Maze Solving with BFS and DFS and Visualization

Anveshan Timsina*, Sandesh Dhital†, and Utsab Dahal‡
Department of Electronics and Computer Engineering, Thapathali Campus
Institute of Engineering, Tribhuvan University, Kathmandu, Nepal
Email: *anveshan.078bei005@tcioe.edu.np, †sandesh.078bei034@tcioe.edu.np, ‡utsab.078bei046@tcioe.edu.np

***Abstract*—This report focuses on implementing two uninformed search techniques: BFS (Breadth-First Search) and DFS (Depth-First Search) to solve a 2D maze. A complex maze structure is encoded using a 2D grid containing start, goal, wall, and free cells. The implementation involves converting this symbolic maze data (containing strings as well as numbers) into numerical form by the use of numpy and visualizing the search process (BFS or DFS, as selected) using animations from matplotlib. The maze is generated using a random maze generator function. Additionally, the path length, steps taken, and execution time for each algorithm is determined.**

***Index Terms*—Breadth-First Search (BFS), Depth-First Search (DFS), Uninformed Search Algorithms, Maze Solving, Maze Visualization, Matplotlib, Numpy, Python.**

I. Introduction

A. Background

In artificial intelligence, the ability to navigate environments and find optimal or feasible paths from one point to another is a foundational problem. Search algorithms like BFS and DFS provide systematic ways to explore a problem space and identify paths to goal states, even without heuristic knowledge about the environment.

The background of this experiment stems from the fundamental theories of state-space search, where each configuration of the environment represents a unique state. In the case of a maze, each cell in the grid can be viewed as a potential state. The problem becomes one of moving from an initial state (the start cell) to a goal state (the goal cell), while avoiding walls and staying within the boundaries of the grid.

B. Objectives

This lab experiment aims to:

- Implement BFS and DFS algorithms on a grid-based maze to solve (or attempt to solve) it and determine the path length for each.
- visualize the search process using animations and highlight the path from start (initial state) to the goal (goal state).

II. Experimentation

A. Searching Algorithms

Searching algorithms are fundamental techniques in artificial intelligence that enable an agent to traverse a state space in pursuit of a solution to a given problem. At their core, these algorithms explore various possible configurations or states of the environment to find a path from an initial state to a desired goal state. Depending on whether the algorithm possesses domain-specific knowledge about the problem space, searching techniques can be broadly classified into two categories: uninformed search and informed search.

1) Uninformed Searching: Uninformed search algorithms, also known as blind search algorithms, operate without any additional information about the goal state beyond the problem definition itself. These algorithms do not consider how close a given state is to the goal; they rely solely on the structure of the state space and the ability to generate successors. Notable examples include Breadth-First Search (BFS), Depth-First Search (DFS), and Uniform Cost Search (UCS). These methods are exhaustive and guaranteed to find a solution if one exists, although not always the most efficient or optimal.

2) Informed Searching: Informed search algorithms, often called heuristic search methods, utilize problem-specific knowledge to guide the search more efficiently. By estimating the cost or distance from the current state to the goal, these algorithms prioritize more promising paths and typically reach solutions faster. Examples include Greedy Best-First Search and A* Search, both of which rely on heuristic functions to evaluate the desirability of nodes. Informed search tends to be more efficient than uninformed methods but requires a carefully designed heuristic to perform well.

In the context of this lab, the focus is on uninformed search algorithms, particularly BFS and DFS, both of which explore the maze with different traversal strategies.

