Project #1 Report

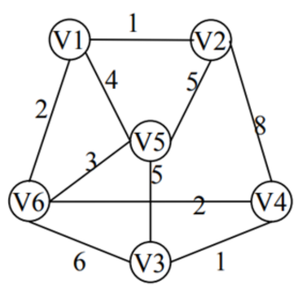
Table of contents

Introduction

This report analyzes the C++ implementation of the **Traveling Salesman Problem (TSP)**. The program reads an input file containing weighted edges between cities, constructs a graph representation, and uses **dynamic programming** to find the shortest possible route that visits all cities exactly once before returning to the starting city.

💡 **Case 1**

e1 1 v1 v2
e2 4 v1 v5
e3 5 v2 v5
e4 2 v1 v6
e5 3 v6 v5
e6 2 v6 v4
e7 6 v6 v3
e8 1 v3 v4
e9 5 v5 v3
e10 8 v4 v2



The answer for this case is:
Optimal Path:
v1 → v6 → v4 → v3 → v5 → v2 → v1
Total Distance: 16

Data Structures

1. `map<int, vector<pair<int, int>>> Nodes`

- A mapping of each node to its list of adjacent nodes along with edge weights.
- e.g. The vertices in case 1 will be stored as

```
for (const auto &kv : Nodes) {
    cout << "Node " << kv.first << " has neighbors: ";
    for (const auto &neighbor : kv.second) {
        cout << "(" << neighbor.first << ", weigth = " << neighbor.second << ") ";
    }
    cout << endl;
}
```

```
Node 0 has neighbors: (1, weigth = 1) (4, weigth = 4) (5, weigth = 2)
Node 1 has neighbors: (0, weigth = 1) (4, weigth = 5) (3, weigth = 8)
Node 2 has neighbors: (5, weigth = 6) (3, weigth = 1) (4, weigth = 5)
Node 3 has neighbors: (5, weigth = 2) (2, weigth = 1) (1, weigth = 8)
Node 4 has neighbors: (0, weigth = 4) (1, weigth = 5) (5, weigth = 3) (2, weigth = 5)
Node 5 has neighbors: (0, weigth = 2) (4, weigth = 3) (3, weigth = 2) (2, weigth = 6)
```

※ Node 0 is v1, Node 1 is v2, ... , and so on.

2. `vector<vector> cities`
    - A matrix representation of the graph where `cities[i][j]` holds the weight of the edge between node `i` and node `j`. If no edge exists, the value is set to a large constant (INF).
    - e.g. The adjacency matrix of case 1.
      Node 1 → Node 3 cost 8

	0	1	2
0	INF	1	INF
1	1	INF	INF
2	INF	INF	INF
3	INF	8	1
4	4	5	5
5	2	INF	6

Algorithm Implementation

Bit masking

Introduction

`mask` is an integer, but it is essentially a **binary number (bitmask)** where each bit represents whether a city has been visited or not.

- For example, if there are 4 cities (labeled as 0, 1, 2, 3), the `mask` will be a 4-bit number, with each bit corresponding to a city:
    - Bit 0 (least significant bit): City 0、Bit 1: City 1、Bit 2: City 2、Bit 3: City 3
- If a bit is `1`, it means the city has been visited; if it is `0`, the city has not been visited.

Examples

Binary Representation	Decimal Value	Meaning
`0001`	1	Only City 0 has been visited
`0011`	3	Cities 0 and 1 have been visited
`0101`	5	Cities 0 and 2 have been visited
`1111`	15	All cities (0, 1, 2, 3) have been visited

Operations

Operation	Description	Method
Check if city `i` is visited	Check if the `i`-th bit is 0 (not visited)	Use the bitwise `&` operator
Mark city `i` as visited	Set the `i`-th bit to 1 (mark as visited)	Use the bitwise `|` operator
Check if all cities are visited	Check if all `n` bits are `1` (all cities visited)	Compare (`==`) with `n` bits of `1`

