Tanta University

Faculty of Computer science and information

Maze problem report

Submitted by:

Eman Ali Ragab Kotb

Shahenda Farag Zaki

Esraa Yehia Mohamed

Mohamed Salah Shehata

Habiba Tamer Elmowafy

Under Guinness of:

Eng / Omar Khaled

2025

Depth-First Search (DFS)

1. Algorithm Description

Depth-First Search (DFS) is **an uninformed search algorithm** that explores the search space by going as deep as possible along a single path before backtracking. The algorithm does not use any heuristic information about the goal state. DFS typically uses a **stack-based** approach, either explicitly with a stack or via recursion.

2. Problem Application:

The goal is to find a path for an agent in a grid-based maze from a start position to a goal position, avoiding obstacles.

- **State Representation**

  Each state is represented as the agent's position **(x, y)**, where:

  - **x** = row index **y** = column index

- **Initial State**

  The starting position of the agent.

- **Goal State**

  The target cell to reach.

- **Actions**

  The agent can move:

  - Up - Down

  - Left - Right

- **A move is valid only if:**

  The new position is inside the maze boundaries
  The cell is not an obstacle
  The cell has not been visited before

3. Cost Function

Each move has a **uniform cost of 1**.

Total path cost = **number of steps** in the final path found.

4. DFS Algorithm Steps

1. Initialize a stack with the start state and its path [start].
2. Mark all states as unvisited.
3. While the stack is not empty:
   a) Pop a state (x, y) and its path.
   b) If (x, y) is the goal, return the path.
   c) If not visited, mark it as visited.
   d) Generate all valid neighbors and push them to the stack with updated paths.
4. Return failure if the goal is not reached.

5. DFS Characteristics in Maze Solver

Property	DFS Behavior
Completeness	Not guaranteed (may get stuck in deep paths)
Optimality	Not optimal
Time Complexity	O(b^m)
Space Complexity	O (b × m)
Memory Usage	Low
Path Cost	May be higher than optimal

6. Python Implementation

```python
def dfs():
    stack = [(start, [start])]
    visited = set()
    nodes_explored = 0

    while stack:
        current, path = stack.pop()
        nodes_explored += 1

        if current == goal:
            return path, nodes_explored

        if current in visited:
            continue

        visited.add(current)

        for n in get_neighbors(current):
            stack.append((n, path + [n]))

    return None, nodes_explored
```

➢ Maze Representation

- **0** → free cell
- **1** → obstacle

➢ Maze

maze = [
    [0, 0, 0, 0, 1],
    [1, 1, 0, 0, 1],
    [0, 0, 0, 1, 0],
    [0, 1, 0, 0, 0],
    [0, 0, 0, 1, 0]]

- **start =** (0, 0)
- **goal =** (4, 4)

7. Experimental Results

```
After applying DFS:
Path found:
[(0, 0), (0, 1), (0, 2), (0, 3), (1, 3), (1, 2), (2, 2), (2, 1), (2, 0), (3, 0), (4, 0), (4, 1), (4, 2), (3, 2),
(3, 3), (3, 4), (4, 4)]

Path Length: 16
Total Path Cost: 16
Nodes Explored: 23
Execution Time: 0.09 ms
```

8. Analysis of DFS Performance

- DFS successfully finds a path from start to goal if one exists.
- Explores deep paths first, which may result in longer paths and unnecessary exploration.
- Uses less memory compared to BFS and UCS.
- Suitable when memory is limited and optimality is not required.
- Limitation: DFS does not guarantee the shortest path; BFS or A* is better for optimal solutions.

Uniform Cost Search (UCS)

1. Algorithm Description

Uniform Cost Search (UCS) is an uninformed search algorithm that expands the node with the lowest cumulative path cost first. Unlike DFS, UCS guarantees finding the optimal (least-cost) path when all step costs are non-negative.
UCS typically uses a priority queue ordered by path cost.

