Pathfinding Prowess: Solving Mazes with Shortest Path Using DFS and BFS

A PROJECT REPORT

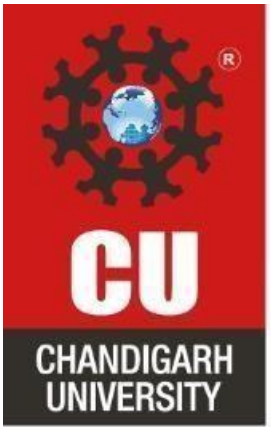
Submitted by
Akalaya Thakur(23BCS12592)

In partial fulfilment for the award of the degree of Bachelor of

Engineering

In

computer science and engineering



CHANDIGARH UNIVERSITY

Discover. Learn. Empower.

Chandigarh University

TABLE OF CONTENTS

1. Abstract
2. Introduction
3. Objective
4. Literature Review
5. Time and Space Complexity
6. Algorithm Flowchart
7. Methodology / Algorithm
8. Pseudocode
9. Implementation
10. Result/Output
11. Applications
12. Conclusion
13. References (suggested)

Abstract

This project focuses on solving mazes by determining the shortest path from a starting point to a goal using classical search algorithms — **Breadth-First Search (BFS)** and **Depth-First Search (DFS)**. BFS guarantees the shortest path in unweighted mazes by exploring nodes level by level, while DFS explores paths deeply and may not always find the optimal route but performs efficiently for path existence checks. The objective of this work is to implement and visualize both algorithms through a maze simulation in a programming environment, demonstrating their traversal behavior, path-finding capabilities, and performance differences. The project also highlights applications in robotics, game development, and autonomous navigation. By comparing BFS and DFS in terms of accuracy, time complexity, and memory usage, the study evaluates each method's strengths and limitations for maze-solving tasks and real-world navigation systems.

Introduction

Maze solving is a fundamental problem in computer science and robotics, where the objective is to navigate from a defined start position to a goal while avoiding obstacles. This problem can be represented using a grid or graph structure, where each cell or vertex corresponds to a state and connections represent valid movements. Efficient pathfinding techniques are essential in situations where optimal or fast navigation is required, such as autonomous robots, game AI, and intelligent navigation systems.

This project explores two classical graph traversal algorithms — **Breadth-First Search (BFS)** and **Depth-First Search (DFS)** — for solving mazes. BFS explores nodes level by level and guarantees the shortest path in an unweighted grid, making it highly effective for maze navigation. DFS, on the other hand, dives deep into possible paths before backtracking, which can efficiently find a path but does not ensure the shortest route. Understanding the contrast between these algorithms helps in selecting the appropriate method based on problem constraints and environment complexity.

The visual simulation in this work demonstrates how BFS and DFS explore the maze, mark visited nodes, and eventually determine the path from the start to the destination. Visualization provides deeper insights into traversal behavior, revealing algorithmic patterns such as BFS's systematic wave-like expansion and DFS's long-chain probing.

This project emphasizes the significance of classical search techniques in modern computing and illustrates their real-world relevance, forming the foundation for advanced pathfinding algorithms like A*, Dijkstra’s, and heuristic-based navigation systems.

Objective

1. **Implement Core Pathfinding Algorithms**

   - Develop working implementations of **Breadth-First Search (BFS)** and **Depth-First Search (DFS)** for maze traversal.

   - Ensure correct path detection and coverage of the maze grid.

2. **Visualize Maze Navigation and Pathfinding**

   - Build a graphical interface that displays maze structure, exploration steps, and final path taken.

   - Highlight visited cells, frontier expansion, and final solution path for both BFS and DFS.

3. **Demonstrate Shortest Path Identification**

   - Validate BFS capability to find the **shortest path** in unweighted mazes.

   - Compare the result with DFS, showing cases where DFS may produce a longer or non-optimal path.

4. **Analyze Algorithmic Behavior & Performance**

   - Compare BFS and DFS based on:
     - Path optimality
     - Time taken to solve the maze
     - Nodes visited
     - Memory usage
   - Discuss the strengths, limitations, and real-world applicability of each method.

