CSE331 - Assignment #2

Minseo Kim*
UNIST
Ulsan, South Korea
sirusister624@unist.ac.kr

1 Introduction

The Traveling Salesman Problem (TSP) stands as a central problem in the field of combinatorial optimization [4], offering both an intellectual challenge and a practical benchmark for evaluating algorithms. First formalized in the early 20th century, but with roots stretching back even further, the TSP asks a question that is easy to state and universally understood: given a collection of cities and the cost of travel between each pair, what is the shortest possible route that visits every city exactly once and returns to the starting point? Despite its simplicity, the TSP is famously difficult to solve optimally, with the size of the search space growing factorially as the number of cities increases.

The importance of the TSP extends far beyond theory. In logistics and supply chain management, the TSP models real-world routing problems such as delivery truck scheduling, printed circuit board manufacturing, and even genome sequencing, where visiting all required locations in an optimal manner saves both time and resources. The TSP is also deeply intertwined with computational complexity theory, as it is a canonical NP-hard problem. Understanding the TSP has guided the development of many of the tools and concepts that underpin modern algorithm design, from greedy methods to dynamic programming and advanced metaheuristics [4].

Over the decades, the TSP has inspired a wealth of research and practical innovation [1]. The quest for better algorithms has led to the creation of entire fields within discrete mathematics and operations research, and advances in solving the TSP have frequently driven progress in other domains. The TSP thus remains as relevant today as ever, providing both a testbed for new ideas and a source of challenging real-world problems.

2 Problem Statement and Theoretical Background

The TSP can be formally described using the language of graph theory [4]. Let $G = (V, E)$ be a complete undirected graph, where V is a set of n vertices (cities), and E contains all possible edges between pairs of vertices. Each edge (u, v) has an associated non-negative cost $c(u, v)$, representing the distance, time, or expense required to travel from city u to city v. The objective is to find a permutation $\pi = (\pi_1, \pi_2, ..., \pi_n)$ of the cities that minimizes the total cost of the tour:

$$\text{Cost}(\pi) = \sum_{i=1}^{n-1} c(\pi_i, \pi_{i+1}) + c(\pi_n, \pi_1)$$

In other words, the tour starts at π_1, visits each city in the order specified by π, and finally returns to the starting city. The challenge arises from the sheer size of the search space: there are $(n-1)!/2$ possible tours for n cities, due to the cyclical and undirected nature of the problem. This exponential growth quickly makes exhaustive search infeasible even for moderate n.

One of the key theoretical insights about the TSP is its NP-hardness. There is no known polynomial-time algorithm that can solve all instances of the TSP to optimality, and the consensus is that no such algorithm exists unless P = NP. Moreover, even the problem of *approximating* the TSP solution within a certain factor is NP-hard in the general case. However, when the cost function c satisfies the *triangle inequality*—that is, $c(u, v) + c(v, w) \geq c(u, w)$ for all $u, v, w \in V$—approximation algorithms with provable guarantees become possible [1]. This "metric" assumption is satisfied in many real-world applications, such as those based on Euclidean distances.

The TSP's connections to other areas of mathematics are deep and multifaceted. For instance, the minimum spanning tree (MST) of G—the cheapest way to connect all cities without cycles—always costs less than or equal to the optimal TSP tour. However, this lower bound can be quite loose, especially for graphs where the MST is "star-like" and the optimal tour is much more circuitous.

Another rich area of study is the polyhedral structure of the TSP. The set of all feasible tours can be viewed as the vertices of a high-dimensional polytope, and cutting-plane methods that exploit this structure are the basis of the fastest exact solvers for large TSP instances today.

3 Existing Algorithms

I implemented and analyzed two classic approaches for the TSP: an MST-based 2-approximation algorithm and the Held–Karp exact dynamic programming algorithm.

3.1 MST-based 2-Approximation Algorithm

One of the simplest algorithms for the metric TSP is the MST-based 2-approximation. The idea is to use a minimum spanning tree (MST) as a scaffold: since any TSP tour must be at least as expensive as the MST, and the triangle inequality allows us to "shortcut" repeated nodes, we can create a feasible tour by walking around the tree and skipping duplicates [1].