2. Problem Application: Maze Solver

The goal is to find the **least-cost path** for a agent in a grid-based maze from a start position to a goal position, avoiding obstacles.

- **State Representation**

  Each state is represented as the agent's position **(x, y)**, where:
  - **x** = row index **y** = column index
- **Initial State**

  The starting position of the agent.
- **Goal State**

  The target cell is to reach.
- **Actions**

  The agent can move:

  - Up - Down
  - Left - Right
- **A move is valid only if:**

  The new position is inside the maze boundaries

  The cell is not an obstacle

  The cell has not been visited with a lower cost before

3. Cost Function

- Each move has a **uniform cost of 1**
- Total path cost = **sum of all move costs**

4. UCS Algorithm Steps

1. Initialize a **priority queue** with the start state, its path, and cost = 0.
2. Initialize a dictionary to store the **minimum cost** to reach each state.
3. While the priority queue is not empty:
   a) Pop the state with the **lowest path cost**.
   b) If the state is the goal, return the path and cost.
   c) Generate all valid neighbors.
   d) For each neighbor, calculate the new cost.
   e) If the neighbor has not been visited before or a lower cost is found, update it and push it into the queue.
4. Return failure if the goal is not reached.

5. UCS Characteristics in Maze Solver

Property	UCS Behavior
Completeness	Guaranteed
Optimality	Guaranteed
Time Complexity	O(b^m)
Space Complexity	O(b^m)
Memory Usage	High
Path Cost	Optimal (minimum)

6. Python Implementation

```python
def ucs():
    pq = [(0, start, [start])]
    visited = {}
    nodes_explored = 0

    while pq:
        cost, current, path = heapq.heappop(pq)
        nodes_explored += 1

        if current == goal:
            return path, nodes_explored

        if current in visited and visited[current] <= cost:
            continue

        visited[current] = cost

        for n in get_neighbors(current):
            heapq.heappush(pq, (cost + 1, n, path + [n]))

    return None, nodes_explored
```

➢ Maze Representation

- **0** → free cell

- **1** → obstacle

➢ Maze

maze = [

[0, 0, 0, 0, 1],

[1, 1, 0, 0, 1],

[0, 0, 0, 1, 0],

[0, 1, 0, 0, 0],

[0, 0, 0, 1, 0]]

- **start =** (0, 0)

- **goal =** (4, 4)

7. Experimental Results

After applying UCS:

- **Path found:**
  [(0, 0), (0, 1), (0, 2), (1, 2), (2, 2), (3, 2), (3, 3), (3, 4), (4, 4)]
- **Path Length:** 8
- **Total Path Cost:** 8
- **Nodes Explored:** 35
- **Execution Time:** 0.06 ms

8. Analysis of UCS Performance

- UCS always finds the **shortest path in terms of cost**.
- Explores nodes systematically based on cumulative cost.
- Uses more memory than DFS due to priority queue storage.
- More efficient than BFS when costs vary.
- Suitable when **optimality is required**.
- Limitation: High memory usage for large mazes.

A* Search Algorithm (A-Star)

1. Algorithm Description

A* Search is an **informed search algorithm** that expands the node with the lowest evaluation function value

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

Where:

- **g(n)**: cost from the start node to the current node
- **h(n)**: heuristic estimate of the cost from the current node to the goal

A* combines the advantages of Uniform Cost Search and Greedy Best-First Search and guarantees finding the optimal path if the heuristic is admissible (never overestimates the true cost).

A* typically uses a **priority queue** ordered by the f(n) value.

2. Problem Application: Maze Solver

The goal is to find the **least-cost path** for a agent in a grid-based maze from a start position to a goal position while avoiding obstacles.

- **State Representation**
  Each state is represented as the agent's position **(x, y)**, where:
  - **x** = row index
  - **y** = column index
- **Initial State**
  The starting position of the agent.
- **Goal State**
  The target cell to reach.

