University of Dhaka

Department of Computer Science and Engineering

CSE-4111 : Artificial Intelligence Lab

Lab Assignment: Comparative study among different searching algorithms

Submitted On:

May 02, 2025

Submitted To:

Dr. Md. Mosaddek Khan

Associate Professor
Department of Computer Science & Engineering
University of Dhaka

Submitted By:

Himel Chandra Roy

Roll No : 13

Registration No : 2020915623

Contents

1 Problem Statement

We are tasked with developing a pathfinding solution for a robot navigating a complex indoor environment, represented as an $n \times n$ 2D grid. The goal is to guide the robot from a designated starting point to a target endpoint, avoiding any blocked zones (obstacles), and minimizing traversal cost where applicable.

Each cell in the grid is defined by a 5-tuple of integers:

- **(0, w, x, y, z)**: The **starting cell**.
- **(1, w, x, y, z)**: The **ending cell**.
- **(3, -1, -1, -1, -1)**: An **obstacle cell** which is completely blocked and cannot be entered or exited.
- **(2, w, x, y, z)**: A **regular traversable cell**, where:
  - w is the cost to move **up** to the adjacent cell,
  - x is the cost to move **right**,
  - y is the cost to move **down**,
  - z is the cost to move **left**.

  All costs w, x, y, z are positive integers (≥ 1), unless a movement direction is blocked (i.e., adjacent to an obstacle), in which case the cost is -1.

Grid Constraints:

- Exactly **one** starting cell and **one** ending cell exist.
- Each pair of adjacent cells must have **symmetric costs**:
  - If cell (i, j) has x as the right movement cost, then cell $(i, j+1)$ must have $z = x$.
  - Similarly for up/down and left/right pairs.
- Obstacle cells are fully blocked in all directions and must be defined with all movement costs as -1.
- The grid must guarantee at least **one valid, cycle-free path** from the start to the end without revisiting any cell or entering obstacle cells.

Objective: Implement and compare multiple graph search algorithms to find a valid path from the starting cell to the ending cell. The algorithms must handle directional movement costs, avoid obstacles, and ensure no cell is visited more than once. Evaluation should include visual heatmaps of node visitation and comparative performance metrics including completeness, optimality, memory consumption, computational time, and total search space explored.

2 Algorithms Overview

2.1 Breadth-First Search (BFS)

Description: Explores all neighbors at the present depth prior to moving on to nodes at the next depth level.

Procedure:

- Use a queue to explore nodes level by level.
- Track visited nodes.
- Stop when the goal node is found.

Pseudocode:

```
function BFS(start, goal):
    queue = [start]
    visited = set()
    while queue:
        node = queue.pop(0)
        if node == goal:
            return path
        for neighbor in node.neighbors:
            if neighbor not in visited:
            visited.add(neighbor)
            queue.append(neighbor)
```

Evaluation:

- Completeness: Yes
- Optimality: Yes (if all step costs are equal). No (if the edge-costs are different)
- Memory: $O(b^d)$
- Time: $O(b^d)$
- Search Space: Exhaustive until the goal is found

2.2 Depth-First Search (DFS)

Description: Explores as far down one branch as possible before backtracking. **Procedure:**

- Start from the initial node and mark it as visited.
- Explore each branch as deeply as possible before backtracking.
- Use a stack (either explicitly or via recursion) to keep track of the path.
- Continue until the goal is found or all nodes are visited.

Pseudocode:

```
function DFS(start, goal):
    stack = [start]
    visited = set()
    while stack:
        node = stack.pop()
        if node == goal:
            return path
        if node not in visited:
            visited.add(node)
            for neighbor in reversed(node.neighbors):
                if neighbor not in visited:
                    stack.append(neighbor)
```

Evaluation:

- Completeness: No (infinite-depth graphs)
- Optimality: No
- Memory: $O(bd)$
- Time: $O(b^d)$
- Search Space: Deep but narrow first

2.3 Depth-Limited Search (DLS)

Description: DFS with a fixed depth limit to avoid infinite descent.

Procedure:

- Same as DFS but with a predefined maximum depth ℓ.
- Do not explore beyond this depth.
- Helps avoid infinite loops in infinite-depth graphs.

