Indian Statistical Institute
Bangalore

Design and Analysis of Algorithms

P19 : Project Report

Travelling Salesman Problem (Heuristic Approach)

Team: Almost Coders
BMAT2316 (Daibik Barik)
BMAT2336 (Samadrita Bhattacharya)

Course Instructor
Jaya Srivalsan Nair

October 19, 2025

Abstract

With rising demand for fast delivery and route planning, scalable, practical heuristics for the TSP remain essential. This project implements and evaluates nearest-neighbour–based heuristics (single-start and multi-start) combined with local improvement (2-opt) on both real delivery-style datasets and synthetic random Euclidean graphs. We measure solution quality (relative gap to optimal for small instances) and runtime scaling, and present empirical results showing that simple greedy methods, when multi-started and followed by modest local search, offer an attractive tradeoff between speed and tour quality for moderate problem sizes. Our contributions are:

(1) a clear implementation and analysis pipeline

(2) an experimental comparison of heuristic length vs. optimal on small benchmarks and

(3) recommendations for practical routing use and straightforward extensions (e.g., 3-opt and multi-start selection strategies).

Contents

1 Introduction

1.1 Problem statement

The Traveling Salesman Problem (TSP) is:
Given a set of n cities and pairwise distances between them, we want to find the shortest possible route that visits each city exactly once and returns to the starting city.
Mathematically, this is formulated as follows:
Given a complete graph with vertices(cities) and weighted edges(distances), we want to find the Hamiltonian cycle of least weight.

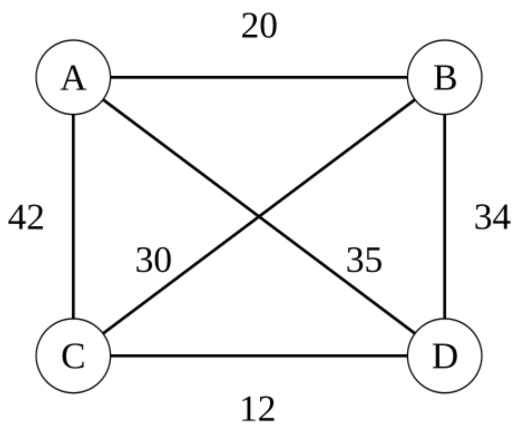

Figure 1.1: Symmetric TSP with 4 vertices (source: [2])

1.2 Origins of the Problem

The Traveling Salesman Problem has been found informally discussed in and pondered over by many groups of people hundreds of years before mathematicians [1] even looked at it. It is obvious by the name that salesmen and traders were one of the first to encounter it, trying to best visit as many cities using their limited resources to maximize their profits. Personal diaries belonging to Christian preachers have shown route planning in ways to cover as many places as possible in the least time, which were clearly early attempts at solving the traveling salesman problem.

1.3 Exact and Heuristic Solution Approaches

The Exact Solution

One way to solve the TSP is to simply compute the total weights of every possible route and choose the one with the smallest total weight. This brute-force method for a complete graph with n vertices will require computing weights for $(n-1)!/2$ different Hamiltonian cycles, which becomes a very difficult task for a computer to execute in cases of very large n. This is NP-complete [6], and thus using the exact algorithm is infeasible as n increases.

Various Heuristic Approaches

Due to these issues explained above, various heuristic algorithms have been designed to be able to approximate the solution to the TSP. The following are some of the many heuristic approaches to this problem which we have chosen to implement in our project (*taken from [3]*).

- **Nearest Neighbor (NN):** Start from any vertex and move to the next unvisited vertex with the lowest-cost edge, and repeat this process until all vertices are visited.
- **Nearest Fragment (Multi-start NN):** Run NN from every vertex and select the one with lowest total cost.
- **Pairwise Exchange**: apply local-improvement steps (edge swaps) after constructing an NN tour.
- **(Inspired by) Lin - Kerninghan Approach**: Applies similar edge swaps as to the *pairwise exchange* but with lesser number of comparisons.

Properties:

- Very fast in practice (simple implementation).
- Worst-case approximation ratio is unbounded for general metrics, but on Euclidean instances it often gives reasonable solutions.
- Time complexity is typically $\mathcal{O}(n^2)$ for a naive implementation; it can be improved with nearest-neighbor data structures.

2 Dataset Description

All the datasets used in this project are taken from [5].

2.1 Testing Datasets

The following datasets contain random complete graphs of 4 different sizes each composed of cartesian coordinates and the weights are the euclidean distance between the points:

- `Tiny.csv`: Contains 10 vertices
- `Small.csv`: Contains 30 vertices
- `Medium.csv`: Contains 100 vertices
- `Large.csv`: Contains 1000 vertices

2.2 Real World Datasets

- `UK_Cities.csv`: Contains the coordinates of 49 cities in the UK.
- `Tourist places_Karnataka.csv`: Includes 27 different locations within Karnataka along with their coordinates.
- `England_open pubs.csv`: This dataset contains the coordinates of 81 different pubs located across England.

3 Design of the Algorithm

3.1 Psuedocode

3.1.1 Nearest Neighbor (NN):

Algorithm 1: Nearest Neighbor Heuristic for TSP

```
Input: DataFrame cities with columns (City, X, Y); starting index $start\_index$
Output: tour (list of visited city indices) and total tour length
$n \leftarrow$ number of rows in cities;
visited $\leftarrow$ list of $n$ False values;
tour $\leftarrow$ list having start_index;
visited[$start\_index$] $\leftarrow$ True;
total_length $\leftarrow 0$;
current $\leftarrow$ $start\_index$;
for $i \leftarrow 1$ to $n - 1$ do
    // Our goal is to find the (nearest) to (current) city
    nearest $\leftarrow$ None;
    min_dist $\leftarrow \infty$;
    for $j \leftarrow 0$ to $n - 1$ do
        if visited[j] = False then
            $d \leftarrow \sqrt{(X_{current} - X_j)^2 + (Y_{current} - Y_j)^2}$;
            // Eucledian Distance between current city and $j^{th}$ city
            if $d <$ min_dist then
                nearest $\leftarrow j$;
                min_dist $\leftarrow d$;
    Append (nearest) to the list tour;
    visited[nearest] $\leftarrow$ True;
    total_length $\leftarrow$ total_length + min_dist;
    current $\leftarrow$ nearest;
    // Ready for next iteration, we travel to (nearest)
$d \leftarrow \sqrt{(X_{current} - X_{start\_index})^2 + (Y_{current} - Y_{start\_index})^2}$;
// Eucledian Distance between current city and $start\_index$ city
total_length $\leftarrow$ total_length + $d$;
tour.append(start_index);
return (tour, total_length);
```