- **Actions**

  The agent can move:

  - Up
  - Down
  - Left
  - Right

- **A move is valid only if:**

  The new position is inside the maze boundaries

  The cell is not an obstacle

  The cell has not been visited with a lower **f-cost** before

3. Cost Function

- Each move has a uniform cost of **1**
- **g(n)** = total cost from start to current node
- **h(n)** = Manhattan Distance to the goal

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

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

4. A* Algorithm Steps

1. Initialize a priority queue with the start state, its path, **g = 0**, and **f = h(start)**.
2. Initialize a dictionary to store the minimum **g-cost** for each state.
3. While the priority queue is not empty:

   a) Pop the state with the lowest **f(n)** value.

   b) If the state is the goal, return the path and cost.

   c) Generate all valid neighboring states.

   d) For each neighbor, calculate **g(n)** and **f(n)**.

   e) If the neighbor has not been visited before or a lower cost is found, update it and push it into the queue.
4. Return failure if the goal is not reached.

5. A* Characteristics in Maze Solver

Property	A* Behavior
Completeness	Guaranteed
Optimality	Guaranteed (with admissible heuristic)
Time Complexity	O(b^m)
Space Complexity	O(b^m)
Memory Usage	High
Path Cost	Optimal (minimum)

6. Python Implementation

```python
def astar():
    pq = [(0, start, [start])]
    visited = {}
    nodes_explored = 0

    while pq:
        f, current, path = heapq.heappop(pq)
        nodes_explored += 1
        g = len(path) - 1

        if current == goal:
            return path, nodes_explored

        if current in visited and visited[current] <= g:
            continue

        visited[current] = g

        for n in get_neighbors(current):
            h = manhattan(n, goal)
            heapq.heappush(pq, (g + 1 + h, n, path + [n]))

    return None, nodes_explored
```

➢ **Maze Representation**

- **0** → free cell
- **1** → obstacle

➢ **Maze**

```
maze = [
    [0, 0, 0, 0, 1],
    [1, 1, 0, 0, 1],
    [0, 0, 0, 1, 0],
    [0, 1, 0, 0, 0],
    [0, 0, 0, 1, 0]]
```

- **start =** (0, 0)
- **goal =** (4, 4)

7. Experimental Results

After applying **A***:

➢ **Path found:**

[(0, 0), (0, 1), (0, 2), (1, 2), (2, 2), (3, 2), (3, 3), (3, 4), (4, 4)]

➢ **Path Length:** 8

➢ **Total Path Cost:** 8

➢ **Nodes Explored:** 13

➢ **Execution Time:** 0.09 ms

8. Analysis of A* Performance

- A* always finds the shortest path when the heuristic is admissible.
- Explores fewer nodes compared to UCS due to heuristic guidance.
- Faster than UCS and BFS in large mazes.
- Uses more memory because it stores open and closed lists.
- Highly suitable for path-finding problems where optimality and efficiency are required.
- Limitation: Performance depends heavily on the quality of the heuristic.

Iterative Deepening Search (IDS)

1. Algorithm Description

Iterative Deepening Search (IDS) is an **uninformed search algorithm** that combines the advantages of Depth-First Search (DFS) and Breadth-First Search (BFS).
IDS explores the search space by performing a **Depth-Limited Search (DLS)** repeatedly, starting with a depth limit of zero and gradually increasing the limit until the goal state is found.
At each iteration, IDS performs a DFS up to the current depth limit.
This allows the algorithm to find the **shallowest (shortest) solution** while using **low memory**.

2. Problem Application

The objective is to find a valid path for an agent in a grid-based maze from a start position to a goal position while avoiding obstacles.

➢ **State Representation**

Each state is represented as the agent's position **(x, y)**, where:

- **x** = row index **y** = column index

➢ **Initial State**

The starting position of the agent in the maze.

➢ **Goal State**

The target cell that the agent must reach.

