CSE6140 Final Project: Travelling Salesman

Aditya Raghavan, Srishti Kalepu, Shraddha Bharadwaj

1st December 2025

1 Introduction

The *Traveling Salesman Problem (TSP)* is an NP-Complete Problem in which given a list of cities and distances between each pair of them we find the least cost route that visits each city exactly once and return to the origin city.

2 Implementation Methodologies

2.1 Brute Force Solution

In the brute force solution, we generated every permutation of all n cities and computed the distance traveled in each permutation until the time cutoff was reached. When the cutoff was triggered, the algorithm returned the lowest cost and the corresponding path found up to that moment. The main implementation details are summarized below.

Data Parsing & Representation.

- The algorithm begins by parsing the `NODE_COORD_SECTION` of the input file.
- The raw text lines are converted into a list of node dictionaries.
- Each node dictionary stores an ID along with its (x, y) coordinates.

Distance Calculation.

- A helper function computes the Euclidean distance between any two nodes.
- The distance is calculated using differences in their x and y coordinates.
- Distances are rounded to the nearest integer following standard TSP conventions.

Initialization & Permutation Strategy.

- The algorithm first computes the tour cost of the nodes in their original input order as a baseline.
- The first node is fixed as the starting point to reduce redundant permutations.
- All permutations of the remaining nodes are generated to explore the full search space.

Path Evaluation & Pruning.

- The algorithm iterates through each permutation sequentially.
- During partial path evaluation, if the cumulative cost exceeds the best cost seen so far, the path is discarded early.
- If a complete permutation yields a lower total cost, the `best_cost` and `best_path` values are updated.

Time Management.

- The system clock is checked every 1,000 permutations to control execution time.
- If the elapsed time exceeds the specified cutoff, the search terminates immediately.
- The best solution found up to that point is returned.

Output Generation.

- The final best path is formatted as a comma-separated list of node IDs.
- The total solution cost and an indicator of whether a full tour was completed are included in the output.
- The same information is written to the appropriate `.sol` file.

2.2 Approximate Algorithm

Pseudocode.

- Compute Euclidean distances between every pair of cities and use them as edge weights.
- Build a Minimum Spanning Tree (MST) using Prim's algorithm:
  - Start from city 0.
  - For the current node, connect the nearest city not yet in the tree.
  - Repeat until all cities are included in the MST.
  - Creates a complete graph.
- Convert the MST into a graph by constructing an adjacency list from its edges.
- Perform a DFS preorder walk on the MST:
  - Start from city 0.
  - Visit each unvisited neighbor recursively.
  - The DFS order forms the tour sequence.
- Form the Hamiltonian cycle by returning to the starting city after completing the DFS order
  - Use the DFS order as a tour and return to the starting city
  - Compute the tour cost
  - Add up distances between consecutive cities in the DFS order, including the traversal between last and first vertex

- Compute the tour cost
  - Add distances between consecutive cities in the DFS order.
  - Include the distance between the last and first city.
- Output the solution:
  - Line 1: total tour cost and solution quality.
  - Line 2: ordered list of city IDs.
  - Write the correct output into a `.sol` file in the correct directory.
- Return the runtime of the algorithm.

Correctness and Quality.

- Any TSP tour is a cycle that visits all cities exactly once.
- Removing any edge from such a cycle yields a spanning tree, so the MST cost is always at most the optimal tour cost.
- The algorithm constructs an MST and performs a DFS preorder walk, which visits each city exactly once and therefore always yields a full Hamiltonian tour.
- Since the MST cost is $\leq$ OPT, the DFS walk uses each MST edge at most twice, giving a tour length $\leq 2 \times$ MST $\leq 2 \times$ OPT.
- By the triangle inequality, shortcutting repeated vertices never increases the total cost as the tour never becomes longer by skipping nodes.
- Therefore, the algorithm always produces a valid **full tour**, and its **solution quality is theoretically guaranteed**.

2.3 Local Search

Pseudocode.

- Convert the input coordinates into two data structures:
  - **Point:** defined as (x, y).
  - **Tour:** an ordered list of Points.
- Construct an initial tour using the nearest neighbor algorithm:
  - Maintain a list of unvisited nodes.
  - Compute the Euclidean distance between the last node in the current tour and all unvisited neighbors.
  - Add the nearest neighbor to the tour.
  - Repeat until all nodes are visited, forming a complete cycle.
- Improve the initial tour using the 2-opt algorithm:
  - Repeatedly reverse segments of the tour to reduce total edge cost.

