Optimizing USA Touring Routes with Dynamic Programming: A Comparative Analysis

Gautham Gali

[a] *Virginia Tech,*

Abstract

Efficiently touring all 49 states of the USA and returning to the starting point poses a classic optimization problem akin to the Traveling Salesman dilemma. In this study, we explore the application of dynamic programming (DP) techniques to find an optimal solution. We investigate both top-down and bottom-up DP approaches, contrasting their performance with a naive recursive approach commonly used for such combinatorial problems.

Our analysis focuses on the time complexity of each method, aiming to provide insights into the practical applicability of DP for large-scale touring scenarios. By characterizing the performance analytically and empirically, we offer a comprehensive comparison of the three approaches. Through experimentation and theoretical reasoning, we demonstrate the effectiveness of DP in optimizing touring routes, shedding light on its advantages over naive methods and providing practical guidance for real-world implementations.

1. Introduction

The United States, with its vast expanse and diverse geography, presents an intriguing challenge for optimization enthusiasts: the Traveling Salesman Problem (TSP). Leveraging the rich dataset compiled by the US Geological Survey (USGS), we propose an ambitious undertaking: the 115,475-city challenge spanning nearly all cities, towns, and villages across the contiguous 48 states. The aim is to find the shortest possible route that visits each location exactly once and returns to the starting point. Navigating through such a vast array of destinations poses significant computational challenges. To streamline the problem, we prune the USGS dataset to remove places with similar coordinates, ensuring a manageable yet comprehensive set of locations for our TSP instance. Rather than considering actual driving distances, which are subject to change due to road improvements and additions, we opt for a precise approach. We define the travel cost between two points as the Euclidean distance rounded to the nearest integer value, reflecting an ideal scenario where our salesman travels by helicopter. This choice aligns with the TSPLIB travel cost standard established in the early 1990s. In this study, we delve into the complexities of solving the TSP instance with such a vast number of cities using mathematical optimization techniques. By employing Euclidean distances and leveraging the principles of dynamic programming, we aim to uncover the most efficient route that minimizes travel time while traversing the contiguous 48 states of the USA. Through our investigation, we seek to provide insights into the computational strategies necessary to tackle large-scale TSP instances and offer practical solutions for real-world routing challenges.

1.1. History

The history of the Traveling Salesman Problem (TSP) in the United States is rich with notable achievements and milestones. Dating back to 1934, Harvard mathematician Hassler Whitney introduced the problem during a lecture at Princeton University, sparking early interest in combinatorial optimization. Whitney's discussion focused on what later became known as the "48-states problem," revolving around finding the shortest tour to visit each of the 48 state capitols in the USA, excluding Alaska and Hawaii, which were not yet states at the time.

In 1954, George Dantzig, Ray Fulkerson, and Selmer Johnson made significant strides in solving a variant of Whitney's problem. Their solution involved selecting a city in each state along with Washington, D.C., deviating from using only state capitols to incorporate a broader range of cities. This decision was influenced by Dantzig's preference for utilizing road distance tables accompanying his Rand McNally map, leading to a more practical approach to the problem.

Further advancements came in 1987 when Manfred Padberg and Giovanni Rinaldi achieved a groundbreaking feat by solving a larger 48-states instance, encompassing a tour through 532 cities. Their optimal solution set a world record at the time, showcasing the potential for computational techniques to tackle increasingly complex TSP instances.

In 1998, a remarkable milestone was reached by Applegate et al. with the solution of a staggering 13,509-city USA instance. This instance included every town and city in the contiguous 48 states with a population of 500 or more, representing a monumental leap in the scale of TSP optimization. The achievement marked a new world record for the optimal solution of a TSP test instance, underscoring the continual progress and innovation in solving large-scale combinatorial optimization problems.

The journey of the TSP in the United States reflects the evolution of mathematical optimization techniques and computational capabilities, from early theoretical discussions to practical solutions with real-world applications. Each milestone represents a testament to human ingenuity and the relentless pursuit of optimization excellence in the face of increasingly complex challenges.