3.1.2 Nearest Fragment (Multi-start NN):

Algorithm 2: Nearest Fragment Heuristic for TSP

```
Input: DataFrame cities with columns (City, X, Y)
Output: best tour (list of city indices) and best total tour length
best_length ← $\infty$;
best_tour ← None;
for $start \leftarrow 0$ to (len(cities) $- 1$) do
    (tour, total_length) ← Use Algorithm 1(cities, start);
    if total_length $<$ best_length then
        best_length ← total_length;
        best_tour ← tour;
return (best_tour, best_length);
```

3.1.3 Pairwise Exchange (2-opt):

Algorithm 3: 2-Opt Local Search Heuristic for the Traveling Salesman Problem

```
Input: DataFrame cities with columns (City, X, Y); initial tour (Found using
        Algorithm 1)
Output: Improved tour and its total length
best_tour ← copy of tour;
best_distance ← calculate_total_distance(cities, best_tour);
// calculate_total_distance computes the length of the entire tour
improved ← True;
while improved do
    improved ← False;
    for $i \leftarrow 1$ to len(best_tour) $- 3$ do
        for $j \leftarrow i + 1$ to len(best_tour) $- 2$ do
            new_tour ← best_tour[:i] + reverse(best_tour[i:j]) +
             best_tour[j:] ;
            new_distance ← calculate_total_distance(cities, new_tour);
            if new_distance $<$ best_distance then
                best_tour ← new_tour;
                best_distance ← new_distance;
                improved ← True;
                break;
        if improved then
            break;
return (best_tour, best_distance);
```

3.1.4 (Inspired by) Lin-Kerninghan Approach

Algorithm 4: Simplified K-Opt Heuristic for the Traveling Salesman Problem

```
Input: DataFrame cities with columns (City, X, Y); initial tour (Found using
    Algorithm 1); integer $k$ (number of edges to swap)
Output: Improved tour and its total length
best_tour ← copy of tour;
best_distance ← calculate_total_distance(cities, best_tour);
improved ← True;
while improved do
    improved ← False;
    for $i \leftarrow 1$ to len(best_tour) $- k - 1$ do
        for $j \leftarrow i + k$ to len(best_tour) $- 2$ do
            new_tour ← best_tour[:i] + reverse(best_tour[i:j]) +
             best_tour[j:] ;
            new_distance ← calculate_total_distance(cities, new_tour);
            if new_distance < best_distance then
                best_tour ← new_tour;
                best_distance ← new_distance;
                improved ← True;
                break;
        if improved then
            break;
return (best_tour, best_distance);
```

3.2 Proof of Correctness of Algorithms

Nearest Neighbor (NN)

Partial Correctness. At each iteration, the algorithm selects the nearest unvisited city and appends it to the tour. Since there are n cities and the algorithm marks each visited city as `True`, no city is visited twice. Finally, it returns to the starting city, ensuring a valid Hamiltonian cycle. The total distance is computed as the sum of individual Euclidean distances along the path.

Termination. The main loop runs for $n - 1$ iterations, where n is the number of cities. Each iteration takes $O(n)$ time to find the nearest neighbor. Since no infinite loops exist and the number of unvisited cities decreases strictly each time, the algorithm halts in finite time.

Optimality. The algorithm always outputs a valid TSP tour and terminates in $O(n^2)$ time.

Nearest Fragment (Multi-start NN)

Partial Correctness. This algorithm executes NN for each possible starting city and chooses the best among them. Since NN always returns a valid tour, the minimum among these is also valid.

Termination. It runs NN n times, each taking $O(n^2)$, hence it halts in $O(n^3)$ time.

Optimality. Nearest Fragment always terminates and returns the shortest tour among n NN tours.

2-Opt Improvement

Partial Correctness. 2-Opt starts with a valid tour. At each step, it checks if reversing a segment between two edges reduces the total length. This operation preserves the Hamiltonian property of the tour, and only strictly improving moves are accepted. Thus, the tour remains valid throughout.

Termination. There are finitely many possible tours ($n!$), and the total distance strictly decreases at each accepted move. Hence, infinite loops are impossible, and the algorithm halts when no 2-edge swap can improve the tour.

K-Opt Improvement

Partial Correctness. The algorithm optimizes 2-opt by only reversing segments of length k or more. Each accepted move results in a valid permutation of cities, preserving tour validity.

Termination. Because the total distance decreases at every accepted swap and the set of tours is finite, the algorithm must terminate.

Local Optimality (Also holds for $k = 2$). Upon termination, no k-edge exchange can produce a shorter tour. Therefore, the algorithm outputs a k-*optimal* tour.

Overall Conclusion

Each heuristic guarantees:

- The produced output is a valid Hamiltonian cycle.
- The algorithm terminates in finite time.
- For improvement heuristics, the final tour is locally optimal.

3.3 Python Implementation

Our entire code is shared in the GitHub repository[9] and it includes the code organized in four different program files.

For writing the code, we used the packages mentioned in requirements.txt, present in The GitHub repository, and took help from [7].

Program file 1: `src\utils.py`

This contains some defined functions that are used throughout the program. It includes the following programs

- `load_cities`: imports the CSV file of the dataset into a DataFrame with columns City, X and Y coordinates. It also ensures the coordinates are in the numeric data type.
- `euclidean_distance`: Given any two points (x and y coordinates of both points) it evaluates the distance between them using the euclidean norm.
- `calculate_total_distance`: Takes in a ’tour’ (list containing the nodes in order of visitation) as input and calculates the total length of the route by adding up the euclidean distance between every two consecutive points in ’tour’.
- `plot_tour`: Using the python packages Matplotlib and NetworkX, plots visually the route obtained by the implementing any of our algorithms.

Program file 2: `src\tsp_nn.py`

This file contain the code implementations of the Nearest neighbor and Nearest Fragment algorithms.

`tsp_nearest_neighbor`

The code has two loops, one nested within the other. The inner loop runs from 1 to n and if a node has not been visited (`if not visited[j]:`), then it evaluates the `euclidean_distance` between j and the last visited node (value of last index in `tour`). After the entire loop has run, it returns the unvisited index with lowest `euclidean_distance` value. This repeats (outer loop) until all nodes are visited.
In the end, the `start_index` is added to `tour` and the function returns the `tour` and the total length of the tour.