  - The algorithm selects two non-adjacent edges and swaps them by reversing the segment between them. Then checks if there was any improvement.
  - Continue applying 2-opt until no further improvement is possible.

- Apply simulated annealing using the 2-opt result as the starting tour:
  - Use a random seed to introduce stochasticity.
  - Determine the initial temperature as a fraction of the average edge length.
  - If a proposed modification reduces the tour length, accept it.
  - If it increases the cost, accept based on the Boltzmann probability to escape local minima.

- Run simulated annealing 10 times with different seeds:
  - Each run introduces independent randomness.
  - The final runtime is computed as the average of the ten runs.

3 Results Summary

Now, we summarize the results of our analysis in the following Tables . Tables 1,2, and 3 compare the performance of three algorithms:

Data (City)	**Cutoff = 30 secs**			**Cutoff = 60 secs**		
	Time	**Quality**	**Full Tour**	**Time**	**Quality**	**Full Tour**
Atlanta	30	3630534.0	Yes	60	3630534.0	Yes
Berlin	30	19448.0	Yes	60	19438.0	Yes
Boston	30	2234116.0	Yes	60	2227323.0	Yes
Champaign	30	218074.0	Yes	60	216391.0	Yes
Cincinnati	0.55	277952.0	Yes	0.55	277952.0	Yes
Denver	30	563620.0	Yes	60	547974.0	Yes
NYC	30	7239945.0	Yes	60	7210976.0	Yes
Philadelphia	30	3710782.0	Yes	60	3624919.0	Yes
Roanoke	30	6857992.0	Yes	60	6849948.0	Yes
SanFrancisco	30	5604793.0	Yes	60	5600573.0	Yes
Toronto	30	9219353.0	Yes	60	9184579.0	Yes
UKansasState	0.41	62962.0	Yes	0.41	62962.0	Yes
UMissouri	30	663479.0	Yes	60	662935.0	Yes

Table 1: Performance of Brute Force Algorithm for Different Cities for Cutoff Times of 30s and 60s

3.1 Runtime

- **Brute Force**:
  - **Complexity**: $O(n!)$ – Since it explores all permutations of cities.

Data (City)	**Cutoff = 300 secs**			**Cutoff = 600 secs**		
	Time	**Quality**	**Full Tour**	**Time**	**Quality**	**Full Tour**
Atlanta	300	3353390.0	Yes	600	3321711.0	Yes
Berlin	300	19249.0	Yes	600	19233.0	Yes
Boston	300	2220896.0	Yes	600	2210534.0	Yes
Champaign	300	209943.0	Yes	600	209943.0	Yes
Cincinnati	0.55	277952.0	Yes	0.55	277952.0	Yes
Denver	300	546699.0	Yes	600	540975.0	Yes
NYC	300	7166313.0	Yes	600	7166313.0	Yes
Philadelphia	300	3624919.0	Yes	600	3624919.0	Yes
Roanoke	300	6847051.0	Yes	600	6767159.0	Yes
SanFrancisco	300	5591805.0	Yes	600	5591805.0	Yes
Toronto	300	9192020.0	Yes	600	9116969.0	Yes
UKansasState	0.41	62962.0	Yes	0.41	62962.0	Yes
UMissouri	300	662935.0	Yes	600	660277.0	Yes

Table 2: Performance of Brute Force Algorithm for Different Cities for Cutoff Times of 300s and 600s

Data	**Approximation**		**Rel_Error_Approx**	**Local Search**			**Rel_Error_LS**
	Time (sec)	**Sol.Quality**		**Time (sec)**	**Avg. Sol.Quality**	**Best Sol.Quality**	
Atlanta	6.32E-05	2380447.56	0.180905	0.1206335068	2039428.06	2016321.99	0.011462
Berlin	3.32E-04	10403.86	0.309738	0.2152244329	8012.06	7941.31	0.008901
Boston	2.06E-04	1150959.11	0.224374	0.1587280989	962247.85	940032.80	0.023643
Champaign	3.83E-04	65714.19	0.248101	0.1758158445	54044.45	52672.08	0.026024
Cincinnati	2.10E-05	301215.87	0.083615	0.06773586273	278569.01	277952.59	0.002216
Denver	8.78E-04	134748.25	0.282157	0.2586569786	106845.23	105100.05	0.016611
NYC	5.39E-04	2027108.38	0.275164	0.2382211685	1639317.07	1589566.54	0.031315
Philadelphia	1.24E-04	1646247.58	0.134929	0.1479566813	1395980.78	1450704.62	-0.037717
Roanoke	0.00576305	838287.49	0.158627	0.8491278887	739983.77	723620.49	0.022655
SanFrancisco	0.00116301	1134996.97	0.340257	0.3048487663	847555.06	846909.72	0.000761
Toronto	0.00130200	1675107.67	0.387294	0.3777042866	1254175.81	1207856.05	0.038373
UKansasState	2.19E-05	68089.16	0.081552	0.0565718174	62962.31	62962.31	0
UMissouri	0.00125909	178251.65	0.265811	0.37414400957	142571.95	140860.73	0.012142