3) BFS: Breadth-First Search is an uninformed search strategy that explores the search space level by level. Starting from the initial state, it visits all neighboring nodes before moving on to nodes at the next depth level. BFS uses a First-In-First-Out (FIFO) data structure, typically implemented using a queue. This property makes BFS complete, meaning it is guaranteed to find a solution if one exists, and optimal, as it finds the shortest path in terms of the number of steps, assuming all step costs are equal. The pseudocode for BFS search is as follows;

Initialize an empty queue `frontier`
Initialize an empty set `visited`
Enqueue $(S, [S])$ into `frontier`
while `frontier` is not empty **do**
  Dequeue $(current, path)$ from `frontier`
  if $current = G$ **then**
    return $path$
  end if
  if $current$ not in `visited` **then**
    Add $current$ to `visited`

for each neighbor n of *current* **do**
if n is not a wall and n not in `visited` **then**
Enqueue $(n, path + [n])$ into `frontier`
end if
end for
end if
end while
return No path found

4) DFS: Depth-First Search is another uninformed search technique that explores the search space by going as deep as possible along one path before backtracking. DFS uses a Last-In-First-Out (LIFO) structure, usually implemented with a stack. Unlike BFS, DFS does not guarantee the shortest path and can get trapped in deep or infinite paths if not properly managed. However, it is often faster than BFS in reaching a solution, especially in large or sparse search spaces. The pseudocode for DFS search is as follows;

Initialize an empty stack `frontier`
Initialize an empty set `visited`
Push $(S, [S])$ into `frontier`
while `frontier` is not empty **do**
Pop $(current, path)$ from `frontier`
if $current = G$ **then**
return $path$
end if
if *current* not in `visited` **then**
Add *current* to `visited`
for each neighbor n of *current* **do**
if n is not a wall and n not in `visited` **then**
Push $(n, path + [n])$ into `frontier`
end if
end for
end if
end while
return No path found

B. Implementation

1) Maze Representation and Preprocessing: Initially, the maze was represented using a two-dimensional list in Python, containing a mix of string and integer values. The characters 'S' and 'G' were used to denote the start and goal positions, respectively, while '0' denoted free paths and '1' represented walls. However, for convenient searching, a function was implemented, which iterated through the original maze and mapped the symbols into integer values (S:2 and G:3) giving a numpy array. Additionally, instead of just solving a pre-coded maze, a random maze generator function was also implemented to test the algorithm for different mazes.

```
original_maze = [
    ['S', 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0,
        1, 0, 0, 1, 0],
    [1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0,
        1, 0, 1, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0,
        1, 0, 0, 1, 0],
    [0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0,
        1, 1, 0, 1, 0],
    [0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0,
        0, 0, 0, 1, 0],
    [0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1,
        1, 1, 1, 1, 0],
    [0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0],
    [0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1,
        1, 1, 1, 1, 0],
    [0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0,
        0, 0, 0, 1, 0],
    [0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1,
        1, 1, 0, 1, 0],
    [0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0,
        0, 1, 0, 0, 0],
    [1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1,
        0, 1, 1, 1, 0],
    [0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1,
        0, 0, 0, 1, 0],
    [0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1,
        1, 1, 0, 1, 0],
    [0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0,
        1, 0, 0, 0, 0],
    [1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0,
        1, 0, 1, 1, 1],
    [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0,
        0, 0, 0, 0, 0],
    [0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1,
        1, 1, 1, 1, 0],
    [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0],
    [1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 'G'],
]

def to_numeric_grid(maze):
    mapping = {'S':2, 'G':3, '1':1, '0':0, 1:1, 0:0}
    numeric_grid =
        np.array([[mapping.get(element,100) for
        element in row] for row in maze], dtype=int)
    return numeric_grid

numerized_maze = to_numeric_grid(original_maze)

 # Alternatively, using a random maze generator
def generate_random_maze(rows, cols, wall_prob=0.3):
    maze = np.random.choice([0, 1], size=(rows,
        cols), p=[1 - wall_prob, wall_prob])
    maze[0][0] = 2
    maze[rows - 1][cols - 1] = 3
    return np.array(maze)

numerized_maze_random = generate_random_maze(10, 10)
```

2) Position Identification: To initiate the search algorithms, it was necessary to determine the coordinates of the cells within the grid (specifically, start and goal cells). This was achieved using a function (named findPos), which scanned the entire grid to locate the cell corresponding to a given value (2 for start and 3 for goal). This function returned the coordinates of the cell as a tuple (row, column), which was then used as the starting point for traversal.

```
def find_pos(maze, value):
    for i in range(len(maze)):  # iterating over
        rows in the maze
        for j in range(len(maze[i])):  # iterating
            over elements in a row
            if maze[i][j] == value:
                return (i, j)
    return None
```