The algorithm proceeds in three main steps:

(1) Compute the MST T of G using an efficient algorithm such as Prim's [2] or Kruskal's. This provides a subset of E connecting all nodes at minimal total cost, without cycles.
(2) Perform a preorder traversal of T starting from any node. This walk visits every node, possibly multiple times.
(3) Shortcut the walk by removing repeated visits to any node, resulting in a Hamiltonian cycle. Thanks to the triangle inequality, this shortcutting never increases the cost.

Formally, the cost of the resulting tour satisfies

$$\text{Cost}_{\text{MST-approx}} \leq 2 \cdot \text{MST}(G) \leq 2 \cdot \text{OPT}$$

*Student ID: 20231051

where OPT is the cost of the optimal TSP tour. This bound is tight for some contrived graphs, but is usually much better in practice.

A key advantage of the MST-based approximation is its speed: it runs in $O(n^2)$ time and can handle very large graphs. However, the main limitation is the possible gap between the MST and the actual shortest tour, which can be significant in instances with irregular distances or "outlier" cities.

3.2 Held–Karp Dynamic Programming Algorithm

For small to moderate n, the Held–Karp dynamic programming algorithm [3] offers a way to compute the exact solution by systematically exploring all subsets of cities. The core of the algorithm is a recurrence that captures the shortest path covering a given set S of cities and ending at city i:

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

with the base case $dp[\{0, i\}][i] = c(0, i)$.

The final answer is obtained by evaluating:

$$\min_{i \neq 0} (dp[V][i] + c(i, 0))$$

The time and space complexity are both $O(n^2 2^n)$, which is exponentially better than brute-force search but still prohibitive for $n \geq 25$. In practice, the Held–Karp algorithm is invaluable for benchmarking and for verifying the performance of heuristics on small graphs.

The Held–Karp algorithm illustrates the power and limitations of dynamic programming for NP-hard problems: while a huge improvement over naive enumeration, exponential growth remains an insurmountable barrier for large n.

4 Proposed Algorithm: Multi-Start Greedy + 2-Opt Local Search

Recognizing the limitations of both exact and simple approximation algorithms, I designed a practical metaheuristic combining randomized greedy initialization with robust local search.

4.1 Design Motivation

The MST-approximation is extremely fast but suboptimal. On the other hand, exact algorithms are limited to very small instances. I aimed to find a balance: an algorithm that achieves near-optimal solutions for large n in practical time.

4.2 Algorithm Description

The approach is:

(1) For a number of iterations or while under a time limit:
   - Pick a random starting node.
   - Construct a tour using the Greedy Nearest Neighbor (NN) heuristic.
   - Apply the 2-Opt local search to iteratively improve the tour.
   - If this tour is better than the current best, record it.

(2) Return the best tour found.

The Greedy NN heuristic quickly produces a tour by always moving to the nearest unvisited node. The 2-Opt algorithm repeatedly removes two edges and reconnects the two resulting paths in a different way, accepting the change if it leads to a shorter tour. This is repeated until no further improvement is possible, at which point the tour is a local minimum in the 2-Opt neighborhood.

Pseudo-code:

```
for repeat = 1 to runs or while time remains:
    pick random start node
    tour <- Greedy NN from start
    tour <- full 2-Opt(tour)
    if tour better than best: update best
return best
```

Although 2-Opt has no theoretical approximation guarantee, it is empirically effective for metric TSP and forms the basis for state-of-the-art local search algorithms [4].

5 Implementation Details

All algorithms were implemented in C, with careful attention to modularity, efficiency, and robust memory management. This section summarizes the architecture and the most important functions of the codebase.

5.1 Graph Representation and Utilities

The TSP graph is represented as a structure containing both a distance matrix and coordinates for each node:

```
typedef struct Graph {
    int n;          // Number of nodes
    int **dist;     // Distance (cost) matrix, n x n
    double **coords;
    // Coordinates: coords[i][0]=x_i, coords[i][1]=y_i
} Graph;
```

Creation and Memory Management:

- `Graph *graph_create(int n)` allocates all matrices for an n-node graph, ensuring proper error handling and cleanup on allocation failure.
- `void graph_free(Graph *g)` deallocates all dynamic memory, avoiding leaks even in exceptional cases.

Distance and Coordinate Management:

- `void graph_set_edge(Graph *g, int u, int v, int w)` sets the edge cost between nodes u and v.
- `int graph_get_weight(const Graph *g, int u, int v)` retrieves the cost, returning a large value (infinity) for invalid queries.
- `void graph_get_xy(const Graph *g, int v, double *x, double *y)` provides access to the original coordinates.

