EAST WEST UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Project Report

Semester: Fall-2024

Course Title: Algorithms

Course Code: CSE246

Sec: 11

Project Name: Traveling Salesman Problem

Group No: 4

Submitted by-

Group Members:

Student Name	Student ID
Nusrat Jahan Chaiti	2023-1-60-255
Nadira Akter	2022-3-60-220
Farhana Akter Tamanna	2022-3-60-133

Submitted to-

Dr. Tania Sultana

Assistant Professor

Department of Computer Science & Engineering

East West University

1. Introduction

The Traveling Salesman Problem (TSP) is a classic optimization problem that requires finding the shortest possible route for a salesman who needs to visit a given set of cities and return to the starting city while minimizing the total distance.

This project aims to implement a solution to the TSP using Dynamic Programming (DP) combined with Bitmasking. By employing this technique, we can solve the TSP efficiently for a moderate number of cities, providing an optimal solution in terms of total travel distance.

2. Objective

The objective of this project is to:

- Solve the Traveling Salesman Problem (TSP) using Dynamic Programming and Bitmasking.
- Minimize the total travel distance for a given set of cities and their pairwise travel distances.
- Provide an easy-to-use solution to compute the optimal route for small-scale datasets (up to 20 cities).

3. Technology Specification

Hardware Requirements:

- **CPU**: Any modern processor.
- **RAM**: 4 GB minimum (8 GB or more recommended).
- **Storage**: A few megabytes of disk space to store the source code and data.

Software Requirements:

- **Programming Language**: C++
- **Compiler**: Any C++ compiler
- **Libraries**: Standard C++ libraries (no external libraries are used).

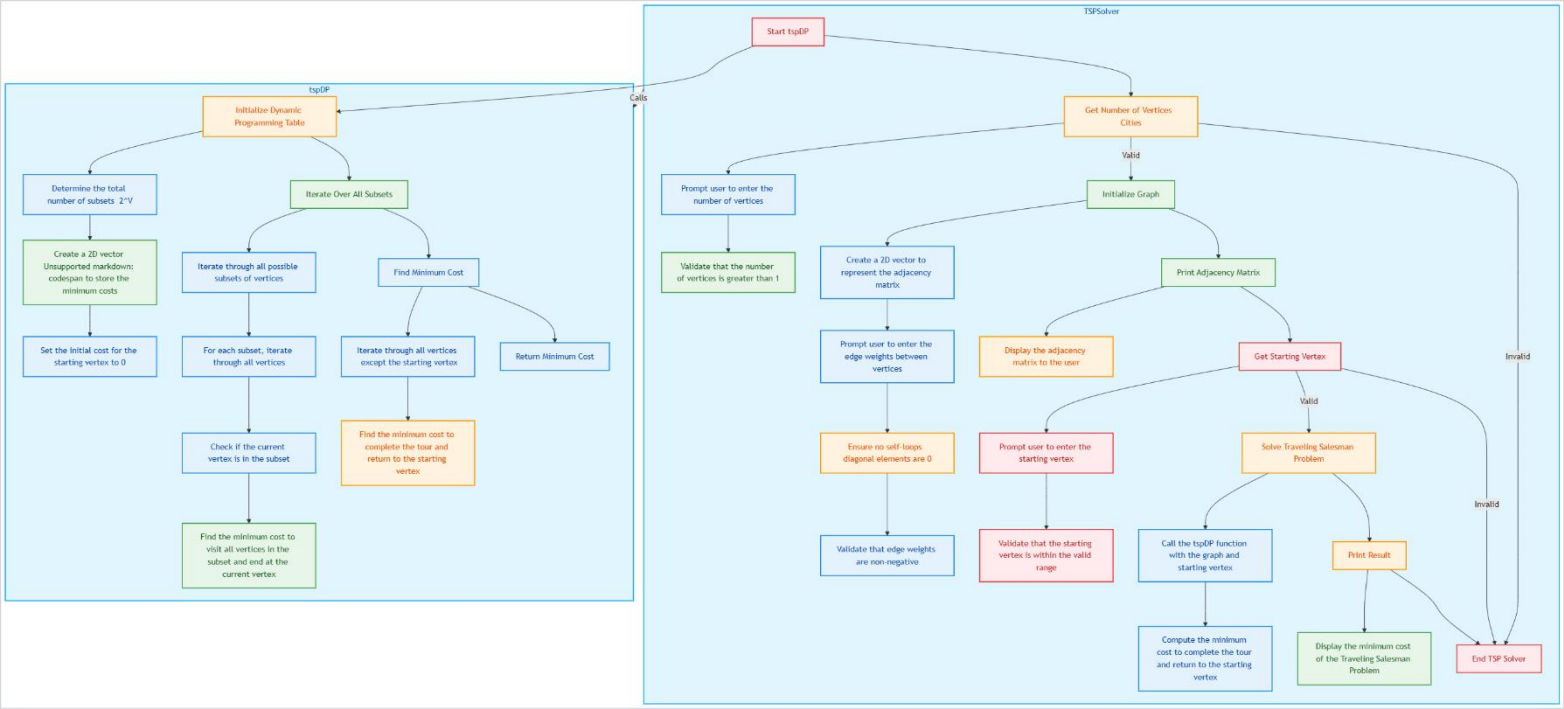
Development Environment:

- **IDE**: Visual Studio Code or any C++ development environment.
- **Operating System**: The project is cross-platform and works on Linux, macOS, and Windows.

4. Data Flow Diagram

The Data Flow Diagram (DFD) for this project illustrates how the input data (cities and distances) is processed to produce the output (optimal travel cost and route).

