Ramesh Chandra Soren

Enrollment No: **2022CSB086**

### Department: **Computer Science and Technology**

Assignment 6

Genetic Algorithm: Travelling Salesman Problem

Steps to implement the Genetic Algorithm:

1. **Create Initial Population**
   Define an initial population, considering `n` number of cities, with a permutation of city indices representing potential routes, like `{1, 2, ..., n}`. Start by considering a small set of cities (5 cities) and gradually increase the number of cities up to 10.

2. **Define Fitness Function**
   The fitness `f` of a solution is determined by the total cost (or distance) of the tour. Use the inverse of the total route cost as the fitness measure, as a lower cost corresponds to a higher fitness.

3. **Selection Process**
   Select the best routes (individuals) based on their fitness to create the next generation. Use an appropriate selection method (e.g., tournament selection or roulette wheel selection) to choose the parents for crossover.

4. **Crossover (Recombination)**
   Perform crossover on the selected routes to produce new offspring. Use a crossover probability of 0.6 to determine if crossover occurs for a pair of parents. Apply a crossover method suitable for permutations.

5. **Mutation**
   Introduce mutations in the population to maintain genetic diversity. Apply a mutation probability of 0.1, swapping two randomly selected cities in a route to create a small variation.

```python
import random

def initialize_population(pop_size, num_cities):
    population = []
    for _ in range(pop_size):
        # Generate a random permutation of cities as a route
        route = random.sample(range(num_cities), num_cities)
        population.append(route)
    return population

import numpy as np

# Distance matrix to hold distances between cities
def calculate_distance_matrix(num_cities):
    # This creates a symmetric matrix for simplicity, but in real
cases, distances can vary
    return np.random.randint(10, 100, size=(num_cities, num_cities))

# Fitness function: lower cost implies higher fitness
def calculate_route_cost(route, distance_matrix):
    cost = 0
    for i in range(len(route) - 1):
        cost += distance_matrix[route[i], route[i + 1]]
    # Add distance to return to the starting city
    cost += distance_matrix[route[-1], route[0]]
    return 1 / cost  # Inverse cost for fitness

def select_parents(population, fitness_scores):
    selected = random.choices(population, weights=fitness_scores, k=2)
    return selected

def crossover(parent1, parent2):
    # Ordered Crossover (OX)
    size = len(parent1)
    start, end = sorted(random.sample(range(size), 2))

    child = [None] * size
    child[start:end] = parent1[start:end]

    # Fill in remaining cities from parent2
    current_pos = end
    for city in parent2:
        if city not in child:
            if current_pos >= size:
                current_pos = 0
            child[current_pos] = city
            current_pos += 1
    return child

def mutate(route, mutation_rate=0.1):
    if random.random() < mutation_rate:
        i, j = random.sample(range(len(route)), 2)
        route[i], route[j] = route[j], route[i]
    return route
```

```python
def genetic_algorithm(num_cities, pop_size=100, generations=500,
crossover_prob=0.6, mutation_rate=0.1):
    distance_matrix = calculate_distance_matrix(num_cities)
    population = initialize_population(pop_size, num_cities)

    for generation in range(generations):
        # Evaluate fitness for each individual
        fitness_scores = [calculate_route_cost(individual,
distance_matrix) for individual in population]

        # Create new population
        new_population = []
        for _ in range(pop_size // 2):  # Generate pop_size
individuals in pairs
            parent1, parent2 = select_parents(population,
fitness_scores)

            # Crossover
            if random.random() < crossover_prob:
                offspring1 = crossover(parent1, parent2)
                offspring2 = crossover(parent2, parent1)
            else:
                offspring1, offspring2 = parent1[:], parent2[:]

            # Mutation
            offspring1 = mutate(offspring1, mutation_rate)
            offspring2 = mutate(offspring2, mutation_rate)

            new_population.extend([offspring1, offspring2])

        # Replace old population with new population
        population = new_population

    # Return the best route found
    best_route = min(population, key=lambda route:
calculate_route_cost(route, distance_matrix))
    best_distance = 1 / calculate_route_cost(best_route,
distance_matrix)
    return best_route, best_distance

# Example run with 5 cities
best_route, best_distance = genetic_algorithm(num_cities=5)
print("Best route:", best_route)
print("Shortest distance found:", best_distance)
```

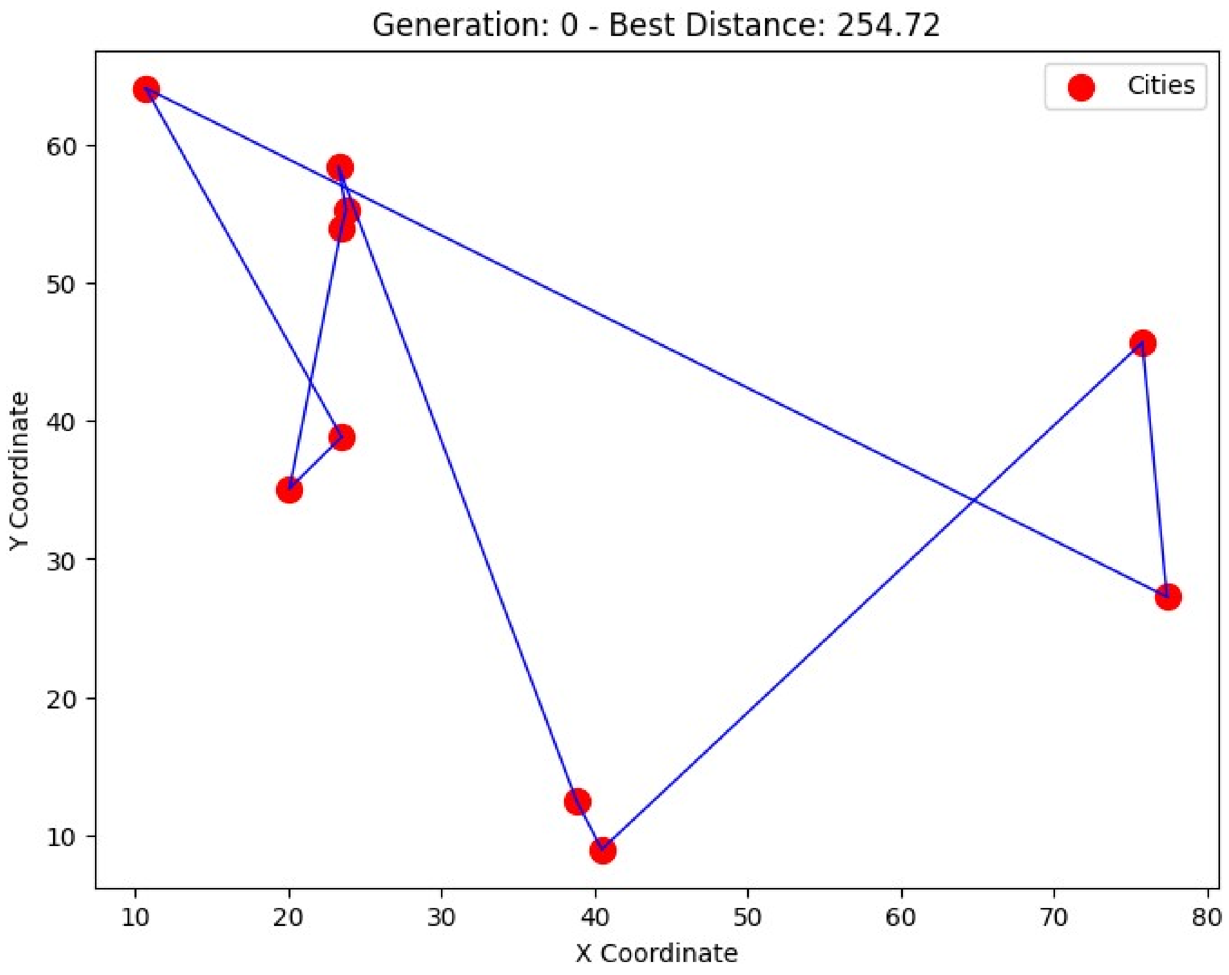
```
Best route: [0, 4, 1, 3, 2]
Shortest distance found: 332.0
```

