Maze Generator And Solver

Depth-First Search

TABLE OF CONTENTS

1. INTRODUCTION & MOTIVATION

INTRODUCTION

•Maze generation algorithms create complex mazes for puzzles, game design, and AI pathfinding training.

•This project focuses on implementing a **maze generation algorithm** using **Depth-First Search (DFS)** with **backtracking**.

•Python and PyGame are used to visualize the algorithm's step-by-step process.

MOTIVATION

•**Educational Value**: This project demonstrates foundational concepts in **data structures and algorithms (DSA)**, particularly **stacks** and **recursion**.

•**Visualization**: By animating the maze generation, we better understand **DFS**, **backtracking**, and **grid-based pathfinding**.

•**Application**: This approach is widely used in game design, AI, and robotics for pathfinding and navigation within grid environments.

2. PROBLEM STATEMENT

In computer science and game design, maze generation is essential for creating engaging, randomized environments that challenge players and test algorithms. Generating a maze involves navigating a grid to create pathways while maintaining structure, coherence, and solvability.

Key Problems:

1. Random Path Generation:
- Creating a unique path layout each time is complex. Randomized paths are needed, but they should form a single, clear solution from start to end to meet the criteria of a perfect maze (one path between any two points).

2. Handling Dead Ends with Efficiency:
- A robust algorithm is required to manage dead ends in paths. Depth-first search (DFS) with backtracking allows efficient dead-end management by revisiting previous cells to ensure the entire grid is accessible.

3. Visualization of the Maze Creation Process:
- Displaying maze generation in real time is essential for visual understanding and debugging, but it requires careful handling of each step to prevent lag or errors in representation.

4. Solution Path Identification:
- After the maze is created, an algorithm must trace the path from the start to the end. This requires a clear system of marking and tracking visited cells to create an accurate solution.

5. Algorithm Selection for Optimal Performance:
- The selected algorithm should handle the grid efficiently, mark cells correctly, and be adaptable for different grid sizes and environments.

3. BACKGROUND STUDY

Maze generation algorithms have evolved significantly, from static, predefined patterns to dynamic, automated systems that produce unique mazes in real time. Early approaches relied on simple grid manipulation and linear paths, while modern algorithms incorporate recursive backtracking and randomized search techniques to create intricate and engaging layouts.

•**Classic Approaches**:

- ***Static Patterns*****:** Early maze designs were often manually created or followed fixed patterns, which limited maze variability and predictability.
- ***Depth-First Search (DFS)*****:** As computing advanced, depth-first search algorithms were introduced to generate mazes by recursively exploring paths and backtracking when reaching dead ends.

•**Modern Techniques**:

- ***Recursive Backtracking*****:** DFS with recursive backtracking is widely used to generate "perfect mazes," which feature only one solution path. This method uses stacks to manage cells and provides control over dead-end and path management.
- ***Randomized Depth-First Search*****:** Adding randomness to DFS enables unique maze designs each time by randomizing path choices, enhancing user engagement through unpredictable layouts.

•**Comparison with Other Algorithms**:

- ***Prim's and Kruskal's Algorithms*****:** These algorithms create mazes using minimum spanning tree methods, often resulting in different structures than recursive backtracking, with fewer dead ends and a more web-like design.
- ***Breadth-First Search (BFS)*****:** Although less common for maze generation, BFS is sometimes used for maze solving due to its ability to find the shortest path, though it is less effective for randomized maze creation.

4. TECHNOLOGY STACK

1. **Python**: Core programming language used for implementing the algorithm, given its readability and support for a wide range of libraries.
2. **Pygame**: A popular Python library used for game development, which provides tools to render graphics, handle user input, and animate movements within the maze.
3. **Data Structures**:
   •***Stack***: Used to manage the depth-first search backtracking process.
   •***Grid/List***: Holds cell coordinates, tracks visited cells, and stores the maze layout.
4. **Algorithms**:
   •***Depth-First Search (DFS)***: The recursive algorithm used for maze generation, enabling each cell to be explored until dead ends are reached, at which point backtracking is applied.
   •***Randomization***: Integrated DFS to ensure maze paths are unique by randomizing cell selection for each step.
5. **Graphics and Visualization**: ***Color Coding***: Utilized within the Pygame library to visually differentiate between maze paths (green for active path, blue for backtracked paths, and yellow for the solution path).
6. **Libraries**: ***Time***: Used to introduce delays, which slow down the execution to visualize the maze generation and pathfinding processes.

