TSP solved using ANT COLONY SYSTEM OPTIMIZATION(ACO)

Student name: NEAGU MIHNEA

1. Problem: Traveling Salesman Problem(TSP)

Problem Explanation: The Traveling Salesman Problem (TSP) is a classic problem in combinatorial optimization. Given a set of cities and the distances between each pair of cities, the objective is to find the shortest possible route that visits each city exactly once and returns to the starting city. The problem is named after the scenario faced by a salesman who needs to visit a set of cities to sell products, aiming to minimize the total distance traveled while visiting each city exactly once. The TSP is considered one of the most well-known NP-hard problems, meaning that finding an optimal solution becomes increasingly difficult as the number of cities increases.

2. ACO Algorithm

Step 1:

```python
import random
import numpy as np


2 usages
class Ant:
    def __init__(self, num_cities, alpha, beta, q0, p, pheromone_matrix, distances):
        self.num_cities = num_cities
        self.alpha = alpha
        self.beta = beta
        self.q0 = q0
        self.p = p
        self.pheromone_matrix = pheromone_matrix
        self.distances = distances
        self.visited = [False] * num_cities
        self.tour = []
        self.tour_length = 0
```

The **Ant** class encapsulates the behavior of an ant in the Ant Colony System (ACS) algorithm for the Traveling Salesman Problem (TSP). It defines attributes such as the number of cities, parameters influencing decision-making (alpha, beta, q0), and matrices for pheromone levels and distances between cities. The ant keeps

track of visited cities, its tour, and the tour's length. This class forms the basis for ant behavior in constructing solutions to the TSP using ACS.

Step 2:

```
def transition_probability(self, current_city, next_city):
    if self.visited[next_city]:
        return 0

    pheromone = self.pheromone_matrix[current_city][next_city]
    distance = self.distances[current_city][next_city]

    if distance == 0:
        return 0

    vision = 1 / distance

    numerator = pheromone ** self.alpha * vision ** self.beta
    denominator = sum((self.pheromone_matrix[current_city][neighbor] ** self.alpha *
                       (1 / self.distances[current_city][neighbor]) ** self.beta)
                       for neighbor in range(self.num_cities) if not self.visited[neighbor])

    return numerator / denominator

1 usage
def select_next_city(self, current_city):
    q = random.random()
    if q < self.q0:
        return self.select_next_city_greedy(current_city)
    else:
        return self.select_next_city_probabilistic(current_city)
```

The transition_probability method in the Ant class computes the probability of transitioning from the current city to the next city based on the amount of pheromone on the edge between them and the distance between them. This probability calculation adheres to the state transition rule of the ACS algorithm. The formula used for calculating the transition probability is as follows:

State transition rule:

$P_{xy} = ((\tau_{xy})^\alpha * (\eta_{xy})^\beta) / (\sum_{z \in allowed} (\tau_{xz})^\alpha * (\eta_{xz})^\beta)$

Where:

- **P_{xy}** is the probability for the ant to move from city x to city y.

- **τ_{xy}** is the amount of pheromone on the edge from city x to city y.

- **η_{xy}** represents the vision of the ant, calculated as 1/d_{xy}, where d_{xy} is the distance between cities x and y.

- **α** and **β** are parameters controlling the relative importance of pheromone and ant vision, respectively.

- allowed is the set of cities that are allowed to be visited from the current city x.is the set of cities that are allowed to be visited from the current city x.

The select_next_city method then utilizes this probability to select the next city for the ant to visit. This probabilistic strategy, based on the state transition rule, ensures that ants select their next city with a balance between exploitation and exploration, contributing to the overall convergence of the algorithm towards optimal solutions.

