Experiment No. 6

Implement Ant colony optimization by solving the Traveling salesman problem using python Problem statement- salesman needs to visit a set of cities exactly once and return to the original city. The task is to find the shortest possible route that the salesman can take to visit all the cities and return to the starting city.

Theory:

The Traveling Salesman Problem (TSP) is a classic optimization problem where a salesman needs to find the shortest route to visit a set of cities exactly once and return to the starting city. One way to solve this problem is by using Ant Colony Optimization (ACO), a metaheuristic algorithm inspired by the foraging behaviour of ants.

In ACO, artificial ants construct solutions by probabilistically selecting cities to visit based on pheromone trails laid down by previous ants. Over time, the pheromone trails are updated based on the quality of the solutions found. This iterative process gradually converges towards an optimal or near-optimal solution.

Implementing ACO for the TSP in Python involves creating a graph representation of the cities and their connections, initializing pheromone levels, defining ant behaviour for constructing solutions, updating pheromone trails, and iterating the process until convergence criteria are met.

By combining the principles of ant behaviour and pheromone communication, ACO offers a powerful and efficient approach to solving combinatorial optimization problems like the TSP.

Steps to be followed:

1. **Initialize Parameters**:
   - Number of ants
   - Number of iterations
   - Pheromone evaporation rate
   - Initial pheromone level
   - Pheromone importance factor
   - Heuristic information importance factor
2. **Initialize Pheromone Trails**:
   - Set initial pheromone level on all edges to the chosen initial value.
3. **Repeat until convergence**:

   a. **Ant Construction Phase**:

   - For each ant:

     i. Choose a random city as the starting point.

     ii. While there are unvisited cities: - Calculate the probabilities of choosing each unvisited city based on pheromone levels and heuristic information. - Select the next city using a probability distribution (e.g., roulette wheel selection). - Move to the selected city.

   - Calculate the length of each ant's tour.

b. **Pheromone Update Phase**:

- Evaporate pheromone on all edges.
- Update pheromone levels on the edges of the tour taken by each ant:
- Add pheromone inversely proportional to the tour length.
- Optionally, add a constant amount of pheromone to encourage exploration.

c. **Global Update**:

- Optionally, update pheromone levels on all edges:
- Add pheromone proportional to the quality of the best solution found so far.

4. **Termination**:
    - Repeat until a stopping criterion is met (e.g., number of iterations, no significant improvement).
5. **Output**:
    - The best tour found.

This algorithm can be further refined by incorporating local search heuristics, adaptive pheromone updates, and various optimization techniques. Additionally, you will need to implement functions to calculate probabilities, update pheromone levels, and evaluate tour lengths based on the problem-specific constraints and objectives

The ant starts from a given node, and will at each step choose from among every node it has not stepped on yet, with a weighted distribution that assigns preference proportional to an edge's pheromone load and to the inverse of its distance, each raised to a power that is a hyperparameter coefficient (*alpha* and *beta* respectively).

That is, the probability of choosing a certain edge will be proportional to:

$$w = \frac{P_{ij}^{\alpha}}{D_{ij}^{\beta}}$$

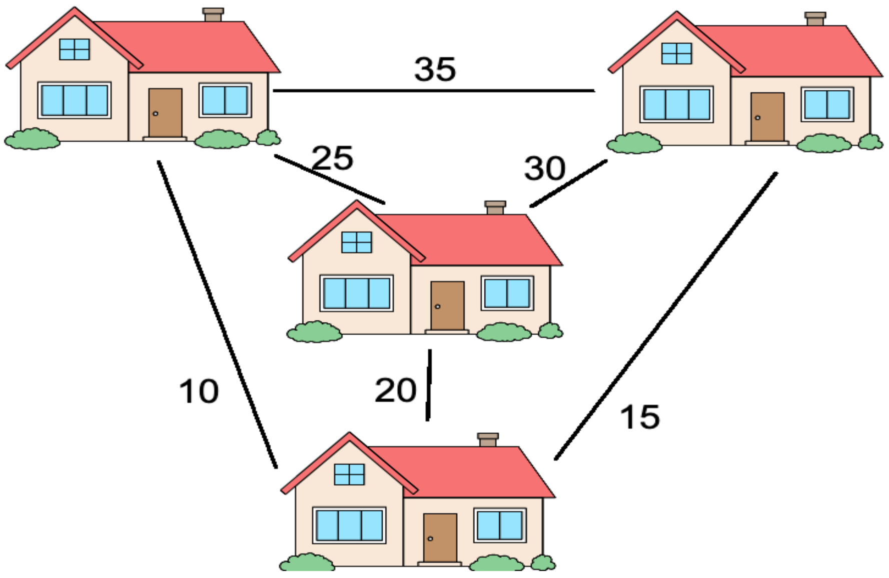
The Traveling Salesman Problem

One especially important use-case for Ant Colony Optimization (ACO from now on) algorithms is solving the Traveling Salesman Problem (TSP).

This problem is defined as follows: *Given a complete graph G with weighted edges, find the minimum weight Hamiltonian cycle. That is, a cycle that passes through each node exactly once and minimizes the total weight sum.*

Note that the graph needs to be *complete*: there needs to exist an edge connecting each possible pair of nodes. For graphs based in real places, this makes sense: you can just connect two places with an edge with a weight equal to their distance, or their estimated travel time.

For a concrete example, look at the following graph.