5. PROJECT IMPLEMENTATION

Recursive implementation

The **depth-first search algorithm** of maze generation is frequently implemented using **backtracking**. This can be described with a following **recursive** routine:

1. Given a current cell as a parameter
2. Mark the current cell as visited
3. While the current cell has any unvisited neighbour cells
    1. Choose one of the unvisited neighbours
    2. Remove the wall between the current cell and the chosen cell
    3. Invoke the routine recursively for the chosen cell

which is invoked once for any initial cell in the area.

INITIAL SETUP AND IMPORTS:

```
import pygame      # For creating the graphical interface
import time       # For adding delays in animation
import random     # For making random choices while generating the maze

# Window settings
WIDTH = 500       # Window width in pixels
HEIGHT = 600      # Window height in pixels
FPS = 30          # Frames per second - controls animation speed

# Color definitions in RGB format
WHITE = (255, 255, 255)    # White color for maze walls
GREEN = (0, 255, 0)        # Green for current cell
BLUE = (0, 0, 255)         # Blue for visited paths
YELLOW = (255, 255, 0)     # Yellow for solution path
```

Project Implementation

PyGame Initialization:

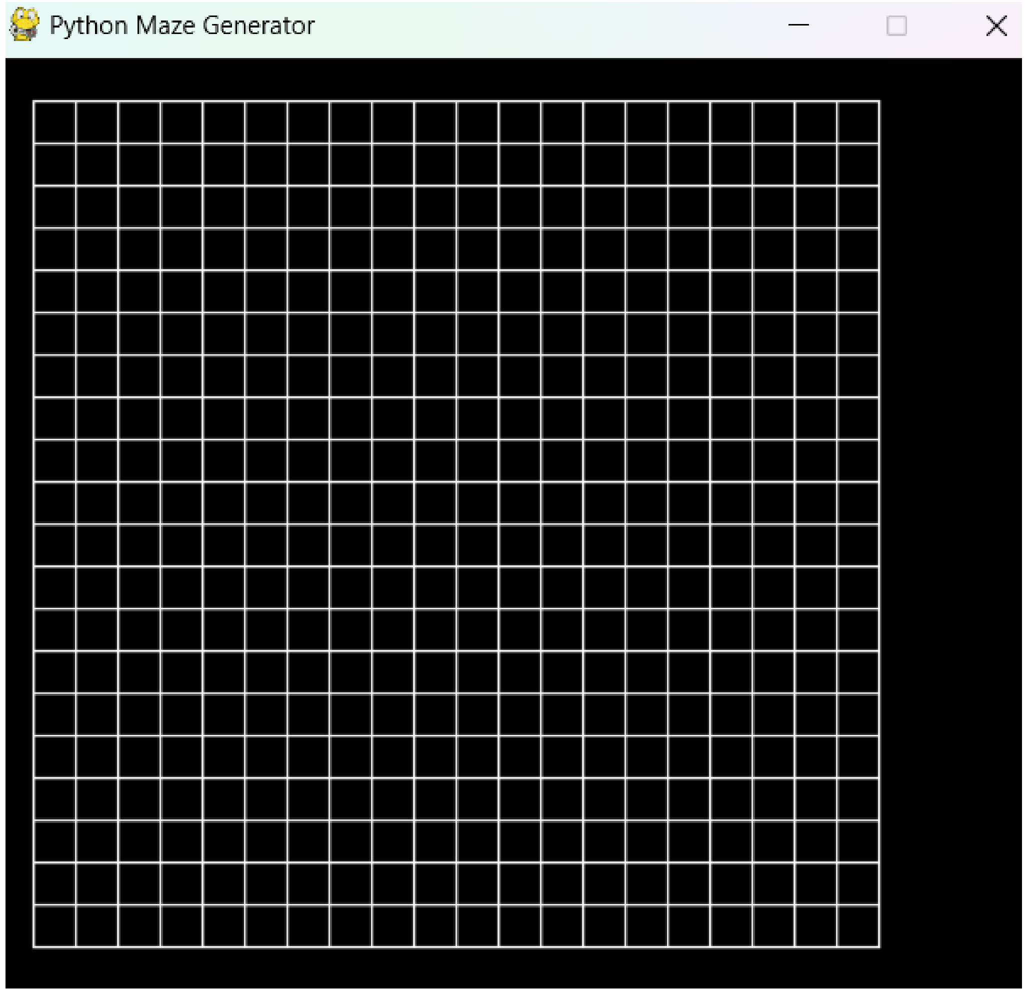
```
pygame.init()                # Initialize all pygame modules
screen = pygame.display.set_mode((WIDTH, HEIGHT))   # Create display window
pygame.display.set_caption("Python Maze Generator") # Set window title
clock = pygame.time.Clock()  # Create clock object to control game speed
```