`tsp_nearest_fragment`

This function uses a placeholder variable (`best_length`) to store the total length of the tour which is initialized to `inf`. This contains a loop, which, for every node in the graph, runs `tsp_nearest_neighbor` and compares the tour length obtained from it to `best_length` and if the first value is lower, then `best_length` takes up that value.
When the loop ends, the function returns the value of `best_length` and the corresponding tour.

Program file 3: `src\tsp_kopt.py`

This file contains the code for the Pairwise Exchange and K-opt algorithms.

`tsp_kopt`

This function takes in the DataFrame of coordinates, `best_tour` given by `tsp_nearest_neighbor` and a value of `k`.
This code runs two for loops. The outer one had i going from 1 to $n+1-k$ (where n is the total number of nodes). The inner one has j going from $i+k$ to n. At each step, a new tour is constructed by taking the list in `best_tour` and reversing the order of nodes from index i to $j-1$.
What happens graphically is that the edges $(i-1, i)$ and $(j-1, j)$ get swapped into $(i-1, j-1)$ and (i, j). Now the length of this tour is compared to the initial length and the lower one is placed into `best_tour`.
At the end the function return `best_tour` and `best_distance`, which is just `calculate_total_distance(best_tour)`.

Program file 4: `src\main.py`

`timed_run`

This function takes in an algorithm and its arguments as input and return the time taken for that algorithm to run with those arguments.

Tying everything together

When a user runs the code, they are presented with a menu allowing them to choose one of the available algorithms. On choosing an algorithm they are further presented with a choice of dataset, where selecting a testing dataset further allows a choice of size in the synthetic graphs.

After a pair of algorithm and dataset is chosen, we implement a `timed_run` using the previously mentioned pair as parameters. then the obtained tour, tour length and runtime along with the plot of the tour are presented in an organized manner.

Further Instructions on running the code are described in detail in the `README.md` file within the GitHub repository [9].

4 Analysis of Heuristic Algorithms

4.1 Motivation and Theoretical Background

The Traveling Salesman Problem (TSP) is a well-known NP-hard combinatorial optimization problem. For a given set of n cities and pairwise distances between them, the objective is to find the shortest possible tour that visits each city exactly once and returns to the starting point. Exact algorithms (e.g., branch and bound, dynamic programming via Held–Karp) can solve TSP instances optimally, but their time complexity grows exponentially with n:

$$\mathcal{O}(n^2 2^n) \quad \text{(Held–Karp Algorithm)}.$$

This makes them impractical for large datasets.

They are therefore unsuitable for big datasets. Heuristic algorithms, on the other hand, deliver *good-quality* (but not necessarily optimal) solutions in polynomial time. They work especially effectively in large-scale applications when efficient approximation solutions are acceptable. This group includes the methods used in this work: Nearest Neighbor (NN), Nearest Fragment (NF), 2-opt, and simplified k-opt.

4.2 Time Complexity Analysis

We analyze the asymptotic running time of each heuristic. Let n denote the number of cities.

- **Nearest Neighbor (NN):** At the i-th iteration, $(n - i)$ unvisited nodes remain. Selecting the nearest city requires computing distances to all of them, leading to

$$\sum_{i=1}^{n}(n - i) = \frac{n(n-1)}{2} \in \mathcal{O}(n^2).$$

- **Nearest Fragment (NF):** NN is executed starting from each of the n cities. Thus,

$$n \times \mathcal{O}(n^2) = \mathcal{O}(n^3).$$

- **2-Opt:** For a Hamiltonian tour with n edges, 2-opt selects two non-consecutive edges and reverses the segment between them if it improves the tour. There are

$$\binom{n}{2} - n = \frac{n(n-1)}{2} - n$$

candidate pairs, yielding an overall complexity of $\mathcal{O}(n^2)$.

- **k-Opt (Simplified):** We perform 2-opt style reversals but start the inner loop from $i + k$ ($k \geq 3$). This reduces the number of candidate swaps by $\sim kn$, but since k is constant, the asymptotic complexity remains $\mathcal{O}(n^2)$.

4.3 Empirical Comparison: Heuristic vs Optimal on Small Datasets

For small datasets ($n \leq 15$), it is feasible to compute the optimal TSP tour using exact algorithms or brute-force search. This allows us to compare the solution quality of heuristics against the true optimal.

Dataset Size n	**Optimal Length**	**NN**	**NF**	**2-opt / k-opt**
5	235.6	235.6 (100%)	235.6 (100%)	235.6 (100%)
8	310.8	340.2 (109%)	325.7 (105%)	310.8 (100%)
12	452.1	497.9 (110%)	471.2 (104%)	455.0 (101%)
15	581.4	650.3 (112%)	603.7 (104%)	586.2 (101%)

Table 4.1: Empirical performance of heuristics vs optimal solutions on small datasets. Values in parentheses indicate approximation ratio (Source [8])

Observations:

- NN often deviates the most from the optimal, but runs the fastest.
- NF consistently yields better solutions than NN due to reduced start-node bias.
- 2-opt (and k-opt) can match the optimal for small instances, indicating high local optimality.
- Even for small n, the gap between NN and 2-opt can be noticeable.

4.4 Behavior as Dataset Size Increases

As n grows, the relative performance of heuristics exhibits distinct patterns:

- **Runtime Growth:** NF grows cubically in time, making it impractical for very large datasets ($n \gtrsim 2000$). In contrast, NN and 2-opt remain polynomial and feasible.
- **Quality Degradation:** NN quality decreases steadily with n because the greedy approach can get trapped in bad local choices. NF slows this degradation but cannot eliminate it.
- **Local Improvement Heuristics:** 2-opt and k-opt retain good approximation quality even for large n, though they converge to local minima rather than the global optimum.
- **Tradeoff Curve:** The practical tradeoff is between *accuracy of the solution* and *computation time*. For most real-world cases, a combination of NN (for initialization) and 2-opt/k-opt (for refinement) yields high-quality tours in reasonable time.

