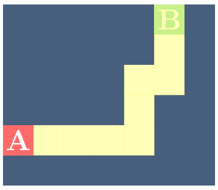
Effat University • Computer Science Department

CS3081: Artificial Intelligence

Spring 2026

Lab 1: Maze Solver

Depth-First Search (DFS) Implementation



Instructor: Dr. Naila Marir
Lab Session: [Date/Time]
Duration: 2 hours

1 Overview

This lab implements a maze-solving algorithm using **Depth-First Search (DFS)** with a stack-based frontier. The program reads a text-based maze, finds a path from start (`A`) to goal (`B`), and visualizes the solution.

What You Will Learn

- How DFS explores a search space using a **stack** (LIFO)
- The difference between DFS and BFS exploration patterns
- State space representation using **nodes**, **states**, and **actions**
- Frontier management and explored set tracking
- Path reconstruction using **parent pointers**

2 Project Structure

Your lab folder should contain the following files:

File	Description
`maze.py`	Main Python program containing the maze solver
`maze1.txt`	Small simple maze (6×7)
`maze2.txt`	Large complex maze (16×29)
`maze3.txt`	Another test maze
`requirements.txt`	Dependencies (pillow for image generation)
`maze.png`	Output visualization (generated after running)

3 Maze File Format

Mazes are stored as text files with the following format:

```
#####B#
##### #
####  #
#### ##
     ##
A######
```

Symbol	Meaning
`#`	Wall (impassable)
`(space)`	Open path (can walk through)
`A`	Start position (exactly one required)
`B`	Goal position (exactly one required)

4 Key Components

4.1 Node Class

maze.py: Lines 3-7

```python
class Node():
    def __init__(self, state, parent, action):
        self.state = state
        self.parent = parent
        self.action = action
```

The `Node` class represents a state in the search space:

Node

- **state**: Current position (row, col)
- **parent**: Previous node in the path
- **action**: Move taken ("up", "down", "left", "right")

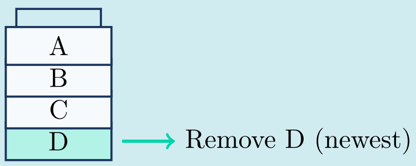
4.2 StackFrontier Class (DFS)

maze.py: Lines 10-29

```
class StackFrontier():
    def __init__(self):
        self.frontier = []

    def add(self, node):
        self.frontier.append(node)

    def contains_state(self, state):
        return any(node.state == state for node in self.frontier)

    def empty(self):
        return len(self.frontier) == 0

    def remove(self):
        if self.empty():
            raise Exception("empty frontier")
        else:
            node = self.frontier[-1]       # Get LAST item
            self.frontier = self.frontier[:-1]  # Remove it
            return node
```

Why Stack = Depth-First Search

- **LIFO** (Last-In, First-Out): The most recently added node is removed first
- This causes the algorithm to go **deep** before going wide
- Like exploring one tunnel completely before trying another



Stack (LIFO)

4.3 QueueFrontier Class (BFS)

maze.py: Lines 32-40

```
class QueueFrontier(StackFrontier):
    def remove(self):
        if self.empty():
            raise Exception("empty frontier")
        else:
            node = self.frontier[0]          # Get FIRST item
            self.frontier = self.frontier[1:]    # Remove it
            return node
```

Why Queue = Breadth-First Search

- **FIFO** (First-In, First-Out): The oldest node is removed first
- This causes the algorithm to go **wide** before going deep
- Explores all neighbors at current depth before moving deeper
- **Note:** This class exists but is NOT used in the current code

Remove A (oldest) → | A | B | C | D |

Queue (FIFO)

4.4 Maze Class

The `Maze` class (lines 42-215) contains the main logic:

Method	Description
`__init__()`	Parses maze file, validates start/goal, builds wall matrix
`neighbors()`	Returns valid adjacent cells (up/down/left-/right)
`solve()`	Main DFS algorithm (lines 119-164)
`print()`	Displays maze in terminal with solution path marked as `*`
`output_image()`	Creates PNG visualization

5 How the Algorithm Works

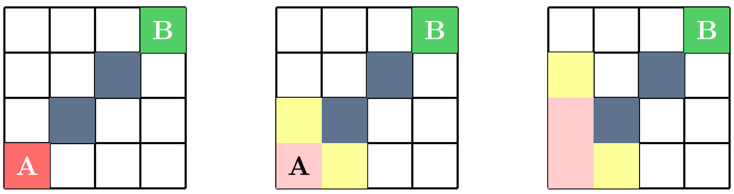
The `solve()` method (lines 119-164) implements DFS:

DFS Algorithm Pseudocode

```
1. Initialize frontier with start node
2. Initialize explored set as empty

3. LOOP until solution found:
   a. If frontier is empty -> No solution exists

   b. Remove node from frontier (LIFO = depth-first)

   c. If node is the goal -> Solution found!
      - Backtrack through parent pointers
      - Reconstruct and return the path

   d. Mark node as explored

   e. For each neighbor of the node:
      - If not explored AND not in frontier:
        - Add neighbor to frontier
```

5.1 Visual Example

Step 1: Start **Step 2: Expand A** **Step 3: Go Deep**



Frontier: [A] Frontier: [right, up] DFS goes UP first

6 Running the Program

6.1 Installation

Navigate to your lab folder and install dependencies:

```
cd Lab1
pip install -r requirements.txt
```

6.2 Execution

Run the maze solver with different maze files:

```
python maze.py maze1.txt
python maze.py maze2.txt
python maze.py maze3.txt
```

6.3 Output

The program produces:

1. **Terminal Output:**
   - Displays the maze before solving
   - Displays the maze after solving (with * showing the path)
   - Reports the number of states explored
2. **Image Output (`maze.png`):**
   - Red = Start position
   - Green = Goal position
   - Yellow = Solution path
   - Salmon = Explored states

Start (A)	Goal (B)	Solution Path
Explored	Wall	Open Path

7 Sample Terminal Output

Before Solving:

```
#####B#
##### #
####  #
#### ##
     ##
A######
```

After Solving:

```
#####B#
#####*#
####**#
####*##
***  ##
A######

States Explored: 10
```

8 Learning Objectives

After completing this lab, you should understand:

1. **Graph Search Algorithms:** The difference between DFS (stack) and BFS (queue)
2. **Data Structures:** How stack vs queue affects search behavior
   - Stack (LIFO) → Depth-First Search
   - Queue (FIFO) → Breadth-First Search
3. **State Space Representation:**

- Nodes contain state, parent, and action
- States are positions (row, col)
- Actions are movements (up, down, left, right)

4. **Frontier Management:** How the frontier stores nodes to be explored
5. **Explored Set:** Why we track visited nodes (to avoid infinite loops)
6. **Path Reconstruction:** Using parent pointers to trace back the solution

9 Discussion Questions

Discussion Questions

1. **Why does the current implementation use `StackFrontier` instead of `QueueFrontier`?**

2. **What would happen if you changed line 127 to use `QueueFrontier()`?**
   - How would the exploration pattern change?
   - Would it find the same path?
   - Would it explore more or fewer states?

3. **Is DFS guaranteed to find the shortest path? Why or why not?**

4. **How does the number of explored states differ between `maze1.txt` and `maze2.txt`?**
   - Run both and compare the "States Explored" output
   - Why is there a difference?

5. **What would happen if we removed the explored set check?**
   - Hint: Think about cycles in the maze

10 Lab Exercises

Complete the following exercises:

△ Important

Exercise 1: Run and Observe

1. Run the program with all three maze files
2. Record the number of states explored for each
3. Compare the solution paths in the generated images

△ Important

Exercise 2: Switch to BFS

1. Modify line 127 to use `QueueFrontier()` instead of `StackFrontier()`
2. Run the program again with all maze files
3. Compare: states explored, path length, and path shape
4. Document your findings

△ Important

Exercise 3: Create Your Own Maze

1. Create a new file `maze4.txt` with your own maze design
2. Make sure it has exactly one `A` and one `B`
3. Test it with both DFS and BFS
4. Design a maze where DFS and BFS find different paths

11 What to Submit

Submit the following to your instructor:

1. **Lab Report** (PDF or Word) containing:
   - Answers to all discussion questions
   - Screenshots of your program running
   - Comparison table: DFS vs BFS (states explored, path length)
   - Your custom maze design (`maze4.txt`)
2. **Modified Code** (if applicable):
   - Your `maze.py` with BFS modification (Exercise 2)
   - Your custom `maze4.txt` file (Exercise 3)

Happy Maze Solving!

Dr. Naila Marir

CS3081 • Artificial Intelligence • Spring 2026