IMPORT VARIABLES:

```
x = 0          # Current x position
y = 0          # Current y position
w = 20         # Width of each cell in pixels
grid = []      # Stores all cell coordinates
visited = []   # Keeps track of visited cells
stack = []     # Used for backtracking
solution = {}  # Stores the solution path
```

build_grid Function:

```
def build_grid(x, y, w):
    # Creates a 20x20 grid of cells
    for i in range(1,21):           # 20 rows
        x = 20                      # Reset x to starting position
        y = y + 20                  # Move down to next row
        for j in range(1, 21):      # 20 columns
            # Draw four lines for each cell
            pygame.draw.line(screen, WHITE, [x, y], [x + w, y])          # Top
            pygame.draw.line(screen, WHITE, [x + w, y], [x + w, y + w])  # Right
            pygame.draw.line(screen, WHITE, [x + w, y + w], [x, y + w])  # Bottom
            pygame.draw.line(screen, WHITE, [x, y + w], [x, y])          # Left
            grid.append((x,y))      # Add cell coordinates to grid list
            x = x + 20              # Move to next column
```



THE MOVEMENT FUNCTIONS:

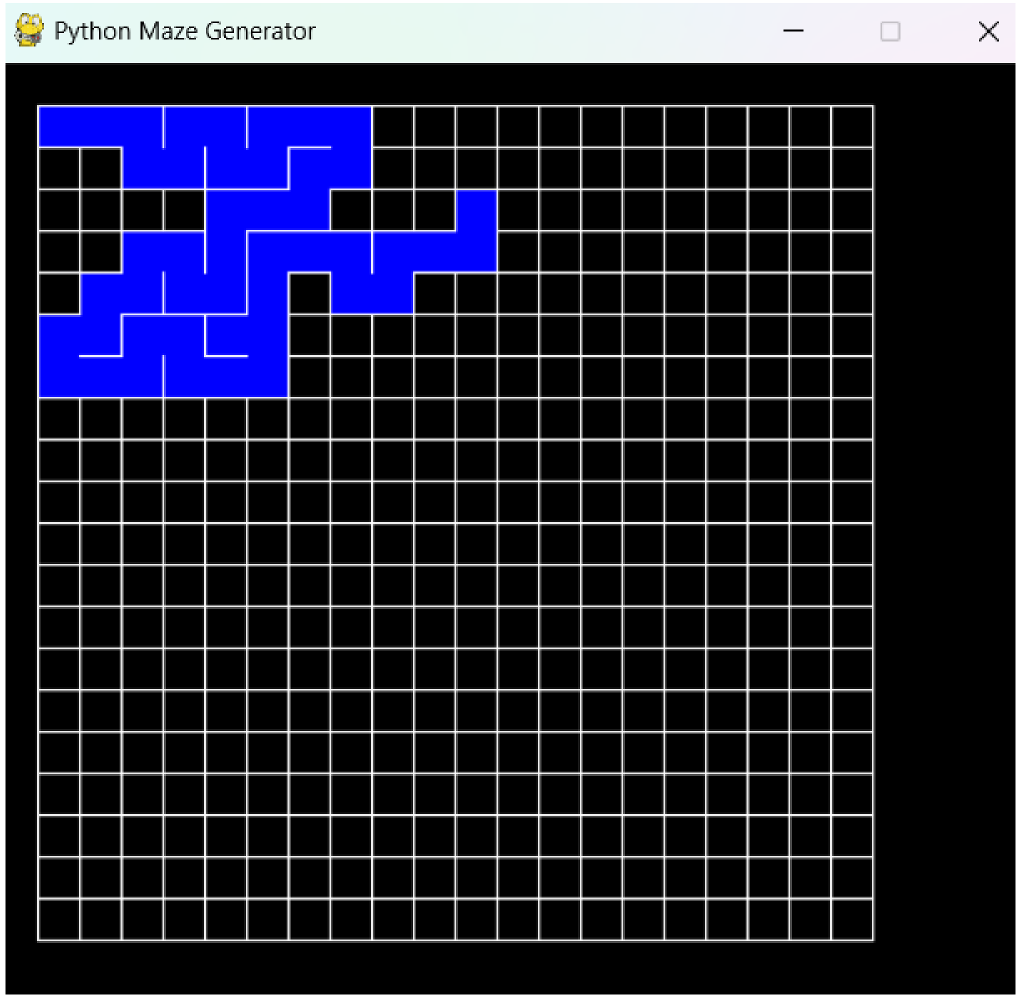
```
def push_up(x, y):
    # Removes wall between current cell and cell above
    pygame.draw.rect(screen, BLUE, (x + 1, y - w + 1, 19, 39), 0)

def push_down(x, y):
    # Removes wall between current cell and cell below
    pygame.draw.rect(screen, BLUE, (x + 1, y + 1, 19, 39), 0)

def push_left(x, y):
    # Removes wall between current cell and cell to the left
    pygame.draw.rect(screen, BLUE, (x - w + 1, y + 1, 39, 19), 0)

def push_right(x, y):
    # Removes wall between current cell and cell to the right
    pygame.draw.rect(screen, BLUE, (x + 1, y + 1, 39, 19), 0)
```