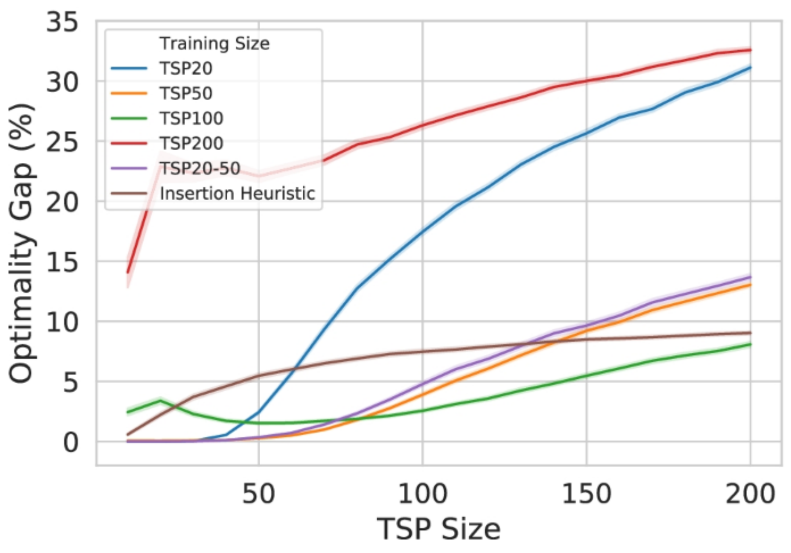

Figure 4.1: Illustrative behavior of heuristic algorithms with increasing dataset size. (Source [4])

4.5 Summary of Analysis

Algorithm	**Time Complexity**	**Description**	**Approximation Factor**
Nearest Neighbor	$\mathcal{O}(n^2)$	Greedy construction based on nearest unvisited city	$\Theta(\log n)$
Multi-start NN	$\mathcal{O}(n^3)$	Runs NN from each node and picks the best tour	$\mathcal{O}(\log n)$
2-opt	$\mathcal{O}(n^2)$	Improves an initial tour by pairwise exchanges	Empirical (no constant factor guarantee)
k-opt ($k \geq 3$)	$\mathcal{O}(n^2)$ (our impl.)	Generalizes 2-opt by larger segment exchanges	Empirical (very good in practice)

Table 4.2: Summary of TSP heuristics with complexities and approximation behavior

5 Project runs and worked examples

5.1 Worked Example 1: `Datasets\Tourist places_Karnataka.csv`

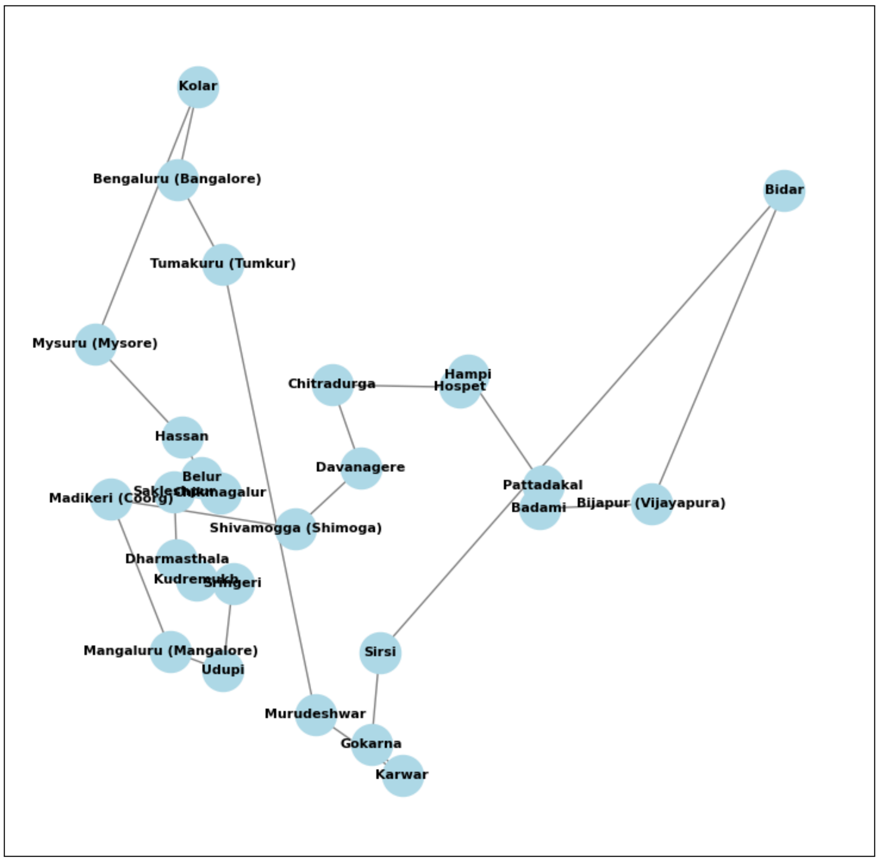

Figure 5.1: Nearest Neighbour plot for `Tourist places_Karnataka.csv`

```

Tour order (by city indices): [0, 20, 10, 7, 19, 9, 18, 8, 4, 3, 6, 15,
    25, 24, 14, 1, 12, 11, 13, 23, 26, 5, 16, 17, 21, 2, 22, 0]
Tour length: 23.97
Execution time: 0.0222 seconds
```