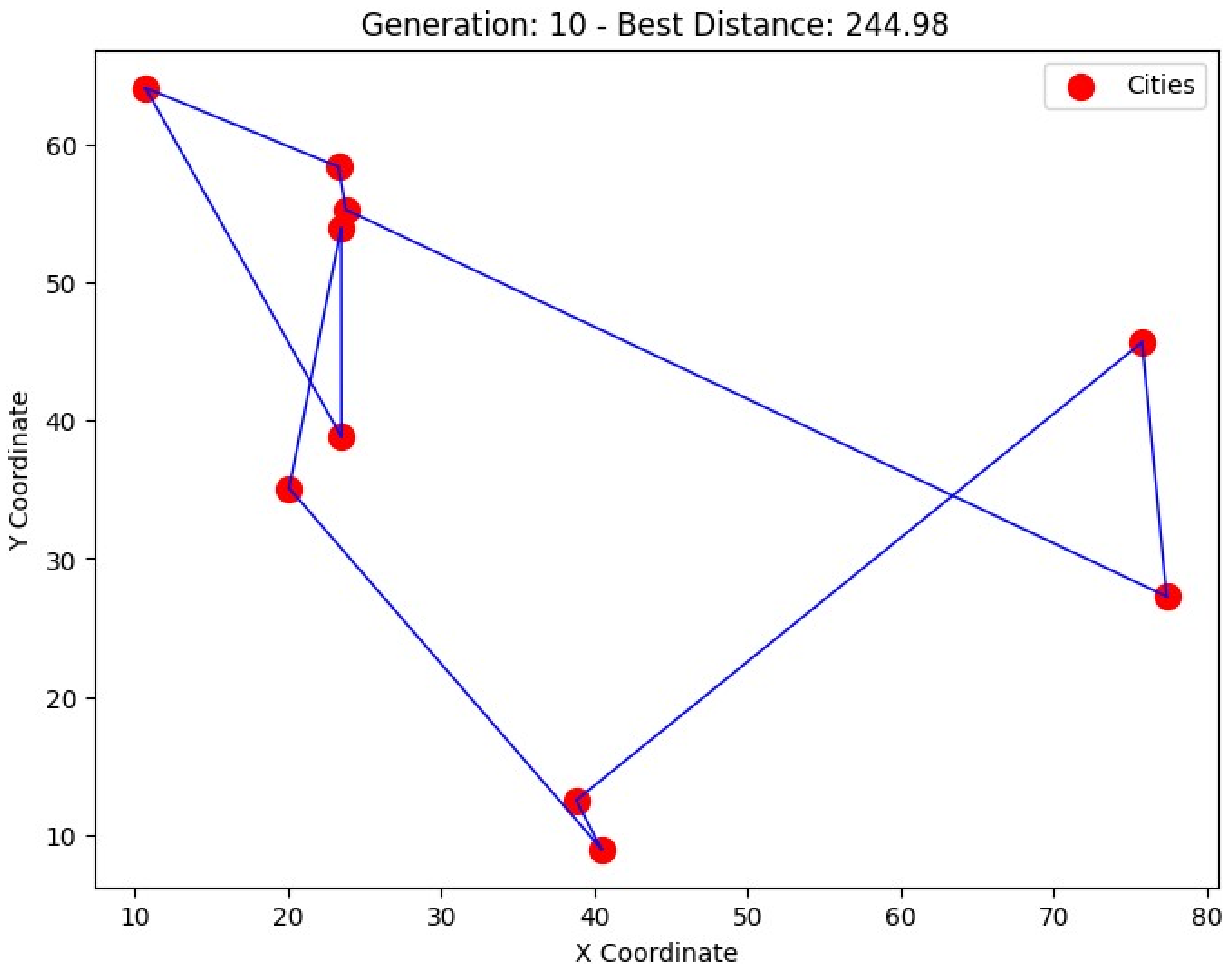
Better Understanding

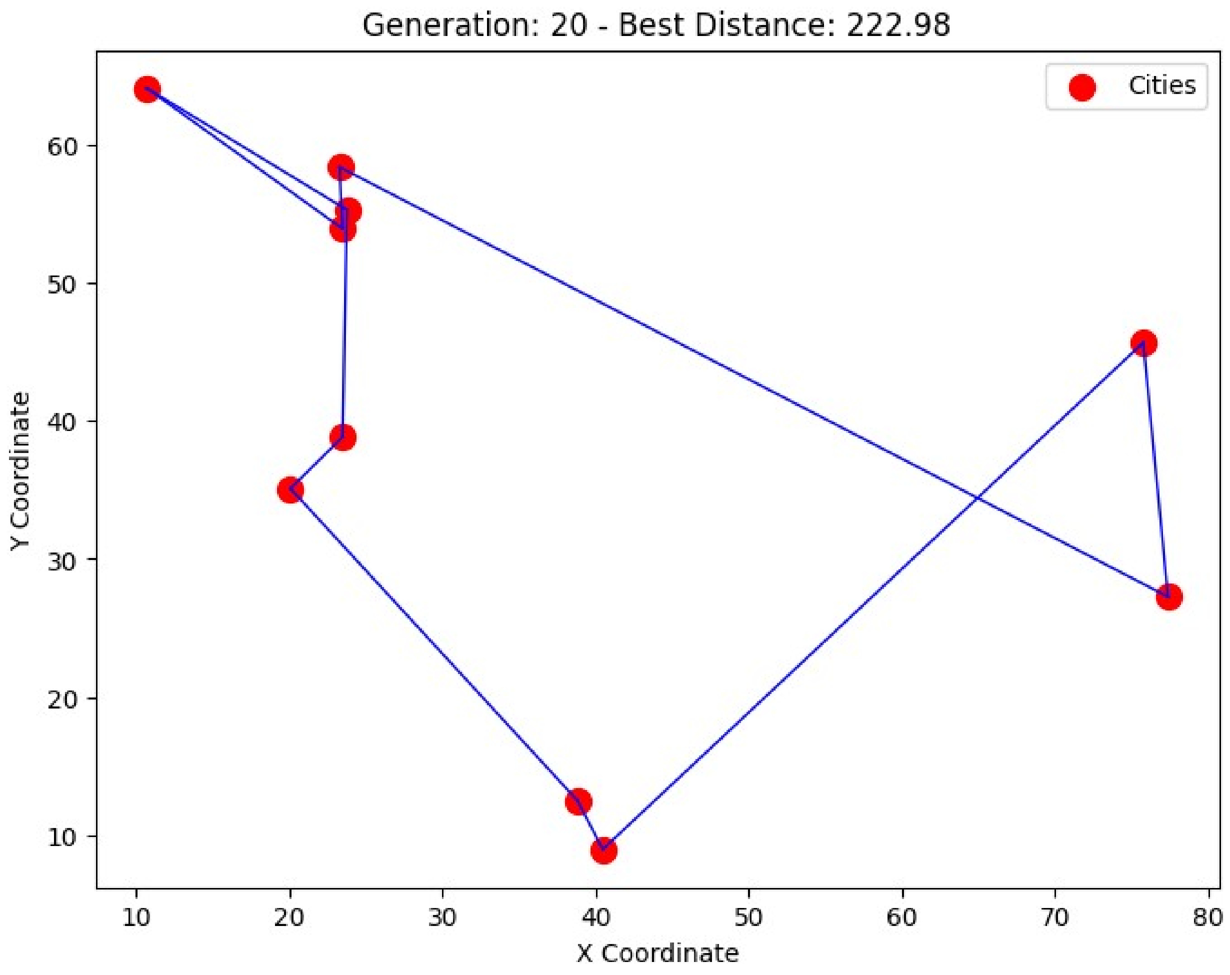
```
pip install matplotlib numpy

Requirement already satisfied: matplotlib in
/usr/local/lib/python3.10/dist-packages (3.8.0)
Requirement already satisfied: numpy in
/usr/local/lib/python3.10/dist-packages (1.26.4)
Requirement already satisfied: contourpy>=1.0.1 in
/usr/local/lib/python3.10/dist-packages (from matplotlib) (1.3.0)
Requirement already satisfied: cycler>=0.10 in
/usr/local/lib/python3.10/dist-packages (from matplotlib) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in
/usr/local/lib/python3.10/dist-packages (from matplotlib) (4.54.1)
Requirement already satisfied: kiwisolver>=1.0.1 in
/usr/local/lib/python3.10/dist-packages (from matplotlib) (1.4.7)
Requirement already satisfied: packaging>=20.0 in
/usr/local/lib/python3.10/dist-packages (from matplotlib) (24.1)
Requirement already satisfied: pillow>=6.2.0 in
/usr/local/lib/python3.10/dist-packages (from matplotlib) (10.4.0)
Requirement already satisfied: pyparsing>=2.3.1 in
/usr/local/lib/python3.10/dist-packages (from matplotlib) (3.2.0)
Requirement already satisfied: python-dateutil>=2.7 in
/usr/local/lib/python3.10/dist-packages (from matplotlib) (2.8.2)
Requirement already satisfied: six>=1.5 in
/usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.7-
>matplotlib) (1.16.0)
```

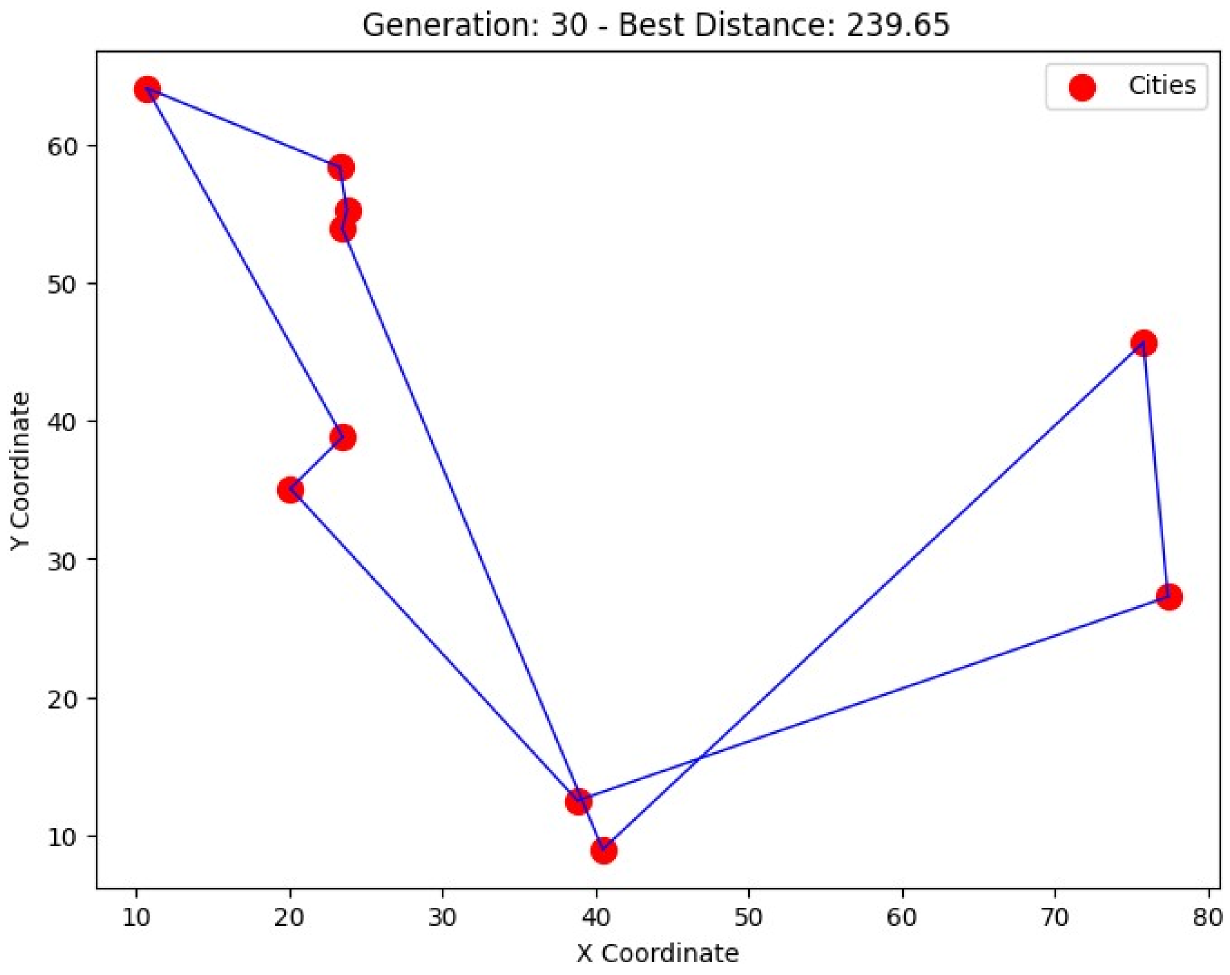
```
import numpy as np
import random
import matplotlib.pyplot as plt

# Generate random cities (coordinates) for the TSP
def generate_cities(num_cities):
    return np.random.rand(num_cities, 2) * 100

# Calculate the distance matrix based on Euclidean distance
def calculate_distance_matrix(cities):
    num_cities = len(cities)
    distance_matrix = np.zeros((num_cities, num_cities))
    for i in range(num_cities):
        for j in range(num_cities):
            distance_matrix[i, j] = np.linalg.norm(cities[i] -
cities[j])
    return distance_matrix

