

AI Pathfinding Visualizer & Algorithms Comparison

Course: Introduction to Artificial Intelligence

Project Topic: Solving Maze Problems using Uninformed, Informed, and Local Search Strategies

Group Number: 44

1. Problem Description and Formulation

1.1 Overview

This project develops an interactive software simulation that visualizes and compares various Artificial Intelligence search algorithms for pathfinding. The core problem involves navigating an agent (robot) through a grid environment from a starting position to a goal position, while avoiding user-placed obstacles. The project demonstrates practical trade-offs between speed, memory usage, and path optimality across different search strategies.

1.2 Formal Problem Formulation

The maze problem is formulated as a search problem with the following components:

- **State Space:** A discrete set of coordinates (i,j) representing grid cells, where $i \in [0, \text{rows}-1]$ and $j \in [0, \text{cols}-1]$
- **Initial State:** The specific coordinate S designated as the Start Node (visualized in green)
- **Goal State:** The coordinate G designated as the Target Node (visualized in red)
- **Action Space:** Four-directional movement to adjacent cells: {Up, Down, Left, Right}. Diagonal movement is not permitted
- **Transition Model:** Movement from cell (i,j) results in state (i',j') only if the new state is within grid boundaries and is not marked as an obstacle (visualized in black)
- **Path Cost:** Uniform cost environment where each valid movement between adjacent cells has a cost of 1

2. Algorithm Implementations

We successfully implemented seven distinct search algorithms, categorized based on their use of domain knowledge.

2.1 Uninformed Search Strategies

These algorithms explore the search space without domain-specific knowledge about the goal's location.

Breadth-First Search (BFS)

Explores the grid layer-by-layer, expanding all nodes at depth d before nodes at depth $d+1$. Uses a FIFO queue and guarantees the shortest path in uniform-cost environments.

Depth-First Search (DFS)

Explores as deeply as possible along one branch before backtracking. Uses a LIFO stack. While memory-efficient, it is neither complete in infinite spaces nor optimal, often producing very long paths.

Iterative Deepening Search (IDS)

A hybrid approach that repeatedly calls DFS with increasing depth limits. Combines the completeness of BFS with the memory efficiency of DFS.

2.2 Informed Search Strategies

These algorithms utilize a heuristic function to guide the search toward the goal more efficiently.

Uniform-Cost Search (UCS)

Expands the node with the lowest cumulative path cost $g(n)$. In our grid where all step costs are 1, its behavior is functionally identical to BFS.

Greedy Best-First Search

Selects the node that appears closest to the goal according to the heuristic $h(n)$. Fast but not guaranteed to find the optimal path.

A Search (A-Star)*

The most robust algorithm, expanding nodes based on the total estimated cost $f(n) = g(n) + h(n)$, where $g(n)$ is the actual cost so far and $h(n)$ is the estimated cost remaining. Both complete and optimal when using an admissible heuristic.

2.3 Local Search Strategy

Hill Climbing

An iterative, greedy algorithm that maintains only the current state. At each step, it moves to the single best neighbor according to the heuristic. Does not maintain a search tree or enable backtracking.

3. Heuristic Design and Justification

For informed search algorithms (A*, Greedy, and Hill Climbing), we selected the **Manhattan Distance** as our heuristic function $h(n)$.

Formula:

$$h(n) = |x_{\text{current}} - x_{\text{goal}}| + |y_{\text{current}} - y_{\text{goal}}|$$

Justification for Selection:

Since the agent's movement is restricted to four cardinal directions without diagonal movement, the shortest path between two points is the sum of horizontal and vertical distances. Euclidean distance would underestimate the actual cost. Manhattan distance provides a tighter, more accurate estimate for grid-based movement.

Admissibility Condition:

Manhattan distance never overestimates the true minimal cost to reach the goal in a grid without obstacles. Therefore, it is an **admissible heuristic**, ensuring that our A* implementation remains optimal.

4. Experimental Methodology

Software Environment:

The simulation was developed using Python 3.x. The GUI was built using the standard Tkinter library, and comparative data visualization was generated using Matplotlib.

Testing Procedure:

We ran different algorithms on identical maze configurations (same start, goal, and obstacle placements) to ensure fair comparison.

Performance Metrics Tracked:

1. **Execution Time:** Real-world time taken to find a solution (in milliseconds)
2. **Space Complexity (Nodes Explored):** Total number of unique states visited and processed by the algorithm
3. **Path Cost:** Length (number of steps) of the final solution path found
4. **Optimality:** Qualitative check of whether the found path represents the shortest possible route

5. Results and Analysis

We conducted extensive testing to evaluate the performance of the implemented algorithms, highlighting significant trade-offs.