Pseudocode:

```
function DLS(node, goal, limit):
    if node == goal:
        return path
    elif limit == 0:
        return cutoff
    else:
        for neighbor in node.neighbors:
            result = DLS(neighbor, goal, limit - 1)
            if result != cutoff:
                return result
        return cutoff
```

Evaluation:

- Completeness: Yes (if depth limit $\geq$ goal depth)
- Optimality: No
- Memory: $O(b\ell)$
- Time: $O(b^{\ell})$
- Search Space: Limited to depth ℓ

2.4 Iterative Deepening Search (IDS)

Description: Repeatedly applies DLS with increasing depth limits.

Procedure:

- Repeatedly apply DLS, increasing the depth limit each time.
- Combines DFS's low memory with BFS's completeness and optimality.

Pseudocode:

```
function IDS(start, goal):
    depth = 0
    while True:
        result = DLS(start, goal, depth)
        if result != cutoff:
            return result
        depth += 1
```

Evaluation:

- Completeness: Yes
- Optimality: Yes (if step costs equal)
- Memory: $O(bd)$
- Time: $O(b^d)$
- Search Space: All paths up to goal depth

2.5 Uniform Cost Search (UCS)

Description: Expands the least-cost node first (Dijkstra's algorithm).

Procedure:

- Use a priority queue ordered by cumulative cost $g(n)$.
- Always expand the least-cost node first.
- Guarantees optimality if costs are non-negative.

Pseudocode:

```
function UCS(start, goal):
    frontier = PriorityQueue()
    frontier.put(start, 0)
    visited = set()
    while not frontier.empty():
        node = frontier.get()
        if node == goal:
            return path
        visited.add(node)
        for neighbor in node.neighbors:
            if neighbor not in visited:
                total_cost = cost_to(node) + cost(node, neighbor)
                frontier.put(neighbor, total_cost)
```

Evaluation:

- Completeness: Yes
- Optimality: Yes
- Memory: High ($O(b^d)$)
- Time: $O(b^{1+\lfloor C^*/\varepsilon \rfloor})$
- Search Space: Cost-based expansion

2.6 Best-First Search

Description: Uses a heuristic to prioritize nodes closest to the goal.

Procedure:

- Use a heuristic function $h(n)$ to estimate cost to goal.
- Expand the node with the lowest heuristic value first.

Pseudocode:

```
function BestFirst(start, goal):
    frontier = PriorityQueue()
    frontier.put(start, h(start))
    visited = set()
    while not frontier.empty():
        node = frontier.get()
        if node == goal:
            return path
        visited.add(node)
        for neighbor in node.neighbors:
            if neighbor not in visited:
                frontier.put(neighbor, h(neighbor))
```

Evaluation:

- Completeness: No
- Optimality: No
- Memory: High
- Time: Depends on heuristic
- Search Space: Heuristic-guided

2.7 A* Search

Description: Combines UCS and heuristic information: $f(n) = g(n) + h(n)$.

Procedure:

- Use $f(n) = g(n) + h(n)$, combining path cost and heuristic.
- Expand the node with the lowest $f(n)$.
- Guaranteed optimal if $h(n)$ is admissible (never overestimates).

Pseudocode:

```
function AStar(start, goal):
    frontier = PriorityQueue()
    frontier.put(start, g(start) + h(start))
    visited = set()
    while not frontier.empty():
        node = frontier.get()
        if node == goal:
            return path
        visited.add(node)
        for neighbor in node.neighbors:
            if neighbor not in visited:
                g_cost = g(node) + cost(node, neighbor)
                f_cost = g_cost + h(neighbor)
                frontier.put(neighbor, f_cost)
```

Evaluation:

- Completeness: Yes
- Optimality: Yes (if h is admissible)
- Memory: High
- Time: Exponential in worst case
- Search Space: Balanced cost and heuristic

2.8 Weighted A* Search

Description: Variant of A* with weighted heuristic: $f(n) = g(n) + w \cdot h(n)$.

Procedure:

- Use $f(n) = g(n) + w \cdot h(n)$ where $w > 1$ to favor heuristics more.
- Faster than A* but may not be optimal.

