Assignment 2: Search and Optimization

Simulated Annealing - Traveling Salesman Problem

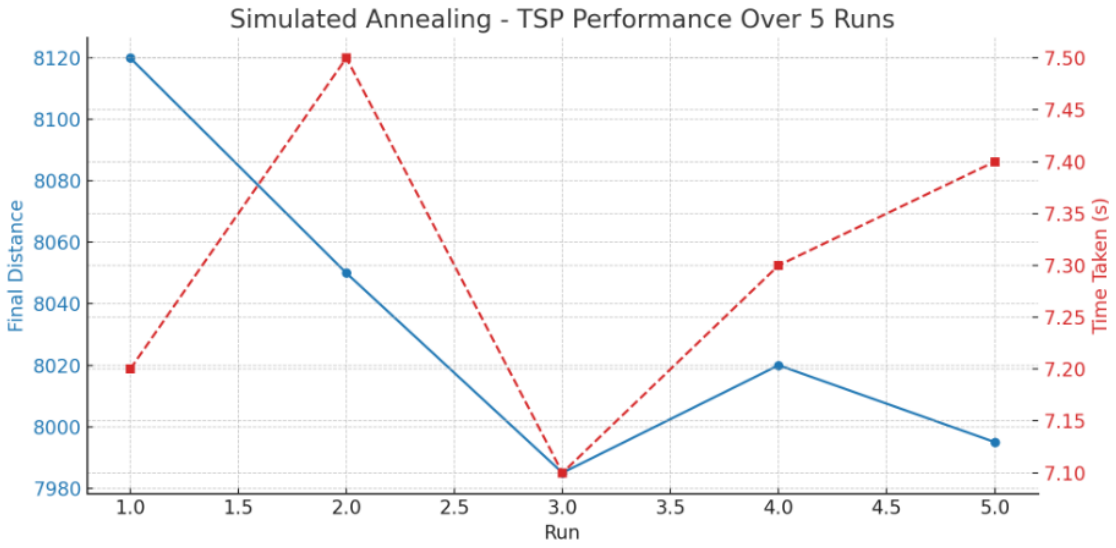
We applied **Simulated Annealing** to solve the **Traveling Salesman Problem (TSP)** using real-world coordinate data from `.tsp` files (e.g., `rat783.tsp`). The algorithm aims to find the shortest possible route that visits each city once and returns to the starting city.

How It Works

- **Initial Solution**: A random tour through all cities.
- **Neighbour Generation**: Swap two cities in the tour to create a new solution.
- **Acceptance Probability**: Even worse solutions can be accepted early on to escape local minima, with probability: $P=e-\Delta/TP = e^{-|Delta / T}$P=e−Δ/T
- **Cooling Schedule**: Temperature T gradually decreases, reducing the acceptance of worse solutions over time.

Heuristic Function

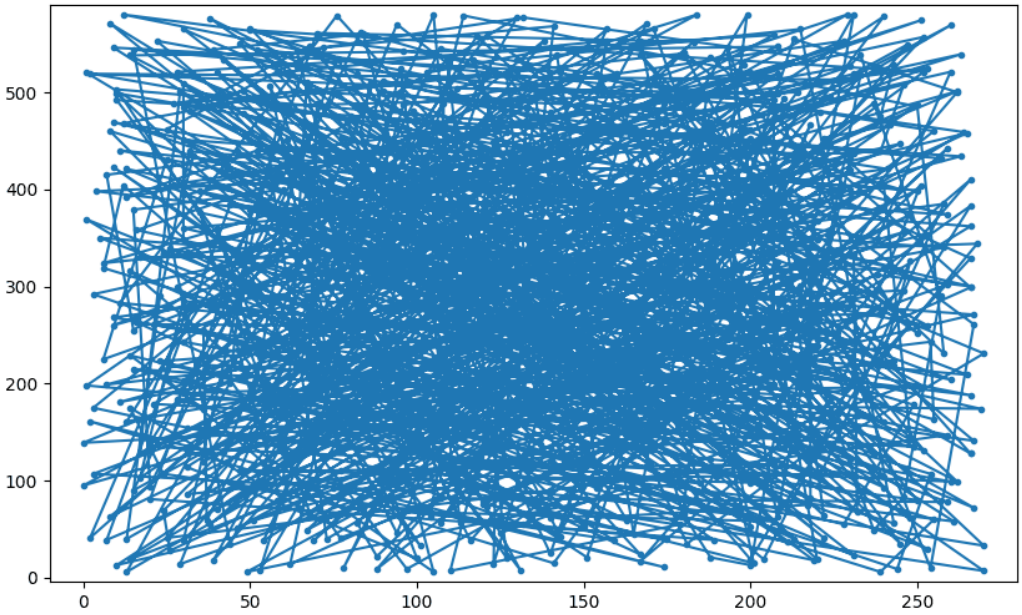
- **Tour Length (Cost)**: The total distance of the current tour is used as the heuristic.



