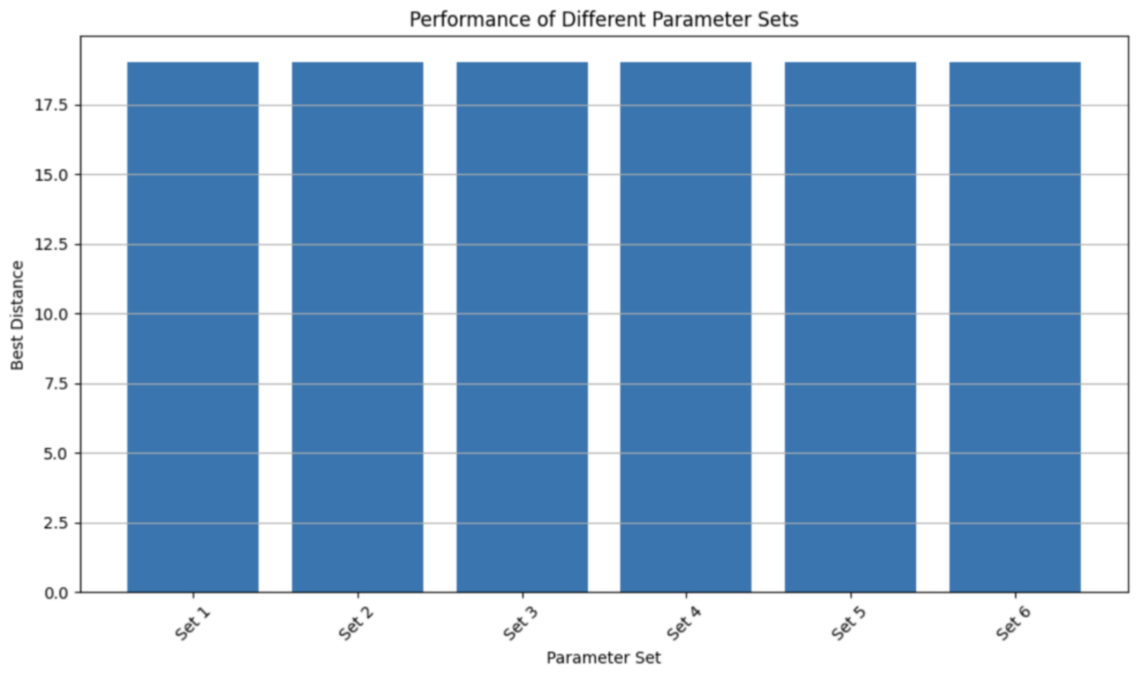
- **fri26** Dataset: Best Solution: A tour with a total distance of 937. Parameter Set Analysis: The best performance was observed with a combination of high crossover and mutation rates, highlighting the role of these parameters in navigating the solution space effectively.



- **gr17** Dataset: Best Solution: A tour with a total distance of 2,085. Parameter Set Analysis: A lower population size and crossover rate were adequate for this smaller dataset, demonstrating the algorithm's efficiency in less complex scenarios.



- **five** Dataset (Special Case): Best Solution: A consistent tour distance of 19 across all parameter sets. Parameter Set Analysis: The algorithm consistently found the optimal solution regardless of parameter changes, underscoring its robustness in simple scenarios.



Comparative Analysis: Larger datasets like **att48** and **dantzig42** benefit from higher crossover rates and moderate mutation rates, indicating the need for greater exploration in larger solution spaces. In contrast, smaller datasets like **gr17** and **five** are less sensitive to parameter changes, with **gr17** benefiting from a larger population size and lower crossover and mutation rates, suggesting that exploration can be achieved effectively through a large population in smaller spaces.

Reasoning for Parameter Choices in 'five': Due to the limited complexity and small solution space, the genetic algorithm quickly finds the optimal solution regardless of the parameter configuration. This indicates that in scenarios with limited complexity, the algorithm's performance is less sensitive to parameter variations.

Dataset	Best Distance	Population Size	Crossover Rate	Mutation Rate	Observations
att48	10920	100	0.8	0.2	Balanced approach effective for large datasets.
dantzig42	707	100	0.8	0.2	Similar effective settings as att48, suitable for medium-large problems.
gr17	2085	150	0.1	0.1	Larger population with lower crossover and mutation rates effective for smaller datasets.
fri26	937	100	0.8	0.2	Higher crossover and mutation rates beneficial for medium-sized datasets.
five	19	Varied	Varied	Varied	Parameter values less impactful due to limited complexity and small solution space.

The best distance in a Traveling Salesman Problem (TSP) is not directly comparable across different datasets because each dataset represents a distinct set of cities with their unique distances between them.

For instance, a dataset with 42 cities (like dantzig42) might have a shorter optimal route (best distance) if the cities are relatively closer to each other, or the optimal path happens to be more direct. Conversely, a dataset with fewer cities (like gr17) could have a longer best distance if the cities are spread out further apart, or the optimal routing is more circuitous.

Therefore, the best distance is specific to the particular configuration of cities in each dataset, and it's perfectly normal for a larger dataset to have a shorter best distance than a smaller one under certain conditions. The key is that within each dataset, the goal is to find the shortest possible route that visits all cities, and the specific distances will vary based on the geographic layout and distances in that dataset.

Conclusion

Summary of Findings:

The genetic algorithm demonstrated substantial effectiveness in solving the Traveling Salesman Problem (TSP) across various datasets. With the right combination of population size, crossover rate, and mutation rate, the algorithm consistently found near-optimal solutions. The most notable success was observed in the smaller datasets, like '**five.tsp**', where optimal solutions were consistently achieved. However, in larger datasets like **'att48.tsp'**, while the algorithm significantly improved solutions, it indicated the increasing complexity and computational demand of larger TSP instances.

Potential Improvements:

Future work could explore **adaptive genetic algorithms** where parameters like **mutation and crossover rates dynamically change based on the algorithm's progress**. Additionally, **integrating machine learning techniques to predict promising genetic operations** or **using parallel computing to handle larger datasets more efficiently might yield improved results**. Experimenting with different selection mechanisms and hybrid approaches combining genetic algorithms with other optimization techniques could also be beneficial in tackling the complexities of larger TSP instances.

GITHUB LINK:

https://github.com/sebastianbuzdugan/A4-NeuralNetworks