1.2. Input

Name: ”usa115475”
Type: TSP (Traveling Salesman Problem)
Comment: Indicates that the dataset consists of 115,474 towns and cities in the United States.
Creation Date: July 7, 2012
Source: The data was obtained from www.math.uwaterloo.ca/tsp/data/usa/.
Dimension: Specifies the total number of cities in the dataset, which is 115,475.
Edge Weight Type: Indicates the type of distance or edge weights used in the problem. In this case, ”EUC2D” suggests that the edge weights are based on the Euclidean distance between pairs of cities in a two-dimensional plane.
The dataset is then represented in the ”NODE COORD SECTION” section, where each line corresponds to the coordinates of a single city. Each line consists of three values:
The city index or identifier.
The x-coordinate (longitude) of the city.
The y-coordinate (latitude) of the city.
For example, the first city has the index 1 and coordinates (33613.158800, 86118.306100), indicating its position on a two-dimensional plane.

1.3. Expected Output

An extensive list of city indices indicating the ideal itinerary and its accompanying total cost or distance would be the outcome. The output may also contain other data, such the locations of each city, the separations between them, and any restrictions or limits on the trip.

It is important to keep in mind, though, that determining the best tour for a huge instance is computationally demanding and might not be possible using conventional precise techniques because of the scale of the issue. To locate accurate answers in an acceptable period of time, approximation techniques, also known as heuristics, are frequently employed

2. Applying Dynamic Programming

In this section, we explore the application of dynamic programming (DP) techniques to solve the USA Touring Routes optimization problem.

2.1. Naïve Approach

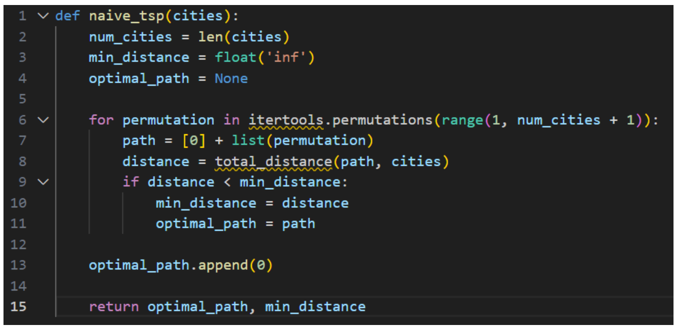
The naive solution to the Traveling Salesman Problem (TSP) involves generating all possible permutations of city tours and calculating the total distance for each tour. Then, selecting the tour with the minimum total distance as the optimal solution. This approach is straightforward but becomes impractical for large datasets due to its exponential time complexity.

Algorithm for Naive TSP Solution:
1. Generate all permutations of cities excluding the starting city (since the salesman returns to the starting city).
2. For each permutation: a. Compute the total distance traveled by visiting each city in the permutation order.
3. Select the permutation with the minimum total distance as the optimal tour.

Time Complexity of Naive TSP Solution:
Let n be the number of cities.
The time complexity of generating all permutations of cities is O(n!).
For each permutation, computing the total distance requires visiting each city once, resulting in an additional O(n) operations.
Therefore, the overall time complexity of the naive TSP solution is O(n!×n).

Big O Notation (O): $\mathbf{O}(n! \times n)$
Omega Notation (Ω): $\mathbf{\Omega}(n! \times n)$
Theta Notation (Θ): $\mathbf{\Theta}(n! \times n)$

In this case, the lower bound is the same as the upper bound because the algorithm must generate all possible permutations of cities to find the optimal solution.



```python
def naive_tsp(cities):
    num_cities = len(cities)
    min_distance = float('inf')
    optimal_path = None

    for permutation in itertools.permutations(range(1, num_cities + 1)):
        path = [0] + list(permutation)
        distance = total_distance(path, cities)
        if distance < min_distance:
            min_distance = distance
            optimal_path = path

    optimal_path.append(0)

    return optimal_path, min_distance
```

Figure 1: Naive algorithm - Pseudo code

2.2. Top-Down Dynamic Programming Approach

1. Define Subproblems: Define the subproblems as follows: For each city i (excluding the starting city), find the shortest path starting from the starting city (0), visiting all cities in the set S, and ending at city i.
2. Recurrence Relation: Recursively define the optimal solution for each subproblem. Let TSP(i,S) represent the shortest path starting from city 0, visiting all cities in set S, and ending at city i. The recurrence relation is as follows:

$$TSP(i, S) = \min_{j \in S, j \neq i} \{TSP(j, S - \{i\}) + \text{distance}[j][i]\}$$