Step 3:

```
def select_next_city_greedy(self, current_city):
    unvisited_cities = [city for city in range(self.num_cities) if not self.visited[city]]
    max_pheromone = float('-inf')
    selected_city = -1
    for city in unvisited_cities:
        pheromone = self.pheromone_matrix[current_city][city] ** self.alpha
        attractiveness = (1.0 / self.distances[current_city][city]) ** self.beta
        if pheromone * attractiveness > max_pheromone:
            max_pheromone = pheromone * attractiveness
            selected_city = city
    return selected_city
1 usage
def select_next_city_probabilistic(self, current_city):
    unvisited_cities = [city for city in range(self.num_cities) if not self.visited[city]]
    probabilities = [self.transition_probability(current_city, next_city) for next_city in unvisited_cities]
    selected_city = random.choices(unvisited_cities, weights=probabilities)[0]
    return selected_city
```

```
def find_tour(self, pheromone_matrix):
    start_city = random.randint( a: 0, self.num_cities - 1)
    self.visited[start_city] = True
    self.tour.append(start_city)
    current_city = start_city
    while len(self.tour) < self.num_cities:
        next_city = self.select_next_city(current_city)
        self.visited[next_city] = True
        self.tour.append(next_city)
        self.tour_length += self.distances[current_city][next_city]
        # Local pheromone update rule
        pheromone_matrix[current_city][next_city] = (1 - self.p) * pheromone_matrix[current_city][
            next_city] + self.p * 1
        current_city = next_city
    self.tour_length += self.distances[self.tour[-1]][self.tour[0]]
```

These methods in the Ant class are responsible for selecting the next city for the ant to visit, employing different strategies based on the ACS algorithm's rules.

The select_next_city_greedy method implements a greedy strategy for selecting the next city based on the attractiveness of each unvisited city. It iterates over all unvisited cities from the current city and calculates the attractiveness of each city based on the amount of pheromone on the edge and the inverse of the distance to that city. The city with the highest product of pheromone and attractiveness is selected as the next city for the ant to visit.

The select_next_city_probabilistic method implements a probabilistic strategy for selecting the next city based on transition probabilities calculated using the transition_probability method. It calculates the probability for each unvisited city and selects the next city probabilistically using these probabilities. The random.choices function is used for probabilistic selection based on the calculated probabilities.

The find_tour method orchestrates the process of finding a complete tour for the ant. It starts by randomly selecting a start city and marks it as visited. Then, it iteratively selects the next city using one of the two selection strategies (select_next_city_greedy or select_next_city_probabilistic) until all cities have been visited. It updates the tour and tour length accordingly. This method is crucial as it incorporates one of the three rules of the ACS algorithm, namely the local pheromone update rule, which ensures the effective exploration of the solution space.

Local Pheromone Update Rule:

After an ant moves from city x to city y, the amount of pheromone on that edge is updated as follows:

$$\tau_{xy}(pas\_curent+1) = (1 - \rho) * \tau_{xy}(pas\_curent) + \rho * \tau 0$$

Where:

τ_{xy}(current_step+1) - the amount of pheromone on the edge from city x to city y after the ant moves

