Traveling Salesman Problem (TSP)

Team Members & Roles

- ## Eng: Shahd Mohamed Hamed Abdelrahman
  - **Task:** Implementation of **Uniform-Cost Search (UCS)**.
  - **Responsibilities:** Developing an uninformed search strategy that guarantees optimal solutions by expanding nodes based on the lowest cumulative path cost.

- ## Eng: Omar Mohamed Ibrahim Badawy
  - **Task:** Implementation of **A* Search Algorithm**
  - **Responsibilities:** Designing an informed search using path cost plus a heuristic estimate ($f(n) = g(n) + h(n)$) to reduce search space and improve speed.

- ## Eng: Noor Hussain Mwafi
  - **Task:** Implementation of **Hill Climbing Search**
  - **Responsibilities:** Developing a local optimization framework using **Random-restart** and **Swap-based neighbor generation** to iteratively improve tour quality.

- ## Eng: Nour Yasser Hashem El-Sheikh
  - **Task:** Implementation of **Nearest Neighbor + 2-opt**.
  - **Responsibilities:** Combining a greedy constructive algorithm with a local search operator (2-opt) to remove "crossover" edges and achieve very fast, near-optimal solutions

- ## Eng: Hend Mohamed Mohamed Fiala
  - **Task:** Implementation of **Genetic Algorithm (GA)**
  - **Responsibilities:** Applying evolutionary concepts (Selection, Crossover, and Mutation) to maintain a population of solutions and perform a global search of the solution space

INTRODUCTION

Navigating the Traveling Salesman Problem (TSP)

The **Traveling Salesman Problem (TSP)** stands as one of the most iconic challenges in combinatorial optimization and computer science. The objective is deceptively simple: find the shortest possible route that visits a set of cities exactly once and returns to the origin. However, due to its **NP-hard** nature, the search space expands factorially, making brute-force solutions impossible for large-scale maps.

To tackle this complexity, our team has implemented a diverse suite of algorithmic strategies—ranging from exact uninformed searches to sophisticated meta-heuristics. Each approach offers a unique trade-off between computational speed and solution optimality.

1. Systematic Search Foundations

- **Uniform-Cost Search (UCS):** Led by **Eng. Shahd Mohamed**, this implementation focuses on an uninformed search strategy. By expanding nodes based on the lowest cumulative path cost, it establishes a rigorous baseline that guarantees an optimal solution.
- *A Search Algorithm:** Developed by **Eng. Omar Mohamed**, this informed search enhances efficiency by integrating path cost with a heuristic estimate ($f(n) = g(n) + h(n)$), significantly pruning the search space without sacrificing accuracy.

2. Local Optimization & Heuristics

- **Hill Climbing Search: Eng. Noor Hussain** implements a local optimization framework. By utilizing **Random-restart** and **Swap-based** neighbor generation, this approach iteratively climbs toward better solutions, effectively navigating the landscape of local optima.
- **Nearest Neighbor + 2-opt: Eng. Nour Yasser** focuses on a hybrid approach. This combines a greedy constructive algorithm for immediate results with a **2-opt** local search operator to eliminate "crossover" edges, achieving high-quality solutions with remarkable speed.

3. Evolutionary Intelligence

- **Genetic Algorithm (GA):** Managed by **Eng. Hend Mohamed**, this strategy applies the principles of natural selection. Through **Selection, Crossover, and Mutation**, the algorithm maintains a diverse population of potential routes, performing a global search to find near-optimal tours in vast solution spaces.

1. Uniform Cost Search (UCS)

The Uniform-Cost Search (UCS) is an uninformed search strategy designed to find the optimal path in a weighted graph[1].

- **Core Strategy**: It expands nodes based on the lowest cumulative path cost from the source.
- **Optimality**: Unlike greedy heuristics, UCS guarantees the discovery of the shortest possible route for any given TSP instance.
- **Objective**: Its primary role in this project is to provide a gold-standard "optimal" baseline to measure the quality of other heuristic methods.

Phase 1: Exploration and Expansion

In this phase, the algorithm explores the search space systematically without any prior knowledge of city locations.

- **Mechanism**: It maintains a priority queue of paths, always choosing the one with the least total distance to expand next.
- **Exhaustive Nature**: It explores all possible directions equally until the goal state (visiting all cities and returning) is reached
- **Uniformity**: Because it does not use a heuristic, it treats all neighboring cities solely based on their actual distance from the current node.

Phase 2: Guaranteeing Optimality

The strength of UCS lies in its mathematical certainty regarding the final result.