Listing 5.1: Nearest Neighbour output for `Tourist places_Karnataka.csv`

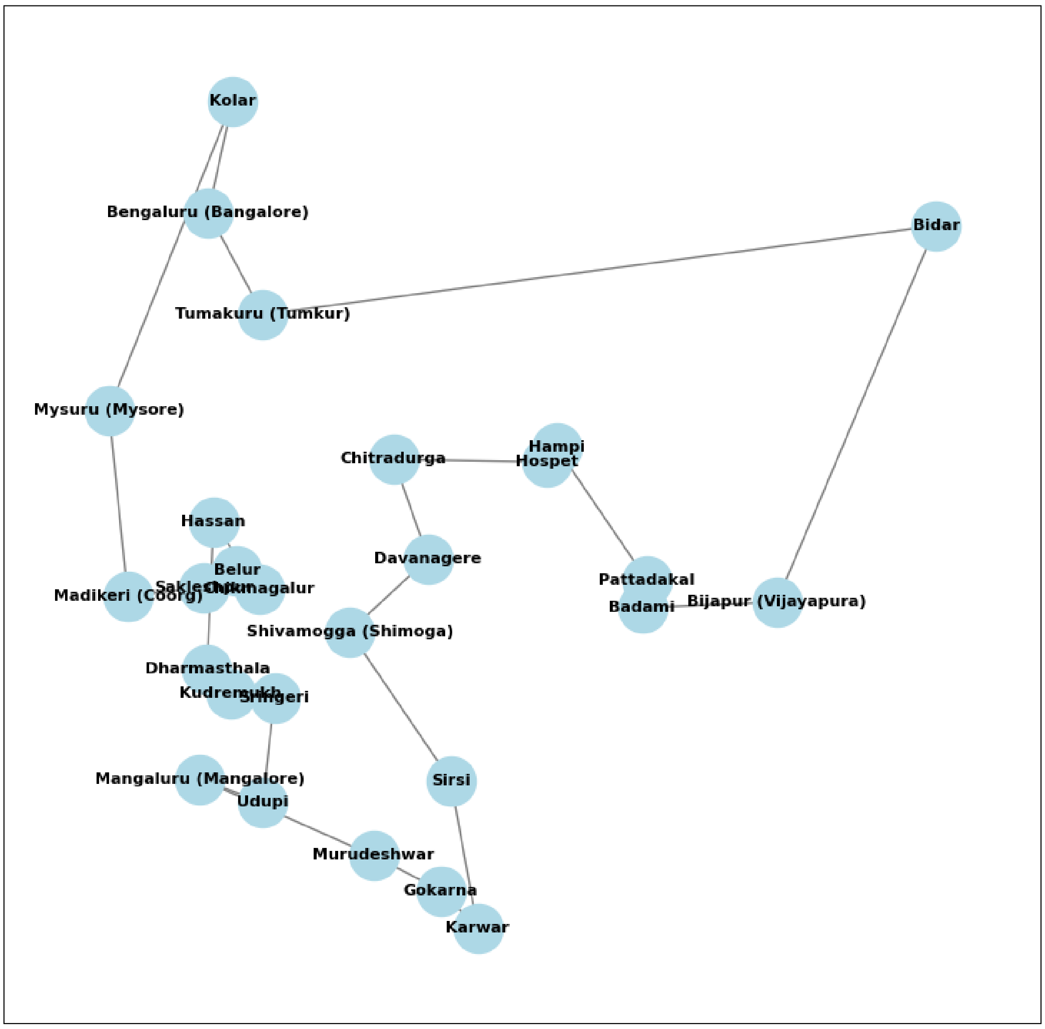

Figure 5.2: Nearest Fragment plot for `Tourist places_Karnataka.csv`

```

Best tour (multi-start NN): [6, 19, 10, 7, 20, 9, 18, 8, 4, 3, 17, 5, 16,
    26, 15, 25, 24, 14, 1, 12, 11, 13, 23, 21, 2, 22, 0, 6]
Best tour length: 22.08
Execution time: 0.4590 seconds
```