5. **Provide Complete Working Code & Documentation**

   - Deliver a runnable program (Python-based maze solver).

   - Include appropriate comments, screenshots, and explanation of algorithm flow.

Literature Review

Maze-solving and pathfinding have long been core research topics in computer science, robotics, and artificial intelligence. Early studies explored graph traversal strategies for systematic exploration, leading to foundational algorithms such as **Depth-First Search (DFS)** and **Breadth-First Search (BFS)**. Both algorithms were formalized in graph theory research during the mid-20th century and are discussed extensively in classical algorithmic literature.

Breadth-First Search (BFS), introduced by Konrad Zuse and later formalized in works by Moore (1959) and Lee (1961), explores nodes level-by-level, ensuring optimal path discovery in unweighted graphs. In maze solving, BFS is known for its guarantee to find the **shortest path** from start to goal by expanding all equally distant nodes before moving deeper. This makes it suitable for navigation tasks where distance minimization is essential.

Depth-First Search (DFS), originally described by Trémaux for maze traversal, uses a stack-based strategy to explore one branch deeply before backtracking. Although DFS may not guarantee the shortest solution, its low memory usage and simplicity make it a practical approach for checking path existence and solving mazes with large but sparsely branching structures. Trémaux's method also forms the basis of several modern backtracking applications in robotics and AI.

Subsequent research has expanded upon these fundamental techniques, introducing heuristic-based algorithms such as A* (Hart, Nilsson & Raphael, 1968), Dijkstra's Algorithm, and Greedy Best-First Search to improve efficiency in large-scale navigation and robotics. However, BFS and DFS remain essential teaching tools due to their clarity, efficiency in grid-based pathfinding, and foundational role in modern AI path-planning systems.

Visualization and simulation tools have contributed significantly to educational understanding of pathfinding. Platforms like **VisuAlgo** and **Maze-Generation & Solver frameworks** demonstrate algorithm behavior step-by-step, enhancing comprehension of traversal order and performance characteristics. Inspired by such educational resources, this project implements and visually compares BFS and DFS on a dynamic maze grid environment to highlight their operational differences, strengths, and limitations.

Time and Space Complexity

Maze solving using BFS (Breadth-First Search) and DFS (Depth-First Search) operates on a grid represented as a graph, where each cell is a node and movement between adjacent cells forms edges. The computational efficiency of both algorithms depends primarily on the number of cells in the maze.

Let **V** represent the total number of vertices (cells) and **E** the number of edges (possible movements between cells).

Breadth-First Search (BFS)
BFS explores the maze level by level and guarantees the shortest path in an unweighted grid.

- **Time Complexity:**

$$O(V + E)$$

For grid-based maze where each cell has at most 4 neighbors,

$$O(4V) = O(V)$$

Thus, BFS runs in **linear time** relative to the number of cells.

- **Space Complexity:**

$$O(V)$$

BFS requires additional memory to store the queue and distance/visited structures. This means BFS consumes more memory than DFS but ensures path optimality.

Depth-First Search (DFS)
DFS explores one branch deeply before backtracking. Although fast in some mazes, it **does not guarantee the shortest path**.

- **Time Complexity:**

$$O(V + E) = O(V)$$

Similar to BFS, DFS also explores all cells in the worst case.