Table 3: Performance of Approximation and Local Search Algorithms for Various Cities

  - Runtime increases exponentially with the number of cities, making it impractical for datasets with more than 10–15 cities. The solution imporves as the cut-off time increses
  - Performance is dependent on cutoff times (e.g., 60, 300, 600 seconds). For larger datasets like Roanoke, the algorithm may take a very long time to converge.

- **Approximation Algorithm (MST-based 2-Approximation)**:
  - **Complexity**: $O(n^2)$ – Prim's MST algorithm on a complete graph with pairwise distances computed in the loop. As for each node we scan all other n nodes, to compute minimum distance neighbour.
  - Runtime is fast even for large datasets, all instances from the data complete in under a few milliseconds. Produces consistent full tour, same order results for every run.

- Generates a feasible tour but solutions are not as good as heuristic local search results.

- **Local Search**:
  - **Complexity**:
    - Nearest-neighbor algorithm: $O(n^2)$
    - 2-opt algorithm: $O(n^2)$
    - Simulated annealing: Cannot be defined by a Big-O since the moves in the algorithm is randomized and has probabilistic moves. The runtime is determined by the cooling schedule, the metropolis steps, and the cost of a single move.

3.2 Relative Error

- Errors from approximation algorithms are consistently greater than those from LS, indicating that the MST-based 2-approx generates valid yet distinctly suboptimal tours in contrast to heuristic refinement.

- LS attains minimal relative errors, demonstrating that local search (2-opt + annealing) consistently identifies solutions near the best-known ones in all datasets

$$\text{RelError}_{\text{Approx}} = \frac{\text{ApproxSolCost} - \text{BestSolCost}}{\text{BestSolCost}} \tag{1}$$

$$\text{RelError}_{\text{LS}} = \frac{\text{AvgLSCost} - \text{BestLSCost}}{\text{BestLSCost}} \tag{2}$$

$$\text{RelError}_{\text{BF}} = 0 \tag{3}$$

4 Effect of Cutoff Times on the Brute Force Solution

To better understand and visualize the factorial nature of the brute-force implementation of the Traveling Salesman Problem (TSP), we consider three cities from the dataset. The first is a small city with 10 nodes (`UKansasState`), the second is a medium-sized city with 68 nodes (`NYC`), and the third is a large city with 230 nodes (`Roanoke`). Shown below are line plots illustrating how the tour cost varies with different cutoff times for each of these cities.

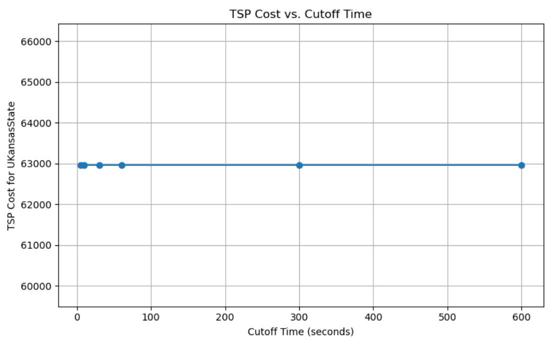

Figure 1: Cost vs Cutoff Time for UKansasState(10 nodes)

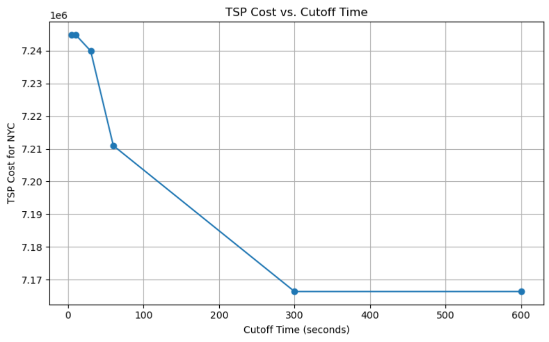

Figure 2: Cost vs Cutoff Time for NYC(68 nodes)