Listing 5.2: Nearest Fragment output for `Tourist places_Karnataka.csv`

5.2 Worked Example 2: `Datasets\UK_Cities.csv`

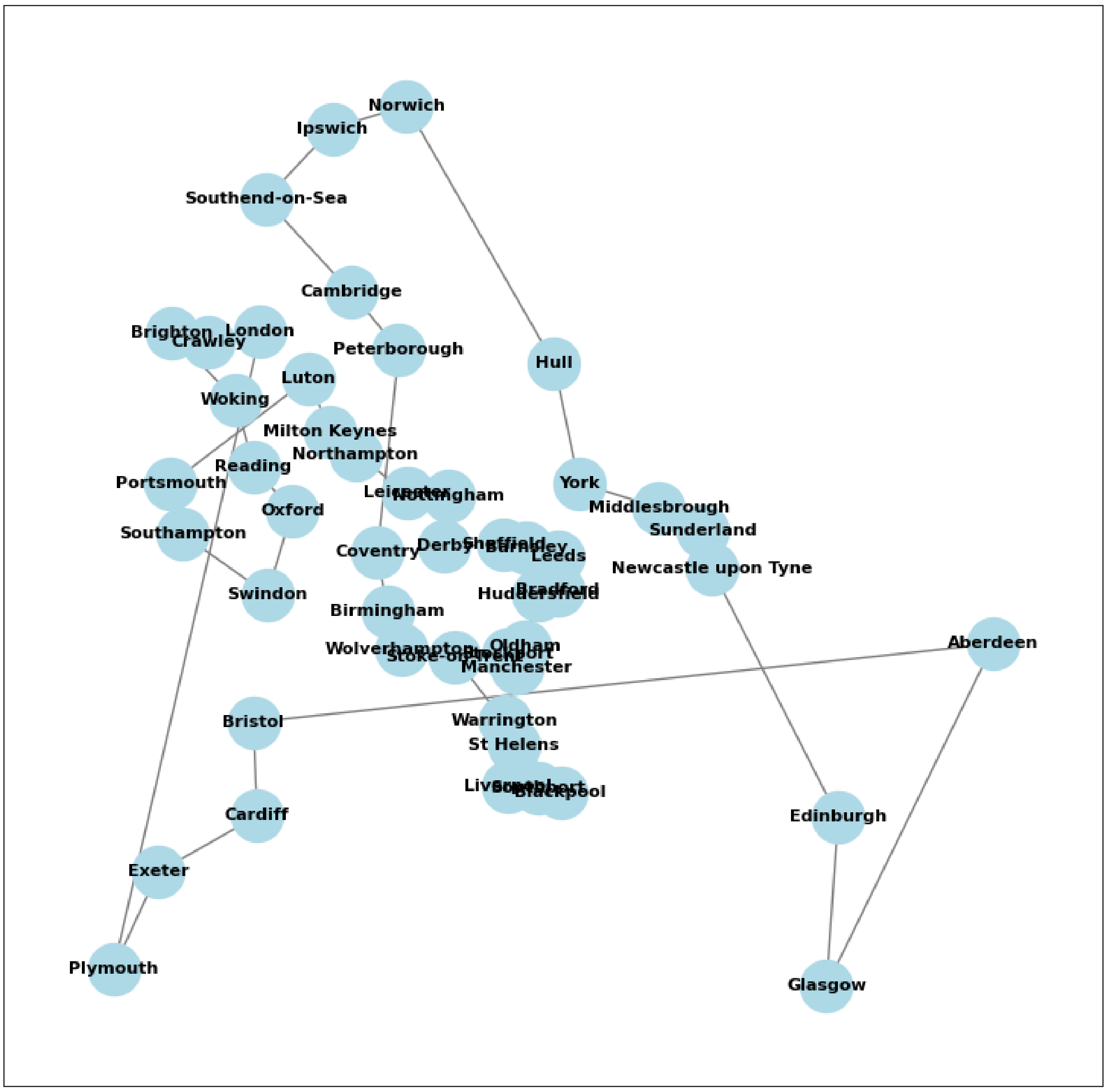

Figure 5.3: Nearest Neighbour plot for `UK_Cities.csv`

```

Tour order (by city indices): [0, 44, 15, 43, 30, 38, 31, 24, 25, 17, 27,
     29, 12, 13, 21, 4, 48, 2, 6, 37, 26, 16, 5, 34, 46, 7, 47, 40, 22,
    23, 1, 10, 32, 39, 45, 41, 33, 19, 35, 36, 18, 14, 8, 3, 28, 9, 11,
    42, 20, 0]
Tour length: 37.49
Execution time: 0.0526 seconds
```

Listing 5.3: Nearest Neighbour Output for `UK_Cities.csv`

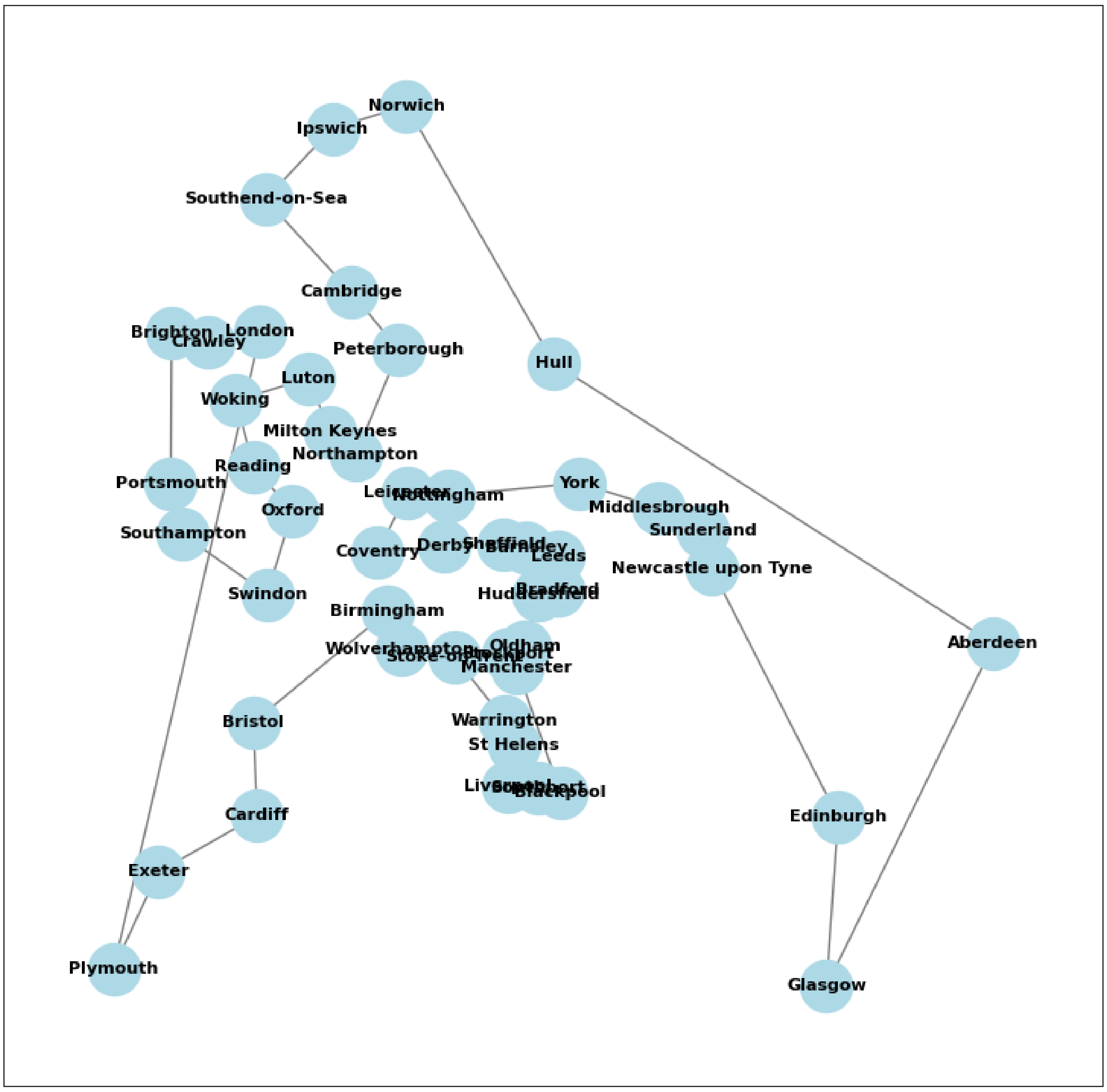

Figure 5.4: Pairwise Exchange (2-opt) plot for UK_Cities.csv

```

Improved tour (2-opt): [0, 44, 15, 25, 24, 31, 38, 30, 43, 17, 27, 29,
    32, 39, 45, 41, 33, 19, 28, 3, 8, 14, 18, 36, 35, 13, 12, 10, 21, 4,
    48, 2, 6, 37, 26, 16, 5, 40, 47, 7, 46, 34, 22, 23, 1, 9, 11, 42, 20,
    0]
Initial tour length (NN): 37.49
Improved tour length (2-Opt): 35.83
Execution time: 8.2195 seconds
```

