**Name: Atharva Salitri**

**Roll No.: 029**

**Branch: TY CSAI-B**

**Batch: B2**

**PRN: 12310120**

# Experiment Number: 06

**Title:** Assignment Based on Dynamic programming strategy to implement traveling salesman problem

|  |  |  |  |
| --- | --- | --- | --- |
| **Title of Experimentation** | **CO**  **Mapping** | **CO-Statements** | **PO**  **Mapping** |
| Assignment Based on Dynamic programming strategy to implement traveling salesman problem | CO2, CO3, CO6 | CO2: To design efficient algorithms for computational problems using appropriate algorithmic paradigm  CO3: To analyze asymptotic complexity of the algorithm for a complex computational problem using suitable mathematical techniques.  CO3: To analyze asymptotic complexity of the algorithm for a complex computational problem using suitable mathematical techniques | PO2, PO3, PO4, PO5 |

**Theory:**

* Dynamic Programming (DP): A problem-solving technique where a problem is divided into overlapping subproblems, and results of subproblems are reused (memorization).
* Traveling Salesman Problem (TSP):
* Problem: A salesman must visit every city exactly once and return to the starting city with minimum travel cost.
* Brute force approach → O(n!) complexity (checking all permutations).
* DP (Held-Karp Algorithm): Solves TSP in O(n² · 2ⁿ) time by storing results of subproblems using bit masking.
* Applications: Vehicle routing, logistics, circuit design, route optimization, DNA sequencing

**Input:**

Number of cities: 4

Cost Matrix:

0 10 15 20

10 0 35 25

15 35 0 30

20 25 30 0

**Output:**

Minimum travel cost: 80

Path: 0 → 1 → 3 → 2 → 0

**Objective of Experiment:**

* To apply Dynamic Programming strategy to solve the Traveling Salesman Problem.
* To compare brute force vs. DP in terms of computational complexity.
* To understand the importance of overlapping subproblems and optimal substructure in TSP.

**Flow Chart/Pseudo Code/Algorithm:**

Algorithm (Held-Karp DP Algorithm for TSP)

1. Let dp[mask][i] = minimum cost to visit the set of cities represented by mask ending at city i.
2. Initialize dp[1<<i][i] = cost[start][i].
3. For each subset of cities (represented as bitmask):
   * For each city i in subset:
     + For each city j in subset, j != i:
     + dp[mask][i] = min(dp[mask][i], dp[mask ^ (1<<i)][j] + cost[j][i])
4. Answer = min(dp[(1<<n)-1][i] + cost[i][start]) for all i.

**Flowchart:**

(You can insert a flowchart here showing recursive splitting and combining steps)

**Source Code, with description and with Output Need to be Uploaded to the VOLP**

**Code:**

import java.util.Scanner;

public class TravellingSalesman {

    static int tsp(int[][] dist, int mask, int pos, int n, int[][] memo) {

        if (mask == (1 << n) - 1) {

            return dist[pos][0];

        }

        if (memo[pos][mask] != -1) {

            return memo[pos][mask];

        }

        int ans = Integer.MAX\_VALUE;

        for (int city = 0; city < n; city++) {

            if ((mask & (1 << city)) == 0) {

                int newAns = dist[pos][city] + tsp(dist, mask | (1 << city), city, n, memo);

                ans = Math.min(ans, newAns);

            }

        }

        memo[pos][mask] = ans;

        return ans;

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.println("Enter number of cities:");

        int n = sc.nextInt();

        int[][] dist = new int[n][n];

        System.out.println("Enter distance matrix:");

        for (int i = 0; i < n; i++) {

            for (int j = 0; j < n; j++) {

                dist[i][j] = sc.nextInt();

            }

        }

        int[][] memo = new int[n][1 << n];

        for (int i = 0; i < n; i++) {

            for (int j = 0; j < (1 << n); j++) {

                memo[i][j] = -1;

            }

        }

        int result = tsp(dist, 1, 0, n, memo);

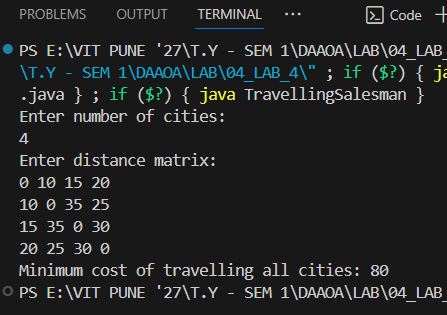
        System.out.println("Minimum cost of travelling all cities: " + result);

        sc.close();

    }

}

**OUTPUT:**

****

**Time and Space Complexity Analysis:**

## Time Complexity

* Number of subproblems: n×2^n (position × visited subset)
* For each subproblem, we check up to n cities.
* Overall time complexity: **O(n^2×2^n)**

## Space Complexity

* Memoization table size: n×2^n
* Extra space for recursion stack: O(n)
* Overall space complexity: **O(n×2^n)**

## Pseudocode with Complexity Comments

text

FUNCTION tsp(dist, mask, pos, n, memo)

IF mask == (1 << n) - 1 // Check if all cities visited; Time: +1

RETURN dist[pos][0] // Return distance back to start; Time: +1

IF memo[pos][mask] != -1 // Memoization check; Time: +1

RETURN memo[pos][mask]

ans ← INFINITY // Time: +1

FOR city FROM 0 TO n-1 // Time: +n (check all possible next cities)

IF (mask & (1 << city)) == 0 // If city not visited; Time: +1 per city

newAns ← dist[pos][city] + tsp(dist, mask | (1 << city), city, n, memo) // Recursion; Time: exponential by subproblem count

ans ← MIN(ans, newAns) // Time: +1 (compare and assign)

memo[pos][mask] ← ans // Store computed answer; Time: +1

RETURN ans // Return minimum cost found; Time: +1

ENDFUNCTION

FUNCTION main

DECLARE scanner // Space: +1

PRINT "Enter number of cities:" // Time: +1

INPUT n // Time: +1

DECLARE 2D array dist[n][n] // Space: +n^2

PRINT "Enter distance matrix:" // Time: +1

FOR i FROM 0 TO n-1 // Time: +n

FOR j FROM 0 TO n-1 // Time: +n per i; Total: \*n^2

INPUT dist[i][j] // Time: +1 per input

DECLARE 2D array memo[n][1 << n] // Space: +n \* 2^n

FOR i FROM 0 TO n-1 // Time: +n

FOR j FROM 0 TO (1 << n) - 1 // Time: +2^n per i; Total: \*n\*2^n

memo[i][j] ← -1 // Time: +1 per initialization

result ← tsp(dist, 1, 0, n, memo) // Recursive call; Time: O(n^2 \* 2^n)

PRINT "Minimum cost of travelling all cities: " + result // Time: +1

CLOSE scanner // Time: +1

ENDFUNCTION

## Complexity Explanation

* **Time Complexity:** There are n×2n possible state combinations of (current city, visited subset). Each state recursively calls up to n cities, so total time complexity is O(n^2 2^n)
* **Space Complexity:** The memoization table stores results for each (city, mask) state: O(n 2^n)space.
* Input distance matrix uses O(n^2) space, but this is dominated by memo space.
* The recursive recursion stack depth at most O(n), which is less significant compared to memo.