- **Space Complexity:**

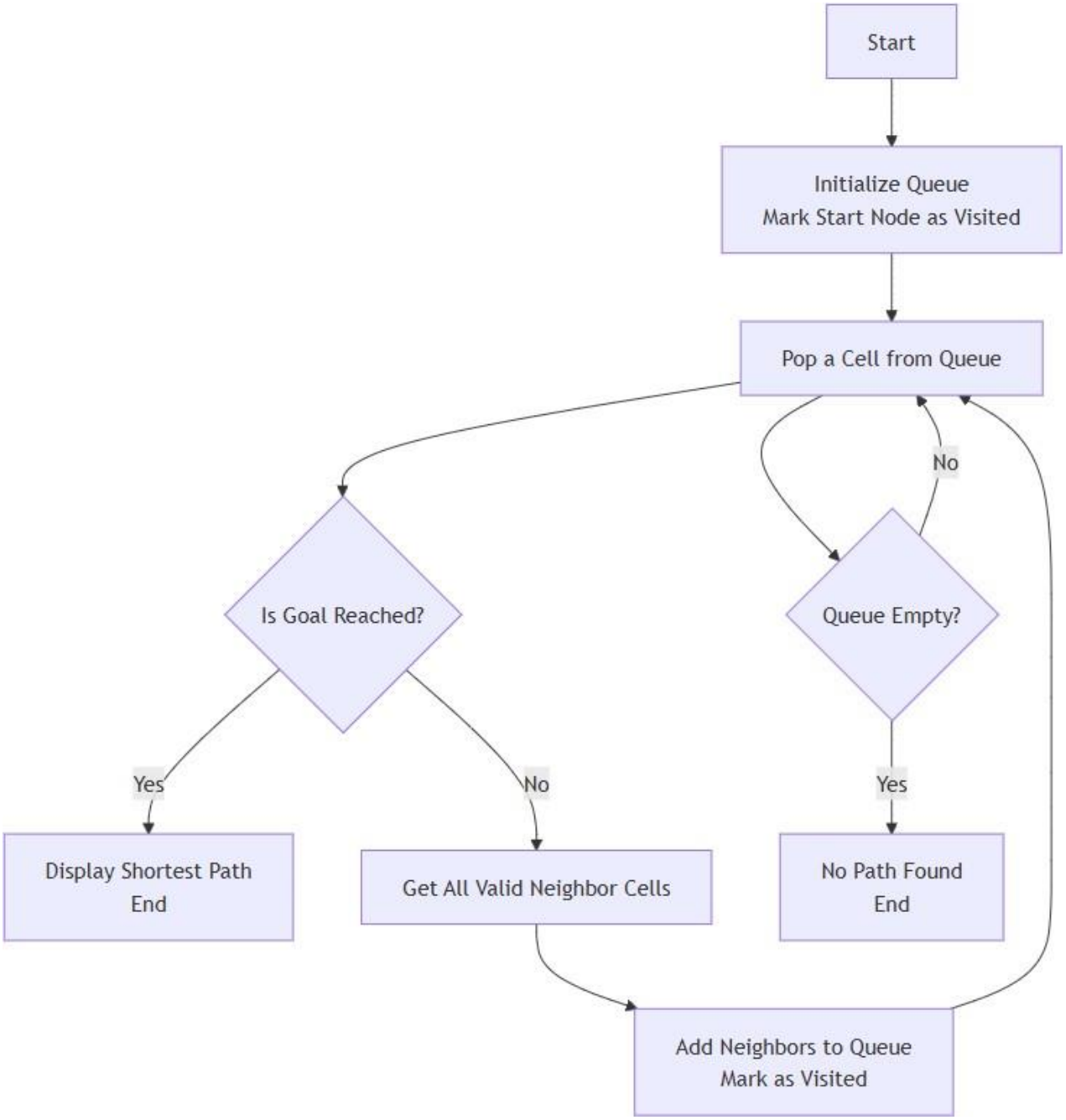
$$O(V)$$

DFS uses a stack (implicit or explicit) to store the recursion/depth path.
In the worst case (e.g., a long corridor maze), stack size may approach **O(V)**.