# Define fitness as the inverse of route cost
def calculate_route_cost(route, distance_matrix):
    cost = sum(distance_matrix[route[i], route[i + 1]] for i in
range(len(route) - 1))
    cost += distance_matrix[route[-1], route[0]]  # Return to start
    return 1 / cost

# Plot cities and route
def plot_route(cities, route, generation=None, best_distance=None):
    plt.figure(figsize=(8, 6))
    plt.scatter(cities[:, 0], cities[:, 1], s=100, color='red',
label="Cities")

    for i in range(len(route)):
        start, end = cities[route[i]], cities[route[(i + 1) %
len(route)]]
        plt.plot([start[0], end[0]], [start[1], end[1]], 'b-', lw=1)

    if generation is not None:
        plt.title(f"Generation: {generation} - Best Distance:
{best_distance:.2f}")
    plt.xlabel("X Coordinate")
    plt.ylabel("Y Coordinate")
    plt.legend()
    plt.show()

# Run the genetic algorithm and plot progress
def genetic_algorithm(num_cities=10, pop_size=100, generations=200,
crossover_prob=0.6, mutation_rate=0.1):
    cities = generate_cities(num_cities)
    distance_matrix = calculate_distance_matrix(cities)

    population = [random.sample(range(num_cities), num_cities) for _
in range(pop_size)]
    best_distances = []

    for generation in range(generations):
        fitness_scores = [calculate_route_cost(ind, distance_matrix)
for ind in population]
        best_route = max(population, key=lambda ind:
calculate_route_cost(ind, distance_matrix))
        best_distance = 1 / calculate_route_cost(best_route,
distance_matrix)
        best_distances.append(best_distance)

        # Plot the best route at certain generations
        if generation % (generations // 10) == 0 or generation ==
generations - 1:
            plot_route(cities, best_route, generation, best_distance)

        # Selection, Crossover, and Mutation (simplified)
        new_population = []
        for _ in range(pop_size // 2):
            parents = random.choices(population,
weights=fitness_scores, k=2)
```