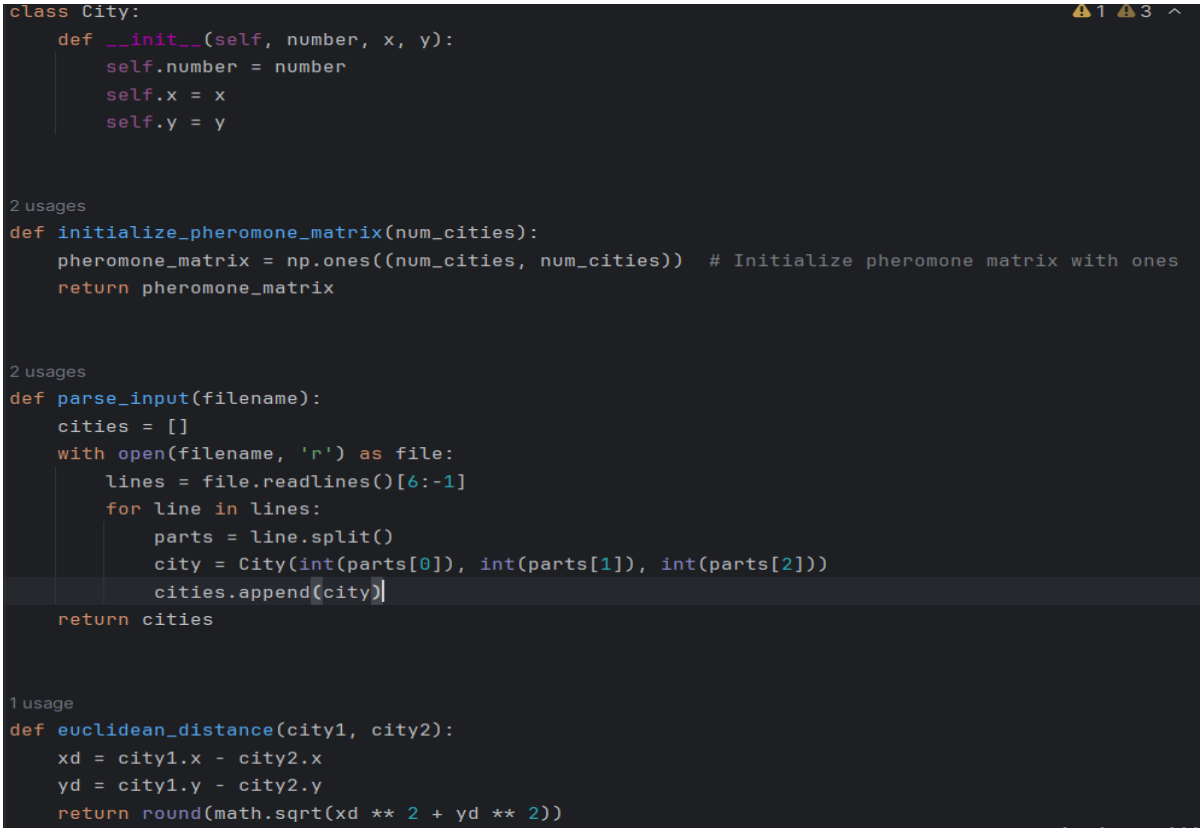
τ_{xy}(current_step) - the current amount of pheromone on the edge from city x to city y

ρ - the local pheromone evaporation rate

$\tau 0$ - a constant representing the initial amount of pheromone(in code written directly as 1 because the initial pheromone matrix is initialized with values of one

These methods collectively ensure that the ant constructs a tour while following the rules of the ACS algorithm, contributing to the search for an optimal solution to the TSP.

Step 4:



```python
class City:
    def __init__(self, number, x, y):
        self.number = number
        self.x = x
        self.y = y


2 usages
def initialize_pheromone_matrix(num_cities):
    pheromone_matrix = np.ones((num_cities, num_cities))  # Initialize pheromone matrix with ones
    return pheromone_matrix


2 usages
def parse_input(filename):
    cities = []
    with open(filename, 'r') as file:
        lines = file.readlines()[6:-1]
        for line in lines:
            parts = line.split()
            city = City(int(parts[0]), int(parts[1]), int(parts[2]))
            cities.append(city)
    return cities


1 usage
def euclidean_distance(city1, city2):
    xd = city1.x - city2.x
    yd = city1.y - city2.y
    return round(math.sqrt(xd ** 2 + yd ** 2))
```

These functions in the provided Python file are integral to solving the Traveling Salesman Problem (TSP) using the Ant Colony System (ACS) algorithm. They facilitate crucial aspects of the algorithm:

initialize_pheromone_matrix(num_cities): This function initializes the pheromone matrix, which plays a vital role in guiding the ants' exploration. It sets the initial amount of pheromone on each edge between cities to one.

parse_input(filename): This function parses the input file containing information about cities. It extracts data such as city number, x-coordinate, and y-coordinate from each line, creating City objects for each city. These City objects are essential for the algorithm's execution.

euclidean_distance(city1, city2): This function calculates the Euclidean distance between two cities based on their coordinates. It computes the straight-line distance between two points in Euclidean space, providing a key metric for evaluating routes in the TSP.

Together, these functions are really important for solving the TSP, handling tasks such as initializing pheromone levels, parsing input data, and computing distances between cities. They are fundamental in orchestrating the ants' exploration and ultimately finding an optimal solution to the TSP.

