TSP Heuristic via Kruskal-Based Local Trees and MST-Driven Global Linking

Sehyun Yun (20231233)
UNIST
South Korea
nawhji@unist.ac.kr

1 Introduction

The Traveling Salesman Problem (TSP) is a fundamental combinatorial optimization problem that asks for the shortest possible route visiting each of n cities exactly once and returning to the starting city. It appears in a variety of practical applications, such as delivery routing, urban planning, and biological data analysis, and is known to be NP-hard.

Due to its exponential time complexity, solving TSP exactly becomes infeasible for large instances. Consequently, various approximation algorithms and heuristics have been studied. Among them, the dynamic programming approach by Held and Karp, and the MST-based 2-approximation algorithm are two well-known examples that are simple and illustrative, though not always practical for large-scale problems.

However, real-world distance matrices often violate the assumptions required by these algorithms, or offer structural opportunities that such methods do not exploit. In this report, we propose a new TSP heuristic inspired by MST principles. The algorithm incrementally builds a complete tour by selecting low-cost connections and aims to strike a balance between computational efficiency and solution quality.

This study evaluates the effectiveness of the proposed method through experiments and discusses its potential as a lightweight heuristic for large-scale TSP instances. For further exploration, the full codebase is available at public repository[1].

2 Problem Statement [1]

Traveling Salesman Problem (TSP).. Given a set of n cities and a pairwise distance function $d(i, j)$ for all city pairs, the goal is to find a minimum-cost tour that visits each city exactly once and returns to the starting point. Formally, let $G = (V, E)$ be a complete undirected graph where $V = \{0, 1, \ldots, n-1\}$ and E is the set of weighted edges. The objective is to find a permutation π of $\{0, 1, \ldots, n-1\}$ that minimizes the total tour cost:

$$\text{TSP}(\pi) = d(\pi(n-1), \pi(0)) + \sum_{i=0}^{n-2} d(\pi(i), \pi(i+1))$$

Decision Version. The decision version of TSP asks: Given a graph G and a number k, does there exist a Hamiltonian cycle whose total cost is less than or equal to k?

Computational Hardness. The TSP is a well-known NP-hard problem. Even its decision version is NP-complete. This means:

- The problem is in NP: given a proposed tour, its cost can be verified in polynomial time.
- The problem is NP-hard: any problem in NP can be reduced to TSP in polynomial time.

NP-hardness Proof Outline. To prove that the TSP is NP-hard, we reduce from the *Hamiltonian Cycle Problem* (HCP), which is known to be NP-complete.

- Given an instance of HCP: an unweighted graph $G' = (V, E')$, determine whether there exists a simple cycle that visits every vertex exactly once.
- Construct a complete graph $G = (V, E)$ where:

$$d(u, v) = \begin{cases} 1 & \text{if } (u, v) \in E' \\ 2 & \text{otherwise} \end{cases}$$

- Let $k = |V|$. Then, G' has a Hamiltonian cycle $\iff$ the corresponding TSP instance has a tour of cost $\leq k$.

This reduction is polynomial in size and proves that TSP is NP-hard.

Computational Complexity. The number of possible tours grows factorially with the number of cities, specifically $O(n!)$, since each valid tour corresponds to a permutation of the cities. This results in an exponential time complexity for exact algorithms. Consequently, solving TSP exactly becomes computationally infeasible as n increases, making it impractical for large-scale instances.

3 Existing Algorithms

3.1 Held and Karp Algorithm [2]

Algorithm Overview. The Held-Karp algorithm is a well-known dynamic programming solution to the *Traveling Salesman Problem (TSP).* It aims to find the minimum-cost tour that visits every city exactly once and returns to the starting point.

The key idea is to build up optimal solutions to subproblems and use them to construct the optimal solution to the full problem. Instead of evaluating all possible city permutations, the algorithm stores and reuses partial results for subsets of cities.

