**Experiment No. : 5**

**Title: Implement Travelling Salesman Problem using Dynamic approach**

**Batch: B2 Roll No.: 16010421119 Experiment No.: 5**

**Aim:** To Implement Travelling Salesman Problem for minimum 6 vertices using Dynamic approach and analyse its time Complexity.

**Algorithm of Travelling Salesman Problem:**

**Algorithm: Traveling-Salesman-Problem**

C ({1}, 1) = 0

for s = 2 to n do

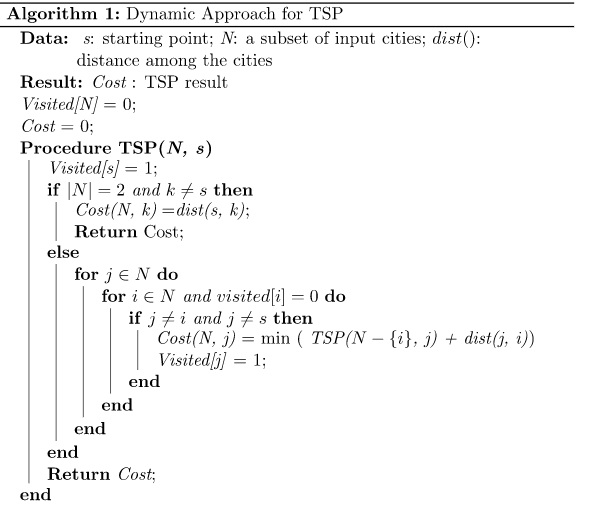
for all subsets S Є {1, 2, 3, … , n} of size s and containing 1

C (S, 1) = ∞

for all j Є S and j ≠ 1

C (S, j) = min {C (S – {j}, i) + d(i, j) for i Є S and i ≠ j}

Return minj C ({1, 2, 3, …, n}, j) + d(j, i)



**Explanation and Working of Travelling Salesman Problem:**

**Problem Statement**

A traveller needs to visit all the cities from a list, where distances between all the cities are known and each city should be visited just once. What is the shortest possible route that he visits each city exactly once and returns to the origin city?

**Solution**

Travelling salesman problem is the most notorious computational problem. We can use brute-force approach to evaluate every possible tour and select the best one. For n number of vertices in a graph, there are (n - 1)! number of possibilities.

Instead of brute-force using dynamic programming approach, the solution can be obtained in lesser time, though there is no polynomial time algorithm.

Let us consider a graph G = (V, E), where V is a set of cities and E is a set of weighted edges. An edge e(u, v) represents that vertices u and v are connected. Distance between vertex u and v is d(u, v), which should be non-negative.

Suppose we have started at city 1 and after visiting some cities now we are in city j. Hence, this is a partial tour. We certainly need to know j, since this will determine which cities are most convenient to visit next. We also need to know all the cities visited so far, so that we don't repeat any of them. Hence, this is an appropriate sub-problem.

For a subset of cities S Є {1, 2, 3, ... , n} that includes 1, and j Є S, let C(S, j) be the length of the shortest path visiting each node in S exactly once, starting at 1 and ending at j.

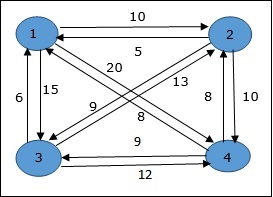
When |S| > 1, we define C(S, 1) = ∝ since the path cannot start and end at 1.

Now, let express C(S, j) in terms of smaller sub-problems. We need to start at 1 and end at j. We should select the next city in such a way that

C(S,j)=minC(S−{j},i)+d(i,j)wherei∈Sandi≠jc(S,j)=minC(s−{j},i)+d(i,j)wherei∈Sandi≠j

## Example

In the following example, we will illustrate the steps to solve the travelling salesman problem.



From the above graph, the following table is prepared.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 |
| 1 | 0 | 10 | 15 | 20 |
| 2 | 5 | 0 | 9 | 10 |
| 3 | 6 | 13 | 0 | 12 |
| 4 | 8 | 8 | 9 | 0 |

The minimum cost path is 35.

Start from cost **{1, {2, 3, 4}, 1}**, we get the minimum value for **d [1, 2]**. When **s = 3**, select the path from 1 to 2 (cost is 10) then go backwards. When **s = 2**, we get the minimum value for **d [4, 2]**. Select the path from 2 to 4 (cost is 10) then go backwards.

When **s = 1**, we get the minimum value for **d [4, 3]**. Selecting path 4 to 3 (cost is 9), then we shall go to then go to **s = Φ** step. We get the minimum value for **d [3, 1]** (cost is 6).

Values

**Derivation of Travelling Salesman Problem:**

Time complexity Analysis

The time complexity of this code is O(n!) where n is the number of cities. This is because we are generating all possible permutations of the cities and calculating the distance for each permutation. In the worst-case scenario, we will have to generate n! permutations.