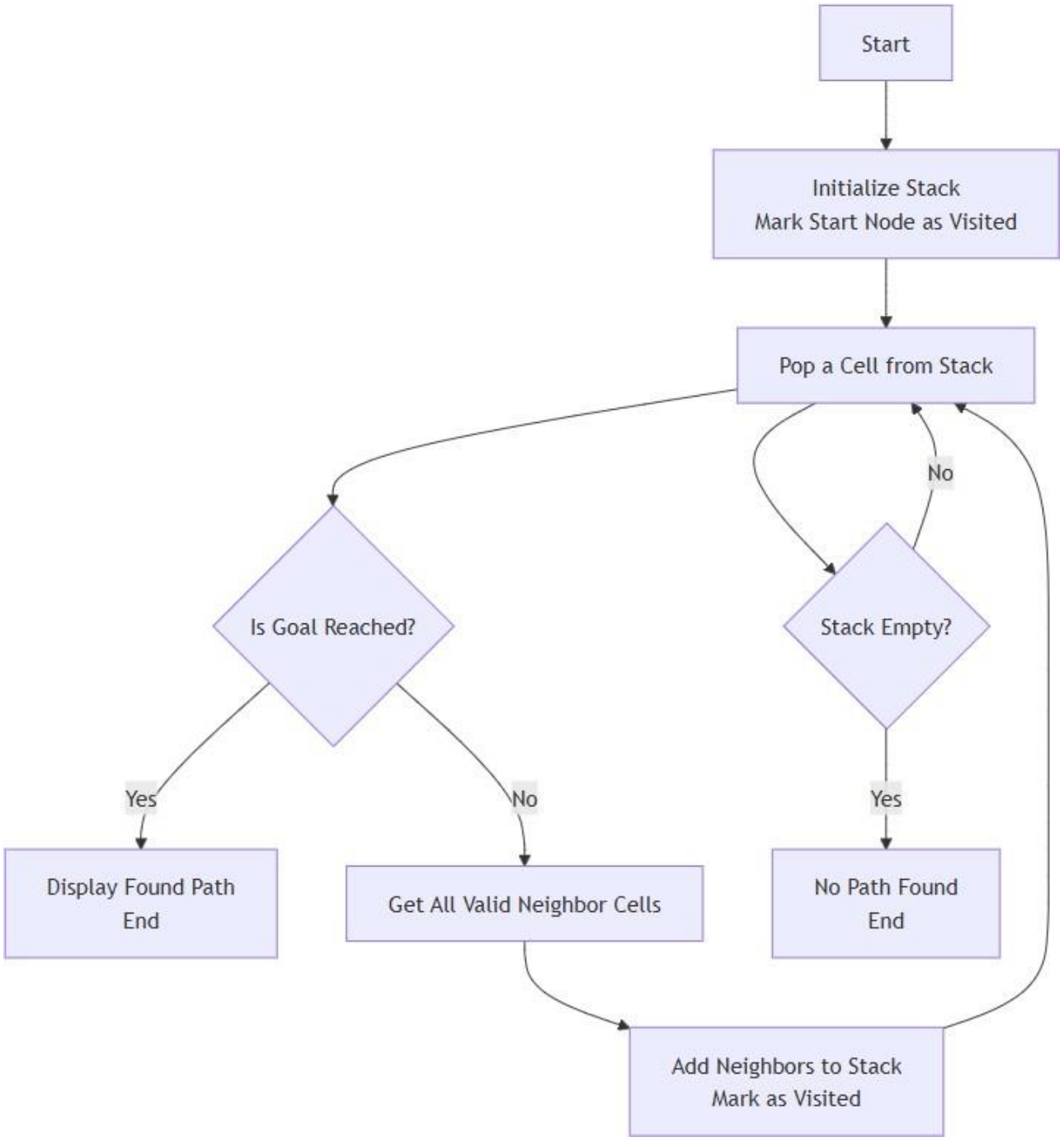
Algorithm Flowchart

Below flowcharts depict working sequences of Breadth-First Search (BFS) and Depth-First Search (DFS) used in this project for maze traversal and pathfinding.

BFS Algorithm Flowchart



DFS Algorithm Flowchart



Methodology / Algorithm

The primary objective of this project is to solve a maze by determining a valid path from the start cell to the goal cell using **Breadth-First Search (BFS)** and **Depth-First Search (DFS)**. The grid-based maze is treated as a graph, where each cell represents a vertex and movement between adjacent walkable cells represents an edge. Both algorithms operate over this graph structure, exploring possible paths until the destination node is reached or all reachable nodes have been exhausted.