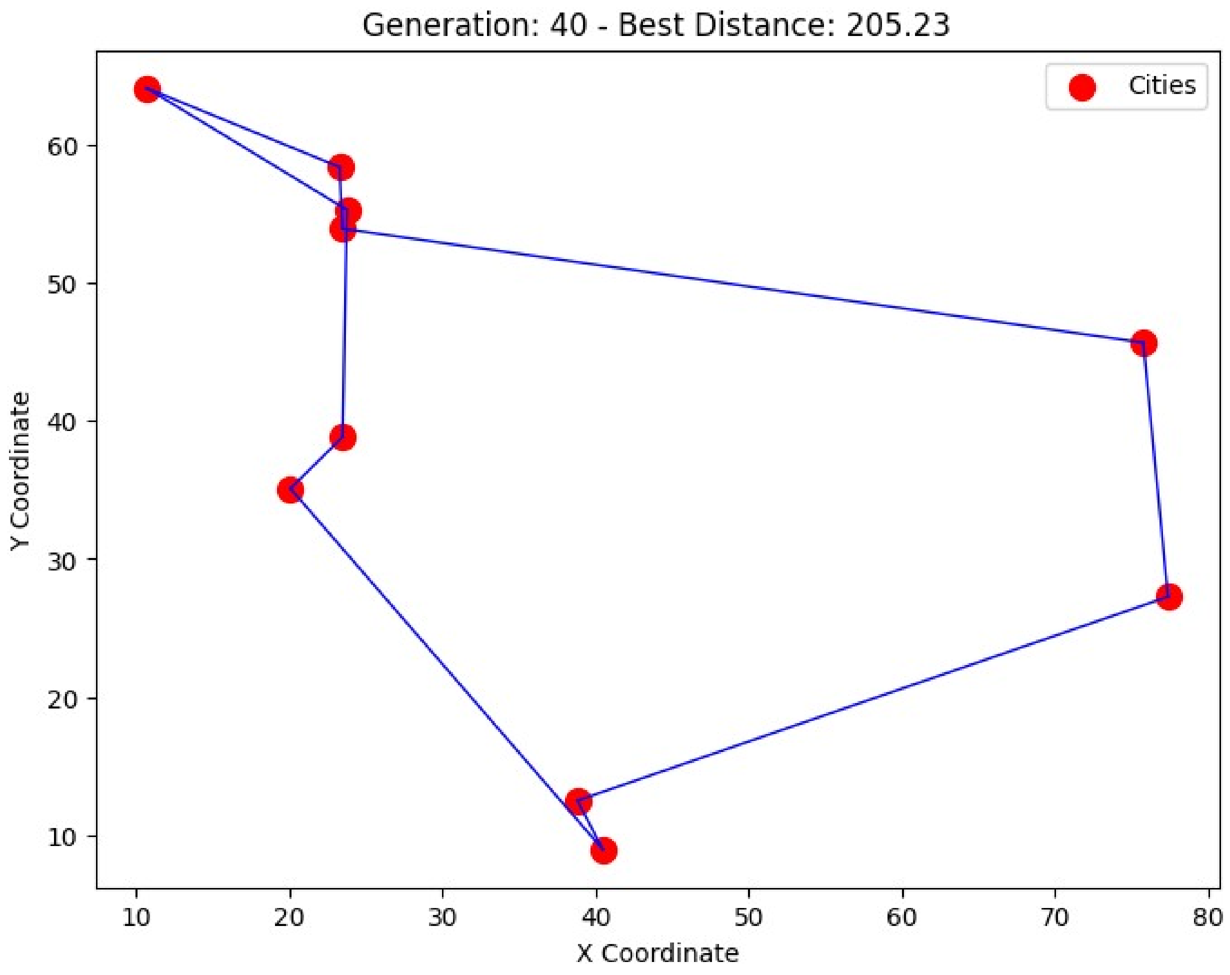
```
            if random.random() < crossover_prob:
                child1, child2 = crossover(parents[0], parents[1])
            else:
                child1, child2 = parents[0][:], parents[1][:]
            new_population.extend([mutate(child1, mutation_rate),
mutate(child2, mutation_rate)])

        population = new_population

    # Plot progress of best distances over generations
    plt.figure(figsize=(10, 5))
    plt.plot(best_distances, marker='o')
    plt.xlabel("Generation")
    plt.ylabel("Best Distance Found")
    plt.title("Progress of GA Optimization for TSP")
    plt.grid(True)
    plt.show()

# Crossover and Mutation functions as defined earlier, simplified for
demo
def crossover(parent1, parent2):
    size = len(parent1)
    start, end = sorted(random.sample(range(size), 2))
    child = [None] * size
    child[start:end] = parent1[start:end]
    current_pos = end
    for city in parent2:
        if city not in child:
            if current_pos >= size:
                current_pos = 0
            child[current_pos] = city
            current_pos += 1
    return child, child

def mutate(route, mutation_rate=0.1):
    if random.random() < mutation_rate:
        i, j = random.sample(range(len(route)), 2)
        route[i], route[j] = route[j], route[i]
    return route

# Run the genetic algorithm with 10 cities and plot the progress
genetic_algorithm(num_cities=10, generations=100)
```

