Date of Submission July 7, 2025.

Maze Solving With BFS and DFS

AMRIT KANDEL[1], (THA078BEI004), PRASISH TIMALSINA[2], (THA078BEI026)
[1]Department of Electronics and Computer Engineering, Thapathali Campus (e-mail: amrit.tha078bei004@tcioe.edu.np)
[2]Department of Electronics and Computer Engineering, Thapathali Campus (e-mail: prasish.tha078bei026@tcioe.edu.np)

ABSTRACT This report discusses the implementation and comparison of two basic graph traversal algorithms—Breadth-First Search (BFS) and Depth-First Search (DFS)—to solve a maze. The maze was a 2D grid, and both algorithms were implemented to search for a path between a start point and a goal point. The program was written to accept user input to choose between BFS and DFS, compute the path, and graph it using Python's matplotlib library. Experimental results showed that BFS always resulted in the shortest path, confirming its optimality, while DFS also reached the goal but with longer and less efficient routes for most instances. The exercise illustrates the real-world difference between BFS and DFS in terms of completeness, optimality, and efficiency.

INDEX TERMS Breadth First Search (BFS), Depth First Search (DFS), Maze Solving

I. INTRODUCTION

A. INTRODUCTION TO GRAPH

Graph is a non-linear data structure consisting of edge and vertices. It is made up of vertices V and edges E and is denoted by G(V, E). The components of a graoh are described below.

- **Vertices**: Vertices are also called nodes of a graph.
- **Edges**: Edges are also called paths or arcs. They connect two nodes of the graph. They are in the form of ordered pairs of two nodes i.e. $E_1 = (V1, V2)$.

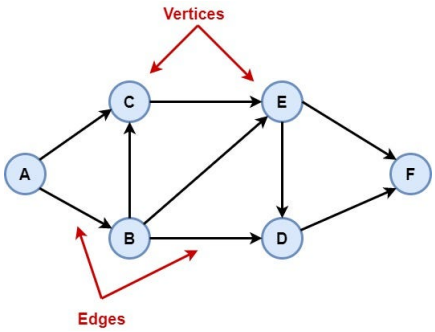

FIGURE 1. Example of a graph

B. GRAPH SEARCHING ALGORITHMS

Graph searching algorithms are techniques used to explore or traverse the nodes and edges of a graph to find specific information, such as paths or connectivity. The two most common algorithms are Breadth-First Search (BFS) and Depth-First Search (DFS). These algorithms are widely used in applications like pathfinding, maze solving, etc.

1) Breadth First Search (BFS)

In breadth-first search, nodes are searched in a horizontal line and later checked for goal state. It proceeds level by level down the search trees. Starting from the root node explores all children of the root node, left to right. If no solution is found, expand the first child of the root node, then expands the second node at depth 1 and so on.

TABLE 1. BFS Algorithm

Algorithm for Breadth First Search
1. Place the start node in the queue
2. Examine the node all the front queue
a. if the point is empty, stop.
b. If the node is the goal, stop.
c. Otherwise, add the children of the node to the end of the queue.

a: Properties of BFS:

- **Completeness**: Complete if the goal node is at finite depth.
- **Optimality**: It is guaranteed to find the shortest path. If the goal node is available, the algorithm would already reach the node first. Hence, we can see that the algorithm is optimal.
- **Time Complexity**: $O(b^d)$
- **Space Complexity**: $O(b^d)$

b: Weakness:

- High time and memory requirement.

2) Depth First Search (DFS)

TABLE 2. DFS Algorithm

Algorithm for Depth First Search
1. Put the start node on the stack
2. While stack is not empty
a. Pop the stack
b. If the top of the stack is goal, stop
c. Otherwise, push the nodes connected to the top of the stack

a: Properties of DFS:

- **Completeness**: Incomplete as it may get stuck down, going down an infinite branch that does not lead to a solution.
- **Optimality**: The first solution found by the DFS my not be the shortest.
- **Time Complexity**: $O(b \times d)$
- **Space Complexity**: $O(b^d)$

b: Weakness

- In DFS, certain state forms an infinite branch that doesn't lead to a solution.

II. EXPERIMENTATION DETAILS

In this experiment, we used two fundamental graph traversal methods: Breadth-First Search (BFS) and Depth-First Search (DFS) to explore a maze, which was simulated as a grid. The methods were applied using the Python programming language, and visualizations were performed using the matplotlib library.

The maze was presented as a 10×10 two-dimensional NumPy array. Any cell within the maze may be assigned one of four numbers: 'S' for start position, 'G' for goal position, 1 for obstacles or walls, and 0 for open paths. The maze was initially converted to a numeric grid by itself before applying the pathfinding algorithm. 'S' was substituted with 2 and 'G' with 3 in this conversion while other numbers stayed unchanged. This numerical depiction made subsequent processing easier while pathfinding.