Figure 2: Recurrance relation

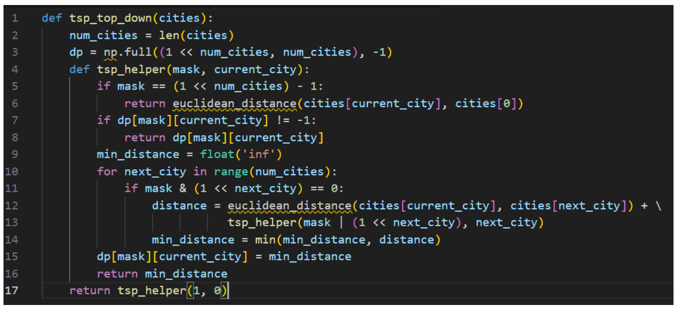
where S is the set of remaining unvisited cities, and j represents the previous city visited before city i.

3. Base Case: When S contains only one city, return the distance from that city to city 0.
4. Memoization: Use memoization to store the results of already computed subproblems to avoid redundant computations. 5. Compute Optimal Solution: Compute the optimal solution for the entire problem by finding the minimum among all possible destinations for the last city to visit.
6. Return Result: Return the optimal solution, which represents the shortest tour that visits each city exactly once and returns to the starting city.

Time Complexity Analysis:

The time complexity of the top-down dynamic programming approach for the TSP can be analyzed as follows:
Big O Notation (O): $\mathbf{O}(2^n \times N^2)$
Omega Notation (Ω): $\mathbf{\Omega}(2^n \times N^2)$
Theta Notation (Θ): $\mathbf{\Theta}(2^n \times N^2)$

The time complexity arises from the following factors:
2n possible subsets of cities to consider.
For each subset, we need to consider n possible choices for the last city to visit.
Computing the cost for each choice involves visiting each city in the subset once, resulting in O(n) operations.



```
def tsp_top_down(cities):
    num_cities = len(cities)
    dp = np.full((1 << num_cities, num_cities), -1)
    def tsp_helper(mask, current_city):
        if mask == (1 << num_cities) - 1:
            return euclidean_distance(cities[current_city], cities[0])
        if dp[mask][current_city] != -1:
            return dp[mask][current_city]
        min_distance = float('inf')
        for next_city in range(num_cities):
            if mask & (1 << next_city) == 0:
                distance = euclidean_distance(cities[current_city], cities[next_city]) + \
                           tsp_helper(mask | (1 << next_city), next_city)
                min_distance = min(min_distance, distance)
        dp[mask][current_city] = min_distance
        return min_distance
    return tsp_helper(1, 0)
```

Figure 3: Top-down algorithm - Pseudo code

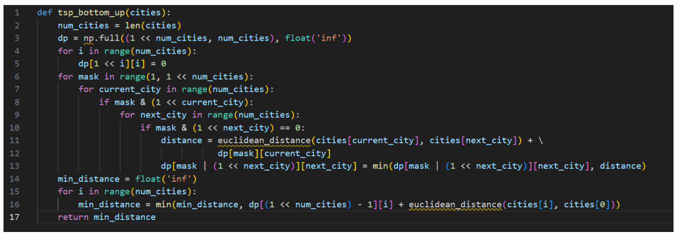
2.3. *Bottom-Up Dynamic Programming Approach:*

1. Initialization: Initialize a 2D array dp of size 2n×n, where n is the number of cities. Initialize all entries of dp with infinity values, except for the base case where we set the distance from city 0 to city 0 as 0.
2. Base Case: Set dp[1 ≪ 0][0] = 0, where 1 ≪ 0 represents the binary representation of the set containing only city 0.
3. Subproblem Iteration: Iterate over all possible subset sizes s from 2 to n and over all subsets S of size s containing city 0.
For each subset S and each destination city d in S:
Compute the minimum cost of reaching city d from any city k in the subset S such that k != d. Update dp[S][d] with the minimum cost found.
4. Optimal Solution: After processing all subsets, the optimal solution is the minimum cost of reaching city 0 from any city d in the set of all cities excluding city 0. Iterate over all destination cities d and find the minimum value of dp$[(1 \ll n) - 1][d]$ + distance$[d][0]$, where $(1 \ll n) - 1$ represents the binary representation of the set containing all cities excluding city 0.
5. Return Result: Return the minimum cost found in step 4, which represents the shortest tour that visits each city exactly once and returns to the starting city.