Coding examples

```
// Check if city i is visited
if (!(mask & (1 << i))) {
      // City i has not been visited
}

// Mark city i as visited
int newMask = mask | (1 << i);

// Check if all cities have been visited
if (mask == (1 << n) - 1) {
   // All cities have been visited
}
```

- `1 << i`: Shift `1` to the left by `i` positions, resulting in a number where only the `i` th bit is `1`.
- `mask & (1 << i)`: If the result is `0`, it means city `i` has not been visited.
- `mask | (1 << i)`: Set the `i` th bit of `mask` to `1`.
- `(1 << n) - 1`: Shift `1` to the left by `n` positions, resulting in a `1` followed by `n` zeros, then subtract `1` resulting in a number with `n` bits all set to `1`.

Examples

Initial State

- `mask = 0001` (decimal value `1`), meaning only City 0 has been visited.

Visit City 2

- Check if City 2 has been visited:
  - `1 << 2` results in `0100`.
  - `mask & 0100` results in `0000`, meaning City 2 has not been visited.
- Mark City 2 as visited:
  - `mask | 0100` results in `0101` (decimal value `5`).

Check if all cities have been visited

- Check if `mask` equals `(1 << n) - 1`:
  - `1 << 4` results in `10000`.
  - `(1 << 4) - 1` results in `1111` (decimal value `15`).
  - `mask != 1111`, meaning **not** all cities have been visited.

Top-Down Approach (Recursive + Memoization)

The top-down approach employs **recursive function calls with memoization** to solve TSP efficiently. The recursion explores different paths and stores previously computed results in a memoization table to avoid redundant calculations.

Key components

- `TSP_TopDown::TSP(int mask, int now_city, const vector<vector<int>>& cost)` is the main recursive function.
- `memo[mask][now_city]` : A memorization table used for dynamic programming to store already computed costs. Stores the minimum cost from `now_city` with a given `mask` representing visited cities.
- `parent[mask][now_city]` records the best next city for path reconstruction. Used to store the best preceding node in the optimal path.

Recurrence relation

Define `TSP(mask, now_city)` as: The minimum cost to travel from the current city `now_city`, visit all unvisited cities in `mask`, and finally return to the starting city.

$$TSP(mask, now\_city) = \begin{cases} cost[now\_city][0] & \text{if all cities are visited} \\ \min_{i \notin mask} \left(cost[now\_city][i] + TSP(mask \mid (1 \ll i), i) \right) & \text{otherwise} \end{cases}$$

1. **Base Case**
    - If `mask == (1 << n) - 1`, it means all cities have been visited. In this case, return the cost to travel from `now_city` back to the starting city, which is `cost[now_city][0]`.
2. **Recursive Case**
    - For each unvisited city `i` (i.e., `i` is not in `mask`), calculate the cost to travel from `now_city` to `i` (`cost[now_city][i]`), and recursively compute the minimum cost to visit the remaining cities starting from `i` (`TSP(mask | (1 << i), i)`).
    - Choose the minimum cost among all possible next cities `i`.

Handling Cases Where No Hamiltonian Path Exists

A Hamiltonian path (tour) exists if there is a way to visit all cities and return to the starting city. If no such path is possible, the algorithm must correctly identify and return an indication of failure.

- In `TSP_TopDown::TSP()`, if there is no edge from `now_city` to the next city, the cost remains `INF`.
- If no valid path is found after iterating through all cities, `minCost` remains `INF`, meaning that a Hamiltonian path cannot be formed.
- In `TSP_TopDown::Solve()`, if `MinCost >= INF`, the function outputs `"Unable to form Hamilton path (tour path)."`

This ensures that the algorithm correctly handles cases where the input graph is disconnected or lacks the required edges.

Time Complexity Analysis

- There are **2^n** possible subsets (`mask` values) and `n` possible cities to be at.
- Each state involves iterating over `n` cities.
- **Total complexity: O(n^2 * 2^n)**, which is much more efficient than a brute-force **O(n!)** approach.