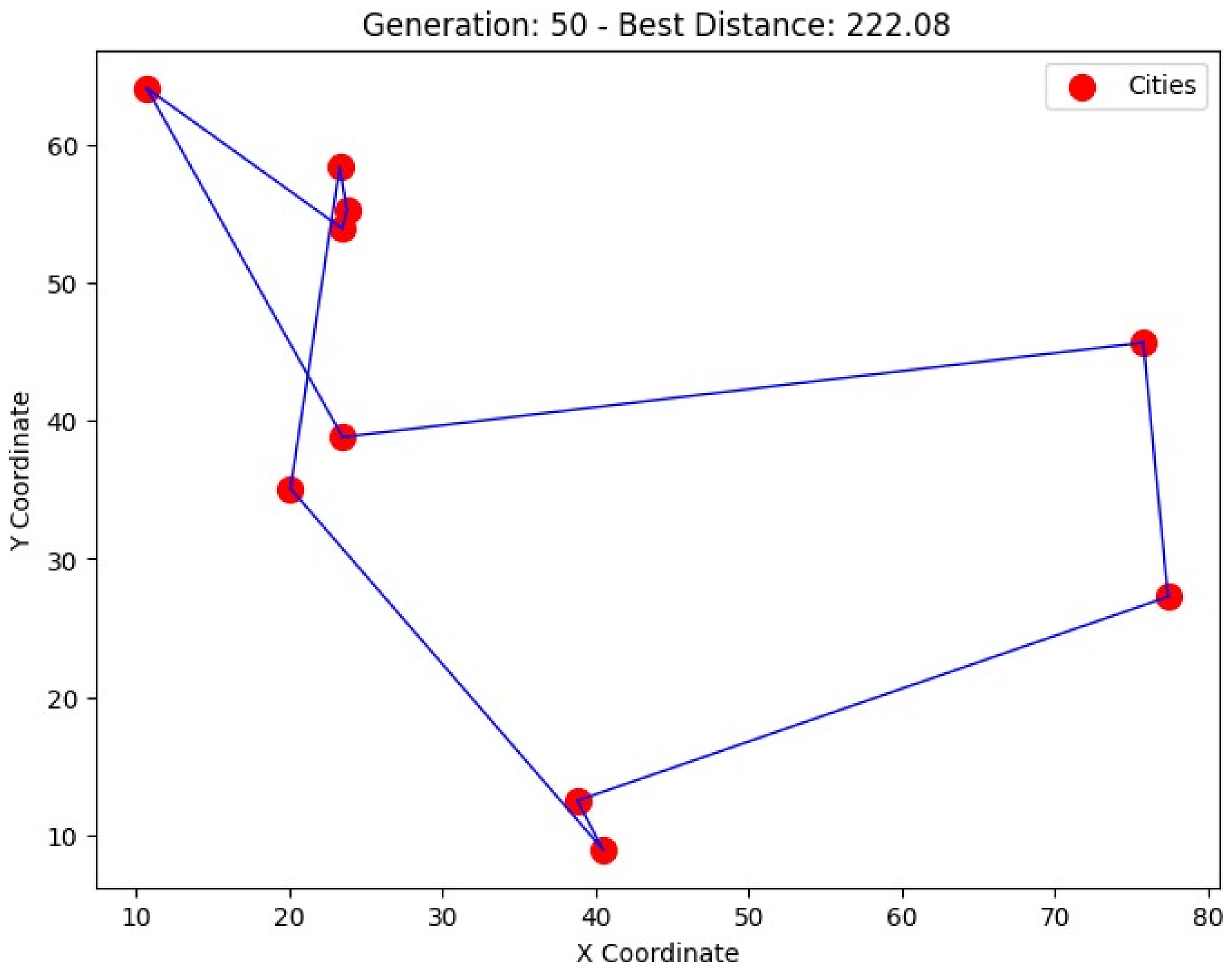

Generation: 0 - Best Distance: 254.72
Cities
Y Coordinate
X Coordinate
10
20
30
40
50
60
70
80