5.1 General Performance Comparison (BFS vs. DFS vs. A*)

The chart below illustrates performance differences on a typical maze with moderate obstacles.

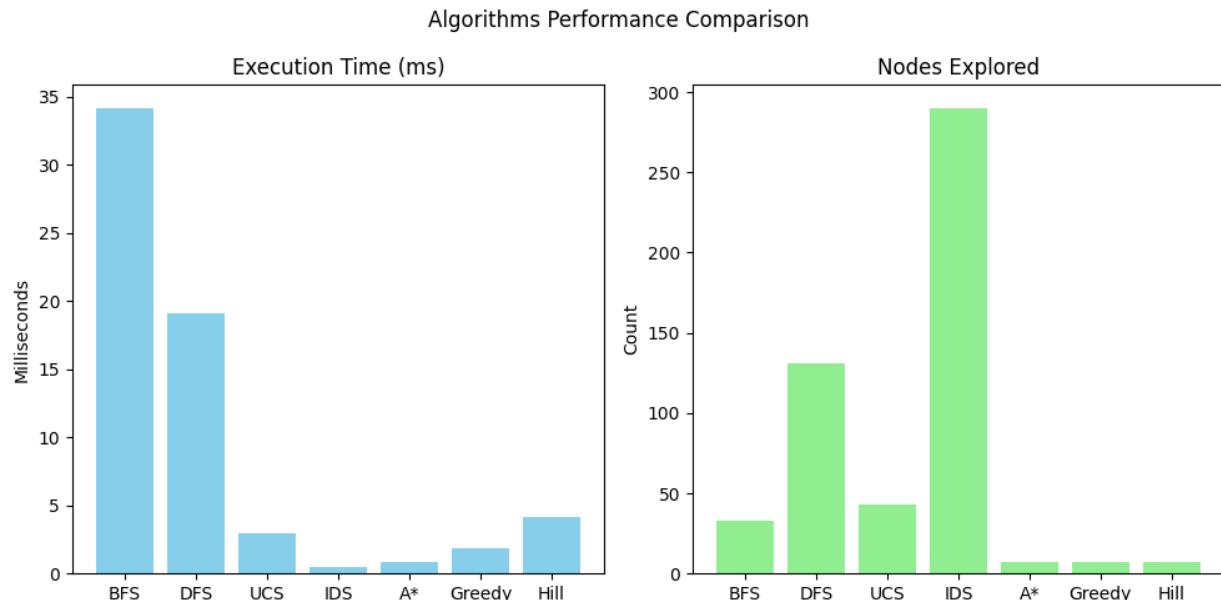


Figure 1: Performance comparison of key algorithms in terms of Execution Time (ms) and Nodes Explored

Analysis:

- **Optimality:** BFS and A* consistently found the optimal path (shortest cost). DFS frequently returned highly suboptimal, winding paths, sometimes 3-5× longer than necessary
- **Efficiency (Nodes Explored):** A* demonstrated superior efficiency by exploring significantly fewer nodes than BFS to find the same optimal path, proving the effectiveness of the Manhattan heuristic in guiding the search
- **Speed:** Greedy Search and A* were generally the fastest

5.2 Observations on Iterative Deepening Search (IDS)

We observed significant UI latency during the execution of IDS. This behavior reflects expected theoretical overhead. Since IDS regenerates the entire search tree from scratch for every new depth limit, it repeatedly visits and redraws the same nodes multiple times, causing high computational and visualization cost compared to single-pass algorithms like BFS.

5.3 The Failure of Local Search (Hill Climbing)

While extremely fast, Hill Climbing failed to find a path in mazes containing concave obstacles (U-shaped walls or dead-ends).