To determine the coordinates of the start and end points, a utility function visited the grid to determine the respective numerical values. Another helper function was used to determine the valid neighboring cells of any cell. The function considered four directions of movement: up, down, left, and right. A neighboring cell was valid only if it was inside the maze and not a wall.

The BFS algorithm used a queue-based approach to explore the maze level-order. It was started at the source node and explored all of its direct neighbors first before exploring further into the maze. BFS is optimal and complete for finding the shortest path if the path exists but comes with a high memory overhead. On the other hand, DFS used a stack-based approach to traverse the maze depth-first. It ventured one branch to its deepest point before backtracking. Although DFS may be more memory-intensive, it is not optimal and incomplete and may become trapped in infinite loops if there exist cycles.

Both algorithms contained a list of visited nodes so that cells wouldn't be visited again and a parent dictionary to keep track of the path. When the goal node was discovered, the path was then formed by going back from the goal to the beginning using the parent information.

After computing the path, the result was plotted with a plotting function. A color map was used to plot the maze grid, blue circular markers were used to mark the path, green was used to highlight the start point, and red was used to mark the target. This provided an indication of the path that the algorithms took through the maze.

The whole experiment process was wrapped around a single function that did preprocessing, algorithm selection, path discovery, and visualization. The user was asked at runtime to choose the algorithm they wanted to work with, either BFS or DFS, and the respective logic was invoked accordingly. This modular interactive aspect guaranteed readability and ease of use while testing and demonstrating.

III. RESULTS

The system prompts the user to input the desired search algorithm either 'bfs' or 'dfs'. Based on the selection, the corresponding algorithm is executed, and the shortest path from the start node ('S') to the goal node ('G') is computed and displayed.

The resulting path is printeddeque as a list of coordinate positions, and a visual representation of the path is generated using matplotlib. The selected path is marked in blue dots, the start node in green, and the goal node in red.

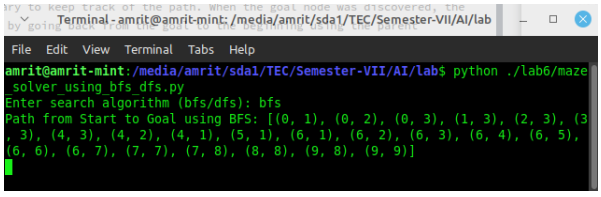

FIGURE 2. Path from start to goal for BFS

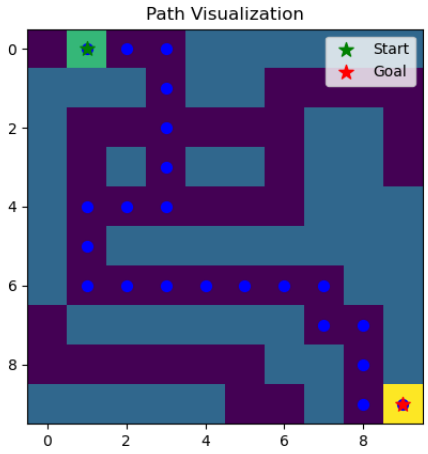

FIGURE 3. Visualization of Path using BFS

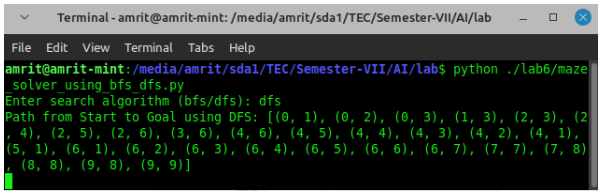

FIGURE 4. Path from start to goal for DFS

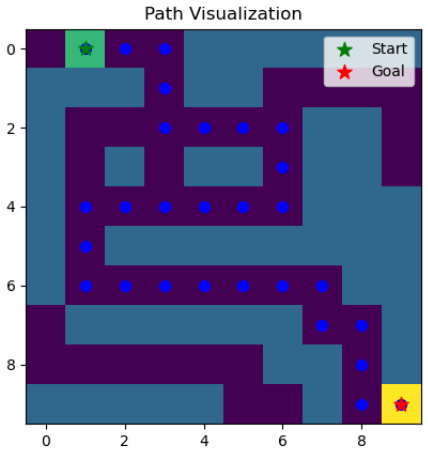

FIGURE 5. Visualization of Path using DFS

IV. DISCUSSION

t was observed from the experiment that Breadth-First Search (BFS) always produced the optimal solution by traversing the goal from the initial position via the shortest path. This was expected because BFS traverses nodes level by level, and it is guaranteed to yield the shortest path in unweighted graphs. On the other hand, Depth-First Search (DFS) did not necessarily provide the best path. It has a tendency to follow a branch all the way down before backtracking, thus often taking longer and less optimal routes. While DFS completed the goal, the solution was not minimal in all cases.

V. CONCLUSION

This project succeeded in demonstrating maze solving using BFS and DFS algorithms. BFS produced the shortest and most optimal paths, while DFS provided an accurate but generally suboptimal solution. The project graphically demonstrated the strengths and weaknesses of each method using graphical outputs and user interaction. The system is modular and can be extended with additional advanced path-finding algorithms as future work.