3) Maze Visualization: The maze, now in the form of a numpy array was visualized with white representing free cells and black representing walls.

```
def visualize_maze(maze):
    start_pos = find_pos(maze, 2)
    goal_pos = find_pos(maze, 3)

    cmap = colors.ListedColormap(['white', 'black'])
    bounds = [-0.5, 0.5, 1.5]
    norm = colors.BoundaryNorm(bounds, cmap.N)

    plt.figure(figsize=(10,10))
    plt.imshow(maze, cmap=cmap, norm=norm)

    plt.scatter(start_pos[0], start_pos[1], s=150,
        c='green', label='Start(S)')
    plt.scatter(goal_pos[0], goal_pos[1], s=150,
        c='red', label='Goal(G)')
    plt.legend()

    plt.xticks([])
    plt.yticks([])

    plt.show()

visualize_maze(numerized_maze)
```

4) Determining Legal Moves: Obviously, in a maze, there are only a few valid positions that can be traversed to from a particular position. To find the legal moves from a given position, the getNeighbor function was implemented. It generated possible moves in the four cardinal directions: up, down, left, and right and also ensured that these moves did not lead outside the bounds of the grid and did not move into wall cells.

```
def get_neighbors(maze, r, c):
    directions = [(-1, 0), (1, 0), (0, 1), (0, -1)]
    rows, columns = len(maze), len(maze[0])

    for (dr, dc) in directions:
        nr = r + dr
        nc = c + dc
        if (0 <= nr < rows) and (0 <= nc < columns):
            if maze[nr][nc] != 1:
                yield (nr, nc)
```

5) Algorithm Implementation and Visualization: The core component of the lab was the implementation of the solveMaze function, which handled both BFS and DFS. The user was prompted to choose either of the two algorithms (input being converted to lowercase). BFS used a queue-like structure implemented using deque (from the collections module), whereas DFS utilized a list structure that functioned as a stack.

The maze-solving process was visualized using the matplotlib library, particularly through the FuncAnimation class. A colormap was defined to assign colors to different elements of the maze. As the algorithm progressed, each visited cell was colored green, and upon reaching the goal, the entire solution path was marked in blue. Additional functionality was included to allow users to terminate the animation by pressing "q".

```
def solve_maze(maze, algorithm='bfs'):
    if algorithm not in ['bfs', 'dfs']:
        print('Invalid algorithm. Choose bfs or
            dfs')
        return

    start = find_pos(maze, 2)
    goal = find_pos(maze, 3)

    frontier = deque([(start, [start])]) if
        algorithm == 'bfs' else [(start, [start])]
    visited = set()

    cmap = colors.ListedColormap(['white', 'black',
        'green', 'red', 'yellow', 'blue'])

    fig, ax = plt.subplots(figsize=(10, 10))
    img = ax.imshow(maze, cmap=cmap)
    ax.set_title(f"{algorithm.upper()} Maze Solver")

    found_goal = False

    def update(frame):
        nonlocal frontier, found_goal
        start_time = time.time()

        if found_goal or not frontier:
            return img

        (r, c), path = frontier.popleft() if
            algorithm == 'bfs' else frontier.pop()

        if (r, c) in visited:
            return img

        visited.add((r, c))

        if (r, c) == goal:
            path_length = len(path)
            end_time = time.time()
            execution_time = end_time - start_time
            print("Goal reached at", (r, c))
            print(f'Path length: {path_length}')
            print(f'Steps taken: {len(visited)}')
            print(f'Total visited nodes:
                {len(visited)}')
            print(f'Execution time:
                {execution_time:.3f} seconds')

            for pr, pc in path:
                if maze[pr][pc] not in [2, 3]:
                    maze[pr][pc] = 5
            img.set_data(maze)
            found_goal = True
            return img

        if maze[r][c] not in [2, 3]:
            maze[r][c] = 4

        for nr, nc in get_neighbors(maze, r, c):
            if (nr, nc) not in visited and
                maze[nr][nc] != 1:
                frontier.append(((nr, nc), path +
                    [(nr, nc)]))

        print(f"Visiting: {(r, c)}, Frontier size:
            {len(frontier)}")
        img.set_data(maze)
        return img

    def close_on_key(event):
        plt.close(fig)

    fig.canvas.mpl_connect('key_press_event',
        close_on_key)
    anim = animation.FuncAnimation(fig, update,
```

```
        interval=300, frames=500, repeat=False)
    display(HTML(anim.to_jshtml()))

algorithm = input("Enter algorithm (BFS or DFS):
    ").strip().lower()
solve_maze(numerized_maze,algorithm)
```