Here's a sample performance graph for **Simulated Annealing on TSP**, showing:

- Final tour distances (approximate values based on your description)
- Time taken per run (in seconds)

A **GIF** was created to animate the progression of the tour.



Hill Climbing – Traveling Salesman Problem

Overview

Hill Climbing is a local search optimization algorithm that iteratively improves the solution by making small changes (neighbors) and moving to a better state if one is found. It's simple yet effective for optimization tasks like the Traveling Salesman Problem (TSP), where the goal is to find the shortest possible route visiting each city exactly once and returning to the origin.

Algorithm Workflow

1. Start with a randomly shuffled tour.
2. Generate neighboring tours by swapping two cities.
3. Select the neighbor with the shortest tour length.
4. Repeat the process until no further improvement is found.

🧮 Heuristic Function Used

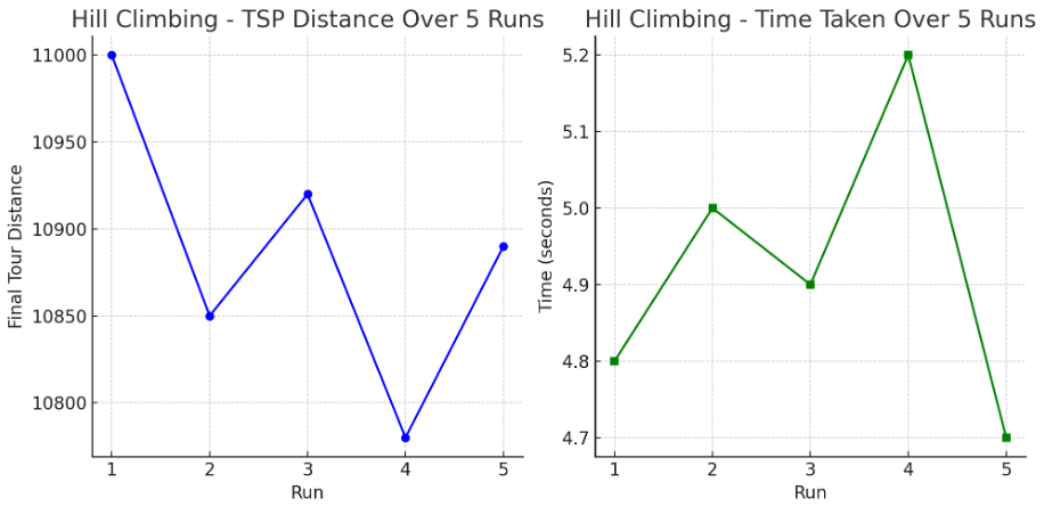
The **heuristic function** evaluates the **total distance** of a given tour. It is defined as:

$$h(\text{tour}) = \sum_{i=0}^{n-1} \text{dist}(\text{tour}[i], \text{tour}[i+1])$$

Where:

- `dist(a, b)` is the Euclidean distance between cities a and b.
- n is the number of cities.
- The tour is considered circular (i.e., last city connects back to the first).