Dynamic Programming Formulation. The Held-Karp algorithm defines a dynamic programming table $dp[S][i]$, where $S \subseteq \{0, 1, \ldots, n-1\}$ is a subset of cities visited so far, and $i \in S$ is the city where the tour ends. The value $dp[S][i]$ represents the minimum cost to reach city i after visiting all cities in S. The table is filled using the recurrence:

$$dp[S][i] = \min_{j \in S,\ j \neq i} (dp[S \setminus \{i\}][j] + \text{cost}[j][i])$$

This relation considers all possible previous cities j in the subset S (excluding i), and selects the one that yields the lowest total cost to reach i.

[1] https://github.com/nawhji/CSE331-2

The base case assumes that city 0 is the starting point. Thus, the only entry initialized explicitly is:

$$dp[2^0][0] = 0$$

All other entries are initialized to infinity, representing unreachable states at the start.

Once the table is fully computed, the optimal tour cost is obtained by completing the cycle: returning from any last visited city $i \neq 0$ back to city 0. The final answer is therefore:

$$\min_{i \neq 0} \left(dp[2^n - 1][i] + \text{cost}[i][0] \right)$$

Algorithm. The pseudocode for Held-Karp is as follows:

Algorithm 1 HELD_KARP($matrix, n$)

Input: An $n \times n$ distance matrix `matrix[0..n-1][0..n-1]`
Output: Minimum cost of a TSP tour

Initialize $dp[S][i] \leftarrow \infty$ for all $S \in [0, 2^n)$ and $i \in [0, n)$
$dp[1 \ll 0][0] \leftarrow 0$
for $i \leftarrow 1$ to $n - 1$ **do**
    $dp[(1 \ll 0) \mid (1 \ll i)][i] \leftarrow matrix[0][i]$
for $S \leftarrow 1$ to $2^n - 1$ **do**
    for $i \leftarrow 1$ to $n - 1$ **do**
        if $(S \gg i)$ & $1 = 0$ **then**
            continue
        for $j \leftarrow 1$ to $n - 1$ **do**
            if $j = i$ or $(S \gg j)$ & $1 = 0$ **then**
                continue
            $dp[S][i] \leftarrow \min(dp[S][i], dp[S \oplus (1 \ll i)][j] + matrix[j][i])$
$end \leftarrow 2^n - 1$
$result \leftarrow \infty$
for $i \leftarrow 1$ to $n - 1$ **do**
    $result \leftarrow \min(result, dp[end][i] + matrix[i][0])$
return $result$

Complexity. The algorithm considers all subsets of cities, and for each subset, it computes the minimum cost for each possible last city. Thus, the overall time complexity is:

$$O(n^2 \cdot 2^n)$$

and the space complexity is:

$$O(n \cdot 2^n)$$

While the Held-Karp algorithm is significantly faster than brute-force enumeration ($O(n!)$), it still has exponential time and space complexity. As a result, it is only practical for relatively small instances of the TSP.

3.2 MST-Based 2-approximation Algorithm [1]

Algorithm Overview. This algorithm provides a 2-approximation for the *Traveling Salesman Problem (TSP)* when the cost function satisfies the **triangle inequality**. It constructs a tour by computing a *Minimum Spanning Tree (MST)* of the input graph, performing a preorder traversal on the MST to determine a visiting order, and then returning to the starting point to complete the tour.

Key Idea. The MST cost serves as a lower bound on the optimal TSP tour. A preorder traversal of the MST visits all vertices and produces a walk that may revisit some nodes. By shortcutting repeated visits—guaranteed to be non-increasing in cost due to the triangle inequality—a valid Hamiltonian tour is obtained. The resulting tour has total cost at most twice that of the MST, and hence at most twice the optimal TSP tour cost.