Parsing:

- `Graph *parse_tsp_file(const char *filename)` parses a TSPLIB file and constructs the `Graph` structure, filling coordinates and the distance matrix using rounded Euclidean distances.

5.2 MST-based 2-Approximation Algorithm

Implemented in `mst_approx.c`, the MST-based algorithm computes a minimum spanning tree and constructs a tour by traversing the tree.

- `int mst_approx_tour(const Graph *g, int *tour_out)` returns the cost and fills the node sequence of the tour.
- `prim_mst_parents(const Graph *g)` uses Prim's algorithm [2] to find MST parent relationships.
- The MST is converted to an adjacency list, and a preorder DFS (`dfs_preorder`) records the traversal order.
- The final tour is created by following this order and shortcutting repeated nodes.
- Time complexity is $O(n^2)$, allowing for very large instances.

5.3 Held–Karp Dynamic Programming Algorithm

Implemented in `held_karp.c`, this exact algorithm computes the optimal solution for small n via bitmask DP [3].

- `int held_karp_tour(const Graph *g, int *tour_out, double time_limit_s)` computes the minimum-cost tour for $n \leq 25$, storing the result in `tour_out`.
- The DP table `dp[mask][i]` tracks the shortest path covering a subset `mask` ending at node i.
- Parent pointers are used for reconstructing the optimal path.
- For $n > 25$, the function skips computation to avoid excessive resource usage.

5.4 Metaheuristic: Multi-Start Greedy + 2-Opt Local Search

Implemented in `my_algo.c`, this algorithm is designed for large-scale instances by combining greedy construction, local search, and multiple restarts.

- `int my_algo_tour(const Graph *g, int *tour_out)` serves as the entry point and selects between a fixed restart count and time-bounded search.
- `greedy_nn`: constructs an initial tour using the Greedy Nearest Neighbor heuristic from a random starting city.
- `full_two_opt`: repeatedly applies 2-Opt swaps within a given span to locally improve the tour.
- `run_fixed`: executes multiple random restarts, retaining the best tour.
- `run_timed`: for very large n, runs as many restarts as possible within a fixed time limit.

Main loop (pseudocode):

```
for repeat = 1 to runs or while time remains:
    pick random start node
    tour <- Greedy NN from start
    tour <- full 2-Opt(tour)
    if tour better than best: update best
return best
```

5.5 Main Driver and CLI

The `main.c` driver parses command-line arguments and coordinates the execution:

```
Usage:
./tsp_solver --mode [mst|held-karp|myalgo] --input <file.tsp>
```

It loads the input file, selects the algorithm, reports cost and tour, and manages memory cleanup.

5.6 Robustness and Modularity

Key aspects of the implementation include:

- Careful error checking and cleanup for all memory allocations.
- Handling infeasible or excessive cases (e.g., skipping Held–Karp for large n).
- Randomization and time-limited search for repeatable and scalable experiments.
- Clean modular organization: each algorithm and utility is in its own source/header file.

5.7 Source Code Availability

All code, build scripts, and experiment drivers are available at: https://github.com/UNISTminseo/algorithms_as2_20231051

This codebase is robust, modular, and efficient, serving as a strong foundation for further research and experimentation on large-scale TSP.

6 Experiments

6.1 Experimental Setup

All algorithms were implemented in C and run on a MacBook Pro (Apple M3, 16GB RAM). The following publicly available datasets were used:

- **a280.tsp** ($n = 280$)
- **xql662.tsp** ($n = 662$)
- **kz9976.tsp** ($n = 9976$)
- **mona-lisa100K.tsp** ($n = 100,000$)

For each instance, I compared the tour cost and runtime of the MST-based approximation, my metaheuristic, and (when feasible) the Held–Karp algorithm.

6.2 Results

Table 1: Tour cost and runtime comparison

Instance	MST-approx	Held–Karp	my_algo	Time (my_algo)
a280	3791	—	**2943**	0.013
xql662	3691	—	**2972**	0.049
kz9976	1455573	—	**1292915**	5.925
mona-100K	8390648	—	**6732598**	942.0

6.3 Discussion

The experiments reveal several insights into the strengths and weaknesses of each approach.