➢ **Actions**

The agent can move:

- Up
- Down
- Left
- Right

- **A move is considered valid only if:**

The new position is within the maze boundaries

The cell is not an obstacle

The cell has not been visited in the current path

3. Cost Function

- Each move in the maze has a **uniform cost of 1**.
- The total path cost is equal to the **number of steps** taken to reach the goal state.

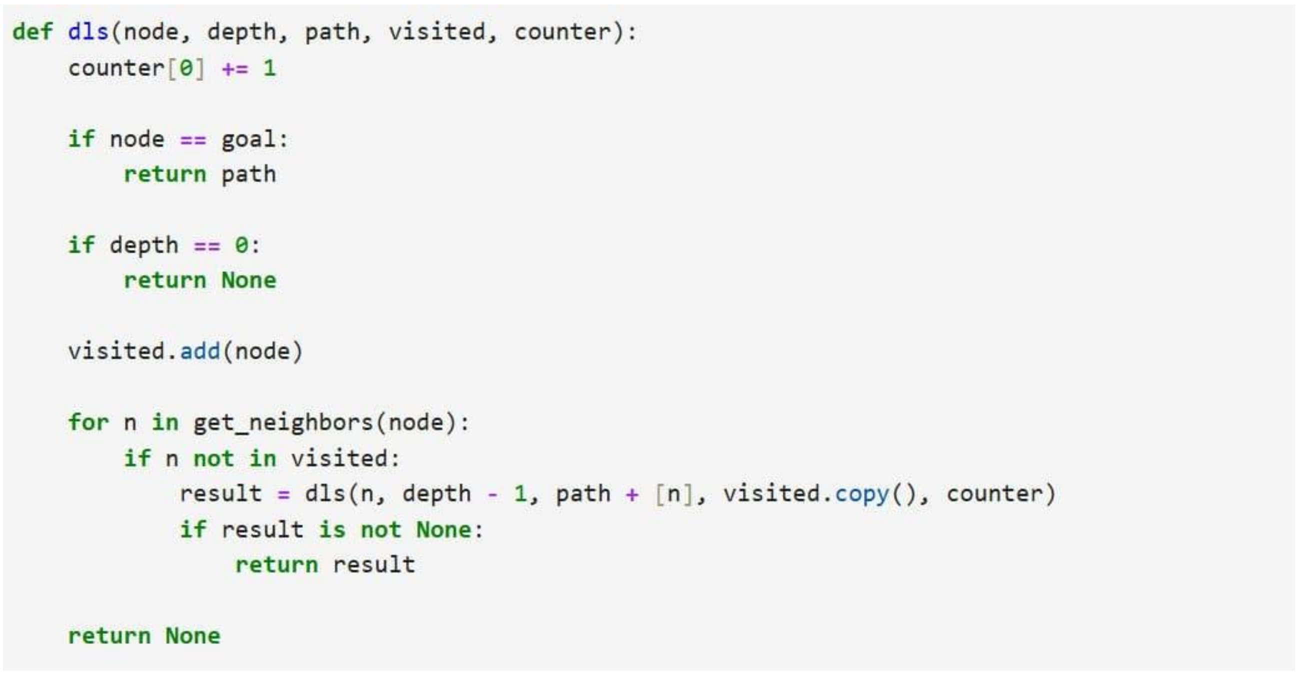
4. IDS Algorithm Steps

1. Initialize the depth limit to **0**.
2. Perform a **Depth-Limited Search (DLS)** from the start state.
3. If the goal state is found, return the path.
4. If the goal is not found, increase the depth limit by **1**.
5. Repeat the process until the goal is reached or the search space is exhausted.