Algorithm. The pseudocode below outlines the procedure:

Algorithm 2 MST-based Approximation for TSP

Input: Distance matrix M of dimension n
Output: Approximate TSP tour cost
function MST_BASED_APPROX_2(M, n)
    $T \leftarrow$ PRIM_MST(M, n)
    $P \leftarrow$ PREORDER_TREE_TRAVERSAL(T)
    $C \leftarrow 0$
    for $i = 0$ to $n - 2$ **do**
        $C \leftarrow C + M[P[i]][P[i + 1]]$
    $C \leftarrow C + M[P[n - 1]][P[0]]$
    return C
function PRIM_MST(M, n)
    $V[0] \leftarrow$ root node with index 0
    while number of vertices in MST $< n$ **do**
        Find minimum edge (u, v) with u in MST, v not in MST
        Add v to MST as child/sibling of u
    return root of MST

Formulation. Let T be the MST, H^* the optimal TSP tour, W the full walk of the tree, and H the final shortcut Hamiltonian tour. The following inequalities hold:

$$c(T) \leq c(H^*) \quad \text{(MST is a lower bound)}$$

$$c(W) = 2c(T) \quad \text{(each edge traversed twice in full walk)}$$

$$c(H) \leq c(W) \Rightarrow c(H) \leq 2c(H^*) \quad \text{(by triangle inequality)}$$

Complexity. The MST is computed using Prim's algorithm in $O(n^2)$ time when using an adjacency matrix. The preorder traversal and final tour construction each take $O(n)$ time. Therefore, the overall time complexity is:

$$\boxed{O(n^2)}$$

This algorithm is efficient and simple, and it guarantees a solution within a factor of 2 of the optimal cost.

4 Proposed Algorithm

Motivation and Design Rationale. The core idea of this algorithm was inspired by Kruskal's approach [3] to constructing a minimum spanning tree, which builds a tree by repeatedly selecting the minimum-weight edge and merging connected components. In large TSP instances, however, sorting all $O(n^2)$ edges becomes computationally expensive. To mitigate this, we introduce a percentile-based cutoff to retain only the shortest edges, thereby reducing the number of candidates considered for merging.

Another important distinction is that Kruskal aims to build a tree, whereas TSP requires a single cyclic tour that visits each city exactly once. To preserve this constraint, we restrict merges to occur only between the endpoints of disjoint paths. This prevents premature cycles or invalid path structures while maintaining the greedy nature of Kruskal's merging process.

Key Ideas.

- **Edge Filtering:** Long edges are discarded based on a percentile threshold, reducing computational overhead.
- **Path-Level Merging:** Entire paths are merged only via endpoints, avoiding cycles or internal insertions.
- **Greedy Strategy:** Edges are processed in increasing order of weight to promote early convergence.
- **Two-Phase Structure:** A local merging phase builds as much of the tour as possible, followed by a global linking phase to complete it.

Algorithm Overview. This heuristic aims to construct an approximate TSP tour efficiently, avoiding the combinatorial explosion of enumerating all possible permutations. The algorithm proceeds in two phases.

In the first phase, only edges below a certain percentile threshold are considered to reduce the search space. Each city is initially treated as a separate path, and edges are processed in order of increasing weight. Two paths are merged only if their endpoints can be connected without violating the tour constraints.

If a single complete tour is not formed, the second phase greedily links the remaining paths by selecting the shortest valid endpoint-to-endpoint connections until a Hamiltonian cycle is obtained.

Rather than constructing a full tree, the algorithm incrementally builds a tour-like structure based on edge cost and path connectivity.

Algorithm. The full procedure is detailed in Algorithm 3. For clarity and reproducibility, the algorithm is separated into a local merging phase followed by a global completion step.

Time Complexity.

