# Week 3 Lab Plan: Depth-First Search (DFS) Pathfinding

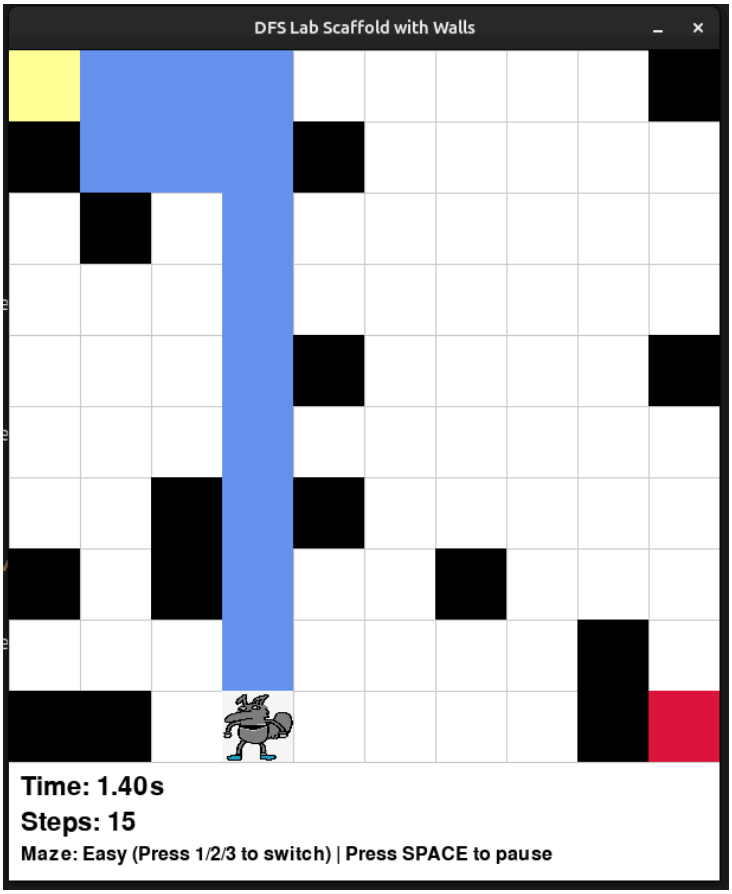


Figure 1: GUI for DFS Lab

## Overview

This lab session focuses on implementing Depth-First Search (DFS) algorithm for maze navigation. Students will work with a Pygame-based visualization tool to understand how DFS explores a maze and compares to random movement strategies. This lab bridges the gap between theoretical search algorithms and practical implementation.

## Learning Objectives

By the end of this lab, students will be able to:

- Understand the DFS algorithm and how it explores state spaces
- Implement DFS for pathfinding in a grid-based maze
- Compare DFS performance against random movement
- Work with stack-based data structures for search
- Analyze search algorithm efficiency (steps taken, time complexity)
- Understand the difference between explored nodes and optimal paths

## Prerequisites

- Completion of Week 2 lab (environment setup)
- Understanding of stacks and recursion
- Basic knowledge of search algorithms from lectures
- Familiarity with Python data structures (lists, sets, tuples)

## Important: Update Your Environment

### Pull Latest Changes

Before starting this lab, make sure to pull the latest changes from the repository:

**Using Git (CLI)**:

```
cd aima-python-eecs118-fall-25
git pull origin master
```

**Using GitHub Desktop**:

- Open the repository in GitHub Desktop
- Click "Fetch origin" button at the top
- If updates are available, click "Pull origin"

### Update Conda Environment

The `environment.yml` file has been updated with new dependencies (including Pygame). Update your conda environment:

```
conda activate aima-python
conda env update -f environment.yml --prune
```

If you encounter issues, you can recreate the environment from scratch:

```
conda deactivate
conda env remove -n aima-python
conda env create -f environment.yml
conda activate aima-python
```

## Lab Activities

### 1. Understanding the Scaffold

### Run the Initial Program

```
python dfs_lab_week3.py
```

### Explore the Interface

Students should familiarize themselves with:

1. **Controls**:
   - `SPACE`: Start/Pause/Resume the simulation
   - `1/2/3`: Switch between Easy/Medium/Hard difficulty mazes
   - The simulation won't start until you press SPACE
2. **Visual Elements**:
   - **Yellow cell**: Start position (top-left corner: 0,0)
   - **Red cell**: Goal position (bottom-right corner)
   - **Black cells**: Walls/obstacles
   - **Blue cells**: Visited positions
   - **Green circle/image**: The agent
3. **Maze Sizes**:
   - **Easy (1)**: 10x10 grid
   - **Medium (2)**: 15x15 grid
   - **Hard (3)**: 20x20 grid
   - The grid only renders the used space for cleaner visualization

4. **Information Display**:
   - **Timer**: Tracks elapsed time (pauses when simulation is paused)
   - **Steps**: Number of moves made by the agent
   - **Maze difficulty**: Current maze and controls