APPENDIX A

SOURCE CODE

The source code is also available in the following link: https://github.com/Amritkandel49/AI_labwork/blob/main/lab6/maze_solver_using_bfs_dfs.py

```
import numpy as np
import matplotlib.pyplot as plt
from collections import deque

def to_numeric_grid(maze):
    # maze = np.array(maze)
    # print(maze)
    for i in range(len(maze)):
        for j in range(len(maze[i])):
            if maze[i][j] == 'S':
                maze[i][j] = 2
            elif maze[i][j] == 'G':
                maze[i][j] = 3
            else:
                maze[i][j] = maze[i][j]
    return maze.astype(int)


def find_pos(maze, value):
    maze = np.array(maze)
    for i in range(len(maze)):
        for j in range(len(maze[i])):
            if maze[i][j] == value:
                return (i, j)



def get_neighbours(r, c, numeric_maze):
    grid = np.array(numeric_maze)
    ROWS = grid.shape[0] - 1
    COLS = grid.shape[1] - 1
    neighbors = []
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)
        ]
    for dr, dc in directions:
        new_r, new_c = r + dr, c + dc
        if 0 <= new_r <= ROWS and 0 <= new_c <=
            COLS and grid[new_r][new_c] != 1:  #
            Exclude walls
            neighbors.append((new_r, new_c))
    return neighbors


def bfs(maze, start, goal):
    queue = deque([start])
    visited = set()
    visited.add(start)
    parent = {start: None}
```

```
    while queue:
        current = queue.popleft()
        if current == goal:
            path = []
            while current:
                path.append(current)
                current = parent[current]
            return path[::-1]
        r, c = current
        for neighbor in get_neighbours(r, c, maze)
            :
            if neighbor not in visited:
                visited.add(neighbor)
                parent[neighbor] = current
                queue.append(neighbor)
    return None

def dfs(maze, start, goal):
    stack = [start]
    visited = set()
    visited.add(start)
    parent = {start: None}

    while stack:
        current = stack.pop()
        if current == goal:
            path = []
            while current:
                path.append(current)
                current = parent[current]
            return path[::-1]

        for neighbor in get_neighbours(*current,
            maze):
            if neighbor not in visited:
                visited.add(neighbor)
                parent[neighbor] = current
                stack.append(neighbor)
    return None


def visualize_path(maze, path, start_pos, goal_pos
    ):
    grid = np.array(maze)
    plt.imshow(grid, cmap='viridis')

    for y, x in path:
        plt.scatter(x, y, c='b', s=50, marker='o')

    # start = np.argwhere(grid == 2)
    # goal = np.argwhere(grid == 3)

    for y, x in [start_pos]:
        plt.scatter(x, y, c='g', s=100, marker='*'
            , label='Start')

    for y, x in [goal_pos]:
        plt.scatter(x, y, c='r', s=100, marker='*'
            , label='Goal')

    plt.title("Path Visualization")
    plt.legend()
    plt.show()


def pipeline(maze, search_alg):
    numeric_maze = to_numeric_grid(maze)
    start_pos = find_pos(numeric_maze, 2)
    goal_pos = find_pos(numeric_maze, 3)

    if search_alg == 'bfs':
        path = bfs(numeric_maze, start_pos,
            goal_pos)
    elif search_alg == 'dfs':
        path = dfs(numeric_maze, start_pos,
            goal_pos)
    else:
        raise ValueError("Invalid search algorithm
            . Use 'bfs' or 'dfs'.")

    print(f"Path from Start to Goal using {
        search_alg.upper()}: {path}")
    visualize_path(numeric_maze, path, start_pos,
        goal_pos)


if __name__ == "__main__":
    maze = np.array([
    [0, 'S', 0, 0, 1, 1, 1, 1, 1, 1],
    [1, 1, 1, 0, 1, 1, 0, 0, 0, 0],
    [1, 0, 0, 0, 0, 0, 0, 1, 1, 0],
    [1, 0, 1, 0, 1, 1, 0, 1, 1, 0],
    [1, 0, 0, 0, 0, 0, 0, 1, 1, 1],
    [1, 0, 1, 1, 1, 1, 1, 1, 1, 1],
    [1, 0, 0, 0, 0, 0, 0, 0, 1, 1],
    [0, 1, 1, 1, 1, 1, 1, 0, 0, 1],
    [0, 0, 0, 0, 0, 0, 1, 1, 0, 1],
    [1, 1, 1, 1, 1, 0, 0, 1, 0, 'G']])

    search_alg = input("Enter search algorithm (
        bfs/dfs): ").strip().lower()
    if search_alg not in ['bfs', 'dfs']:
        print("Invalid input. Please enter 'bfs'
            or 'dfs'.")
    else:
        pipeline(maze, search_alg)
```