CSE331 - Assignment #2

Ho Mai Phuong (20232006)
UNIST
South Korea
phuonggii@unist.ac.kr

1 INTRODUCTION

In this project, I aim to analyze three algorithms for solving the Traveling Salesman Problem (TSP), which include two existing algorithms, namely Christofides and Held-Karp, along with another algorithm that I designed myself.

For each algorithm, I evaluated their running time and outputs accuracy, then compared them with ground truth values to learn about their performance practicality in different cases. The datasets I used vary in input sizes: berlin52.tsp, a280.tsp, xql662.tsp, kx9976.tsp, mona-lisa100K.tsp.

The algorithms implementation can be found on GitHub[1]

2 PROBLEM STATEMENT

The Travelling Saleman Problem states that: given a set of n cities and a distance matrix that represents the distances between every pair of cities, the goal is to find the shortest possible route (Hamiltonian cycle with minimum cost) that:

- Starts at a specified city (or any city, since the problem is symmetric)
- Visits each city exactly once
- Return to the starting city

The TSP is known to be NP-hard, meaning there is no known polynomial-time algorithm to solve it exactly for arbitrary instances. Despite its simplicity, it has practical applications in fields such as logistics, circuit design, genome sequencing, and robotics. Because of its computational intractability at scale, a wide range of heuristic and approximation algorithms have been developed to find near-optimal solutions within reasonable time constraints.

3 EXISTING ALGORITHMS

3.1 Christofides heuristic algorithm

The Christofides algorithm is an approximation algorithm to find approximate solution to the TSP problem. It guarantees to output a Hamiltonian circuit with a total weight of at most $\frac{3}{2}$ times the weight of the optimal tour, given a complete undirected graph with non-negative edge weights satisfying the triangle inequality. The algorithm constructs a Minimum Spanning tree of the graph (my implementation used Kruskal's algorithm), then identifies the vertices with odd degree in the MST. By handshaking lemma, the number of odd-degree vertices is even. Compute a minimum-weight perfect matching for these odd-degree vertices, forming a subgraph where each vertex has degree 2. The MST is combined with the matching to form a connected multigraph in which each vertex has even degrees. From this multigraph, find an Eulerian circuit. Finally turn the Eulerian circuit into a Hamiltonian circuit by "shortcutting" (skip repeated vertices), leveraging the triangle inequality to ensure the total weight does not increase.

In worst case, the algorithm runs in $O(n^3)$ time complexity [1]. Specifically, buillding the MST takes $O(mlogn)$ with Kruskal algorithm (m: number of edges, n: number of vertices). Odd-degree vertex identification runs in linear time. Minimum-weight perfect matching runs in $O(n^3)$ using greedy approach. Eulerian circuit and shortcut to Hamilton cycle take $O(m)$ and $O(n)$ running time respectively.

3.2 Held–Karp algorithm

The Held-Karp algorithm is a dynamic programming solution to the TSP and finds an exact solution to the problem. The algorithm breaks down the problem into subproblems instead of checking every possible route.

We define a state containing 2 information: subset of cities (citiies visited so far), and the last visited city in the subset. We create a DP table where each entry $dp[mask][i]$ represents the shortest path cost to visit all the citires in the subset mask and end at city i (mask is bitmask representing which cities have been visited). For each city i in the subset, compute the shortest path to another city j not yet visited and update the table. Finally, we examine the last entries in the table and add the return cost to the starting city to find the exact solution.

The time complexity depends on the number of states and the cost to compute each state.

- Given n cities, there are 2^{n-1} possible subsets S containing the city 0. For each subsets S with k cities, there are $k-1$ possible ending cities j(not 0). The total number of states is bounded by $O(n2^n)$.
- Transition cost: to compute DP table values, we iterate all possible previous cities i in S (not including the ending city). There are at most $n-1$ such cities.

Hence, overall time complexity is $O(n^2 2^n)$. The space complexity used to store DP table is $O(n2^n)$, proportional to the number of states.

4 PROPOSED ALGORITHM (MULTI-START GREEDY)

The algorithm I proposed based on the novel idea of greedy algorithm. Therefore, the design rationale is based on greedy heuristic approach, however there is a slight modification, which is multiple starting points:

- Greedy tour construction: From a starting city, iteratively choose the nearest unvisited city

[1] https://github.com/kirari804/UNIST-CSE-Assignment/tree/main/Algorithm/HW2

- Multiple starting points: Since greedy algorithm normally chooses the random starting point, my idea is to allow starting from every possible city to improve the chances of finding a near-optimal tour.

The main motivation for my idea is to improve greedy accuracy. By trying all starting cities, we improve robustness and get better results than choosing a single starting city. It is also fast for small and moderate-sized inputs.

- The greedyTour function builds a tour by picking the nearest unvisited city. It runs n iterations. In each iteration, it checks all other unvisited cities ($n-1, n-2, \ldots, 1$ comparisons). This results in $O(n^2)$ running time
- In main function, lines 6 to 11 call greedyTour for n cities and compare lengths for each starting city to find the most optimal one. This process runs in $n * O(n^2) = O(n^3)$

Algorithm 1 Euclidean Distance

```
1: function euclideanDistance(a, b)
2:     dx ← a.x − b.x
3:     dy ← a.y − b.y
4:     return √(dx² + dy²)
5: end function
```

Algorithm 3 Compute Tour Length

```
1: function tourLength(cities, tour)
2:     total ← 0
3:     n ← length of tour
4:     for i ← 0 to n − 1 do
5:         from ← cities[tour[i]]
6:         to ← cities[tour[(i + 1) mod n]]
7:         total ← total+ euclideanDistance(from, to)
8:     end for
9:     return total
10: end function
```

5 EXPERIMENTS

5.1 Accuracy

Consider the smallest dataset berlin52.tsp, where the ground truth value is 7542, the Greedy algorithm I design has an output of 8182.19 and Christofides produces 9235.15. My own design algorithm is significantly better since it is only +8.5% worse, compared to Christofides +22.4% worst.

