Smart Route Planner

Greedy vs Dynamic Programming — TSP Solver

Project Report

C++ Console Application

Submitted By

Mohith-AP24110010745

Bhaskar-AP24110010598

Sudhir-AP24110010716

Leena-AP24110010584

Sahithi-AP24110010570

1. Project Overview

The Smart Route Planner is a C++ console application that solves the Travelling Salesman Problem (TSP) — finding the shortest route that visits every city exactly once and returns to the starting point. The application implements two distinct algorithmic approaches and compares their results side by side.

The core objective is to demonstrate the classic trade-off in algorithm design between speed and optimality. The two approaches implemented are:

- **Greedy Algorithm:** Greedy (Nearest Neighbor) — a fast heuristic that does not guarantee the globally optimal route.
- **Dynamic Programming:** Dynamic Programming with Bitmask — an exact algorithm that always finds the globally optimal solution.

2. Algorithms

2.1 Greedy Algorithm — Nearest Neighbor

Starting from City A (index 0), the algorithm greedily moves to the closest unvisited city at every step. Once all cities are visited, it returns to the start to complete the cycle.

- Simple and extremely fast
- Produces a good but generally sub-optimal route
- Time Complexity: $O(N^2)$ — at each of the N cities, searches up to N cities for the nearest unvisited city

2.2 Dynamic Programming — Bitmask TSP

Uses a bitmask to represent the set of visited cities and a 2D DP table dp[mask][pos] to cache subproblem results. A parent table records the best decision at each state, enabling full path reconstruction.

- mask — a binary integer where bit i is 1 if city i has been visited
- pos — the current city we are at
- dp[mask][pos] — minimum cost to complete the tour from this state
- Guarantees the globally optimal route every time
- Time Complexity: $O(N^2 \cdot 2^N)$ — feasible for N ≤ 20; this application constrains N ≤ 10

3. Algorithm Comparison

Feature	Greedy	Dynamic Programming
Strategy	Nearest-neighbor heuristic	Bitmask DP with memoization
Time Complexity	$O(N^2)$	$O(N^2 \cdot 2^N)$
Optimality	Not guaranteed	Guaranteed optimal
Speed	Very fast	Exponential (slower)
Best for	Large N, fast approximation	Small N, exact solution
Max cities (app)	10	10

State Growth by City Count

N (Cities)	Greedy States	DP States (2^N)
4	16	64
6	36	384
8	64	2,048
10	100	10,240

4. Sample Run

4.1 Input — 4-City Distance Matrix

```
     A    B    C    D
A    0   10   15   20
B   10    0   35   25
```

```
C   15   35    0   30
D   20   25   30    0
```

4.2 Greedy Output

```
[Greedy Execution Steps]
  Step 1: From A -> B (cost = 10), Running Total = 10
  Step 2: From B -> D (cost = 25), Running Total = 35
  Step 3: From D -> C (cost = 30), Running Total = 65
  Step 4: Return C -> A (cost = 15), Running Total = 80

Greedy Route: A -> B -> D -> C -> A
Cost: 80
```

4.3 DP Optimal Output

```
OPTIMAL ROUTE SUMMARY
Route: A -> B -> D -> C -> A
Cost Breakdown: A->B(10) + B->D(25) + D->C(30) + C->A(15) = 80
Total Cost: 80

Improvement: 0.0% (Greedy found the optimal route).
```

In this particular dataset, the greedy algorithm happened to find the optimal solution. This is not always the case — on different inputs, the DP algorithm will outperform greedy by finding a strictly shorter route.

5. Project Structure

```
smart_route_planner/
  |
  +-- smart_route_planner.cpp
  |     |-- main()            Input handling, output, comparison
  |     |-- greedyRoute()     Nearest-neighbor greedy algorithm
  |     |-- tspDP()           DP path reconstruction & step output
  |     +-- tspHelper()       Recursive memoized TSP solver
  |
  +-- README.md
```

6. Key Implementation Details

6.1 Bitmask Representation

Each subset of visited cities is encoded as an integer. For N cities there are 2^N possible subsets. City i is marked visited by setting bit i using: mask | (1 << i).

6.2 Memoization

dp[mask][pos] = -1 signals an uncomputed state. Once solved, the result is cached so repeated subproblems are answered in O(1) time, preventing exponential recomputation.

6.3 Path Reconstruction

The parent[mask][pos] table stores the best next city chosen at each state. After the optimal cost is computed, the path is traced forward from the initial state (mask=1, pos=0) to reconstruct the full tour.

6.4 Cycle Completion

Both algorithms return to City A after visiting all cities, making the comparison fair — both solve the same closed-tour TSP problem under identical constraints.

7. Getting Started

7.1 Prerequisites

- A C++ compiler supporting C++11 or later (e.g., g++, clang++)

7.2 Compile and Run

```
g++ -std=c++11 -o smart_route_planner smart_route_planner.cpp
./smart_route_planner
```

7.3 Input Options

1. Predefined sample dataset — 4-city distance matrix for instant demo.
2. Custom dataset — enter your own N×N adjacency matrix (2 ≤ N ≤ 10).

Note: Negative distances are rejected. The matrix must be square and non-negative.

8. Constraints

- Maximum 10 cities (enforced by input validation)
- All distances must be non-negative integers
- Distance matrix must be square (N × N)
- Always starts and ends at City A (index 0)

9. Concepts Demonstrated

- Travelling Salesman Problem (TSP)
- Greedy algorithms and their limitations
- Dynamic Programming with bitmask (subset DP)
- Memoization and state-space search
- Path reconstruction using parent tables
- Time complexity analysis and algorithm comparison

10. Conclusion:

The Smart Route Planner project successfully demonstrates two fundamental algorithmic approaches for solving the Travelling Salesman Problem (TSP): the Greedy (Nearest Neighbor) method and Dynamic Programming using bitmasking. Through this implementation, the project clearly highlights the trade-off between computational efficiency and solution optimality.

The greedy algorithm provides a fast and simple solution, making it suitable for larger datasets where quick approximations are required. However, it does not guarantee the best possible route. In contrast, the dynamic programming approach ensures an optimal solution by exhaustively exploring all possible city combinations using memoization, but at the cost of higher computational complexity.

By comparing both approaches side by side, the project offers a practical understanding of how different algorithms behave under the same problem constraints. It reinforces the importance of selecting the right algorithm based on problem size and requirements.

Overall, this project not only strengthens concepts in algorithm design, dynamic programming, and optimization but also demonstrates their real-world relevance in applications such as logistics, route planning, and network optimization.