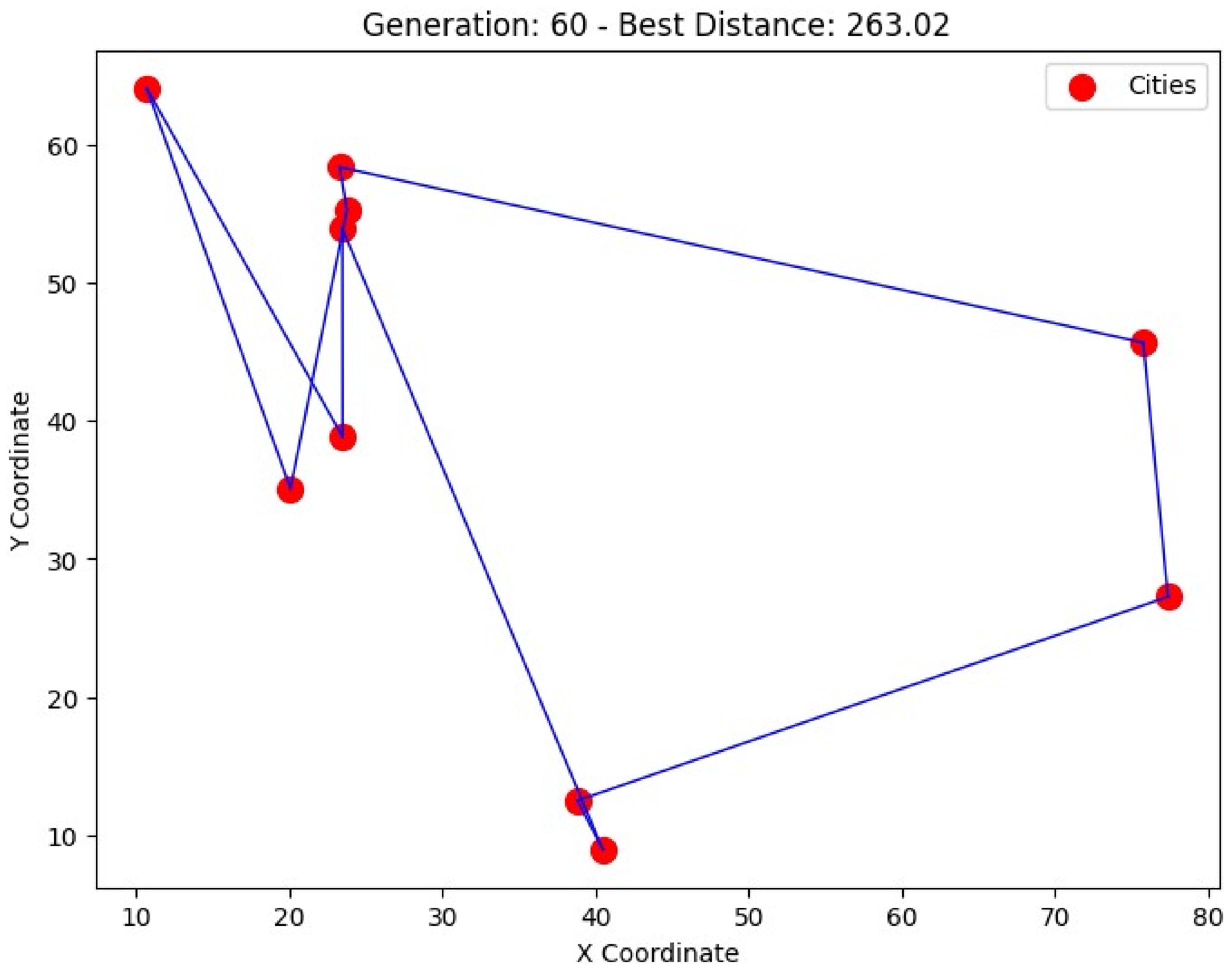

Generation: 10 - Best Distance: 244.98
Cities
Y Coordinate
X Coordinate
10
20
30
40
50
60
70
80
10
20
30
40
50
60

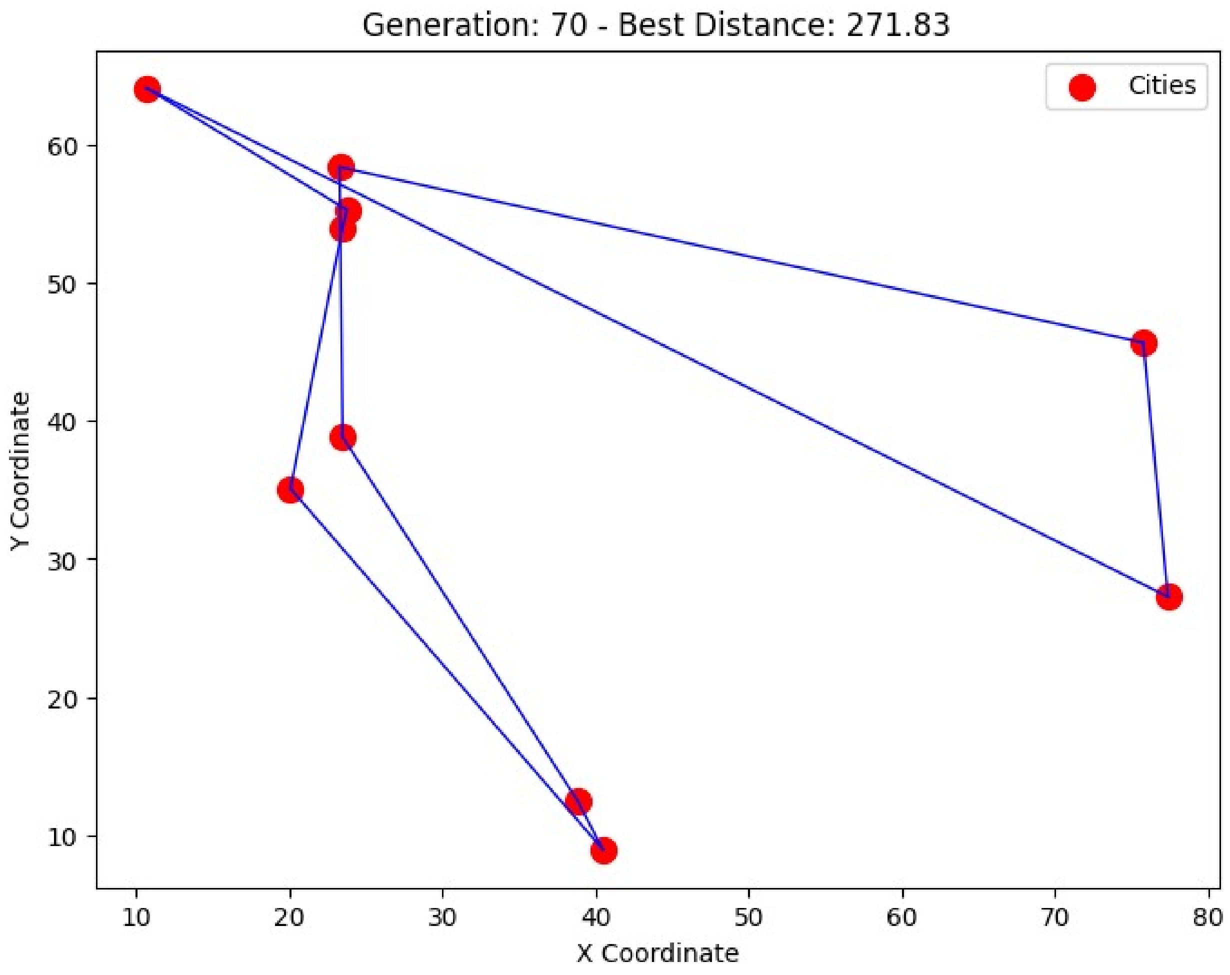

Generation: 20 - Best Distance: 222.98
Cities
Y Coordinate
X Coordinate
10
20
30
40
50
60
70
80
10
20
30
40
50
60