Bottom-Up Approach (Iterative Dynamic Programming)

This approach uses a **table-based iterative method** to build solutions for increasing subsets of cities.

- `dp[mask][i]` stores the minimum cost to visit all cities in `mask` and end at city `i`.
- The table is filled iteratively by checking feasible transitions from `i` to `j`.
- Once all subsets are processed, the shortest path is determined by adding the return cost to the starting city.

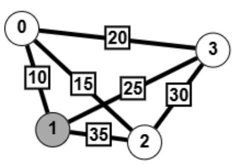
DP Transition

- Initialization: dp[1][0]=0
    - This means starting at city 0, with only city 0 visited, the cost is 0.
- State Transition
    - For each state `dp[mask][i]`, try to extend to an unvisited city `j`, and update `dp[mask | (1 << j)][j]`.

$$dp[mask \mid (1 \ll j)][j] = \min\left(dp[mask \mid (1 \ll j)][j],\, dp[mask][i] + cost[i][j]\right)$$

Example

Suppose there are 4 cities (labeled as 0, 1, 2, 3), and the distance matrix `cost` is as follows:



	0	**1**	**2**
0	0	10	15
1	10	0	35
2	15	35	0
3	20	25	30

Step 1: Initialization

- `dp[0001][0] = 0` (only city 0 is visited, and we are at city 0).

Step 2: State Transitions

• **From** `mask = 0001` : ◦ Visit city 1: ▪ newMask = 0001 | 0010 = 0011. ▪ dp[0011][1] = dp[0001][0]+cost[0][1] = 0+10 = 10. ◦ Visit city 2: ▪ newMask = 0001 | 0100 = 0101. ▪ dp[0101][2] = dp[0001][0]+cost[0][2] = 0+15 = 15. ◦ Visit city 3: ▪ newMask = 0001 | 1000 = 1001. ▪ dp[1001][3] = dp[0001][0]+cost[0][3] = 0+20 = 20.	• **From** `mask = 0101` : ◦ Visit city 1: ▪ newMask = 0101 | 0010 = 0111. ▪ dp[0111][1] = dp[0101][2]+cost[2][1] = 15+35 = 50. ◦ Visit city 3: ▪ newMask = 0101 | 1000 = 1101. ▪ dp[1101][3] = dp[0101][2]+cost[2][3] = 15+30 = 45.
• **From** `mask = 0011` : ◦ Visit city 2: ▪ newMask = 0011 | 0100 = 0111. ▪ dp[0111][2] = dp[0011][1]+cost[1][2] = 10+35 = 45. ◦ Visit city 3: ▪ newMask = 0011 | 1000 = 1011. ▪ dp[1011][3] = dp[0011][1]+cost[1][3] = 10+25 = 35.	• **From** `mask = 1001` : ◦ Visit city 1: ▪ newMask = 1001 | 0010 = 1011. ▪ dp[1011][1] = dp[1001][3]+cost[3][1] = 20+25 = 45. ◦ Visit city 2: ▪ newMask = 1001 | 0100 = 1101. ▪ dp[1101][2] = dp[1001][3]+cost[3][2] = 20+30 = 50.

Continue computing transitions until reaching the final state (mask = 1111)...

Step 3: Compute the Shortest Path

- For the final state `mask = 1111`, compute the cost to return to the starting city from each city:
  - From city 1: `dp[1111][1] + cost[1][0] = ? + 10 = a`.
  - From city 2: `dp[1111][2] + cost[2][0] = ? + 15 = b`.
  - From city 3: `dp[1111][3] + cost[3][0] = ? + 20 = c`.
- Compare a, b, and c, and the minimum cost is the final answer.

Complexity analysis

Time Complexity

- States: **2^n×n** (all subsets of cities × current city).
- Transitions: **n** (try all unvisited cities).
- Total: **O(2^n×n2)**.