- Edge filtering from the distance matrix: $O(n^2)$
- Sorting candidate edges: $O(m \log m)$ where $m = O(n^2)$
- Path merging using ID tracking (e.g., via union-find): $O(n\alpha(n))$
- Final linking among remaining paths: $O(n^2)$

Thus, the total time complexity of the algorithm is:

$$\boxed{O(n^2 \log n)}$$

This makes it comparable to classical greedy TSP heuristics in terms of computational efficiency, while offering three notable advantages:

First, the algorithm performs structure-aware merging by treating entire subpaths as units, rather than connecting isolated nodes. This prevents the formation of premature cycles and allows for a more coherent tour structure to emerge naturally.

Second, the algorithm introduces adaptability through kth-percentile-based edge filtering, where only the shortest edges below a certain percentile cutoff k are retained. This k value acts as a tunable hyperparameter: for large-scale instances with many nodes, setting a lower k significantly reduces the number of edges considered, improving computational efficiency. Conversely, when the number of nodes is small, a higher k can be used to preserve more edge information without incurring a high computational cost. This flexibility allows the algorithm to adapt to varying problem scales while maintaining meaningful structural guidance. In particular, the filtering mechanism helps ignore noisy or outlier edges in complex or non-Euclidean distance matrices, leading to more coherent intermediate paths and reducing the risk of early commitment to suboptimal connections.

Third, the two-phase architecture allows the algorithm to first construct partial solutions efficiently and then refine them in a targeted way. This progressive strategy balances greediness with global awareness, often yielding better-quality solutions in practice than simpler greedy approaches.

Algorithm 3 Kruskal-based Heuristic for TSP

Input: Distance matrix M of dimension n
Output: Approximate TSP tour cost
$E \leftarrow$ all edges (u, v) with $M[u][v] \leq$ kth percentile
Sort E in increasing order of weight
Initialize each node as its own path
$C \leftarrow 0$ // total cost
for each edge (u, v) in E **do**
    if u and v belong to different path ends **then**
        Merge paths of u and v
        Update path-to-node mappings
        $C \leftarrow C + M[u][v]$
    if only one multi-node path remains **then**
        break
if more than one path remains **then**
    $X \leftarrow$ all endpoint-to-endpoint edges between remaining paths
    Sort X by weight and greedily merge disjoint paths
    Add each selected edge to C
    Close final cycle by connecting first and last node
return C

5 Experiments

5.1 Experimental Setup

All experiments were conducted on a personal computer equipped with:

- **Processor**: 13th Gen Intel(R) Core(TM) i5-1340P @ 1.90,GHz
- **Operating System**: Windows 11
- **Execution Mode**: Single-threaded
- **Compiler**: `g++ (Rev6, Built by MSYS2 project) 13.2.0`

The algorithms were implemented in C++ and compiled using the above compiler without any additional optimization flags (compiled with `-o` only).

5.2 Test Instances

We evaluated the following three algorithms:

(1) **Held-Karp Algorithm** — classical dynamic programming approach with time complexity of $O(n^2 \cdot 2^n)$
(2) **MST-based 2-Approximation Algorithm** — greedy approximation method with time complexity of $O(n^2)$

(3) **Proposed Heuristic**: *TSP Heuristic via Kruskal-Based Local Trees and MST-Driven Global Linking* — time complexity of $O(n^2 \log n)$

The algorithms were tested on the following datasets:

- **small** ($n = 20$): synthetic small-size test case
- **a280**[2]
- **xql662**[3]
- **kz9976**[4]
- **mona-lisa100K**[5]

Due to its exponential time complexity, the Held-Karp algorithm was only executed on the `small` dataset. The other two algorithms were evaluated on all datasets.

