AISSMS
INSTITUTE OF INFORMATION TECHNOLOGY
ADDING VALUE TO ENGINEERING

# Experiment No. 6

Title: Implementation of Travelling Salesman Problem using Python.

Aim:
To explore and implement the Nearest Neighbor algorithm as a heuristic approach to solve the Traveling Salesman Problem (TSP), aiming to find a suboptimal solution that visits each city exactly once and returns to the starting city.

Objectives:

1. Implement the Nearest Neighbor algorithm for solving TSP.
2. Demonstrate the effectiveness and efficiency of the algorithm in finding a suboptimal tour for small to moderate-sized instances of TSP.

Problem Statement:

Given a list of cities and the distances between them, the objective is to find the optimal tour that visits each city exactly once and returns to the starting city, minimizing the total distance traveled.

Software/s: Python (version 3.x)

Theory:
The Traveling Salesman Problem (TSP) is one of the most well-known and studied problems in the field of combinatorial optimization. It is a classic problem that has applications in various real-world scenarios, such as logistics, transportation, and circuit design. The problem is simple to state yet challenging to solve optimally, making it a subject of great interest for researchers and practitioners alike.
In the Traveling Salesman Problem, a salesman needs to visit a set of cities exactly once and return to the starting city, covering the shortest possible distance. This problem can be represented as a graph, where cities are nodes and the distances between them are edges. The objective is to find a tour that visits each city exactly once and returns to the starting city, minimizing the total distance traveled.
Despite its seemingly straightforward formulation, finding the optimal solution to the Traveling Salesman Problem is computationally intensive, especially as the

number of cities increases. In fact, TSP is classified as an NP-hard problem, meaning that there is no known

algorithm that can solve it optimally in polynomial time for all instances. As a result, researchers have developed various heuristic and approximation algorithms to tackle TSP and find good-quality solutions within reasonable time frames.

One such heuristic approach is the Nearest Neighbor algorithm, which provides a simple yet effective method for constructing a suboptimal tour for the Traveling Salesman Problem. The Nearest Neighbor algorithm starts from an arbitrary city and iteratively selects the nearest unvisited city as the next stop until all cities are visited. While this algorithm may not always produce the optimal solution, it is computationally efficient and easy to implement, making it suitable for practical applications, especially for small to moderate
sized instances of TSP.

In this assignment, we explore the Nearest Neighbor algorithm as a heuristic approach to solving the Traveling Salesman Problem. We discuss the algorithm's principles, outline its procedure, provide a Python implementation, and analyze its effectiveness in finding suboptimal tours for TSP instances. Through this exploration, we aim to gain insights into the challenges of TSP and the practical approaches to address them using heuristic techniques like the Nearest Neighbor algorithm.

Algorithm:

1. Start at an arbitrary city as the current city. Berlin
2. Mark the current city as visited.
3. Repeat the following steps until all cities have been visited:
    a. Find the nearest unvisited city.
    b. Move to the nearest unvisited city.
    c. Mark the nearest unvisited city as visited.
    d. Add the distance between the current city and the nearest unvisited city to the total tour length.
4. Once all cities have been visited, return to the starting city to complete the tour

Pseudocode:

```
nearest_neighbor_tsp(cities):
 current_city = randomly_select_starting_city(cities)
 tour = [current_city]
 mark_city_as_visited(current_city)

 while unvisited_cities_exist(cities):
 nearest_city = find_nearest_unvisited_city(current_city, cities)
 tour.append(nearest_city)
```

```
 mark_city_as_visited(nearest_city)
 current_city = nearest_city

 tour.append(tour[0]) // Complete the tour by returning to the starting
city   return tour

randomly_select_starting_city(cities):
 return random.choice(cities)

unvisited_cities_exist(cities):
 return any(city['visited'] == False for city in cities)

find_nearest_unvisited_city(current_city, cities):
 nearest_distance = infinity
 nearest_city = None
 for city in cities:
 if not city['visited']:
 distance = calculate_distance(current_city, city)
 if distance < nearest_distance:
 nearest_distance = distance
 nearest_city = city
 return nearest_city

mark_city_as_visited(city):
 city['visited'] = True

calculate_distance(city1, city2):
 return ((city2['x'] - city1['x'])^2 + (city2['y'] - city1['y'])^2)^0.5
```

Conclusion:

Department of Artificial Intelligence and Data Science