Pseudocode:

```
function WeightedAStar(start, goal, w):
    frontier = PriorityQueue()
    frontier.put(start, g(start) + w * h(start))
    visited = set()
    while not frontier.empty():
        node = frontier.get()
        if node == goal:
            return path
        visited.add(node)
        for neighbor in node.neighbors:
            if neighbor not in visited:
                g_cost = g(node) + cost(node, neighbor)
                f_cost = g_cost + w * h(neighbor)
                frontier.put(neighbor, f_cost)
```

Evaluation:

- Completeness: Yes (if $w < \infty$)
- Optimality: No (unless $w = 1$)
- Memory: High
- Time: Faster than A* with high w
- Search Space: Heuristic-biased

3 Results

3.1 Execution time and memory usage

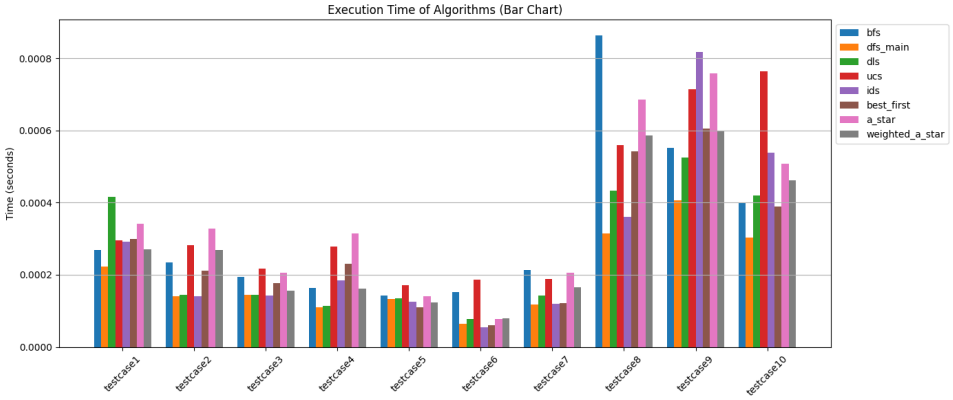

Figure 1: Execution time of different algorithm for different testcases

Figure 1 shows the execution time of all these algorithms for different test cases. These test cases are sorted according to the ascending order of the value n described in the problem statement.
The following figure shows the memory usage of the algorithms for the same set of test cases.