- **Optimal Solution**: It is proven to find the global minimum path cost.
- **Comparison to Heuristics**: While methods like Nearest Neighbor or Hill Climbing might settle for a "good" path, UCS continues until it confirms no shorter path exists.
- **Cost Focus**: Every decision is based on the actual path cost (g(n)), ensuring the most efficient route is never overlooked[10].

1. Uniform Cost Search (UCS)

Best Performance Conditions:

- **Absolute Precision:** Best used when the **optimal solution** (shortest path) is strictly required.
- **Small-Scale Tasks:** Highly effective for small datasets where the state space is manageable.
- **Optimal Baseline:** Ideal for creating a reference point to evaluate the quality of other heuristic algorithms.

Factors Affecting Performance:

- **Computational Expense:** Performance drops drastically as the number of cities increases due to **extremely slow** execution.
- **Memory Exhaustion:** It suffers from **extremely high memory usage** because it stores all explored paths.
- **Lack of Guidance:** Unlike A*, it operates without heuristics, exploring the search space in all directions, which limits its scalability.

Number of Cities (N)	Execution Time (Speed)	Memory Usage	Solution Quality (Optimality)
N = 5 (Small)	Fast	Manageable	100% Optimal (Best Choice)
N = 15 (Medium)	Slow	Very High	100% Optimal
N = 20 (Large/Complex)	Extremely Slow	Critical (May crash)	100% Optimal (Impractical)

2.A* Search

A* is an Informed Search Algorithm used to find the shortest path between nodes. It combines the strengths of Uniform Cost Search (BFS with weights) and Greedy Best-First Search.

The Core Formula

The algorithm evaluates nodes using the fitness function:

f(n) = g(n) + h(n)

- **g(n)**: The actual cost from the start node to the current node n.
- **h(n)**: The **Heuristic** estimated cost from node n to the goal (the "informed" part).
- **f(n)**: Total estimated cost of the cheapest solution through node n.

Mechanism

A* uses a **Priority Queue** (Min-Heap) to always expand the node with the lowest f(n) value. This ensures that the algorithm explores the most promising paths first, significantly reducing the number of nodes visited compared to Uninformed Search.

What makes A* Perfect (Optimal)? (Conditions)

To guarantee the shortest path, the Heuristic h(n) must satisfy two conditions:

1. **Admissibility**: The heuristic must **never overestimate** the actual cost to reach the goal. It must be optimistic or exact (h(n) < h^*(n)).
2. **Consistency (Monotonicity)**: For every node n and its successor n' , the estimated cost to the goal from n is no greater than the step cost to n' plus the estimated cost from n'.

Factors that Reduce Efficiency (Conditions)

1. **Poor Heuristic**: If h(n) is too low (close to 0), A* behaves like BFS (very slow, explores everything).
2. **State Space Explosion**: In problems like TSP, as the number of cities increases, the number of possible paths grows factorially (n!), leading to high **Memory Consumption**.
3. **Inadmissible Heuristic**: If h(n) overestimates, the algorithm might find a solution faster, but it **won't be the optimal (shortest)** solution.

---

2. A* Search

In the Traveling Salesperson Problem (TSP), A* struggles with "Complexity" as the number of cities grows. Here is a comparison of what happens during the evaluation:

Cities 20	Cities 15	Cities 5	Metric
Slow : Performance drops significantly (Exponential growth).	**Moderate :**Can take seconds to a few minutes depending on CPU.	**Instant :**Results in milliseconds.	**Execution Time**
Critical : Likely to hit memory limits or cause "Out of Memory" errors.	**High :**Storing unvisited states consumes significant RAM.	**Very Low :**The priority queue stays small.	**Memory Usage**
Absolute Minimum :)If the search completes.(	**Absolute Minimum :** Guaranteed to be the shortest possible route.	**Absolute Minimum :**Finds the perfect shortest path.	**Path Cost (Tour Length)**
Imbalance : Speed becomes too slow to maintain optimality.	**Shift :**Maintaining optimality starts to sacrifice speed.	**Perfect Balance :** High speed and 100% optimal.	**Optimality vs. Speed**
Poor : Not scalable for larger real-world datasets.	**Struggling :** Reaching the limits of exact search methods.	**Excellent :**Handles small sets with ease.	**Scalability**

The provided Python code implements A* for TSP using:

1. **heapq:** To manage the Priority Queue based on the lowest f(n).
2. **distance function:** Calculates the Euclidean distance (g(n)).
3. **heuristic function**: Uses the distance to the nearest unvisited city as a simple estimation to guide the search.
4. **solve method**: Continuously pops the best path, checks if all cities are visited, and returns to the start city to complete the cycle.