Algorithm Approach

Breadth-First Search (BFS)

1. Initialize an empty queue
2. Mark the start cell as visited and push it into the queue
3. While queue is not empty:
    1. Remove the front cell
    2. If it is the goal → stop
    3. Add all valid and unvisited neighboring cells to the queue and mark them visited
4. Reconstruct path if goal found

Depth-First Search (DFS)

1. Initialize an empty stack
2. Mark the start cell as visited and push it into the stack
3. While stack is not empty:
    1. Pop a cell from the stack
    2. If it is the goal → stop
    3. Add all valid and unvisited neighbors to the stack and mark them visited
4. Reconstruct path if goal found

Maze Execution Flow

1. Load / generate maze
2. Set start and goal positions
3. Select BFS or DFS
4. Run algorithm and animate progress
5. Highlight visited nodes and final path

Pseudocode

Breadth-First Search (BFS)

```
BFS(Grid, Start, Goal):

    create an empty queue Q
    create an empty set Visited
    create a dictionary Parent

    enqueue Start into Q
    add Start to Visited

    while Q is not empty:
        current = dequeue Q

        if current == Goal:
            return ReconstructPath(Parent, Goal)

        for each neighbor in GetNeighbors(current):
            if neighbor not in Visited:
                add neighbor to Visited
                Parent[neighbor] = current
                enqueue neighbor into Q

    return "No Path Found"
```

Depth-First Search (DFS)

```
DFS(Grid, Start, Goal):

    create an empty stack S
    create an empty set Visited
    create a dictionary Parent

    push Start into S
    add Start to Visited

    while S is not empty:
        current = pop S

        if current == Goal:
            return ReconstructPath(Parent, Goal)

        for each neighbor in GetNeighbors(current):
            if neighbor not in Visited:
```

```
            add neighbor to Visited
            Parent[neighbor] = current
            push neighbor into S

    return "No Path Found"
```

Neighbor Function

```
GetNeighbors(cell):

    neighbors = []
    directions = [UP, RIGHT, DOWN, LEFT]

    for direction in directions:
        next_cell = cell + direction

        if next_cell inside grid AND next_cell is not a WALL:
            neighbors.append(next_cell)

    return neighbors
```

Path Reconstruction

```
ReconstructPath(Parent, Goal):

    path = []
    current = Goal

    while current exists in Parent:
        append current to path
        current = Parent[current]

    reverse(path)
    return path
```

Implementation

This project is implemented in **Python** using the **Pygame** library to visualize maze generation and maze solving using BFS and DFS. The program creates a grid-based maze, allows interactive placement of the start and goal points, generates walls, and visually demonstrates the path-finding steps of each algorithm.

1. Grid Initialization

The grid stores the maze structure, cell status, start and goal nodes, and visited/path cells.

```
class Grid:
    def __init__(self, width, height, cell_size):
        self.width = width
        self.height = height
        self.cell_size = cell_size
        self.cells = [[CellType.EMPTY for _ in range(width)] for _ in range(height)]
        self.start = None
        self.goal = None
        self.visited = set()
        self.frontier = set()
        self.path = []
```

2. Maze Generation (Recursive Backtracking)

```
class MazeGenerator:
    @staticmethod
    def generate(grid):
        stack = [(start_x, start_y)]
        while stack:
            x, y = stack[-1]
            neighbors = get_unvisited_neighbors(x, y)
            if neighbors:
                nx, ny = random.choice(neighbors)
                carve_path(x, y, nx, ny)
                stack.append((nx, ny))
                yield
            else:
                stack.pop()
                yield
```

This generates a perfect maze with one unique path between any two points (no loops).