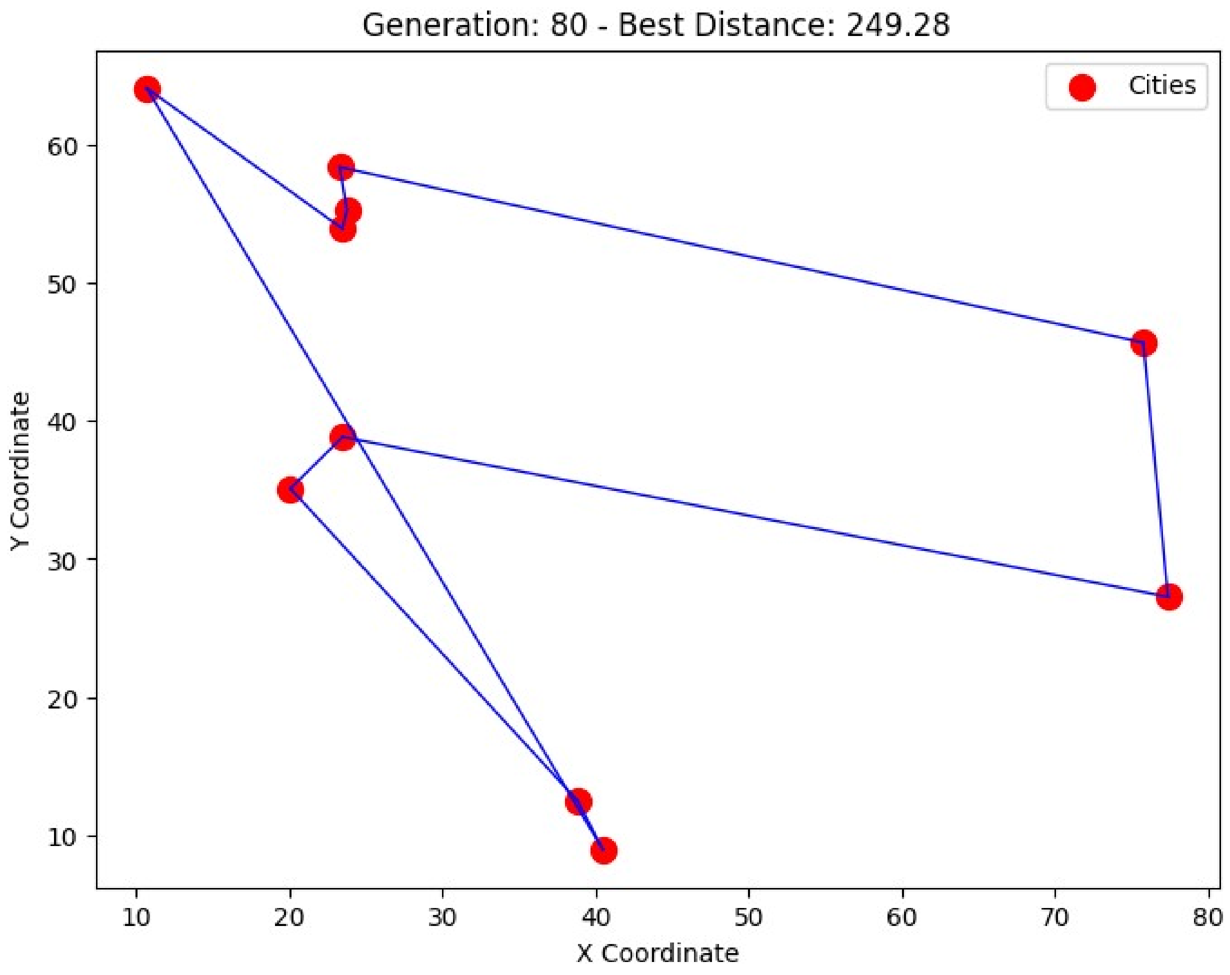

Generation: 30 - Best Distance: 239.65
Cities
Y Coordinate
X Coordinate
10
20
30
40
50
60
70
80

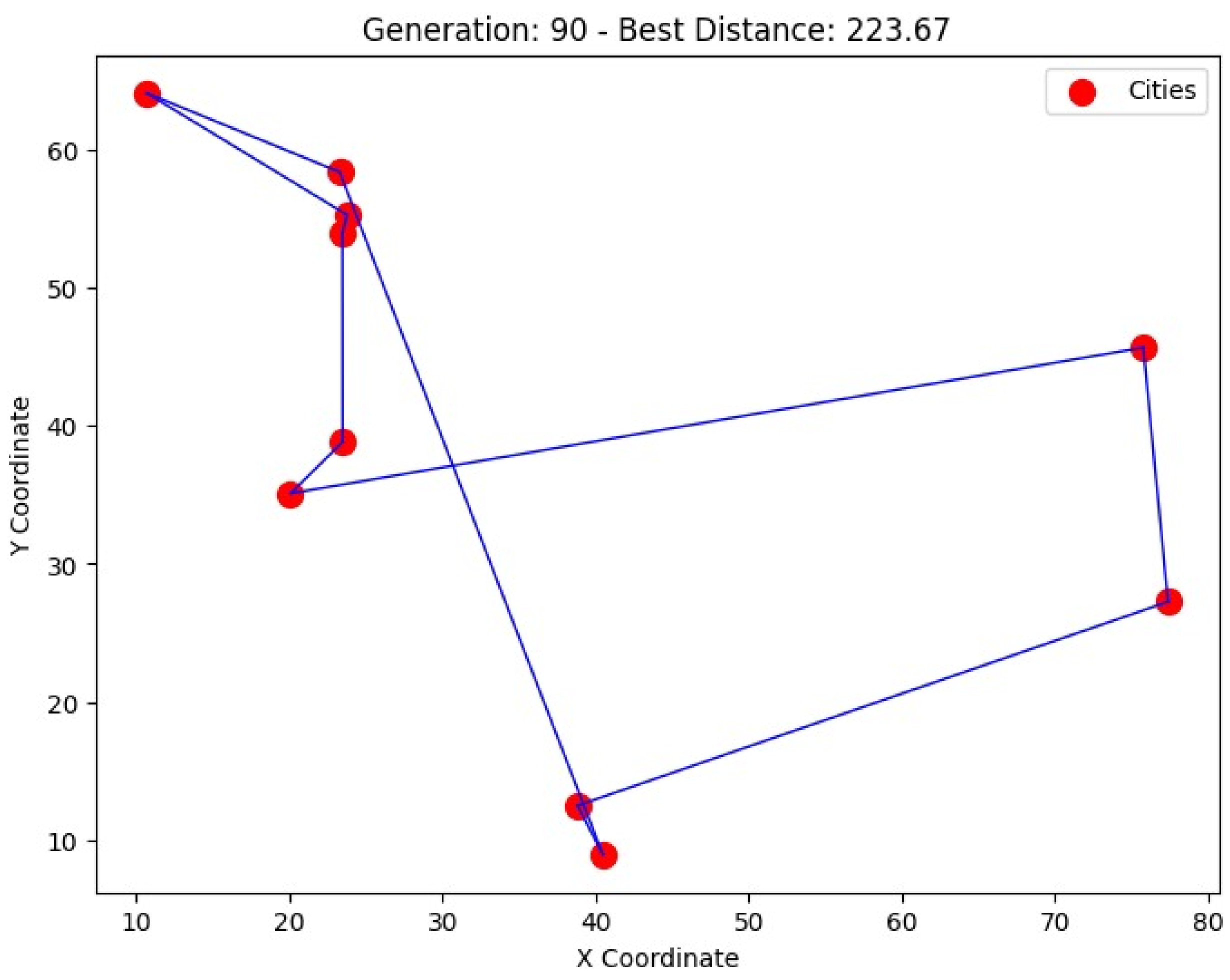

Generation: 40 - Best Distance: 205.23
Cities
Y Coordinate
X Coordinate
10
20
30
40
50
60
70
80