However, Christofides slightly outperforms multi-start Greedy for a280.tsp dataset. Christofides gives the path length of 3033.87 (17.6% longer than ground truth length 2579), and multi-start Greedy outputs 3094.28 (20% worse).

In xql662.tsp dataset, my designed Greedy algorithm is slightly better (0.9% improvement).

From this, I draw to some conclusion:

- For small to medium instances, multi-start Greedy is often the best.

Algorithm 2 Greedy TSP Tour

```
1: function greedyTour(cities, start_index)
2:     n ← length of cities
3:     visited ← array of n false
4:     tour ← empty list
5:     current ← start_index
6:     visited[current] ← true
7:     append current to tour
8:     for step ← 1 to n − 1 do
9:         min_dist ← ∞
10:        next_city ← −1
11:        for i ← 0 to n − 1 do
12:            if not visited[i] then
13:                dist ← euclideanDistance(cities[current],
   cities[i])
14:                if dist < min_dist then
15:                    min_dist ← dist
16:                    next_city ← i
17:                end if
18:            end if
19:        end for
20:        if next_city = −1 then
21:            error: no unvisited city found
22:        end if
23:        visited[next_city] ← true
24:        append next_city to tour
25:        current ← next_city
26:    end for
27:    assert length(tour) = n
28:    return tour
29: end function
```

Algorithm 4 Main Function

```
1: function main(filename)
2:     cities ← parsed from filename
3:     best_length ← ∞
4:     best_tour ← empty list
5:     for i ← 0 to length(cities) - 1 do
6:         tour ← greedyTour(cities, i)
7:         length ← tourLength(cities, tour)
8:         if length < best_length then
9:             best_length ← length
10:            best_tour ← tour
11:        end if
12:    end for
13:    output best_length
14:    output best_tour using city IDs
15: end function
```

- Christofides is more stable and gives good results consistently across instances, especially for larger ones like kz9976.tsp, while Greedy isn't, possibly due to running time
- Held-Karp is exponential and due to its space complexity, there is not enough RAM to run the algorithm on my device.

Algorithms/Data sets	berlin52.tsp	a280.tsp.gz	xql662.tsp	kz9976.tsp	mona-lisa100K.tsp
Christofides	9235.15	3033.87	3072.55	1297604.01	n/a
Held-Karp	n/a	n/a	n/a	n/a	n/a
Greedy (consider all starting points)	8182.19	3094.28	3045.27	n/a	n/a

Table 1: Result of each algorithm

Algorithms/Data sets	berlin52.tsp	a280.tsp.gz	xql662.tsp	kz9976.tsp	mona-lisa100K.tsp
Christofides	0.002s	0.020s	0.119s	35.35s	n/a
Held-Karp	n/a	n/a	n/a	n/a	n/a
Greedy (consider all starting points)	0.012s	1.220s	14.975s	n/a	n/a

Table 2: Running time of each algorithm

5.2 Running time

From observations, I can conclude that Christofides is fastest across datasets, with time grows slowly along input sizes. The algorithm is especially good for large instances like kz9976.tsp as it runs in acceptable time of 35.35s. Since Christofides runs in polynomial time, it scales well, and even works for a dataset of thousands of cities.

Multi-start Greedy is much slower than Christofides in larger instances (for a280.tsp, it is 60 times slower, and for xql662.tsp, it is over 125 times slower). This is due to my implementation of the algorithm, which considers all starting cities, results in $O(n^3)$ total time. For example, taking the dataset xql662.tsp (n=662), one greedy run is $O(n^2) = 666^2$, around $448K$ operations, running n times can lead up to 289 million steps. Hence, for very large instances (kz9976.tsp or mona-lisa100K.tsp), it may take up to a week and many days to complete.

While running Held-Karp algorithm, the program is killed due to insufficient memory. Its space complexity is $O(n2^n)$, so even for $n = 25$, it starts exceeding GBs of RAM (my device RAM is 16GB). Therefore, Held-Karp is only feasible for very small instances (n<25).

6 CONCLUSION

Based on experiments and observation, I can summarize the **ranking** based on running time and accuracy:

Running time: Christofides » Multi-start Greedy » Held-Karp

Accuracy:

- *for small instances*, Held-Karp » Multi-start Greedy » Christofides
- *for medium instances*, Multi-start Greedy » Christofides » Held-Karp (exceed RAM)
- *for large instances*, Christofides » Multi-start Greedy (very long running tine) » Held-Karp (exceed RAM)

My designed algorithm - multi-start Greedy has both pros and cons. Although it gives a more optimal output than Christofides and not restricted to small datasets like Held-Karp, its running time is extremely long for very large datasets. Hence, a hybrid algorithm, which combines the use of Held-Karp for $n < 20$ instances, multi-start Greedy for $n < 1K$ instances, and Christofides for $n > 1K$ instances would be a more optimal algorithm, considering characteristics of each algorithm as discussed above.

REFERENCES

[1] Nicos Christofides. 2022. Worst-case analysis of a new heuristic for the travelling salesman problem. In *Operations Research Forum*, Vol. 3. Springer, 20.