Step 5:

```python
def calculate_distances(cities):
    num_cities = len(cities)
    distances = np.zeros((num_cities, num_cities))
    for i in range(num_cities):
        for j in range(i + 1, num_cities):
            distance = euclidean_distance(cities[i], cities[j])
            distances[i][j] = distance
            distances[j][i] = distance
    return distances


1 usage
def update_pheromone(pheromone_matrix, ants, evaporation_rate, p, L_plus):
    num_cities = len(pheromone_matrix)
    # Evaporation
    pheromone_matrix *= (1 - evaporation_rate)
    # Pheromone deposit
    for ant in ants:
        for i in range(len(ant.tour) - 1):
            city1, city2 = ant.tour[i], ant.tour[i + 1]
            delta_tau = 1 / L_plus if ant.tour_length == L_plus else 0
            pheromone_matrix[city1][city2] += p * delta_tau
```

calculate_distances(cities):

This function calculates the distances between all pairs of cities using the Euclidean distance formula. It constructs a symmetric distance matrix where each entry represents the distance between two cities. This matrix is essential for evaluating the tour lengths and guiding the ants during their exploration.

update_pheromone(pheromone_matrix, ants, evaporation_rate, p, L_plus):

This function updates the pheromone levels on each edge based on the ants' tours and the ACS algorithm's rules. It first performs pheromone evaporation by reducing the pheromone levels on all edges. Then, it deposits pheromone on edges traversed by the ants, with the amount deposited depending on the length of the ant's tour and the algorithm's parameters.

Global Pheromone Update Rule:

$\Delta\tau_{xy} = (1 - \rho) * \tau_{xy} + \sum(1 / L_{plus}) * p$

Where:

- **$\Delta\tau_{xy}$** is the amount of pheromone deposited on the edge (x, y).

- **ρ** is the global pheromone evaporation rate.
- **τ_xy** is the current amount of pheromone on the edge (x, y).
- **L_plus** is the length of the best tour found until that iteration.
- **p** is a parameter controlling the pheromone deposit.

Together, these functions contribute to the global pheromone update process in the ACS algorithm, playing a crucial role in shaping the ants' exploration and convergence towards optimal solutions.

Step 6:

```
def ant_colony_system(num_ants, alpha, beta, q0, p, evaporation_rate, cities, distances, max_iterations):
    for _ in range(10):
        num_cities = len(cities)
        pheromone_matrix = initialize_pheromone_matrix(num_cities)
        best_solution = None
        best_solution_length = float('inf')
        route_lengths = []

        start_time = time.time()

        for iteration in range(max_iterations):
            ants = [Ant(num_cities, alpha, beta, q0, p, pheromone_matrix, distances) for _ in range(num_ants)]
            for ant in ants:
                ant.find_tour(pheromone_matrix)
                if ant.tour_length < best_solution_length:
                    best_solution = ant.tour
                    best_solution_length = ant.tour_length
            route_lengths.append(best_solution_length)
            update_pheromone(pheromone_matrix, ants, evaporation_rate, p, best_solution_length)

        end_time = time.time()
        runtime = end_time - start_time

        return best_solution, best_solution_length, route_lengths, runtime, pheromone_matrix
```

The ant_colony_system function implements the Ant Colony System (ACS) algorithm for solving the Traveling Salesman Problem (TSP). Let's delve into its code:

Initialization: The function starts by initializing essential variables, including the number of cities, the pheromone matrix, and metrics to track the best solution and route lengths explored.

Main Loop: It executes a loop to perform multiple runs of the ACS algorithm (here set to 10 runs). Each run consists of the following steps:

a. Pheromone Matrix Initialization: At the beginning of each run, a new pheromone matrix is initialized.

b. Ant Construction: Ants are created, each representing a potential solution to the TSP. They construct their tours by probabilistically selecting the next city to visit based on pheromone levels and heuristic information (alpha and beta parameters).

c. Solution Update: After each ant completes its tour, the algorithm checks if it has found a better solution than the current best one. If so, it updates the best solution and its length accordingly.
d. Pheromone Update: The pheromone matrix is updated based on the quality of the solutions found by the ants. Better solutions contribute more to the pheromone levels on the edges traversed.
Runtime Calculation: The function records the runtime of the ACS algorithm for each run.
Result Return: At the end of each run, it returns crucial information, including the best solution found, its length, the route lengths explored during iterations, the runtime, and the final pheromone matrix.
This function encapsulates the core mechanics of the ACS algorithm, iterating over multiple runs to converge towards an optimal solution for the TSP.