5.IDS Characteristics in Maze Solver

Property	IDS Behavior
Completeness	Guaranteed if a solution exists
Optimality	Optimal for uniform step cost
Time Complexity	O(b^m)
Space Complexity	O(b × m)
Memory Usage	Low
Path Cost	Shortest path

6. Python Implementation



```python
def dls(node, depth, path, visited, counter):
    counter[0] += 1

    if node == goal:
        return path

    if depth == 0:
        return None

    visited.add(node)

    for n in get_neighbors(node):
        if n not in visited:
            result = dls(n, depth - 1, path + [n], visited.copy(), counter)
            if result is not None:
                return result

    return None
```

➢ **Maze Representation**

- **0** → free cell
- **1** → obstacle

➢ **Maze**

maze = [
    [0, 0, 0, 0, 1],
    [1, 1, 0, 0, 1],
    [0, 0, 0, 1, 0],
    [0, 1, 0, 0, 0],
    [0, 0, 0, 1, 0]]

- **start =** (0, 0)
- **goal =** (4, 4)

7. Experimental Results

After applying IDS to the maze:

- **Path found:**
  [(0, 0), (0, 1), (0, 2), (1, 2), (2, 2), (3, 2), (3, 3), (3, 4), (4, 4)]
- **Path Length**: 8
- **Total Path Cost**: 8
- **Nodes Explored**: 81
- **Execution Time**: 0.22 ms

8. Analysis of IDS Performance

- IDS successfully finds a path from the start state to the goal state if **one exists**.
- The algorithm guarantees the shortest path due to its iterative depth increase.
- Uses significantly **less memory** than BFS.
- Repeated exploration causes extra time overhead compared to BFS.
- IDS is suitable for maze problems where **memory efficiency** is important and optimal solutions are required.

Breadth-First Search (BFS)

1. Algorithm Description

Breadth-First Search (BFS) is **an uninformed search algorithm** that explores the search space level by level. It expands all nodes at the current depth before moving to the next depth level. BFS does not use any heuristic information and typically uses a **queue** data structure.

2. Problem Application: Maze Solver

The objective is to find a valid and **shortest path** for an agent in a grid-based maze from a start position to a goal position while avoiding obstacles.

- **State Representation**

  Each state is represented as the agent's position **(x, y)**, where:

  - **x** = row index **y** = column index

- **Initial State**

  The starting position of the agent in the maze.

- **Goal State**

  The target cell that the agent must reach.

- **Actions**

  The agent can move:

  - Up - Down

  - Left - Right

- **A move is valid only if:**

  The new position is inside the maze boundaries

  The cell is not an obstacle

The cell has not been visited before

3. Cost Function

- Each move has a **uniform cost of 1.**
- Total path cost = **number of steps in the final path found.**

4. BFS Algorithm Steps

1. Initialize a **queue** with the start state and its path.

2. Mark the start state as visited.

3. While the queue is not empty:

    a) Dequeue the front state.

    b) If the current state is the goal, return the path.

    c) Generate all valid neighboring states.

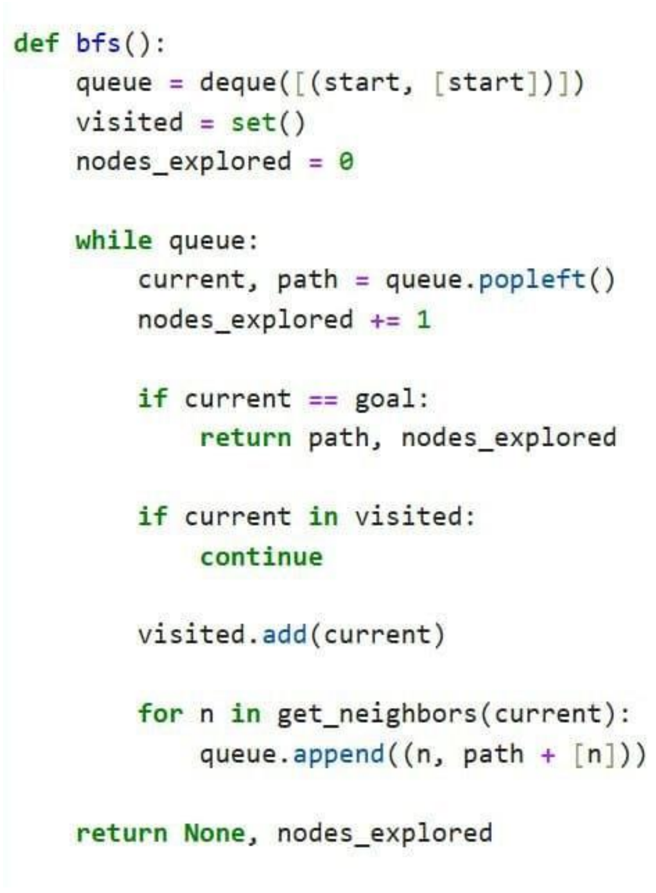
    d) For each valid neighbor that has not been visited, mark it as visited and enqueue it with the updated path.