5.3 Results and Discussion

Table 1: Performance and Accuracy Comparison of TSP Algorithms

Instance	Algorithm	Runtime (s)	Tour Cost	Accuracy (%)
small (n=20)	Held-Karp	3.678	75	100.00
	2-Approximation	0.004	94	79.79
	TSP Heuristic	0.0015	79	94.94
a280	2-Approximation	0.0037	3575.45	72.37
	TSP Heuristic	0.151	3125	82.78
xql662	2-Approximation	0.0094	3648	68.91
	TSP Heuristic	0.797	2867.00	87.64
kz9976	2-Approximation	0.799	1458280	72.82
	TSP Heuristic	265.82	1250550	84.89
mona-lisa100K	2-Approximation	1050.73	8328202	69.09
	TSP Heuristic	2636.65	6506320	88.49

As shown in Table 1, the performance of each algorithm varies significantly depending on the size and complexity of the problem instance.

For the **small** dataset ($n = 20$), the Held-Karp algorithm computed the exact optimal cost of 75. The proposed *TSP Heuristic* achieved a highly competitive result with a tour cost of 79 (94.94% accuracy) and completed in only 0.0015 seconds, demonstrating strong approximation capability even for small-scale instances.

As the problem size increases, the performance gap between algorithms becomes more pronounced. The **MST-based 2-Approximation** consistently yielded the lowest accuracy, particularly on large datasets such as `mona-lisa100K`, where accuracy dropped to 69.09%. This result aligns with the theoretical bound of 2-approximation algorithms and highlights their practical limitations.

In contrast, the **proposed heuristic** maintained over 80% accuracy across all large-scale instances, reaching 84.89% on `kz9976` and 88.49% on `mona-lisa100K`. This suggests that the combination of MST-based local linking and global edge selection becomes increasingly effective as instance size grows.

The heuristic also demonstrated competitive runtime performance. On medium-sized datasets such as `a280` and `xql662`, high-quality solutions were produced within a few seconds or less. Overall, the method offers a practical trade-off between solution quality and computational efficiency, making it well-suited for large or time-sensitive TSP instances.

5.4 Effect of Cutoff Parameter k

Table 2: Effect of Cutoff Parameter k on TSP Heuristic (Instance: `kz9976`)

Cutoff k	Runtime (s)	Tour Cost	Accuracy (%)
0.90	265.82	1250550	84.89
0.50	277.652	1250550	84.89
0.30	283.259	1250550	84.89
0.10	310.973	1250550	84.89
0.05	287.993	1250550	84.89

The effect of the cutoff parameter k, which controls the proportion of shortest edges retained during the initial filtering stage, was also examined. Table 2 presents the results for the `kz9976` instance.

Interestingly, the tour cost remained constant across all tested k values, indicating that the algorithm consistently produced the same solution regardless of the size of the retained edge set. However, runtime showed a slight increase as k decreased. This is likely because smaller k values reduce the number of available merge candidates, resulting in longer resolution paths during the linking phase.

A likely explanation for the invariant tour cost is the algorithm's greedy merge strategy, which always processes edges in ascending order of distance. As a result, even with fewer edges retained, the most influential ones—those most likely to be included in the optimal tour—are still present. This leads to an almost identical merging sequence and the same final tour.

It is important to note that the impact of k may vary across datasets. In sparser or irregularly distributed graphs, overly aggressive filtering may eliminate essential connections, potentially degrading solution quality or feasibility. Thus, the choice of k should be adapted based on the scale and structure of the input instance.

6 Conclusion

This work introduced a Kruskal-based heuristic for the TSP that constructs tours by greedily merging endpoint paths using short edges. Experimental results demonstrate improved accuracy and flexibility compared to the classical 2-approximation, particularly on large-scale instances. The method also showed robustness to variations in the cutoff parameter, making it a practical approach for efficient tour generation.

References

[1] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

[2] Michael Held and Richard M Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied mathematics*, 10(1):196–210, 1962.

[3] Jon Kleinberg and Eva Tardos. *Algorithm design*. Pearson Education India, 2006.

[2] http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp/

[3] https://www.math.uwaterloo.ca/tsp/vlsi/index.html

[4] https://www.math.uwaterloo.ca/tsp/world/countries.html

[5] https://www.math.uwaterloo.ca/tsp/data/ml/monalisa.html