Cell coloring functions:

```
def single_cell(x, y):
    # Colors current cell green
    pygame.draw.rect(screen, GREEN, (x + 1, y + 1, 18, 18), 0)

def backtracking_cell(x, y):
    # Colors cell blue when backtracking
    pygame.draw.rect(screen, BLUE, (x + 1, y + 1, 18, 18), 0)

def solution_cell(x,y):
    # Draws yellow dot in cell for solution path
    pygame.draw.rect(screen, YELLOW, (x + 8, y + 8, 5, 5), 0)
```

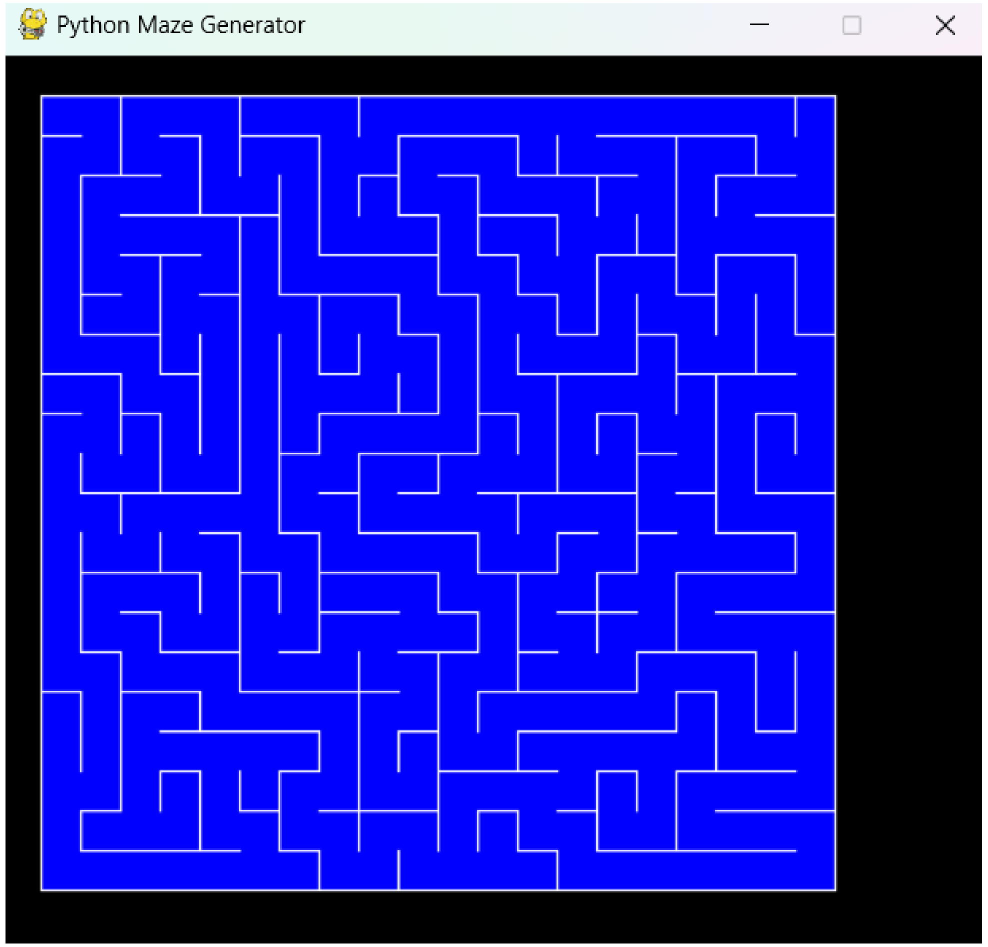
MAZE GENERATION FUNCTION:

```
def carve_out_maze(x,y):          # Uses depth-first search with backtracking to generate maze
    single_cell(x, y)                            # starting positing of maze
    stack.append((x,y))                          # place starting cell into stack
    visited.append((x,y))                        # add starting cell to visited list
    while len(stack) > 0:                        # loop until stack is empty
        time.sleep(.07)                          # slow program now a bit
        cell = []                                # define cell list
        # Check all four directions
        # Add direction to cell list if neighbor exists and is unvisited
        if (x + w, y) not in visited and (x + w, y) in grid:      # right cell available?
            cell.append("right")                                  # if yes add to cell list

        if (x - w, y) not in visited and (x - w, y) in grid:      # left cell available?
            cell.append("left")

        if (x , y + w) not in visited and (x , y + w) in grid:    # down cell available?
            cell.append("down")

        if (x, y - w) not in visited and (x , y - w) in grid:     # up cell available?
            cell.append("up")

        if len(cell) > 0:                                         # check to see if cell list is empty
            cell_chosen = (random.choice(cell))                   # select one of the cell randomly
            # Move in chosen direction and update necessary lists
            if cell_chosen == "right":                            # if this cell has been chosen
            push_right(x, y)                                      # call push_right function
                solution[(x + w, y)] = x, y                       # solution = dictionary key = new cell, other = current cell
```