**Observe Random Movement**

- Start the simulation and watch the agent move randomly
- Note how inefficient random movement is for reaching the goal
- Try different difficulty levels

### 2. Understanding the Code Structure

The code is now organized into two files:

#### `maze_search_week3.py` - Your Implementation File

This is where you'll implement your DFS algorithm. It contains:

**`get_neighbors(pos, walls, rows, cols)`**

```
def get_neighbors(pos, walls, rows, cols):
    x, y = pos
    moves = [(1, 0), (-1, 0), (0, 1), (0, -1)]
    valid = [
        (x + dx, y + dy)
        for dx, dy in moves
        if 0 <= x + dx < cols and 0 <= y + dy < rows and (x + dx, y + dy) not in
walls
    ]
    return valid
```

- Returns all valid neighboring cells (not out of bounds, not walls)
- Movement directions: right, left, down, up
- **Question**: Why does order matter for DFS?

**`random_move(pos, walls, rows, cols)`**

```
def random_move(pos, walls, rows, cols):
    import random
    neighbors = get_neighbors(pos, walls, rows, cols)
    if neighbors:
        return random.choice(neighbors)
    return pos
```

- Current baseline: picks a random valid neighbor

**`dfs(start, goal, walls, rows, cols)`**

- Currently returns empty lists: `return [], []`
- **Task**: This is what you'll implement!
- Should return: `(path, visited_order)`

#### `dfs_lab_week3.py` - Visualization Code

- Contains all the pygame visualization
- You don't need to modify this file
- It imports and uses functions from `maze_search_week3.py`

### 3. Implementing DFS (Main Exercise - 45 minutes)

#### Understanding DFS Algorithm

Depth-First Search uses a **stack** to explore paths. Here's the conceptual algorithm:

```
DFS(start, goal):
    1. Initialize a stack with the start position
    2. Initialize an empty set for visited nodes
    3. While the stack is not empty:
        a. Pop a position from the stack
        b. If this position is the goal, success!
        c. If this position hasn't been visited:
            - Mark it as visited
            - Get all valid neighbors
            - Push unvisited neighbors onto the stack
    4. If stack is empty and goal not found, no path exists
```

#### Your Task

Implement the `dfs()` function in `maze_search_week3.py`.

**Function Signature**:

```python
def dfs(start, goal, walls, rows, cols):
    """
    Args:
        start: Starting position as (x, y) tuple
        goal: Goal position as (x, y) tuple
        walls: Set of wall positions
        rows: Number of rows in the maze
        cols: Number of columns in the maze

    Returns:
        Tuple of (path, visited_order) where:
        - path: List of positions from start to goal
        - visited_order: List showing order of exploration
    """
```

**Key Considerations**:

1. Your DFS should compute the **entire path** at once (not frame-by-frame)
2. Use a stack to keep track of positions to explore
3. Keep track of the path taken to reach each position
4. Use `get_neighbors()` to find valid neighboring cells
5. Return both the solution path and the order cells were visited

**What to modify**:

- Only edit `maze_search_week3.py`
- Implement the `dfs()` function
- The `get_neighbors()` function is already provided - use it!

**What NOT to change**:

- Don't modify `dfs_lab_week3.py`
- Don't change `get_neighbors()` or `random_move()` (yet)

### 4. Testing and Comparison

#### Test Your Implementation

1. Run your DFS implementation on the Easy maze

2. Observe the search pattern - does it explore depth-first?
3. Check if it successfully reaches the goal
4. Test on Medium and Hard mazes

**Questions for Discussion**:
1. Does DFS always find a path if one exists?
2. Does DFS find the *shortest* path?
3. What happens when DFS encounters dead ends?
4. How does wall density affect DFS performance?

**5. Extensions (Optional Challenge Activities)**

**Challenge: Compare with BFS**
Implement Breadth-First Search (BFS) as well:
- Use a queue instead of a stack
- Compare path length and exploration patterns
- Which finds shorter paths?

## Common Issues and Debugging

**Issue 1: Agent Gets Stuck**
**Problem**: Agent stops moving but hasn't reached goal

**Solution**: Check if visited set is being updated correctly and if all neighbors are being explored

**Issue 2: Stack Overflow**
**Problem**: Recursion depth exceeded (if using recursive DFS)

**Solution**: Use iterative DFS with explicit stack instead

## Deliverables
Students should be able to demonstrate:
1. Working DFS implementation in `maze_search_week3.py` that successfully navigates all three maze difficulties
2. Comparison data between random movement and DFS
3. Understanding of when DFS is appropriate vs other search algorithms
4. Code that properly imports and uses functions from `maze_search_week3.py`

## Additional Resources
- AIMA Chapter 3: Solving Problems by Searching
- Pygame documentation: pygame.org
- Visualization of DFS: visualgo.net/en/dfsbfs
- Python data structures (stack/queue): docs.python.org