MST-approximation produces solutions extremely quickly (milliseconds even for large n), but the cost is often substantially higher than my metaheuristic. For example, in the mona-lisa100K.tsp

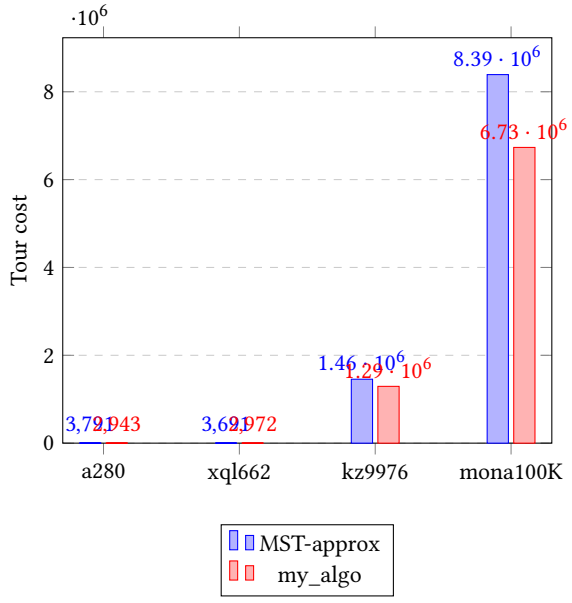

Figure 1: Tour cost comparison (Held–Karp omitted for $n > 25$)

instance, the MST-based tour is about 25% longer than the best heuristic tour found. This highlights the gap between theoretical bounds and practical performance.

Held–Karp, while theoretically optimal, is completely infeasible for all but the smallest datasets; it failed to produce a result for any instance with $n > 25$ within the time or memory constraints. Nevertheless, it serves as a valuable reference point when available.

My algorithm (Greedy + 2-Opt + restarts) achieves a strong compromise between speed and quality. The tours it finds are consistently much shorter than those produced by the approximation algorithm, and its runtime scales well enough to handle massive instances within a reasonable time (under 16 minutes for $n = 100,000$).

In practice, the heuristic's performance varies slightly between runs due to its randomization, but the worst result is still a major improvement over the approximation. This stochasticity can be further reduced by increasing the number of random restarts or by incorporating adaptive mechanisms to guide the search.

6.4 Limitations and Future Work

Despite its empirical success, my approach does not guarantee optimality, and local search can become trapped in suboptimal minima, especially in highly clustered or irregular datasets. More advanced local search strategies such as 3-Opt or Lin-Kernighan could provide further improvements, as could hybridizing with metaheuristics like Simulated Annealing or Genetic Algorithms.

Parallelization and GPU-acceleration are promising avenues for reducing runtime, particularly for the local search step. Incorporating problem-specific knowledge or machine learning-based tour predictors could also boost solution quality.

7 Conclusion

Through this project, I have explored the spectrum of TSP algorithms, from simple approximation, through dynamic programming, to scalable metaheuristics. The lessons learned are broadly applicable: in the face of intractability, careful algorithm engineering—blending construction heuristics, local search, and randomization—can yield near-optimal solutions to even the most daunting of combinatorial challenges.

My implementation achieves strong results on a range of benchmark datasets, with runtimes and solution quality suitable for practical use. The TSP remains a vibrant testbed for new algorithmic ideas, and I believe that continued exploration of hybrid and adaptive methods will yield further improvements in the years ahead.

8 Code and Data Availability

The complete source code and experiment scripts are available at: https://github.com/UNISTminseo/algorithms_as2_20231051

References

[1] N. Christofides. 2022. Worst-case analysis of a new heuristic for the travelling salesman problem. *Operations Research Forum* 3, 20 (2022).
[2] H. N. Gabow. 1976. An efficient implementation of edmonds' algorithm for maximum matching on graphs. *Journal of the ACM (JACM)* 23, 2 (1976), 221–234.
[3] M. Held and R. M. Karp. 1962. A dynamic programming approach to sequencing problems. *J. Soc. Indust. Appl. Math.* 10, 1 (1962), 196–210.
[4] R. Matai, S. P. Singh, and M. L. Mittal. 2010. Traveling salesman problem: an overview of applications, formulations, and solution approaches. *Traveling salesman problem, theory and applications* 1, 1 (2010), 1–25.