3. BFS Solver Implementation

```
class BFSSolver(Solver):
    def solve(self):
        queue = deque([self.grid.start])
        came_from = {}
        self.grid.visited.add(self.grid.start)

        while queue:
            current = queue.popleft()

            if current == self.grid.goal:
                self.grid.path = self.reconstruct_path(came_from, current)
                yield True; return True

            for neighbor in self.grid.get_neighbors(*current):
                if neighbor not in self.grid.visited:
                    self.grid.visited.add(neighbor)
                    came_from[neighbor] = current
                    queue.append(neighbor)

            yield False
```

4. DFS Solver Implementation

```
class DFSSolver(Solver):
    def solve(self):
        stack = [self.grid.start]
        came_from = {}
        self.grid.visited.add(self.grid.start)

        while stack:
            current = stack.pop()

            if current == self.grid.goal:
                self.grid.path = self.reconstruct_path(came_from, current)
                yield True; return True

            for neighbor in self.grid.get_neighbors(*current):
                if neighbor not in self.grid.visited:
                    self.grid.visited.add(neighbor)
                    came_from[neighbor] = current
                    stack.append(neighbor)

            yield False
```

5. Visualization & Controls (Pygame)

The GUI displays the maze, colors for visited/frontier/path cells, and a control interface.

if event.key == pygame.K_1: start_solver(BFSSolver(grid))
if event.key == pygame.K_2: start_solver(DFSSolver(grid))

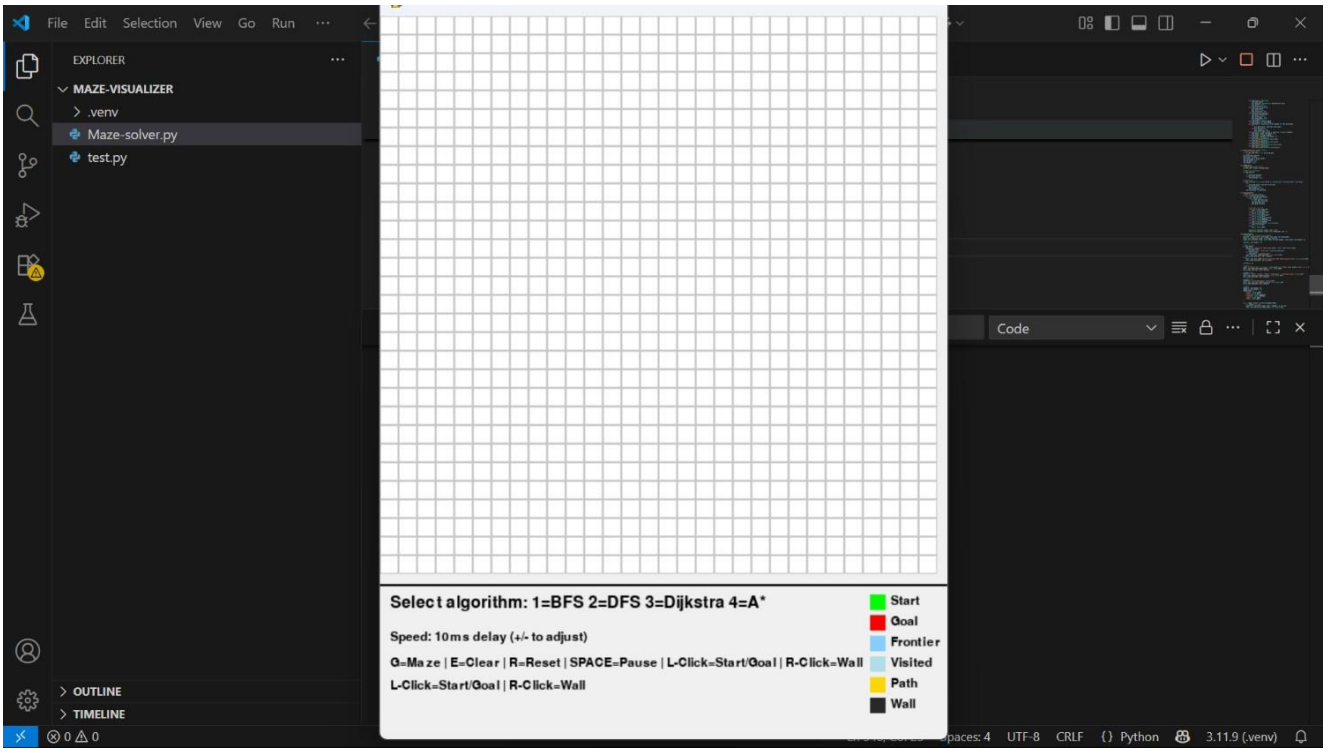
Output Display

- Start node: **Green**
- Goal node: **Red**
- Walls: **Black**
- Frontier cells: **Sky Blue**
- Visited cells: **Light Blue**
- Final shortest path: **Yellow**