Step 7:

```
def plot_pheromone_matrix(pheromone_matrix, cities):
    num_cities = len(cities)
    plt.figure(figsize=(10, 8))
    plt.imshow(pheromone_matrix, cmap='hot', interpolation='nearest')
    plt.colorbar(label='Pheromone Level')
    plt.xticks(range(num_cities), [city.number for city in cities], rotation=45)
    plt.yticks(range(num_cities), [city.number for city in cities])
    plt.xlabel('To City')
    plt.ylabel('From City')
    plt.title('Pheromone Matrix')
    plt.show()
```

The plot_pheromone_matrix function generates a heatmap visualization of the pheromone matrix, representing pheromone levels on edges between cities. Lighter colors indicate higher pheromone levels, which are more frequently used by the ants, while darker colors represent lower pheromone levels, which are less utilized. It takes the pheromone_matrix and a list of cities as inputs and displays the heatmap plot with labeled axes and a color bar indicating pheromone levels.

Step 8:

```python
def main_ant_colony_system():
    num_ants = 20
    alpha = 1
    beta = 2
    q0 = 0.9
    p = 0.1
    evaporation_rate = 0.1
    max_iterations = 100

    # Parse input
    cities = parse_input("tsp.txt")
    distances = calculate_distances(cities)

    # Run ACS
    (best_solution, best_solution_length, route_lengths, runtime,
     pheromone_matrix)= ant_colony_system(num_ants, alpha, beta, q0, p,
                                          evaporation_rate, cities, distances,
                                          max_iterations)

    print("Best solution:", best_solution)
    print("Best solution length:", best_solution_length)
    print("Runtime:", runtime, "seconds")

    # Calculate average route value
    average_route_value = sum(route_lengths) / len(route_lengths)
    print("Average route value over iterations:", average_route_value)

    # Plot pheromone matrix for the best solution
    plot_pheromone_matrix(pheromone_matrix, cities)
```

The main_ant_colony_system function serves as the entry point for running the Ant Colony System (ACS) algorithm on a given TSP instance. Here's a breakdown of its functionality:

Parameter Initialization: It sets up the parameters required for configuring the ACS algorithm, such as the number of ants, alpha and beta parameters controlling the influence of pheromone levels and heuristic information, the probability parameter q0, the pheromone update rate p, the evaporation rate, and the maximum number of iterations.

Input Parsing: It reads the TSP input file (tsp.txt) and parses the city coordinates to create a list of cities. Additionally, it calculates the distances between each pair of cities using the calculate_distances function.

ACS Execution: It invokes the ant_colony_system function with the initialized parameters and input data. This executes the ACS algorithm, finding the best solution to the TSP instance.

Results Display: After the ACS algorithm completes, it prints out essential information about the obtained solution, including the best solution found, its length, and the runtime in seconds.

Average Route Value Calculation: It calculates the average route value over the iterations of the ACS algorithm by averaging the route lengths recorded during each iteration.

Pheromone Matrix Plotting: Finally, it plots the pheromone matrix corresponding to the best solution found by the ACS algorithm. This visualization helps in understanding which edges have higher pheromone levels, indicating more frequently traveled routes.

3. Plots and parameters optimization.

Firstly, we will create a data table that will compare the best values of 3 different algorithms that all have the aim to solve the TSP problem. The 3 algorithms are: ACO(Ant Colony Optimization), Evolutive and SA(Simmulated Annealing). Secondly, we will showcase the pheromone matrix for the optimal parameter setting for the ACS problem. Then, we will experiment with 2 important parameters used in the algorithm: the number of ants and the coefficient with which the pheromone evaporates.