4. Return failure if the goal is not reached.

5. BFS Characteristics in Maze Solver

Property	BFS Behavior
Completeness	Guaranteed
Optimality	Guaranteed (with equal step cost)
Time Complexity	O(b^m)
Space Complexity	O(b^m)
Memory Usage	High
Path Cost	Optimal (shortest path)

6. Python Implementation



```python
def bfs():
    queue = deque([(start, [start])])
    visited = set()
    nodes_explored = 0

    while queue:
        current, path = queue.popleft()
        nodes_explored += 1

        if current == goal:
            return path, nodes_explored

        if current in visited:
            continue

        visited.add(current)

        for n in get_neighbors(current):
            queue.append((n, path + [n]))

    return None, nodes_explored
```

- **Maze Representation**

  - **0** → free cell — - **1** → obstacle

- **Maze**

  maze = [

  [0, 0, 0, 0, 1],

  [1, 1, 0, 0, 1],

  [0, 0, 0, 1, 0],

  [0, 1, 0, 0, 0],

  - **start =** (0, 0)
  - **goal =** (4, 4)

```
        [0, 0, 0, 1, 0]]
```

7. Experimental Results

After applying BFS:

- **Path found:**
  [(0, 0), (0, 1), (0, 2), (1, 2), (2, 2), (3, 2), (3, 3), (3, 4), (4, 4)]
- **Path Length:** 8
- **Total Path Cost:** 8
- **Nodes Explored:** 32
- **Execution Time:** 0.09 ms

8. Analysis of BFS Performance

- BFS successfully finds the **shortest path** from the start state to the goal state.
- It guarantees **optimal solutions** when all moves have **equal cost**.
- BFS explores a large number of nodes, which leads to high memory usage.
- Compared to DFS, BFS produces better (shorter) paths but consumes more memory.
- BFS is **suitable for maze problems** where **optimality is required** and the maze size is **manageable**.

Algorithm Comparison Based on Evaluation Metrics:

Algorithm	Number of Explored Nodes	Execution Time	Path Length	Ability to Reach the Goal	Solution Optimality
Depth-First Search (DFS)	low	Fast	Not guaranteed shortest	Yes (may fail in some cases)	No
Breadth-First Search (BFS)	High	Slow in large search spaces	Shortest path	Yes	Yes
Uniform-Cost Search (UCS)	Medium to High	Slower due to cost calculations	Least-cost path	Yes	Yes
Iterative Deepening Search (IDS)	Medium	Medium	Shortest path	Yes	Yes
A*Search (Manhattan Distance)	Low	Fastest	Shortest path	Yes	Yes

Conclusion

Among the evaluated algorithms, **A* Search with the Manhattan Distance heuristic proved to be the most effective** for solving the maze problem.
It explored the fewest nodes, required less execution time, and consistently found the shortest path to the goal.
Unlike BFS and IDS, which explore many unnecessary states, and DFS, which does not guarantee an optimal solution, A* efficiently guides the search toward the goal using heuristic information.