**Program 6**

# Write a Program to Implement Travelling Salesman Problem using Python.

# City 1 - 10 - City 2

# | |

# 20 15

# | |

# City 4 - 25 - City 3

**Problem Description :**

The **Traveling Salesman Problem (TSP)** is a classic optimization problem in computer science and operations research. The problem is defined as follows:

A **salesman** starts at a specific city, visits a set of cities exactly once, and then returns to the starting city.

The goal is to determine the shortest possible route that allows the salesman to visit all the cities and return to the starting point.

**Problem Definition:**

**Cities**: A list of cities (points) that the salesman needs to visit.

**Distances**: The distances between each pair of cities, which may be represented as a matrix (for example, a distance matrix).

**Objective**: Minimize the total travel distance or cost.

The TSP is NP-hard, meaning that there is no known polynomial-time algorithm to solve it. As the number of cities increases, the possible routes grow factorially, making it computationally expensive to solve for large datasets. However, various **heuristics** and **approximation algorithms** can provide solutions in reasonable time for large numbers of cities.

**Problem Representation:**

We can represent the cities and the distances between them as a **graph**, where:

1. The cities are **nodes**.

2. The distances between the cities are **edges**.

3. The task is to find the **shortest Hamiltonian cycle**, where a Hamiltonian cycle visits

each node exactly once and returns to the starting point.

**.**

**SOURCE CODE :**

from collections import deque

def tsp\_bfs(graph):

n = len(graph) # Number of cities

startCity = 0 # Starting city

min\_cost = float('inf') # Initialize minimum cost as infinity

opt\_path = [] # To store the optimal path

# Queue for BFS: Each element is (cur\_path, cur\_cost)

dq = deque([([startCity], 0)])

print("Path Traversal:")

while dq:

cur\_path, cur\_cost = dq.popleft()

cur\_city = cur\_path[-1]

# Print the current path and cost

print(f"Current Path: {cur\_path}, Current Cost: {cur\_cost}")

# If all cities are visited and we are back at the startCity

if len(cur\_path) == n and cur\_path[0] == startCity:

total\_cost = cur\_cost + graph[cur\_city][startCity]

if total\_cost < min\_cost:

min\_cost = total\_cost

opt\_path = cur\_path + [startCity]

continue

# Explore all neighboring cities (add in BFS manner)

for next\_city in range(n):

if next\_city not in cur\_path: # Visit unvisited cities

new\_path = cur\_path + [next\_city]

new\_cost = cur\_cost + graph[cur\_city][next\_city]

dq.append((new\_path, new\_cost))

return min\_cost, opt\_path

# Example graph as a 2D adjacency matrix

graph = [

[0, 10, 15, 20],

[10, 0, 35, 25],

[15, 35, 0, 30],

[20, 25, 30, 0]

]

# Solve TSP using BFS

min\_cost, opt\_path = tsp\_bfs(graph)

print("\nOptimal Solution:")

print(f"Minimum cost: {min\_cost}")

print(f"Optimal path: {opt\_path}")

**OUTPUT :**

Path Traversal:

Current Path: [0], Current Cost: 0

Current Path: [0, 1], Current Cost: 10

Current Path: [0, 2], Current Cost: 15

Current Path: [0, 3], Current Cost: 20

Current Path: [0, 1, 2], Current Cost: 45

Current Path: [0, 1, 3], Current Cost: 35

Current Path: [0, 2, 1], Current Cost: 50

Current Path: [0, 2, 3], Current Cost: 45

Current Path: [0, 3, 1], Current Cost: 45

Current Path: [0, 3, 2], Current Cost: 50

Current Path: [0, 1, 2, 3], Current Cost: 75

Current Path: [0, 1, 3, 2], Current Cost: 65

Current Path: [0, 2, 1, 3], Current Cost: 75

Current Path: [0, 2, 3, 1], Current Cost: 70

Current Path: [0, 3, 1, 2], Current Cost: 80

Current Path: [0, 3, 2, 1], Current Cost: 85

Optimal Solution:

Minimum cost: 80

Optimal path: [0, 1, 3, 2, 0]