Data table:

Problem Instance	Iterations/ Generations	Best Value	Average Value	Nr of executions	Average runtime
ACO1 pr76.tsp	100	120951	125044.51	10	8.7
ACO2 pr76.tsp	1000	119554	121597.51	10	155.26
ACO3 pr76.tsp	10000	107293	118390.83	10	317.8
SA1 pr76.tsp	1000	439180	500805	10	0.15
SA2 pr76.tsp	10000	330146	452766	10	1.5
SA3 pr76.tsp	100000	116808	173432	10	15.4

Evolutive1 pr76.tsp	10000	414256	498214	10	33.3
Evolutive2 pr76.tsp	100000	400803	481723	10	330

Data Interpretation: Based on the provided data table, it's evident that the Ant Colony Optimization (ACO) algorithm consistently outperforms Simulated Annealing (SA) and Evolutionary algorithms across different iterations or generations. In terms of both the best and average values obtained, ACO consistently achieves superior results compared to SA and Evolutionary algorithms. However, it's noteworthy that ACO tends to have longer runtimes compared to SA and Evolutionary algorithms. Despite this, the higher quality of solutions obtained by ACO makes it the preferred choice for solving the given problem instance.

Experiment 1:

```
num_ants = 20
alpha = 1
beta = 2
q0 = 0.9
p = 0.1
evaporation_rate = 0.1
max_iterations = 100
```

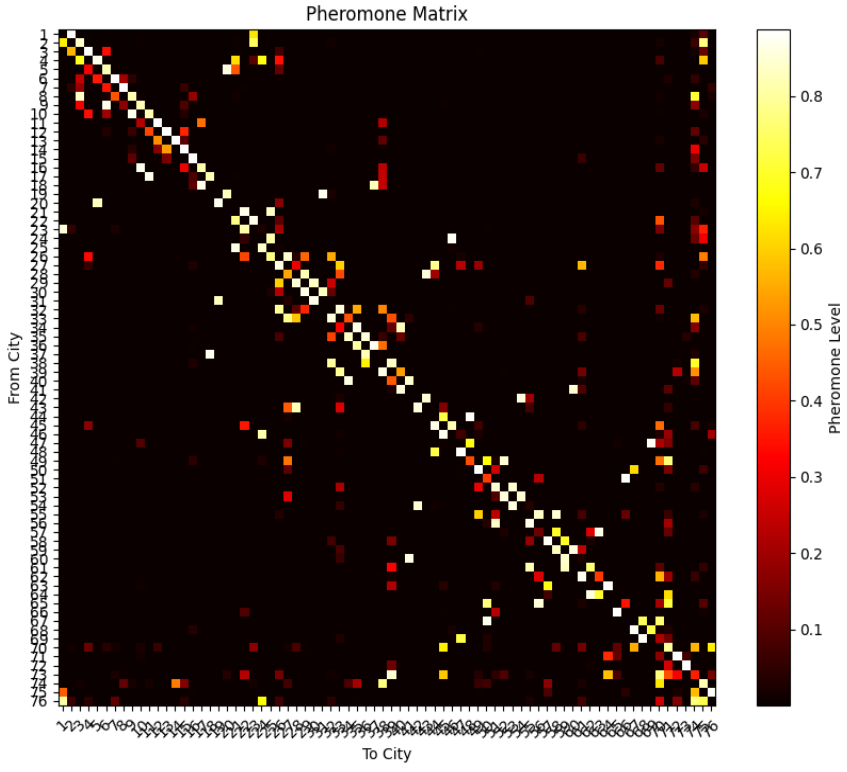
Output 1:

Best solution length: 125161.0

Runtime: 9.133601903915405 seconds

Average route value over iterations: 126442.5

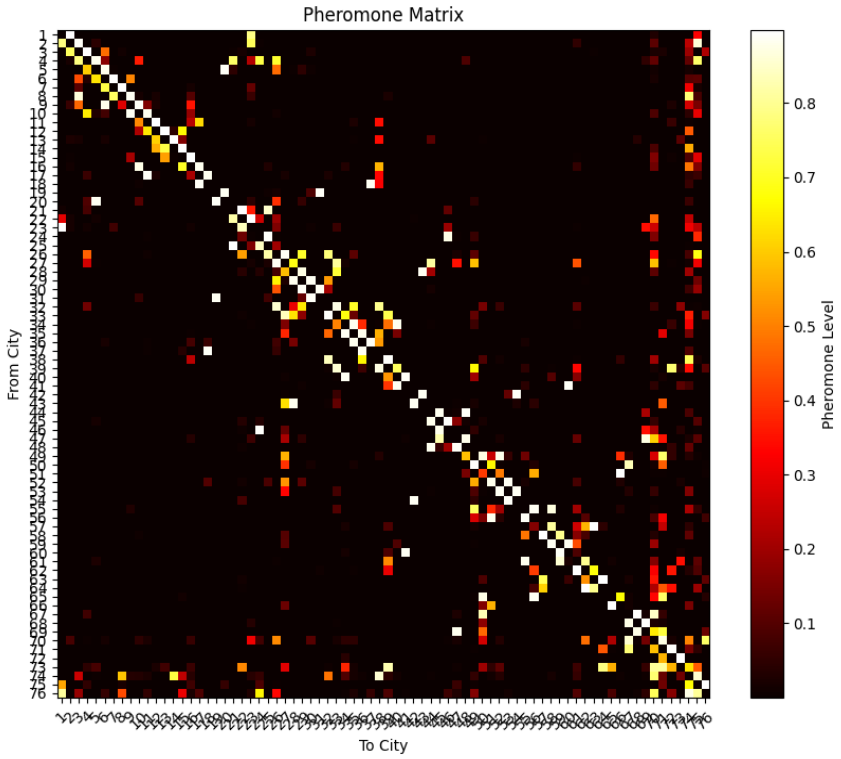
Plot 1:



Experiment 2:

```
num_ants = 40
alpha = 1
beta = 2
q0 = 0.9
p = 0.1
evaporation_rate = 0.1
max_iterations = 100
```

Output 2: Best solution length: 123102.0
Runtime: 26.27910351753235 seconds
Average route value over iterations: 125123.15
Plot 2:



Experiment 1 and 2 Comparison: The two experiments conducted with different numbers of ants, 20 and 40, using the same dataset demonstrated that employing a smaller number of ants (20) yielded superior results compared to a larger number (40). Specifically, the experiment with 20 ants achieved better values for the best route length, average route length, and runtime compared to the

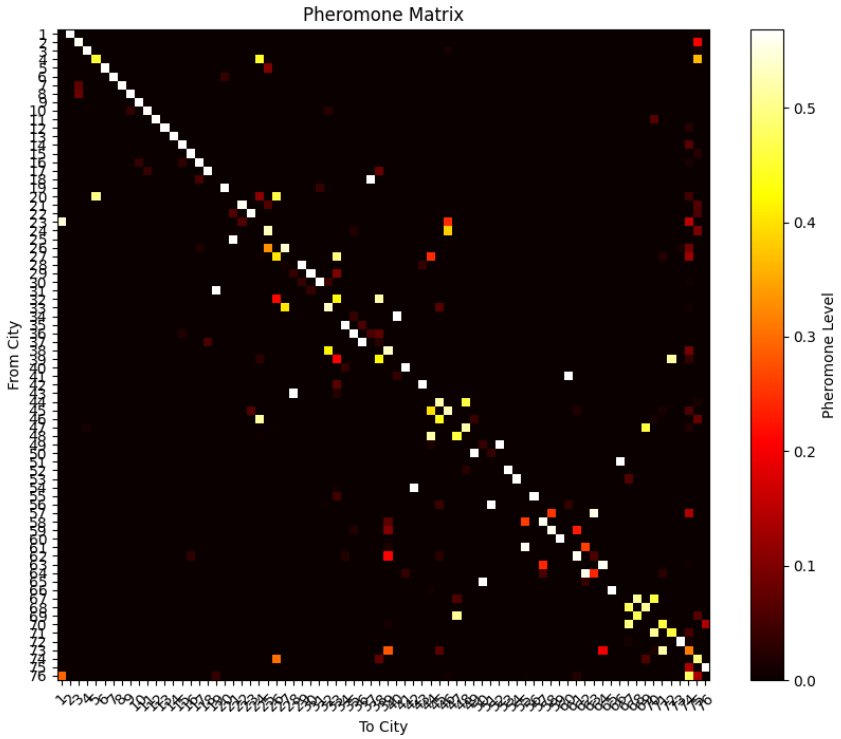
experiment with 40 ants. This suggests that a smaller number of ants led to more efficient exploration of the solution space, resulting in improved optimization outcomes.