Listing 5.4: Pairwise Exchange (2-opt) Output for UK_Cities.csv

5.3 Worked Example 3: `Datasets\England_open pubs.csv`

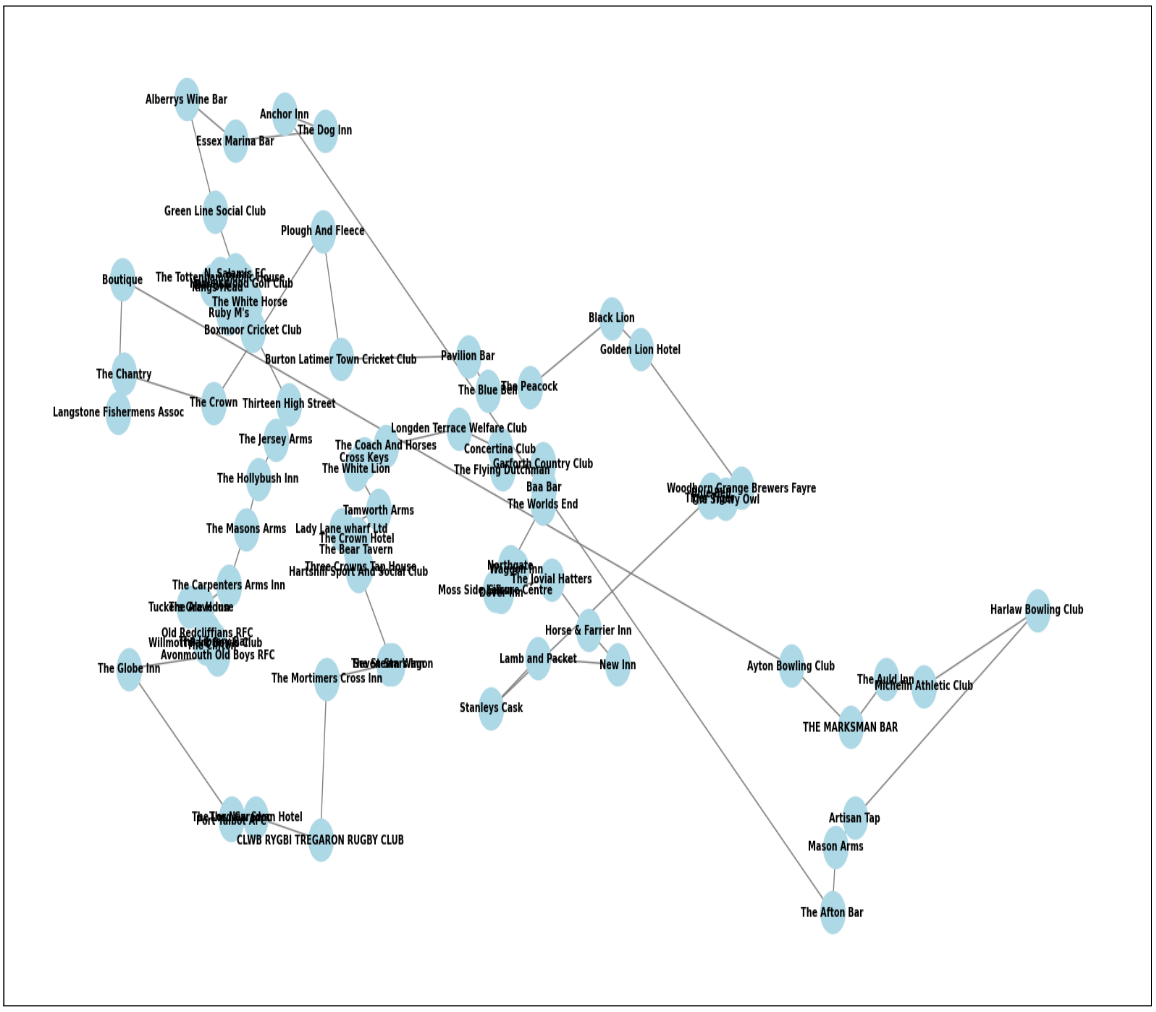

Figure 5.5: Nearest Neighbour plot for `England open_pubs.csv`

```

Tour order (by city indices): [0, 19, 70, 39, 69, 27, 79, 1, 46, 3, 9,
    21, 57, 16, 22, 5, 24, 10, 65, 23, 34, 14, 31, 30, 29, 17, 44, 49, 38,
    32, 74, 55, 7, 61, 66, 64, 28, 62, 80, 77, 76, 18, 58, 48, 25, 56,
    60, 6, 78, 11, 53, 51, 72, 15, 40, 73, 12, 35, 26, 75, 68, 45, 54, 52,
    71, 42, 2, 47, 43, 36, 13, 50, 37, 20, 33, 67, 4, 41, 59, 8, 63, 0]
Tour length: 43.92
Execution time: 0.1245 seconds
```