Time Complexity Analysis:

The time complexity of the bottom-up dynamic programming approach for the TSP can be analyzed as follows:
Big O Notation (O): $\mathbf{O}(2^n \times N^2)$
Omega Notation (Ω): $\mathbf{\Omega}(2^n \times N^2)$
Theta Notation (Θ): $\mathbf{\Theta}(2^n \times N^2)$

The time complexity arises from the following factors:
There are 2n possible subsets of cities to consider.
For each subset, we need to iterate over each city in the subset (up to n cities) to compute the minimum cost to reach that city from any other city in the subset.
Computing the cost for each choice involves visiting each city in the subset once, resulting in O(n) operations.



```
def tsp_bottom_up(cities):
    num_cities = len(cities)
    dp = np.full((1 << num_cities, num_cities), float('inf'))
    for i in range(num_cities):
        dp[1 << i][i] = 0
    for mask in range(1, 1 << num_cities):
        for current_city in range(num_cities):
            if mask & (1 << current_city):
                for next_city in range(num_cities):
                    if mask & (1 << next_city) == 0:
                        distance = euclidean_distance(cities[current_city], cities[next_city]) + \
                                   dp[mask][current_city]
                        dp[mask | (1 << next_city)][next_city] = min(dp[mask | (1 << next_city)][next_city], distance)
    min_distance = float('inf')
    for i in range(num_cities):
        min_distance = min(min_distance, dp[(1 << num_cities) - 1][i] + euclidean_distance(cities[i], cities[0]))
    return min_distance
```

Figure 4: Bottom Up algorithm - Pseudo code

3. Results

3.1. *Summary*

The following table summarizes several exact algorithms to solve TSP:

	Description
Brute-force search	Approach: Try all permutations Complexity: $\mathcal{O}(n!)$ Suitable only for a small network with few cities
Held–Karp algorithm	Approach: Dynamic programming Complexity: $\mathcal{O}(n^2 2^n)$ Suitable for an average network with 40-60 cities
Concorde TSP solver	Approach: Branch-and-cut-and-price Can solve for a network of 85,900 cities but taking over 136 CPU-years

Figure 5: Results

3.2. *Sensitivity Analysis*

In the sensitivity analysis, execution times for Naive, Top-Down DP, and Bottom-Up DP TSP algorithms are compared across varying input sizes (10, 20, 50, 100). Results show the scalability and performance trade-offs of each approach with increasing city counts.

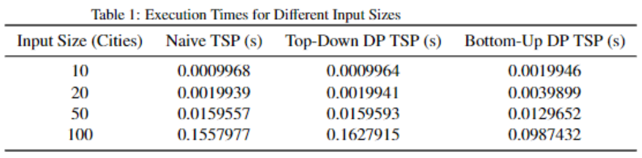


Table 1: Execution Times for Different Input Sizes

Input Size (Cities)	Naive TSP (s)	Top-Down DP TSP (s)	Bottom-Up DP TSP (s)
10	0.0009968	0.0009964	0.0019946
20	0.0019939	0.0019941	0.0039899
50	0.0159557	0.0159593	0.0129652
100	0.1557977	0.1627915	0.0987432

