Traveling Salesman Problem (TSP)

Problem Description

The Traveling Salesman Problem (TSP) is an optimization problem where a salesman must visit a set of cities exactly once and return to the starting city while minimizing the total travel distance. It is a classic NP-hard problem widely used to evaluate search algorithms.

Why This Problem?

TSP is important because it has a very large search space and clearly shows the difference between optimal and heuristic-based algorithms. It is ideal for comparing UCS, A*, and Greedy search strategies.

State Representation

In this project, cities are **generated randomly** on a 2D plane, where each city is represented by its `(x, y)` coordinates. The distances between cities are calculated dynamically using the Euclidean distance formula.

A state in the Traveling Salesman Problem (TSP) is represented by:

The current city index

The list of cities visited so far

The total path cost accumulated up to the current state

Project Domain

The Traveling Salesman Problem (TSP) belongs to the **Optimization Problems** domain in Artificial Intelligence.
It focuses on finding the best possible solution among a large number of alternatives by minimizing the total travel cost.

Successor Function

The successor function generates all unvisited cities that can be visited next from the current city.

Cost Function

The cost function is the sum of distances between consecutive cities, including the return to the starting city.

Algorithms Implemented

1. Uniform Cost Search (UCS): Finds the optimal solution but is computationally expensive.
2. A* Search: Uses a heuristic to reduce search space while guaranteeing optimality.
3. Greedy Search: Fast but does not guarantee the optimal solution.

Expected Results

UCS and A* produce optimal solutions but differ in execution time. Greedy search produces faster but near-optimal solutions.

Comparison between algorithms

Uniform Cost Search (UCS):

UCS explores all possible paths in order of their cumulative cost (g(n)) until it finds the goal. It is essentially Dijkstra's algorithm adapted for a state-space search.

How it's calculated:

In TSP, a "State" consists of (Current City + The set of visited cities).

The number of possible subsets of cities is 2^n (where n is the number of cities).

For each subset, you could be at any of the$n cities.

Thus, the total number of states is roughly $n*2^n$.

At each state, the algorithm checks transitions to all remaining cities (n transitions).

Big O Complexity:

$$\mathbf{O(n^2*2^n)}$$

So **provably optimal** (finds the shortest path), but it is very slow and memory-intensive because it is "blind"—it doesn't know the direction of the goal.

A* Search (with MST Heuristic):

A* uses the formula f(n) = g(n) + h(n), where h(n) is an estimated cost to reach the goal.

Minimum Spanning Tree (MST) is the heuristic that used in A* algorithms.

How it's calculated:

The worst-case state space remains the same as UCS:

$$\mathbf{O(n^2 * 2^n)}$$

However, in each node, you calculate the MST using Prim's algorithm, which takes $O(n^2)$ time.

Practical Big O: While the theoretical worst-case is the same, A* explores **significantly fewer states** than UCS because the MST heuristic guides the search toward the most promising paths.

So we can say A* is the **smartest optimal algorithm.**It guarantees the shortest path (like UCS) but reaches the solution much faster by "guessing" the remaining distance efficiently.

Greedy Search (Nearest Neighbor):

This is a simple "Heuristic" or "Rule of Thumb" approach: "From my current city, always go to the nearest unvisited city."

How it's calculated:

You start at one city.

You look for the closest among (n-1) cities, then

(n-2), and so on.

This is a single loop of n steps, and inside it, finding the minimum distance takes O(n)

Big O Complexity:

$$\mathbf{O(n^2)}$$

So we can say greedy algorithm is the **fastest by far**, but it is **not optimal**. It often makes "short-sighted" decisions that lead to a very long final edge when returning to the start.

Notes:

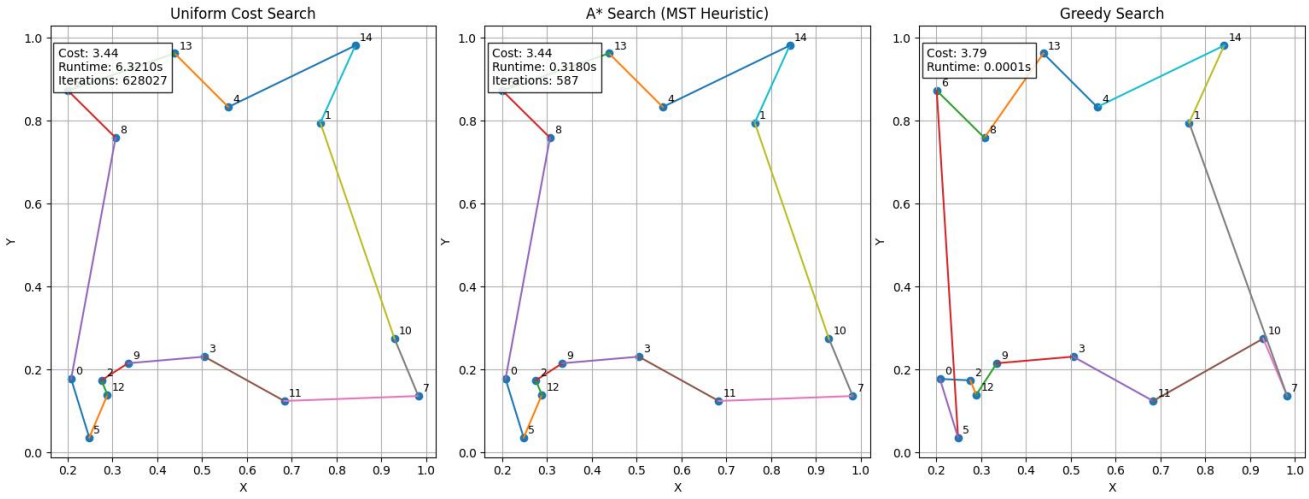
UCS vs. A*: Both will give you the exact same result (the absolute shortest path). However, A* is like using a GPS, while UCS is like

driving down every single street until you happen to find your destination.