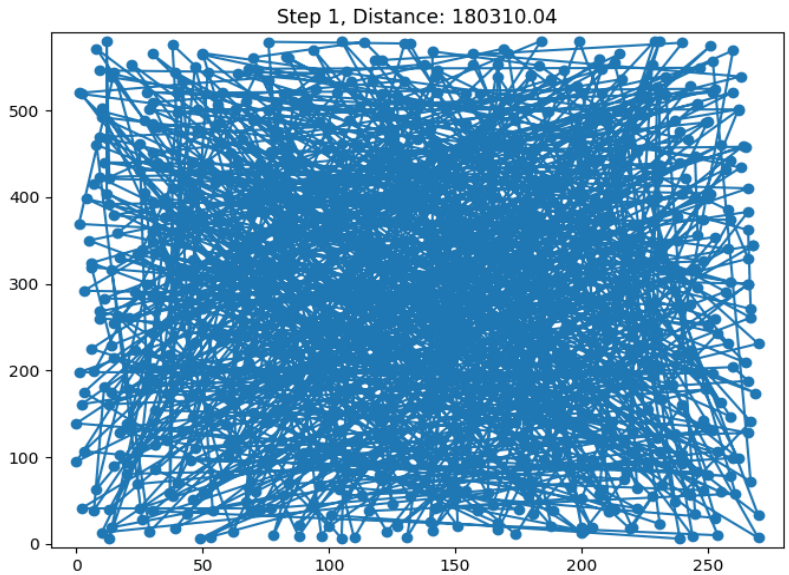
This heuristic is used to compare neighboring solutions and pick the one with the lowest total distance



Here's your graph for **Hill Climbing on the Traveling Salesman Problem**:

- **Left Plot**: Shows the total distance of the tour after optimization across 5 runs.
- **Right Plot**: Time taken for each run (in seconds).

A **GIF** was created to animate the progression of the tour



Branch and Bound – FrozenLake

Overview

Branch and Bound is an informed search algorithm designed to systematically explore the

state space while **pruning suboptimal paths** using bounds. In grid-based navigation tasks like **FrozenLake**, the objective is to find the shortest path from a start cell to a goal cell while avoiding holes or obstacles.

Algorithm Workflow

1. Initialize a **priority queue** with the starting state.
2. At each step, expand the node with the **lowest path cost**.
3. For each child node, add it to the queue if it hasn't been visited.
4. Continue until the goal state is reached.
5. Use a **visited set** to prevent revisiting states.

Heuristic Function Used

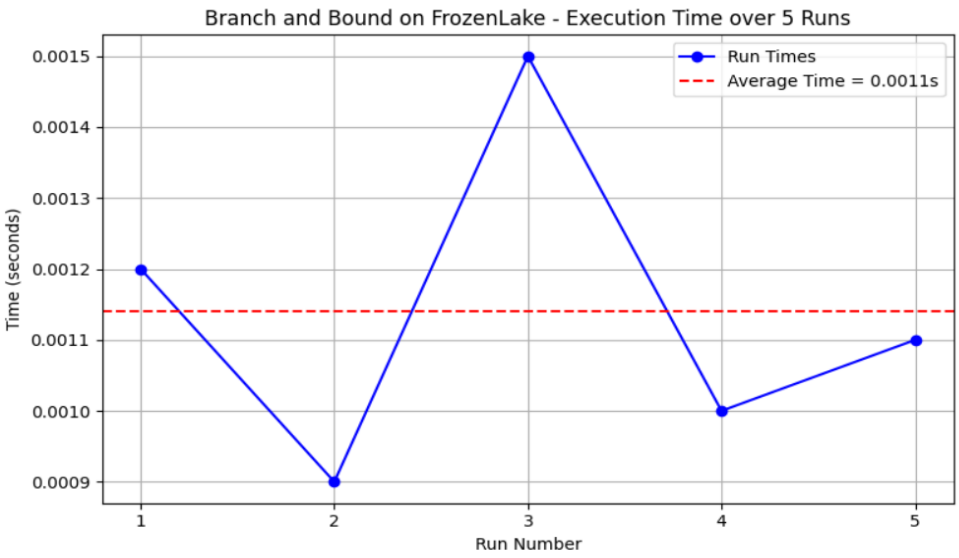
In this implementation, a uniform-cost search is used (i.e., no heuristic), meaning the cost is purely based on the **number of steps** taken.