Space Complexity

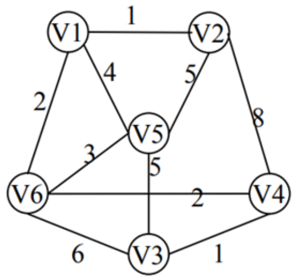
- DP Table: **O(2^n×n)**.

Advantages and Disadvantages

Approach	Pros	Cons
Top-Down (Recursive + Memoization)	Intuitive, easy to implement, only computes necessary states	High recursive overhead, may lead to stack overflow for large `n`
Bottom-Up (DP Table)	Avoids recursion overhead, generally faster due to explicit iteration	Requires a large DP table in memory

Result

💡 **case 1**



Total Distance: 16
Optimal Path: v1 → v2 → v5 → v3 → v4 → v6 → v1

Execution Time (Top-Down): 1.58e-05 seconds
Execution Time (Bottom-Up): 2.52e-05 seconds

💡 **case 2**

e1 9 v1 v2
e2 3 v1 v5
e3 5 v1 v6
e4 5 v2 v3
e5 4 v2 v6
e6 2 v3 v4
e7 8 v3 v6
e8 1 v4 v5
e9 7 v4 v6
e10 5 v5 v6

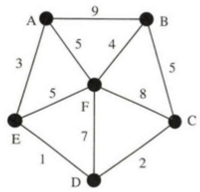


Figure 5.3 An example traveling salesman problem instance.

Total Distance: 20
Optimal Path: v1 → v5 → v4 → v3 → v2 → v6 → v1

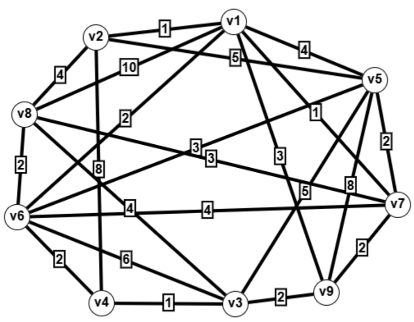
Execution Time (Top-Down): 4.98e-05 seconds
Execution Time (Bottom-Up): 7.99e-05 seconds

💡 **case 3**

e1 1 v1 v2
e2 4 v1 v5
e3 5 v2 v5
e4 2 v1 v6
e5 3 v6 v5
e6 2 v6 v4
e7 6 v6 v3
e8 1 v3 v4
e9 5 v5 v3
e10 8 v4 v2
e11 3 v7 v8

e12 4 v6 v7
e13 2 v7 v9
e14 1 v7 v1
e15 2 v7 v5
e16 10 v8 v1
e17 4 v8 v3
e18 2 v8 v6
e20 3 v9 v1
e21 4 v8 v2
e22 2 v9 v3
e23 8 v9 v5

Number of cities: 9



Total Distance: 20
Optimal Path: v1 → v2 → v8 → v6 → v4 → v3 → v9 → v7 → v5 → v1

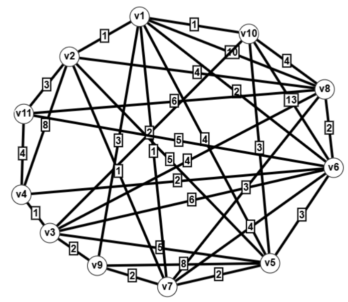
Execution Time (Top-Down): 7.61e-05 seconds
Execution Time (Bottom-Up): 0.0001033 seconds

case 4

e1 1 v1 v2
e2 4 v1 v5
e3 5 v2 v5
e4 2 v1 v6
e5 3 v6 v5
e6 2 v6 v4
e7 6 v6 v3
e8 1 v3 v4
e9 5 v5 v3
e10 8 v4 v2
e11 3 v7 v8
e12 4 v6 v7
e13 2 v7 v9
e14 1 v7 v1
e15 2 v7 v5
e16 10 v8 v1

e17 4 v8 v3
e18 2 v8 v6
e20 3 v9 v1
e21 4 v8 v2
e22 2 v9 v3
e23 8 v9 v5
e24 1 v10 v1
e25 2 v10 v3
e26 13 v10 v6
e27 3 v10 v5
e28 4 v10 v8
e29 3 v11 v2
e30 4 v11 v4
e31 5 v11 v6
e32 6 v11 v8
e33 1 v7 v2