Optimal vs. Greedy: If you have 15 cities, A* is perfect. If you have 1,000 cities, UCS and A* will crash your computer because of the 2^n growth, while Greedy will still finish in a fraction of a second but will not reach to the **optimal solution**.

Try to run the code when cities number is (15):

Like in picture below:



We can see UCS and A* algorithm reach to the **optimal solution**.

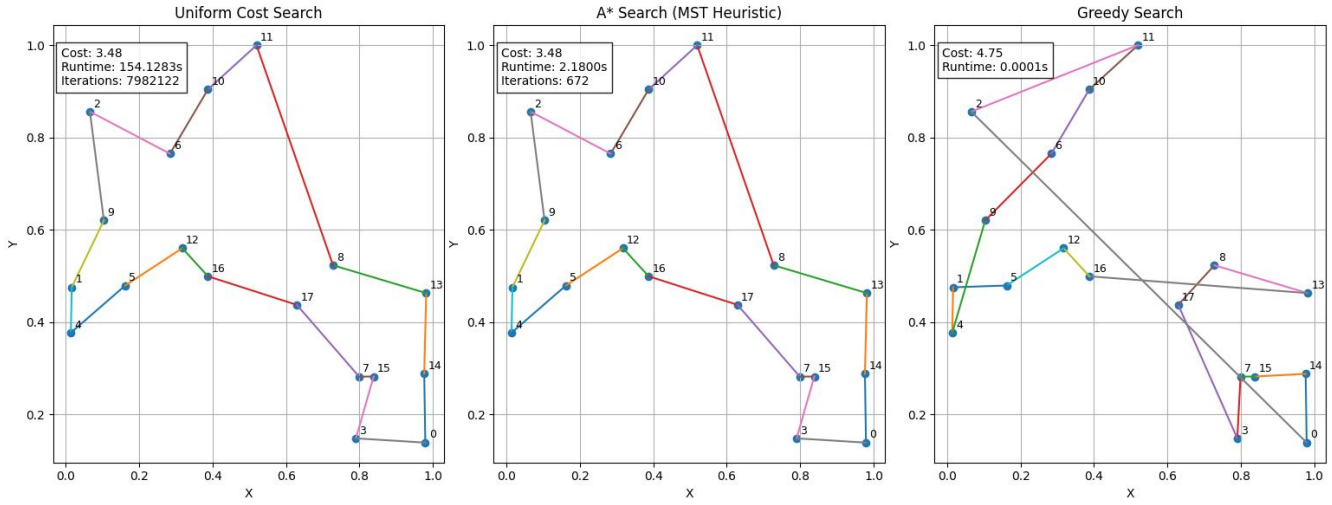
But the UCS take more time and more iterations unlike A*.

In other hand the greedy algorithm did not reach to **optimal solution, have the most cost** and the **fastest algorithm.**

In USC case the “mean” of runtime will be about 6 seconds.

Try to run the code when cities number is (18):

Like in picture below:



When we try to find the way to 18 cities the UCS runtime become more than two minutes.

And need to do more iterations.

Node Iterations:

Greedy Search (Nearest Neighbor):

Iterations: Exactly n (where n is the number of cities).

Behavior: It is a "straight-line" approach. It never looks back, never evaluates alternatives, and never reconsiders a decision.

Efficiency: Highest (in terms of work), as it does the absolute minimum amount of effort.

Analogy: A person walking through a maze who only turns at the first available corner without ever looking at a map.

Uniform Cost Search (UCS):

Iterations: Extremely High ($O(n*2^n)$).

Behavior: It is a "brute-force" expansion. It expands nodes in every direction, starting with the cheapest paths first. It explores every possible combination until it proves no other path could be shorter.

Efficiency: Lowest, as it explores the "entire base" of the search tree.

Analogy: A flood of water filling a maze; it goes everywhere simultaneously until it eventually reaches the exit.

A* Search (with MST):

Iterations: Medium (Significantly fewer than UCS).

Behavior: It uses **"Pruning."** Because the MST heuristic h(n) gives a realistic estimate of the remaining distance, A* can identify "dead-end" paths early and stop exploring them.

Efficiency: High, It is mathematically the "most efficient" optimal algorithm (it expands the minimum nodes necessary to prove optimality).

Analogy: Someone with a compass and a rough map. They might take a few wrong turns, but they generally head in the right direction and ignore paths that clearly lead the wrong way.

Efficiency Ratio:

This comparison between A* and UCS because they can reach to the optimal solution.

We can measure the efficiency of **A*** compared to **UCS** using the following equation:

Efficiency = {UCS Iterations} / {A* Iterations}

If the result is **5**, it means that **A*** was 5 times more efficient in terms of saving computational effort and reducing the number of nodes explored to reach the same optimal result.

Why is this ratio important?:

his ratio demonstrates the "power" of the **Heuristic** you used . The higher this number, the more accurate the heuristic is at guiding the algorithm and avoiding useless paths.

What to expect?

Small n: For 5 or 10 cities, the ratio might be small (e.g., 1.5x or 2x).

Larger n: As the number of cities increases to 15 or 20, you will see the ratio jump significantly (e.g., 10x or 50x), showing how A* "saves" the computer from crashing.