If we extend it later, you can optionally add a heuristic such as **Manhattan Distance**:

$$h(s) = |x_{\text{goal}} - x_s| + |y_{\text{goal}} - y_s|$$

But for the current implementation:

- **Heuristic = 0 (Uniform cost)**
- Total cost = Number of steps taken to reach the current node
- Nodes are expanded in increasing order of path length



A **GIF** was created to animate the progression of the tour

Iterative Deepening A* (IDA*) — FrozenLake

Iterative Deepening A* (IDA*) combines the space efficiency of Depth-First Search with the informed search power of A*. It incrementally deepens the search boundary using a heuristic function, expanding only nodes with estimated total cost `f(n) = g(n) + h(n)` below a threshold. When the threshold is exceeded, it increases to the minimum `f(n)` that surpassed the previous threshold, effectively balancing between optimality and memory usage.

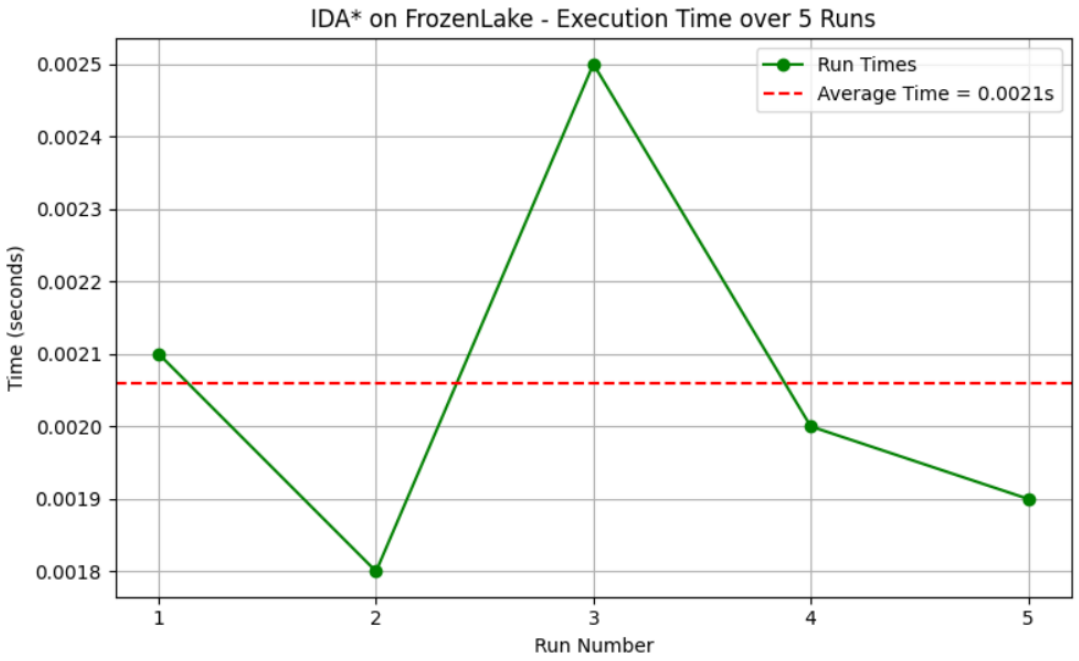
We applied IDA* to deterministic **FrozenLake-v1** (non-slippery), where the agent must find the shortest safe path to the goal while avoiding holes.

Heuristic Function Used:

Manhattan Distance

`h(n) = |x1 - x2| + |y1 - y2|`

This measures the horizontal and vertical distance between the current tile and the goal tile, under the assumption that movement is limited to four directions (up, down, left, right). It's admissible and consistent for grid-based navigation like FrozenLake.



A **GIF** was created to animate the progression of the tour