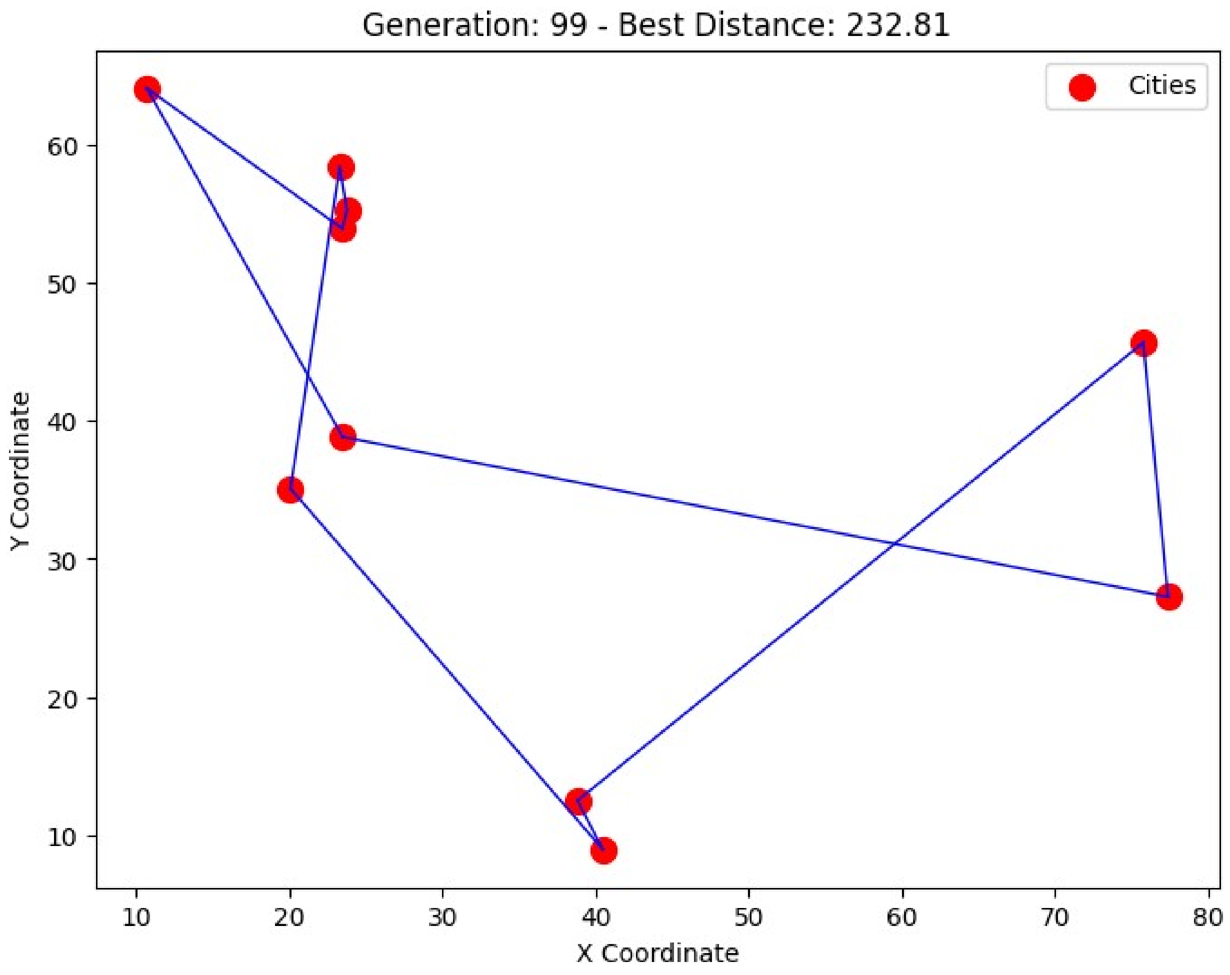

Generation: 50 - Best Distance: 222.08
Cities
Y Coordinate
X Coordinate
10
20
30
40
50
60
10
20
30
40
50
60
70
80

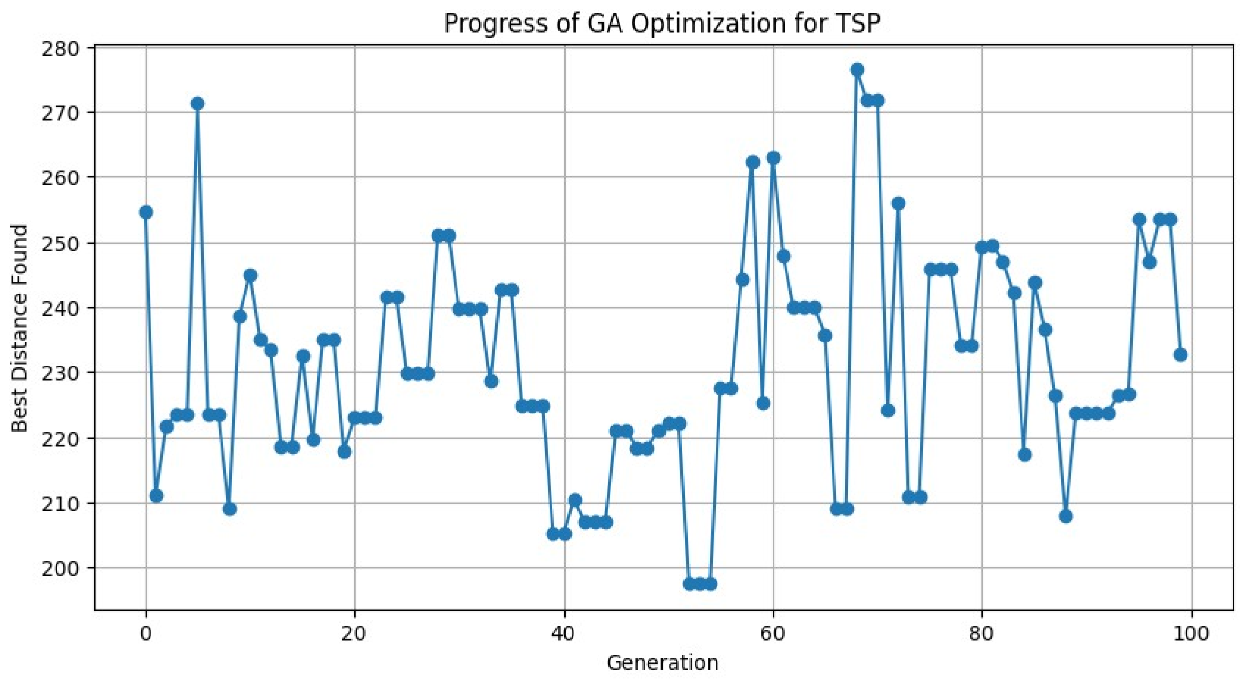

Generation: 60 - Best Distance: 263.02
Cities
Y Coordinate
X Coordinate
10
20
30
40
50
60
70
80


Generation: 70 - Best Distance: 271.83
Cities
Y Coordinate
X Coordinate
10
20
30
40
50
60
70
80


Generation: 80 - Best Distance: 249.28
Cities
Y Coordinate
X Coordinate
10
20
30
40
50
60
70
80


Generation: 90 - Best Distance: 223.67
Cities
Y Coordinate
X Coordinate
10
20
30
40
50
60
70
80


Generation: 99 - Best Distance: 232.81
Cities
Y Coordinate
X Coordinate


Progress of GA Optimization for TSP
Best Distance Found
Generation