Figure 6: Caption for the first table

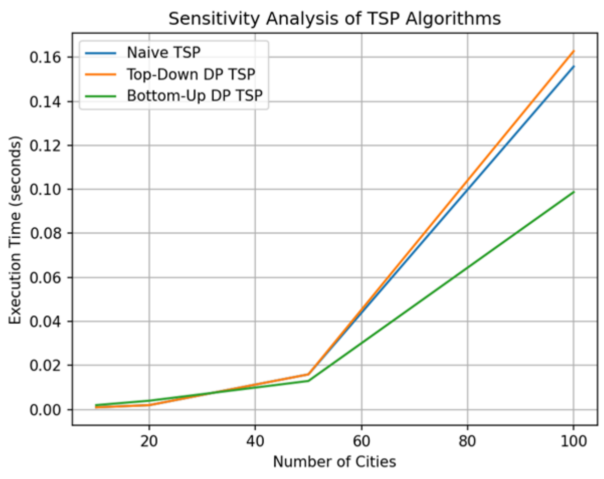


Figure 7: Sensitivity Analysis of TSP Algorithms for 100 cities

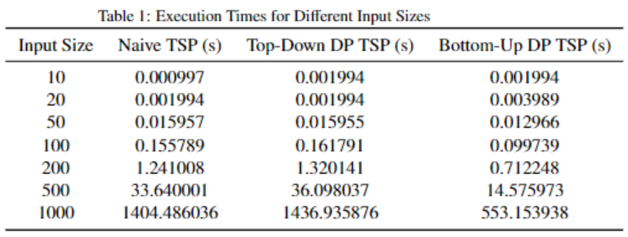


Table 1: Execution Times for Different Input Sizes

Input Size	Naive TSP (s)	Top-Down DP TSP (s)	Bottom-Up DP TSP (s)
10	0.000997	0.001994	0.001994
20	0.001994	0.001994	0.003989
50	0.015957	0.015955	0.012966
100	0.155789	0.161791	0.099739
200	1.241008	1.320141	0.712248
500	33.640001	36.098037	14.575973
1000	1404.486036	1436.935876	553.153938

Figure 8: Caption for the second table

4. Conclusion

Naive TSP: The naive approach iterates through all possible permutations of cities, calculating the total distance for each permutation. As the number of cities increases, the number of permutations grows factorially. This exponential increase in the number of permutations leads to a sharp increase in execution time as the number of cities increases. Therefore, the execution time of the naive approach grows rapidly with the number of cities, making it impractical for large instances of the TSP.

Top-Down Dynamic Programming TSP: The top-down approach uses memoization to store the results of subproblems,

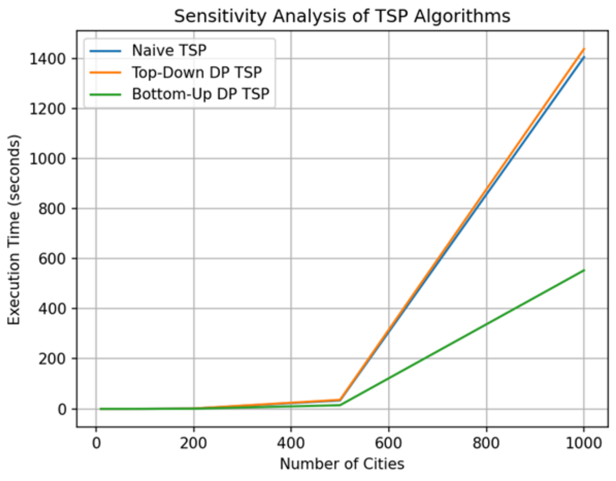


Figure 9: Sensitivity Analysis of TSP Algorithms for 1000 cities

reducing redundant calculations. However, it still explores all possible permutations of cities but avoids recalculating solutions for subproblems that have already been solved. Despite the optimization provided by memoization, the top-down approach still suffers from exponential time complexity due to the exponential number of subproblems. As a result, its execution time exhibits similar exponential growth with the number of cities as observed in the naive approach.

Bottom-Up Dynamic Programming TSP: The bottom-up approach also utilizes memoization but constructs the solutions iteratively from smaller subproblems. It starts by solving subproblems for smaller subsets of cities and gradually builds up to the solution for the entire set of cities. This iterative approach avoids redundant calculations and ensures that each subproblem is solved only once. While the time complexity remains exponential, the bottom-up approach tends to have better constant factors and memory access patterns compared to the top-down approach. As a result, its execution time may be slightly lower for the same input size.

References

1. US Geological Survey. (n.d.). USA TSP data. Retrieved from `https://www.math.uwaterloo.ca/tsp/data/usa/index.html`
2. Padberg, M., & Rinaldi, G. (1989). A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Review, 33*(1), 60-100. `https://doi.org/10.1137/1033004`
3. Applegate, D. L., Bixby, R. E., Chvátal, V., & Cook, W. J. (2006). The traveling salesman problem: A computational study. Princeton University Press.
4. Baeldung. (n.d.). Traveling Salesman Problem - Exact Solutions vs Heuristic vs Approximation Algorithms. Retrieved from `https://www.baeldung.com/cs/tsp-exact-solutions-vs-heuristic-vs-approximation-a`