Experiment 3:

```
num_ants = 20
alpha = 1
beta = 2
q0 = 0.9
p = 0.1
evaporation_rate = 0.4
max_iterations = 100
```

Output 3:

Best solution length: 122972.0
Runtime: 7.556994676589966 seconds
Average route value over iterations: 124115.8
Plot 3:



Experiment 4:

```
num_ants = 20
alpha = 1
beta = 2
q0 = 0.9
p = 0.1
evaporation_rate = 0.001
max_iterations = 100
```

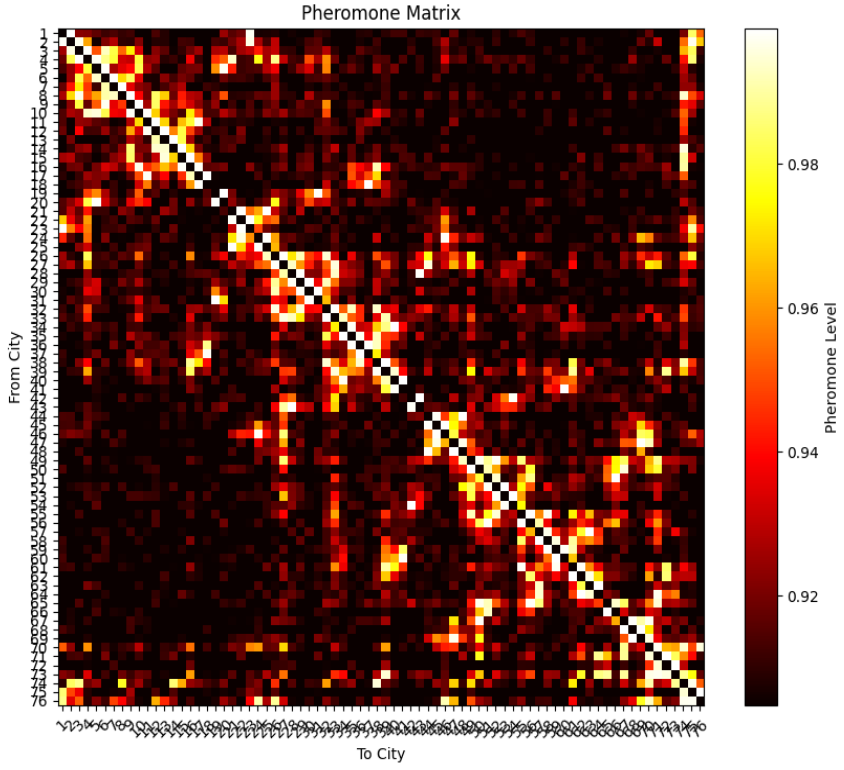
Output 4:

Best solution length: 125488.0

Runtime: 9.24313449859619 seconds

Average route value over iterations: 126502.15

Plot 4:



Experiment 3 and 4 Comparison: The experiments conducted with different evaporation rates, 0.4 and 0.001, yielded notably different outcomes. In the experiment with an evaporation rate of 0.4, the best solution length was 122972.0, with a runtime of 7.56 seconds, and an average route value over iterations of 124115.8. Conversely, the experiment with an evaporation rate of 0.001 resulted in a slightly higher best solution length of 125488.0, a longer runtime of 9.24 seconds, and a higher average route value over iterations of 126502.15.

The major difference between these experiments is reflected in the plots of the pheromone matrix. In the experiment with a high evaporation rate (0.4), the pheromone trails are barely visible, indicating rapid decay of pheromone levels. Conversely, in the experiment with a low evaporation rate (0.001), the pheromone trails are vivid and colorful, with many roads visibly marked by pheromones. This suggests that a lower evaporation rate allows for better retention of pheromone information, leading to more persistent and effective exploration of the solution space by the ants