III. Result

An example maze has been shown in Figure 1. This was solved using both BFS and DFS (Figure 2, 3). The start node was represented by orange, goal by red, visited path by green and final solution by blue. For further analysis of each algorithm, the length of the path with steps taken, number of visited nodes and total execution time was determined (Table I).

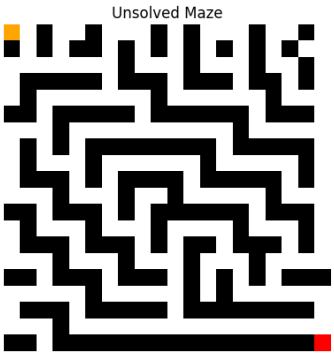
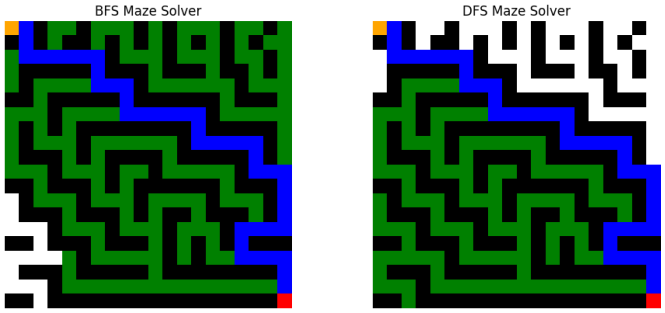


Fig. 1: Unsolved Maze Visualized



(a) Maze Solved using BFS (b) Maze Solved using DFS

Fig. 2: Solved Mazes Visualized

TABLE I: Comparison of BFS and DFS on Maze Solving

Metric	BFS	DFS
Path length	45	45
Steps taken	210	164
Total visited nodes	204	163
Execution time (s)	0.6976	0.1637

IV. Discussion

The process of implementing and visualizing the maze-solving algorithms using BFS and DFS was both insightful and technically demanding. Throughout the development of the project, several challenges arose, particularly in relation to data handling, real-time animation, and ensuring the correctness of the search algorithms under varying conditions.

An issue related to the real-time rendering of the animation was observed during this lab session. Initially, although the logic of BFS and DFS were executing correctly, the animation did not update frame by frame. Instead, it displayed the final state only after the entire algorithm had completed. This was traced to the improper handling of the animation output in the Jupyter Notebook environment. Specifically, the HTML (anim.tojshtml()) function needed to be wrapped inside the display() function to actually render the animation inline.

Color mapping errors were also encountered during development. The values assigned to different maze elements sometimes fell outside the defined range of the colormap, leading to incorrect colors or rendering failures. This was addressed by ensuring that the maze values were limited to the expected range and that the colormap was defined accordingly.

Another technical detail that required careful attention was managing the frontier and visited states. It was important to ensure that previously visited cells were not re-added to the frontier, which could lead to infinite loops or redundant processing. This was accomplished by the use of a boolean named foundGoal.

V. Conclusion

This lab effectively demonstrated the practical differences between BFS and DFS in solving grid-based mazes. On testing with a variety of randomly generated mazes, BFS proved effective in finding the shortest path due to its level-wise exploration, whereas DFS offered faster traversal in some scenarios but did not guarantee optimal paths. Through visualization, the exploration patterns of each algorithm became more intuitive. The project enhanced understanding of search mechanics, legal state transitions, and animation in Python.