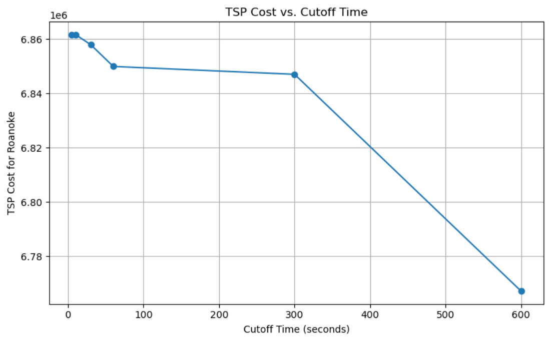

Figure 3: Cost vs Cutoff Time for Roanoke(230 nodes)

As seen in Figures 1, 2, and 3, the progression of the tour cost depends strongly on the number of nodes in the graph. For a small graph such as `UKansasState`, the cost is computed very quickly and remains the same whether the cutoff is 5 seconds or 600 seconds. For a medium-sized graph like `NYC`, the cost gradually decreases until around 300 seconds, after which it stabilizes. In contrast, for the large graph `Roanoke`, the cost continues to decrease even up to 600 seconds, and may continue to improve beyond that, reinforcing that larger graphs require substantially more time to approach the optimal tour cost due to the factorial nature of the brute force implementation.

5 Results Structure

5.1 Code Structure

- Github Link: `https://github.com/shraddha5bharadwaj/TSP`
- Main.py – Main driver that accepts parameters and generates output.
- **brute_force.py** – a brute-force solution to the problem
  - `__init__(filename, cutoff_time)` – Initializes the solver with input file, cutoff time, and default data structures.

  - `parse_file()` – Reads the TSPLIB file and extracts node coordinates and metadata.
  - `calc_distance(n1, n2)` – Computes the rounded Euclidean distance between two nodes.
  - `solve_brute_force()` – Enumerates all permutations within cutoff time to find the minimum-cost Hamiltonian cycle.
  - `write_output(method_code)` – Writes the best solution (cost, tour, full-tour flag) to the appropriate `.sol` file.

- **approximation.py** – an approximate solution with quality guarantees.
  - `read_tsp_file(file_path)` – Reads a TSPLIB .tsp file and returns a list of vertex coordinates in index = ID–1 order.
  - `euclid_dist(coords, i, j)` – Computes the Euclidean distance between vertices i and j.
  - `tour_length(coords, tour)` – Computes the total TSP tour cost by summing distances along the cycle (including last to first).
  - `mst_approx_algorithm(coords)` – Constructs an MST using Prim's algorithm and returns a preorder DFS traversal as a 2-approximation TSP tour.
  - `run_tsp_approx(file_path, cutoff, seed)` – Runs the MST-based approximation algorithm, measures runtime, prints the solution, and writes the `.sol` file.
- **local_search.py** – a heuristic algorithm with no formal guarantees, but effective in practice.
  - `nearest_neighbor` – returns a tour that connects all coordinates based on nearest unvisited neighbor.
  - `two_opt` – improves a tour by reversing edges to reduce total cost.
  - `simulated_annealing` – uses probabilistic hill-climbing with Boltzmann acceptance to escape local minima.
  - `local_search_tsp` – runs the LS pipeline and returns a TSP solution.
- Build – executable build.
- Data – the .tsp files.
- Output
  - approximate – output files for approximation algorithm.
  - local_Search – output files for LS algorithm.
  - brute_force – output files for brute force algorithm.
- The executable (`exec`) can be run as follows:

```
cd code
./exec -time <cutoff> -seed <seed> -alg [BF|Approx|LS] -inst ‘‘file_path’’
```

Example Run Commands

Creating executable

```
pyinstaller --onefile main.py
mv dist/main exec
chmod +x exec
```

Run of executable from root

```
cd code
./exec -time 30 -seed 5 -alg Approx -inst ‘‘/Users/shrads/Desktop/Algos/Data/Atlanta.tsp’’
./exec -time 30 -seed 5 -alg LS -inst ‘‘/Users/shrads/Desktop/Algos/Data/Atlanta.tsp’’
./exec -time 30 -seed 5 -alg BF -inst ‘‘/Users/shrads/Desktop/Algos/Data/Atlanta.tsp’’
```

Group Details

• **Shraddha Bharadwaj**	`sbharadwaj83@gatech.edu`
• **Srishti Kalepu**	`skalepu3@gatech.edu`
• **Aditya Raghavan**	`araghavan68@gatech.edu`