Keyboard Controls:

Key	Action
1	Run BFS
2	Run DFS
G	Generate Maze
E	Clear Grid
R	Reset Search
Left click	Place Start/Goal
Right click	Toggle Walls
Space	Pause
+/-	Speed Control

Results/Output



Applications

1. Robotics

- Robots use path-finding to move without hitting obstacles.
- BFS/DFS help robots explore and find routes in warehouses, rescue areas, etc.

2. Games

- Maze-solving and path-finding are used in games for:
  - Enemy chasing
  - Player movement
  - Navigation in game maps
- Example: Characters finding a way in a maze or around walls.

3. Computer Networks

- Used to find the shortest route for data to travel between computers.
- BFS helps in routing packets efficiently.

4. Artificial Intelligence

- AI uses BFS/DFS for searching solutions in problems like:
  - Puzzle solving
  - Planning steps to reach a goal
- They are basic search techniques used in bigger AI systems.

5. Map Navigation

- Concepts of BFS are used in:
  - GPS systems
  - Self-driving cars
  - Drones
- They help find the shortest or best path on a map.

6. Learning & Teaching

Maze solvers are great tools to learn:

- How search algorithms work
- Difference between BFS & DFS
- How computers explore and decide paths

7. Image Processing

Used in:

- Area filling (like paint bucket tool)
- Finding connected shapes in images

Conclusion

In this project, we successfully implemented and visualized maze solving using **Breadth-First Search (BFS)** and **Depth-First Search (DFS)** algorithms. Both algorithms were applied on a grid-based maze environment, and their behavior was observed in real time. BFS explored the maze level-by-level and always found the **shortest path** from the start to the goal, while DFS explored deeper routes first and sometimes produced longer paths, depending on the maze structure.

The visual simulation helped clearly understand how each algorithm works internally — how nodes are visited, how paths are formed, and how search patterns differ. BFS proved to be more efficient for finding the shortest route, whereas DFS was more suitable for simply determining whether a path exists, especially in deep or narrow mazes.

This project demonstrates the usefulness of classical search algorithms in real-world applications such as robotics, gaming, maps, and intelligent navigation systems. It also builds a strong foundation for learning advanced pathfinding approaches like A*, Dijkstra's Algorithm, and heuristic-based navigation. Overall, the study shows that BFS and DFS remain powerful and essential techniques for solving grid-based pathfinding problems and understanding graph search strategies.

References

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein,
   Introduction to Algorithms (CLRS), 3rd Edition, MIT Press, 2009.

2. Edsger W. Dijkstra,
   "A Note on Two Problems in Connexion with Graphs,"
   Numerische Mathematik, 1959.

3. Konrad Zuse,
   Early Concepts of Computer Algorithms and Search Techniques, 1945.

4. E. F. Moore,
   "Shortest Path Through a Maze,"
   Proceedings of the International Symposium on the Theory of Switching, Harvard University, 1959.

5. Lee, C. Y.,
   "An Algorithm for Path Connections and Its Applications,"
   IRE Transactions on Electronic Computers, 1961.

6. Hart, P. E., Nilsson, N. J., Raphael, B.,
   "A Formal Basis for the Heuristic Determination of Minimum Cost Paths,"
   IEEE Transactions on Systems Science and Cybernetics, 1968.

7. Russell, S., Norvig, P.,
   Artificial Intelligence: A Modern Approach, 3rd Edition, Pearson, 2010.

8. Pygame Development Team,
   Pygame Documentation,
   https://www.pygame.org/docs/

9. VisuAlgo,
   "Graph Algorithms and Visualizations,"
   https://visualgo.net/

10. GeeksforGeeks,
    "BFS and DFS in Graph,"
    https://www.geeksforgeeks.org/bfs-vs-dfs-breadth-first-search-and-depth-first-search/