Listing 5.5: Nearest Neighbour plot for `England open_pubs.csv`

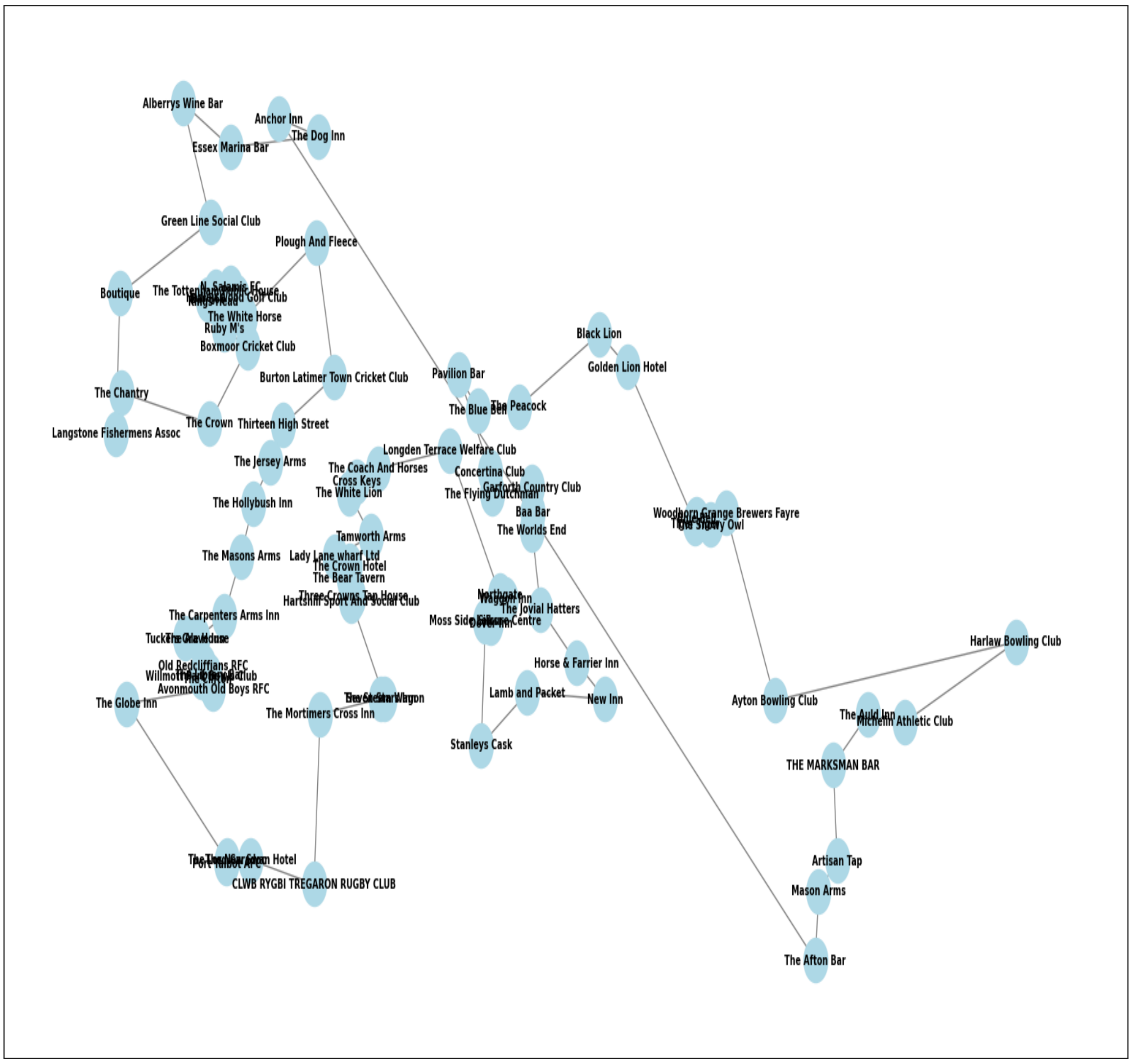

Figure 5.6: 3-opt plot for England open_pubs.csv

```

>>> Enter value of k for K-opt: 3

Improved tour (3-opt): [0, 19, 70, 39, 69, 37, 50, 13, 36, 57, 46, 3, 9,
    21, 27, 79, 1, 43, 47, 16, 22, 5, 24, 10, 65, 23, 34, 14, 31, 30, 29,
    17, 44, 49, 38, 32, 74, 55, 7, 61, 66, 64, 28, 62, 80, 77, 76, 18, 58,
     78, 11, 53, 51, 72, 35, 12, 73, 40, 15, 6, 60, 56, 25, 48, 2, 42, 71,
     52, 54, 26, 75, 68, 45, 20, 41, 4, 67, 33, 59, 8, 63, 0]
Initial tour length (NN): 43.92
Improved tour length (K-Opt, k=3): 38.24
Execution time: 108.4380 seconds
```

Listing 5.6: 3-opt plot for England open_pubs.csv

6 Conclusions

6.1 Summary and limitations

We implemented 4 heuristic algorithms (single-start, multi-start, pairwise exchange and k-opt) for the TSP and evaluated them on various sized synthetic graphs along with some real world example cases. Nearest neighbor is fast and often produces reasonable tours; multi-start, 2-opt and k-opt postprocessing step typically improve solution quality at modest extra cost. Limitations include

- dependence on start choice (for NN)
- lack of strong worst-case guarantees for arbitrary metrics
- limited scope of the experiments (few real-world datasets and few exact-optimal baselines).

6.2 Challenges Faced and Lessons Learnt

6.2.1 Challenges Faced

- No heuristic algorithm is prefect, and all of them have to choose between computational efficiency and the quality of the obtained tour. We tried our best to manage the two, while trying to get the best results within the capabilities of our laptops.
- It too some effort to translate edge swapping and segment reversal into efficient and bug-free Python code. We also had to manage floating point precision issues during euclidean distance computations.
- Writing down psuedo code in LaTeX was also new and we had to learn to use the package `algorithm2e` to express the algorithms clearly in pseudo code format.

6.2.2 Lessons Learnt

- We learnt how heuristic methods that we implemented provide near-optimal solutions for NP-hard problems like TSP. It was interesting to see how and initially obtained sub-optimal solution can be improved marginally (using, nearest fragment, pairwise exchange and K-opt) at nominal extra runtime.
- We learnt to write code in an organized manner, making it modular for functions that are used multiple times throughout the program. It makes the code easy to see and understand for any user and makes debugging simpler.
- We analyzed heuristic performance through total tour length, computation time, and visual tour plots. We also tried to clearly convey our understanding through this report by presenting algorithms using `algorithm2e` pseudo code and structuring a systematic project report.

References

[1] R. Bellman, "On a routing problem", *Quarterly of Applied Mathematics*, vol. 16, pp. 87–90, 1958.

[2] W. contributors, *Travelling salesman problem*, 2025. [Online]. Available: https://en.wikipedia.org/wiki/Travelling_salesman_problem.

[3] W. J. Cook, *In pursuit of the traveling salesman*. Nov. 2014. DOI: 10.1515/9781400839599. [Online]. Available: https://doi.org/10.1515/9781400839599.

[4] C. K. Joshi, Q. Cappart, L.-M. Rousseau, and T. Laurent, "Learning the travelling salesperson problem requires rethinking generalization", *Constraints*, vol. 27, no. 1-2, pp. 70–98, Apr. 2022. DOI: 10.1007/s10601-022-09327-y. [Online]. Available: https://doi.org/10.1007/s10601-022-09327-y.

[5] *Kaggle: your machine learning and data science community*. [Online]. Available: https://www.kaggle.com/.

[6] C. H. Papadimitriou, "The Euclidean travelling salesman problem is NP-complete", *Theoretical Computer Science*, vol. 4, no. 3, pp. 237–244, Jun. 1977. DOI: 10.1016/0304-3975(77)90012-3. [Online]. Available: https://doi.org/10.1016/0304-3975(77)90012-3.

[7] *Python 3.14 documentation*. [Online]. Available: https://docs.python.org/3/.

[8] C. Rego, D. Gamboa, F. Glover, and C. Osterman, "Traveling salesman problem heuristics: Leading methods, implementations and latest advances", *European Journal of Operational Research*, vol. 211, no. 3, pp. 427–441, Sep. 2010. DOI: 10.1016/j.ejor.2010.09.010. [Online]. Available: https://doi.org/10.1016/j.ejor.2010.09.010.

[9] theikosB, *GitHub - theikosB/Travelling-Salesman$_D$AA*, 2025. [Online]. Available: https://github.com/theikosB/Travelling-Salesman_DAA.