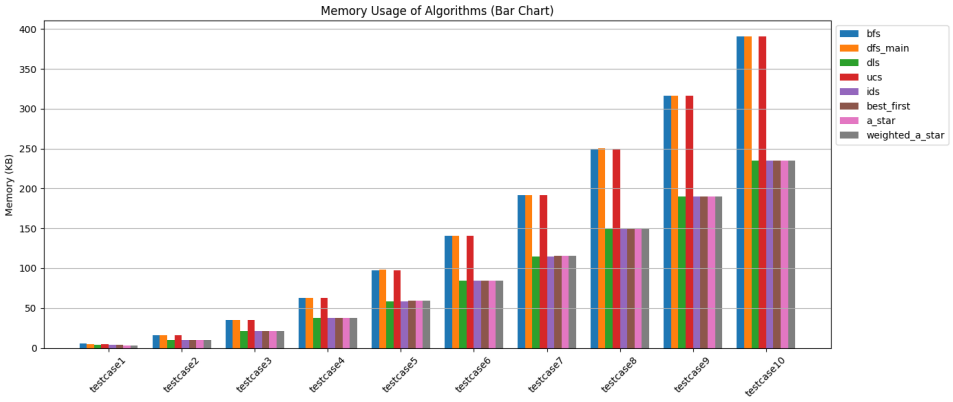

Figure 2: Memory usage of different algorithms for different testcases

3.2 Path Finding

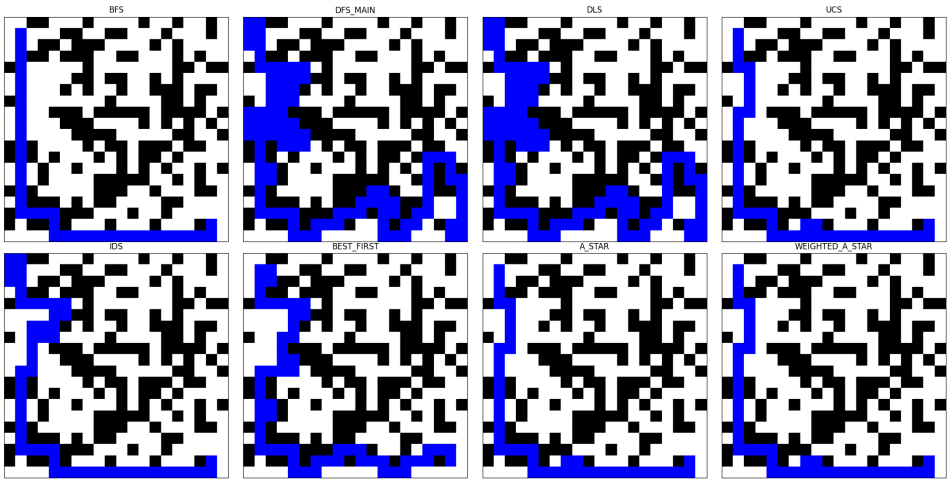

Figure 3: Path finding of different algorithms

Figure 3 shows the grid for testcase - 3. the blue line is the path given by each algorithms. The black dots are the obstacles and the white spaces are possible pathways.

3.3 Frequency heat-map

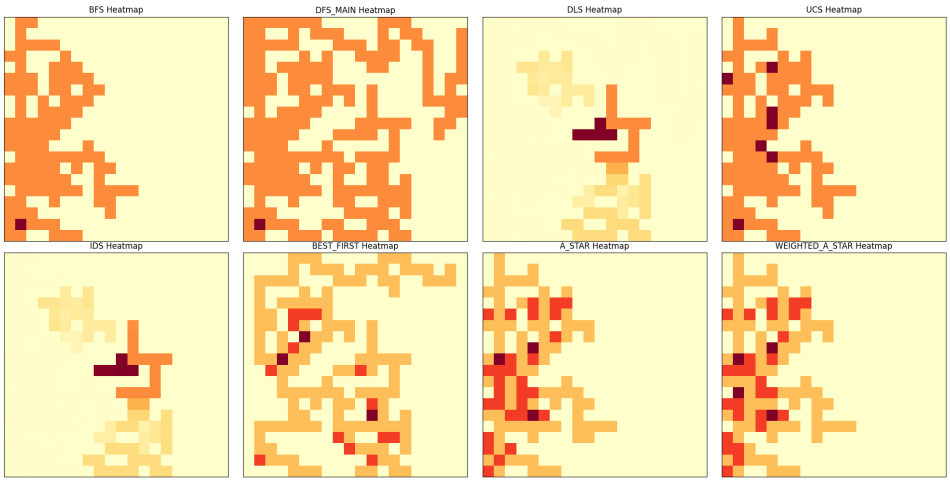

Figure 4: Path finding of different algorithms

Figure 4 presents heatmaps for eight different pathfinding algorithms: BFS, DFS, DLS, UCS, IDS, Best-First Search, A*, and Weighted A*. Each subplot corresponds to one algorithm and illustrates the frequency with which each cell in the 30x30 grid was visited during the search process.

The color intensity in each cell represents how many times that cell was explored or revisited, with a gradient from light yellow (fewer visits) to deep red (more visits). This visualization highlights differences in exploration strategies—such as the broad, level-wise expansion of BFS, the deep and sometimes redundant searching of DFS and IDS, and the more directed search patterns of A*, Best-First, and UCS. Areas with intense red

indicate zones of heavy search activity, often caused by loops, dead-ends, or heuristic uncertainty.

These heatmaps provide insight into each algorithm's efficiency and search behavior, helping compare their exploration patterns beyond just the final path found.

4 Conclusion

In this study, we have analyzed and implemented a variety of classical and heuristic-based graph search algorithms, including Breadth-First Search (BFS), Depth-First Search (DFS), Depth-Limited Search (DLS), Iterative Deepening Search (IDS), Uniform Cost Search (UCS), Best-First Search, A* Search, and Weighted A* Search. Each algorithm was evaluated based on five key criteria: completeness, optimality, memory usage, computational time, and search space coverage.

Our experiments demonstrate that the choice of algorithm significantly impacts performance depending on the structure and constraints of the environment. Algorithms like BFS and IDS offer completeness and, in some cases, optimality, but at the cost of higher memory usage. On the other hand, DFS and DLS are more memory-efficient but risk incompleteness and suboptimal paths. Heuristic-based methods such as A* and Weighted A* effectively balance cost and directionality, especially when admissible heuristics are used.

Visualization via node visitation heatmaps also provided critical insight into how different algorithms traverse the search space. Algorithms like A* and UCS tend to explore more promising nodes first, whereas uninformed strategies cover broader regions before finding the goal.

Overall, no single algorithm is universally superior; rather, the best choice depends on the specific requirements of the problem space, such as available memory, need for optimality, and knowledge of the domain in the form of heuristics.