Number of cities: 11



Total Distance: 25
Optimal Path: v1 → v2 → v11 → v4 → v3 → v9 → v7 → v5 → v6 → v8 → v10 → v1

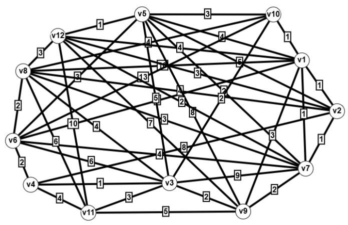
Execution Time (Top-Down): 0.0003232 seconds
Execution Time (Bottom-Up): 0.0004446 seconds

case 5

e1 1 v1 v2
e2 4 v1 v5
e3 5 v2 v5
e4 2 v1 v6
e5 3 v6 v5
e6 2 v6 v4
e7 6 v6 v3
e8 1 v3 v4
e9 5 v5 v3
e10 8 v4 v2
e11 3 v7 v8
e12 4 v6 v7
e13 2 v7 v9
e14 1 v7 v1
e15 2 v7 v5
e16 10 v8 v1
e17 4 v8 v3
e18 2 v8 v6
e20 3 v9 v1
e21 4 v8 v2

e22 2 v9 v3
e23 8 v9 v5
e24 1 v10 v1
e25 2 v10 v3
e26 13 v10 v6
e27 3 v10 v5
e28 4 v10 v8
e29 3 v11 v3
e30 4 v11 v4
e31 5 v11 v9
e32 6 v11 v8
e33 1 v7 v2
e34 4 v12 v1
e35 3 v12 v2
e36 10 v12 v11
e37 7 v12 v9
e38 3 v12 v8
e39 1 v12 v5
e40 2 v12 v7
e41 9 v7 v3

Number of cities: 12

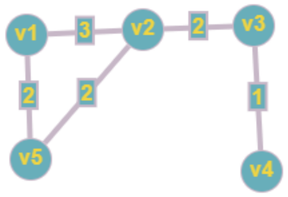


Total Distance: 25
Optimal Path: v1 → v2 → v7 → v9 → v3 → v11 → v4 → v6 → v8 → v12 → v5 → v10 → v1

Execution Time (Top-Down): 0.0006999 seconds
Execution Time (Bottom-Up): 0.0009176 seconds

> The following is the case we set, mainly used to test the situation where **Hamilton cycle** cannot be generated.

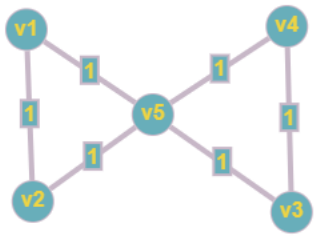
> 💡 **case 6**
>
> e1 3 v1 v2
> e2 2 v2 v3
> e3 1 v3 v4
> e4 2 v5 v1
> e5 2 v5 v2



Unable to form Hamilton cycle (tour path).

Execution Time (Top-Down): 1.16e-05 seconds
Execution Time (Bottom-Up): 1.2e-05 seconds

> 💡 **case 7**
>
> e1 1 v1 v2
> e2 1 v1 v5
> e3 1 v2 v5
> e4 1 v3 v4
> e5 1 v3 v5
> e6 1 v4 v5



Unable to form Hamilton cycle (tour path).

Execution Time (Top-Down): 1.39e-05 seconds
Execution Time (Bottom-Up): 1.61e-05 seconds

Conclusion

This implementation efficiently solves the TSP using **dynamic programming** and **bit masking**, ensuring an optimal route is found. The use of memoization and backtracking allows for significant performance improvements over brute force methods. The program is well-structured and correctly handles input parsing, graph representation, and TSP computation.