3. Hill Climbing Algorithm

Algorithm Mechanics, Challenges, and Implementation

- ## Local Optimum & Random Restarts:
  - **The Trap:** A **Local Optimum** occurs when the algorithm reaches a state better than its immediate neighbors but inferior to the best possible global solution.
  - **The Solution: Random Restarts** act as a safeguard; by re-launching the search from different random starting points, we increase the probability of finding the **Global Optimum**.
  - **Scaling:** As the number of cities increases, the landscape becomes more "rugged" with more local traps, necessitating an increase in the number of restarts.
- ## Reproducibility (Seed vs. Restart):
  - **Random Seed:** Ensures the "randomness" is deterministic. It allows different users to get the **exact same results** every time they run the code, which is essential for scientific evaluation.
  - **Random Restart:** Operates *within* a single run to explore various regions of the search space. The Seed simply ensures that these "random" regions are identical across different executions.
- ## Technical Features:
  - **Heuristic Calculation:** Real-world distances are calculated in **Kilometers (km)** using the **Haversine Formula**.
  - **Visualization:** Interactive **Folium** maps highlight the **Start City in Green** and provide city names and visitation order upon clicking.

3. Hill Climbing Algorithm

Performance Metrics, Conditions for Success, and Conclusion

- ## Comparative Analysis (5 vs. 15 vs. 20 Cities):
  - **Execution Time:** Increases with city count and restarts; 5 cities are near-instant, while 20 cities require more computation time to ensure quality.
  - **Memory Usage:** Extremely efficient (O(n)) across all cases as it only stores the current and best paths.
  - **Path Cost & Optimality:** 5 cities consistently find the global optimum. For

    15-20 cities, the algorithm provides a high-quality approximation (Near-Optimal) rather than a guaranteed perfect solution.

- ## Optimal Conditions for Hill Climbing:
  - **Best Scenario:** Small to medium datasets **(N < 30)** where speed is prioritized over absolute perfection.
  - **Landscape:** Performs best when the search space has a clear gradient (slope) toward the shorter path

Impact of Increasing N	Cities 20	Cities 15	Cities 5	Metric
Very Fast . Even with more cities, it only checks a few neighbor states.	0.01~s	0.005~s	s0.0005~	**Execution Time**
Extremely Low . It only stores the "Current Path" and the "Best Path".	Minimal.	Minimal.	.Minimal	**Memory Usage**
Quality decreases . With 20 cities, there are too many "traps" (local optima).	Sub-.optimal	Fair / Variable.	.Optimal	**Path Cost**
Great for speed, but reliability drops as the search space grows.	Low Accuracy.	Good.	.Excellent	**Optimality vs Speed**

.

4. Nearest Neighbor + 2-opt

Overview of the Hybrid Approach

This project utilizes a hybrid heuristic strategy that combines construction and optimization.

- **Methodology**: It integrates the Nearest Neighbor (NN) algorithm with 2-opt Local Search.
- **Primary Objectives**: The goal is to achieve fast execution, low memory consumption, and a near-optimal solution.
- **Practicality**: This approach offers a balanced trade-off between the quality of the solution and computational efficiency.

Phase 1: Construction via Nearest Neighbor (NN)

The process begins by generating a feasible starting tour.

- **Mechanism**: Starting from a chosen city, the algorithm moves to the closest unvisited city until the cycle is complete.
- **Advantages**: It is simple to implement, extremely fast, and requires very little memory.
- **Role**: It serves as a baseline for further improvement, though it is greedy and does not guarantee an optimal solution.

Phase 2: Refinement via 2-opt Local Search

The rough initial tour is refined to remove obvious inefficiencies.

- **Mechanism**: The algorithm selects two edges and reverses the segment between them to see if it reduces the total distance.
- **Edge Crossing Removal**: A key purpose of this stage is to eliminate "edge crossings" in the tour.
- **Termination**: The process repeats iteratively until no further local improvements can be found

4. Nearest Neighbor + 2-opt

Best Performance Conditions:

- **Scalability:** Best suited for moderate to large-scale TSP instances where fast solutions are required.
- **Efficiency:** Ideal when memory consumption must be kept to a minimum.
- **Goal:** Preferred when a high-quality, near-optimal solution is acceptable rather than a strictly optimal one.

Factors Affecting Performance:

- **Starting Point:** The quality of the final tour is highly dependent on the initial city selected during the NN phase.
- **Local Optimum Trapping:** The algorithm stops once no nearby improvements exist, meaning it can miss the global optimal solution.
- **Problem Size:** For very small problems, its performance is "worse" than UCS or A* because it doesn't guarantee absolute optimality.