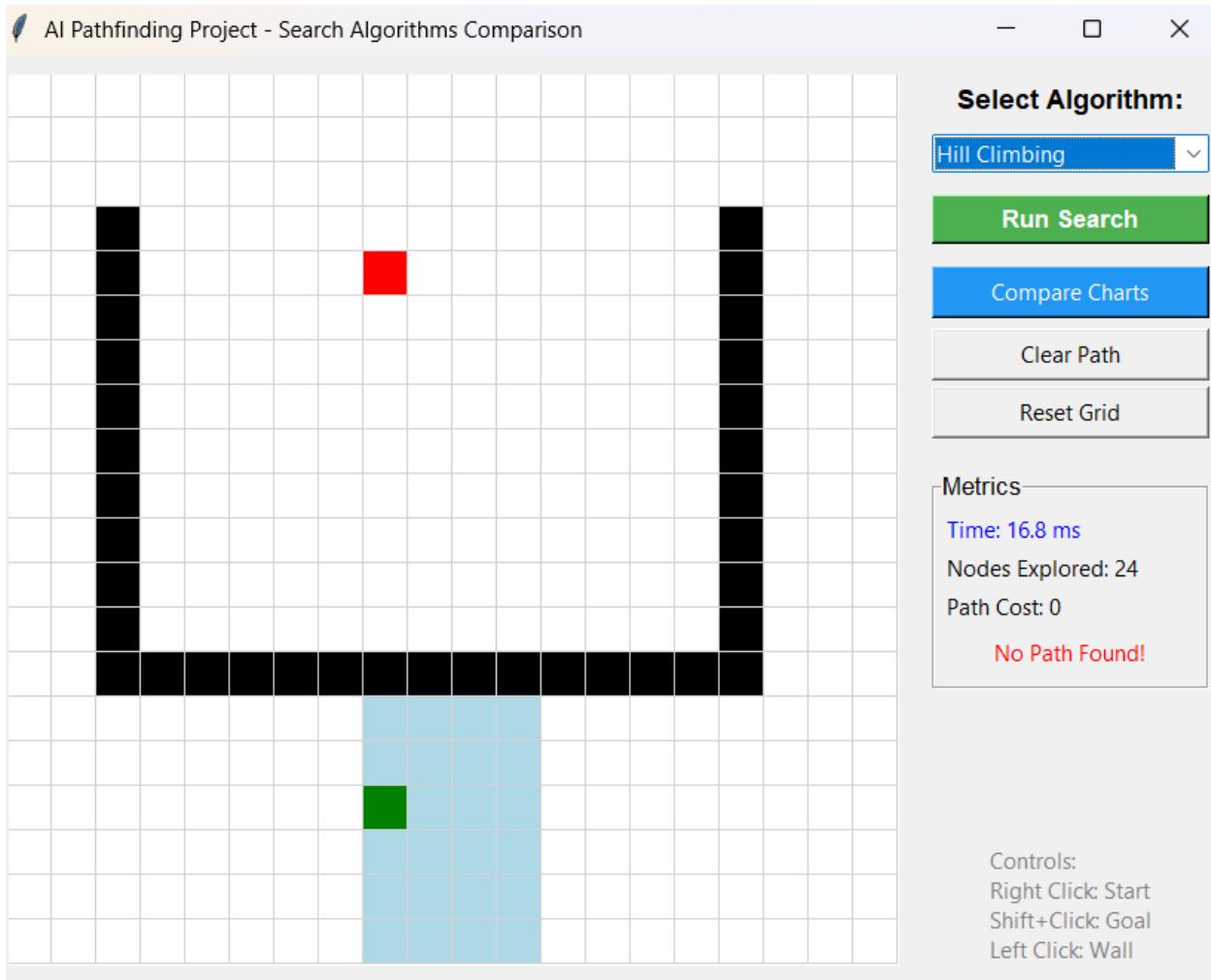


Figure 2: Hill Climbing algorithm stuck in a Local Optimum

Analysis:

As shown in Figure 2, the agent entered a state where all available neighbors had a higher heuristic cost (farther from the goal) than the current state. Without the ability to backtrack or maintain a search queue, the algorithm got stuck in a **local maximum** and terminated unsuccessfully, failing to complete the maze.

6. Conclusion

This project successfully developed a comprehensive tool for visualizing and analyzing AI search strategies. The experimental results confirm theoretical expectations:

1. A *Search** provides the best overall balance for this domain, offering guaranteed optimality with high efficiency due to the Manhattan heuristic
2. **Uninformed searches (BFS)** are reliable for finding shortest paths but scale poorly in terms of memory as grid size increases
3. **Local search (Hill Climbing)** is unsuitable for complex navigation due to its inability to escape local optima, despite its speed

7. References

1. Python Software Foundation. (n.d.). *Tkinter — Python interface to Tcl/Tk*. Retrieved December 2025, from <https://docs.python.org/3/library/tkinter.html>
2. Matplotlib Development Team. (n.d.). *Matplotlib: Visualization with Python*. Retrieved December 2025, from <https://matplotlib.org/>