The space complexity of this code is O(n) because we are using an array of size n to store the current path, the shortest path, and the bitmask. We are also using a 2D array of size n x n to store the distances between the cities. The recursion stack will also consume O(n) space. Therefore, the total space complexity of the code is O(n + n^2) which simplifies to O(n^2).

**Program(s) of Travelling Salesman Problem:**

#include<bits/stdc++.h>

using namespace std;

const int N = 10;

int dist[N][N];

int n;

int ans = INT\_MAX;

vector<int> path, shortest\_path;

void tsp(int mask, int curr, int cost) {

if (mask == (1 << n) - 1) {

if (dist[curr][0] != -1) {

cost += dist[curr][0];

if (cost < ans) {

ans = cost;

shortest\_path = path;

shortest\_path.push\_back(0);

}

}

return;

}

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

if ((mask & (1 << i)) == 0 && dist[curr][i] != -1) {

path.push\_back(i);

tsp(mask | (1 << i), i, cost + dist[curr][i]);

path.pop\_back();

}

}

}

int main() {

cin >> n;

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

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

cin >> dist[i][j];

}

}

tsp(1, 0, 0);

cout << ans << endl;

for (int i : shortest\_path) {

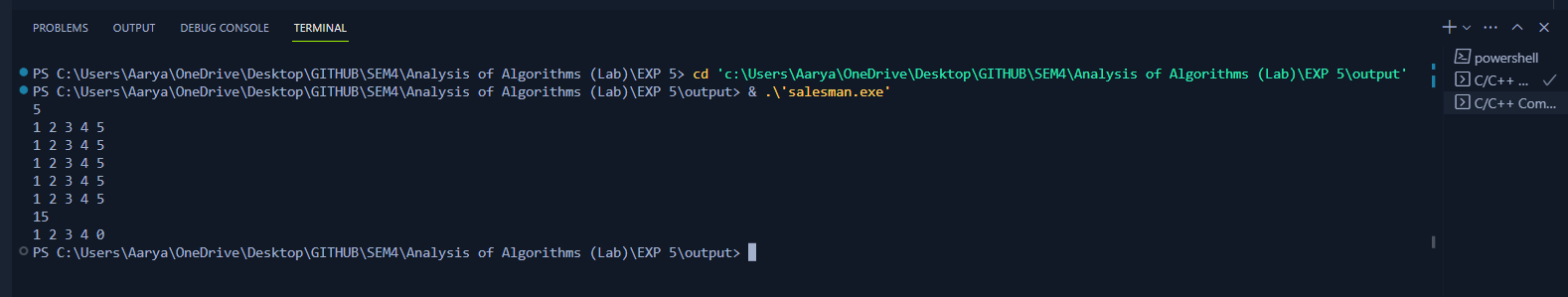
cout << i << " ";

}

return 0;

}

**Output(o) of Travelling Salesman Problem:**



**Post Lab Questions:-**

Explain how Travelling Salesman Problem using greedy method is different from Dynamic Method in detail.

Ans **:**

**Greedy Method**

The Greedy Method is an approach that makes the locally optimal choice at each step of the algorithm, hoping that it will lead to a globally optimal solution. In the context of TSP, the Greedy Method starts from an arbitrary city and selects the closest unvisited city as the next destination. The algorithm continues this process, selecting the closest unvisited city until all cities have been visited. Once all the cities are visited, the algorithm returns to the starting city.

The Greedy Method is easy to implement and computationally efficient. However, it does not always produce the optimal solution. In fact, it often produces suboptimal solutions that can be far from the optimal solution. This is because the Greedy Method is myopic and makes decisions based only on the current step without considering the future steps. This leads to the algorithm getting stuck in a local optimum that is far from the global optimum.

**Dynamic Programming Method**

The Dynamic Programming Method is a more sophisticated approach that solves the problem by breaking it down into smaller subproblems and solving them recursively. In the context of TSP, the Dynamic Programming Method creates a table where each entry represents the shortest possible path to visit a subset of the cities and return to the starting city. The table is filled using a bottom-up approach, starting with the smallest possible subset of cities and gradually building up to the full set of cities.

The Dynamic Programming Method guarantees the optimal solution and is often used to solve TSP for small to medium-sized instances. However, it has high computational complexity, with a time complexity of O(n^2 \* 2^n) where n is the number of cities. This makes it impractical for large instances of TSP.

**Conclusion: (Based on the observations):**

**We can conclude that we have learnt about the Travelling Salesman Problem.**

**Outcome:**

**CO 2. Implement Greedy and Dynamic Programming algorithms.**

**References:**

1. Richard E. Neapolitan, " Foundation of Algorithms ", 5th Edition 2016, Jones & Bartlett Students Edition
2. Harsh Bhasin , " Algorithms : Design & Analysis", 1st Edition 2013, Oxford Higher education, India
3. T.H. Coreman ,C.E. Leiserson,R.L. Rivest, and C. Stein, " Introduction to algorithms", 3rd Edition 2009, Prentice Hall India Publication
4. Jon Kleinberg, Eva Tardos, " Algorithm Design", 10th Edition 2013, Pearson India Education Services Pvt. Ltd.