Number of Cities (N)	Execution Time (Speed)	Memory Usage	Solution Quality (Optimality)
N = 5 (Small)	**Instantaneous**	**Minimal**	**Near-Optimal** (UCS/A* are better for absolute optimality)
N = 15 (Medium)	**Very Fast**	**Low**	**High Quality** (Best balance of speed and precision)
N = 20 (Large/Complex)	**Highly Efficient**	**Consistently Low**	**Reliable** (May hit local optima, but stays practical)

5. Genetic Algorithm (GA)

Overview of Evolutionary Intelligence

The Genetic Algorithm (GA) represents a meta-heuristic approach inspired by the process of natural selection.

- **Global Search:** Unlike local search methods, GA explores the entire solution space to find near-optimal routes.
- **Population-Based:** It maintains a diverse set of potential solutions (tours) simultaneously.
- **Goal:** To achieve high-quality results in large-scale TSP problems where other methods might fail.

Phase 1: Population and Selection

The process begins by creating and filtering potential solutions.

- **Initialization:** A population of random or heuristic-based tours is generated to start the evolution.
- **Selection:** Solutions with shorter path lengths (higher fitness) are prioritized for "reproduction."
- **Role:** This phase ensures that the best traits of current routes are passed down to future generations.

Phase 2: Crossover and Mutation

This stage introduces variation and exploration to improve the results.

- **Crossover:** Parts of two parent tours are combined to create "offspring," attempting to inherit the best segments of both.
- **Mutation:** Small, random changes are made to individual tours to maintain diversity and prevent the algorithm from getting stuck in local optima.
- **Purpose:** This simulates biological evolution to explore new regions of the search space.

5. Genetic Algorithm (GA)

Evaluation and Best Performance

GA is evaluated based on its ability to handle complex problems.

- **Best Performance:** It is most effective for large-scale TSP instances where global exploration is required.
- **Resource Cost:** It requires more execution time and computational resources compared to simpler heuristics like NN.
- **Optimality:** While it doesn't guarantee the absolute mathematical optimum like UCS, it is much more likely to find a global optimum than Hill Climbing.

Performance Comparison Table

Number of Cities (N)	Execution Time	Memory Usage	Solution Quality
Small (N=5)	**Slow** (Compared to NN/A*)	**Moderate**	**High** (Overkill for small sets)
Medium (N=15)	**Moderate**	**High**	**Very High** (Very reliable results)
Large (N=20+)	**Best Choice**	**Intensive**	**Global Optimum** (Best at avoiding local traps)

GA Performance Conditions

- **Best Performance Conditions:** Excels in large-scale problems [8]and scenarios where global exploration is necessary to avoid local optima[9].

- **Conditions Affecting Performance:** The size of the population, the mutation rate, and the available computational time directly impact the quality of the final solution

Performance Metrics Overview

Metric	UCS	A*	Hill Climbing	NN + 2-Opt	Genetic (GA)
Memory Usage	**Extremely High**	High	Minimal	Minimal	Moderate
Scalability	Very Poor	Limited	Excellent	**Excellent**	Excellent
Best Case	Small Data	Precision Needs	Quick Drafts	**Industrial Use**	Large-scale TSP
Path Cost (Length)	**Near-Minimum**	Minimum (Optimal)	Minimum (Optimal)	**Near-Minimum**	**Minimum (Optimal)**

Best Algorithm For This Problem

From Algorithms We Used

NN + 2-Opt (Why?)

Best practical choice because it provides a near-optimal solution almost instantly by effectively removing path intersections.

At All

LKH (Lin-Kernighan) (Why ?)

Best overall globally because it is the world-standard algorithm capable of solving massive TSP problems with thousands of cities at extreme speed.

Comparative Analysis by Number of Cities

5 Cities (Small)

- **All algorithms** perform well here.
- **UCS & A***: Find the absolute shortest path instantly (only 4! = 24 permutations).
- **Complexity**: Negligible for all.

15 Cities (Medium)

- **UCS**: Starts to struggle. The search tree expands to millions of nodes, consuming significant RAM.
- **A***: Still optimal and efficient **if** using a strong heuristic like MST (Minimum Spanning Tree).
- **NN + 2-Opt**: Finds a solution almost identical to A* but in a fraction of a second.
- **Genetic**: Requires several generations to converge, might be slower than 2-Opt for this size.

20 Cities (The Breaking Point)

- **UCS**: Likely to **crash** or run out of memory. 19! is a massive number of states.
- **A***: Becomes very slow; memory consumption becomes the primary bottleneck.
- **Hill Climbing**: Often gets "stuck" in a sub-optimal path (Local Optimum) and fails to find the best route.
- **NN + 2-Opt**: This is the **sweet spot**. It provides a near-optimal solution

  (within 1-3% of the best) almost instantly.

- **Genetic**: Highly effective here. It explores diverse areas of the map and usually beats Hill Climbing in quality.


Thank You For Helping Me