1. **User Inputs Data**: The user provides the number of cities and the adjacency matrix of distances between the cities.
2. **Input Validation**: The input is validated to ensure there are no negative weights and that the number of cities is greater than one.
3. **Graph Construction**: A graph (adjacency matrix) is constructed, representing the distances between each pair of cities.
4. **Dynamic Programming with Bitmasking**: The TSP is solved using a DP approach combined with bitmasking. The state is updated for every possible subset of cities and the optimal route is calculated.
5. **Display Results**: The program displays the optimal route and the corresponding minimum cost.



5. Features List

1. **Input Validation**: Ensures that the number of cities is greater than 1 and that the edge weights are non-negative.
2. **Graph Construction**: Builds an adjacency matrix to represent the travel distances between cities.
3. **Dynamic Programming with Bitmasking**: Implements the main algorithm to solve the TSP optimally.
4. **Output of Optimal Route and Cost**: Displays the minimum travel cost and the optimal route for visiting all cities.

6. Implementation

1. Input Validation

The program begins by reading the number of vertices (cities) from the user, and then it validates that the number is greater than 1, as TSP requires at least two vertices.

```
int V;
cout << "Enter the number of vertices (cities): ";
cin >> V;
if (V <= 1) {
    cout << "Number of vertices must be greater than 1." << endl;
    return 1;
}
```

2. Adjacency matrix Initialization

An adjacency matrix is initialized, and the program reads edge weights between cities. If the edge weight is negative, the program prompts the user to enter the weight again.

```
vector<vector<int>> graph(V, vector<int>(V));

cout << "\nEnter the adjacency matrix (enter 0 for no self-loops):\n";
for (int i = 0; i < V; i++) {
    for (int j = 0; j < V; j++) {
        if (i == j) {
            graph[i][j] = 0;
            continue;
        }
        cout << "Edge weight from vertex " << i << " to vertex " << j << ": ";
        cin >> graph[i][j];
        if (graph[i][j] < 0) {
            cout << "Edge weight cannot be negative. Please re-enter." << endl;
            j--;
        }
    }
}
```

3. Dynamic Programming Table (DP Table) Initialization

The main part of the solution lies in the dynamic programming approach, where we use a DP table to store the minimum costs.

```
int tspDP(vector<vector<int>>& graph, int s)
{
    int V = graph.size();
    int n = (1 << V);
    vector<vector<int>> dp(n, vector<int>(V, INF));
    dp[1 << s][s] = 0;
```

4. DP Table Filling

The dynamic programming solution fills in the DP table by iterating through all subsets of vertices (mask) and for each vertex u in the subset, attempts to add each other vertex v not yet in the subset. If adding v results in a smaller cost, we update the DP table.

```
for (int mask = 0; mask < n; mask++) {
    for (int u = 0; u < V; u++) {
        if (mask & (1 << u)) {
            for (int v = 0; v < V; v++) {
                if (!(mask & (1 << v))) {
                    int nextMask = mask | (1 << v);
                    dp[nextMask][v] = min(dp[nextMask][v], dp[mask][u] + graph[u][v]);
                }
            }
        }
    }
}
```

5. Finding the Minimum Cost to Complete the Tour

Once all subsets are processed, the next step is to find the minimum cost to complete the tour by visiting all cities and returning to the starting city. We check the last entry in the DP table for each possible endpoint u and return the minimum cost plus the cost of returning to the starting city s.

```
    int min_cost = INF;
    for (int u = 0; u < V; u++) {
        if (u != s) {
            min_cost = min(min_cost, dp[n - 1][u] + graph[u][s]);
        }
    }

    return min_cost;
}
```

5. Output

Finally, the program prints the minimum cost of the traveling salesman tour.

```
int result = tspDP(graph, s);
cout << "\nThe minimum cost of the Traveling Salesman Problem is: " << result << endl;
```

7. Total Complexity Analysis

The overall time complexity of the project is dominated by the dynamic programming with bitmasking algorithm, which is O(V^2 * 2^V)**,** where V is the number of cities. This makes the approach feasible for small instances (up to 20-25 cities).

The space complexity is also O(V * 2^V) due to the DP table storing the state for all subsets of cities.

8. Results

The program successfully computes the minimum cost of the TSP for any given set of cities. For example, with 3 cities and specific distances between them, the program calculates and displays the optimal travel cost and the route taken.

```
Enter the number of vertices (cities): 3

Enter the adjacency matrix (enter 0 for no self-loops):
Edge weight from vertex 0 to vertex 1: 10
Edge weight from vertex 0 to vertex 2: 15
Edge weight from vertex 1 to vertex 0: 35
Edge weight from vertex 1 to vertex 2: 24
Edge weight from vertex 2 to vertex 0: 26
Edge weight from vertex 2 to vertex 1: 30

The adjacency matrix is:
   0  10  15
  35   0  24
  26  30   0

Enter the starting vertex: 0

The minimum cost of the Traveling Salesman Problem is: 60
```

9. Limitations

- **Scalability**: The solution is not scalable to large numbers of cities due to its exponential time complexity. For more than 20 cities, the solution becomes impractical.
- **Memory Usage**: The space complexity of **O(V * 2^V)** can lead to memory limitations for larger instances.

10. Conclusion

This project successfully solves the Traveling Salesman Problem using Dynamic Programming with Bitmasking. It provides an optimal solution for small instances and demonstrates an efficient approach to solving a classic NP-hard problem. However, the solution is limited by its time and space complexity, which makes it impractical for larger datasets.