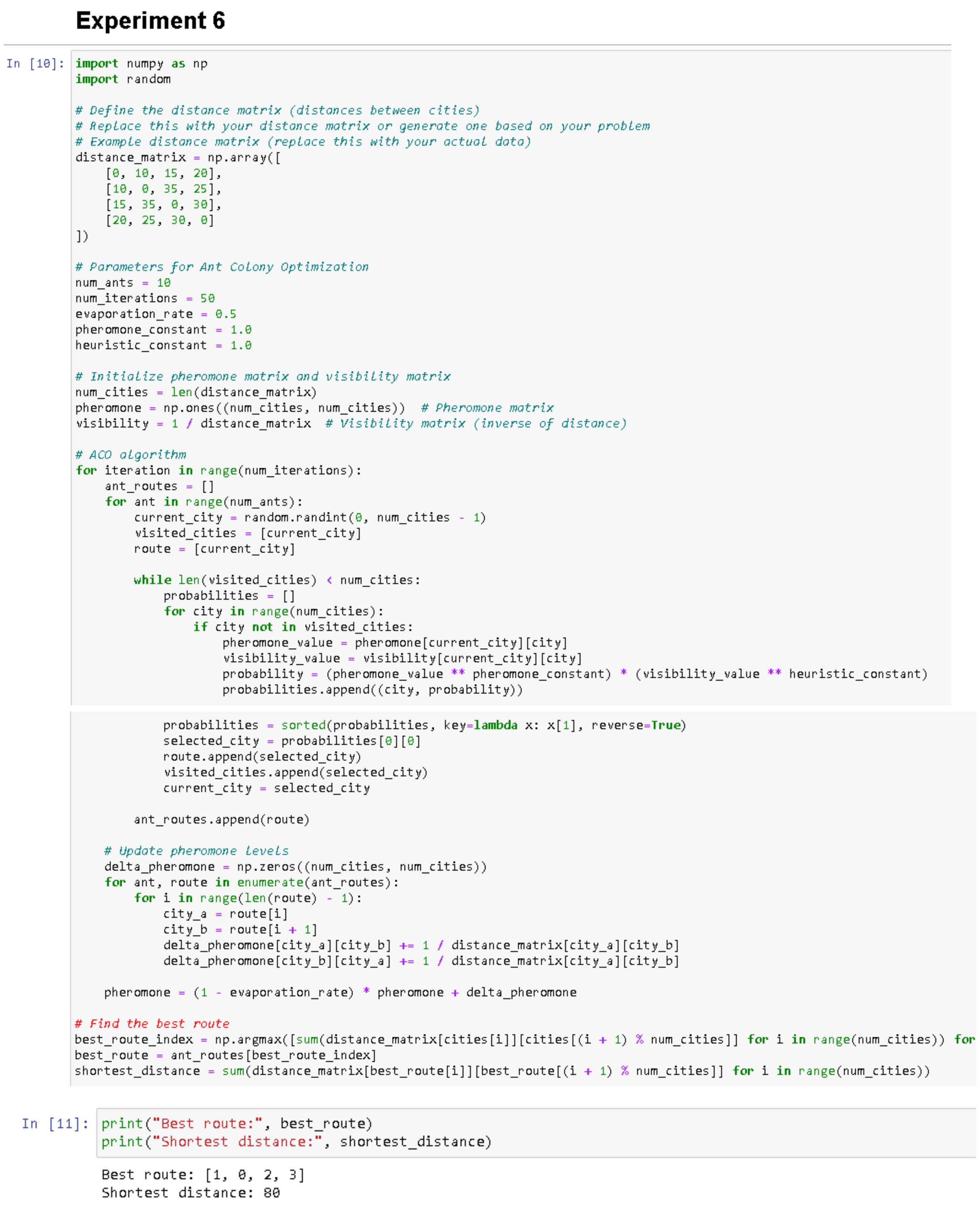


In this case, the salesman wants to visit every home once and get back to where it started. Each edge joining two houses has a numeric label, representing the travel time between them in minutes. The salesman is a busy man, and would prefer to take as little time as possible in visiting all the houses. What would be the most efficient route?

As an example, if we started from the house on the top left, we would want to go bottom, right, center, left again for a total of 80 minutes of travel. You can take a little time to convince yourself that is the right answer by hand, since this is a small case. Try to find a different route that would take less time to visit the four houses.

Conclusion: Thus, we have successfully implemented TSP.

Code and Output:



Experiment 6

```
In [10]: import numpy as np
         import random

         # Define the distance matrix (distances between cities)
         # Replace this with your distance matrix or generate one based on your problem
         # Example distance matrix (replace this with your actual data)
         distance_matrix = np.array([
             [0, 10, 15, 20],
             [10, 0, 35, 25],
             [15, 35, 0, 30],
             [20, 25, 30, 0]
         ])

         # Parameters for Ant Colony Optimization
         num_ants = 10
         num_iterations = 50
         evaporation_rate = 0.5
         pheromone_constant = 1.0
         heuristic_constant = 1.0

         # Initialize pheromone matrix and visibility matrix
         num_cities = len(distance_matrix)
         pheromone = np.ones((num_cities, num_cities))  # Pheromone matrix
         visibility = 1 / distance_matrix  # Visibility matrix (inverse of distance)

         # ACO algorithm
         for iteration in range(num_iterations):
             ant_routes = []
             for ant in range(num_ants):
                 current_city = random.randint(0, num_cities - 1)
                 visited_cities = [current_city]
                 route = [current_city]

                 while len(visited_cities) < num_cities:
                     probabilities = []
                     for city in range(num_cities):
                         if city not in visited_cities:
                             pheromone_value = pheromone[current_city][city]
                             visibility_value = visibility[current_city][city]
                             probability = (pheromone_value ** pheromone_constant) * (visibility_value ** heuristic_constant)
                             probabilities.append((city, probability))

                     probabilities = sorted(probabilities, key=lambda x: x[1], reverse=True)
                     selected_city = probabilities[0][0]
                     route.append(selected_city)
                     visited_cities.append(selected_city)
                     current_city = selected_city

                 ant_routes.append(route)

             # Update pheromone levels
             delta_pheromone = np.zeros((num_cities, num_cities))
             for ant, route in enumerate(ant_routes):
                 for i in range(len(route) - 1):
                     city_a = route[i]
                     city_b = route[i + 1]
                     delta_pheromone[city_a][city_b] += 1 / distance_matrix[city_a][city_b]
                     delta_pheromone[city_b][city_a] += 1 / distance_matrix[city_a][city_b]

             pheromone = (1 - evaporation_rate) * pheromone + delta_pheromone

         # Find the best route
         best_route_index = np.argmax([sum(distance_matrix[cities[i]][cities[(i + 1) % num_cities]] for i in range(num_cities)) for
         best_route = ant_routes[best_route_index]
         shortest_distance = sum(distance_matrix[best_route[i]][best_route[(i + 1) % num_cities]] for i in range(num_cities))
```

```
In [11]: print("Best route:", best_route)
         print("Shortest distance:", shortest_distance)

         Best route: [1, 0, 2, 3]
         Shortest distance: 80
```