MAZE GENERATION FUNCTION:

```
                        x = x + w                         # make this cell the current cell
        visited.append((x, y))                       # add to visited list
        stack.append((x, y))                         # place current cell on to stack
    elif cell_chosen == "left":
        push_left(x, y)
        solution[(x - w, y)] = x, y
        x = x - w
        visited.append((x, y))
        stack.append((x, y))
    elif cell_chosen == "down":
        push_down(x, y)
        solution[(x , y + w)] = x, y
        y = y + w
        visited.append((x, y))
        stack.append((x, y))
    elif cell_chosen == "up":
        push_up(x, y)
        solution[(x , y - w)] = x, y
        y = y - w
        visited.append((x, y))
        stack.append((x, y))
else:
    x, y = stack.pop()                            # if no cells are available pop one from the stack - backtrack
    single_cell(x, y)                            # use single_cell function to show backtracking image
    time.sleep(.05)                              # slow program down a bit
    backtracking_cell(x, y)                        # change colour to green to identify backtracking path
```



SOLUTION PLOTTING FUNCTION:

```
def plot_route_back(x,y):
    # Traces path from end to start using solution dictionary
    while (x, y) != (20,20):       # Until reaching start
        solution_cell(x, y)        # Mark current cell
        x, y = solution[x, y]      # Move to next cell in solution
```

Run The Program

```
x, y = 20, 20                  # starting position of grid
build_grid(40, 0, 20)            # 1st argument = x value, 2nd
argument = y value, 3rd argument = width of cell
carve_out_maze(x,y)              # call build the maze  function
plot_route_back(400, 400)        # call the plot solution function
```

Main Game Loop

```
running = True
while running:
    # keep running at the at the right speed
    clock.tick(FPS)
    # process input (events)
    for event in pygame.event.get():
        # check for closing the window
        if event.type == pygame.QUIT:
            running = False
```

6. OUTPUT

OUTPUT OF THE MAZE GENERATION SYSTEM:

•The system visually generates a maze on a 20x20 grid using **depth-first search** with **backtracking**.

COLOR CODES FOR EACH PHASE:

•**Green Cells**: Represent the current path being carved out in the maze. This shows the active cell being explored.

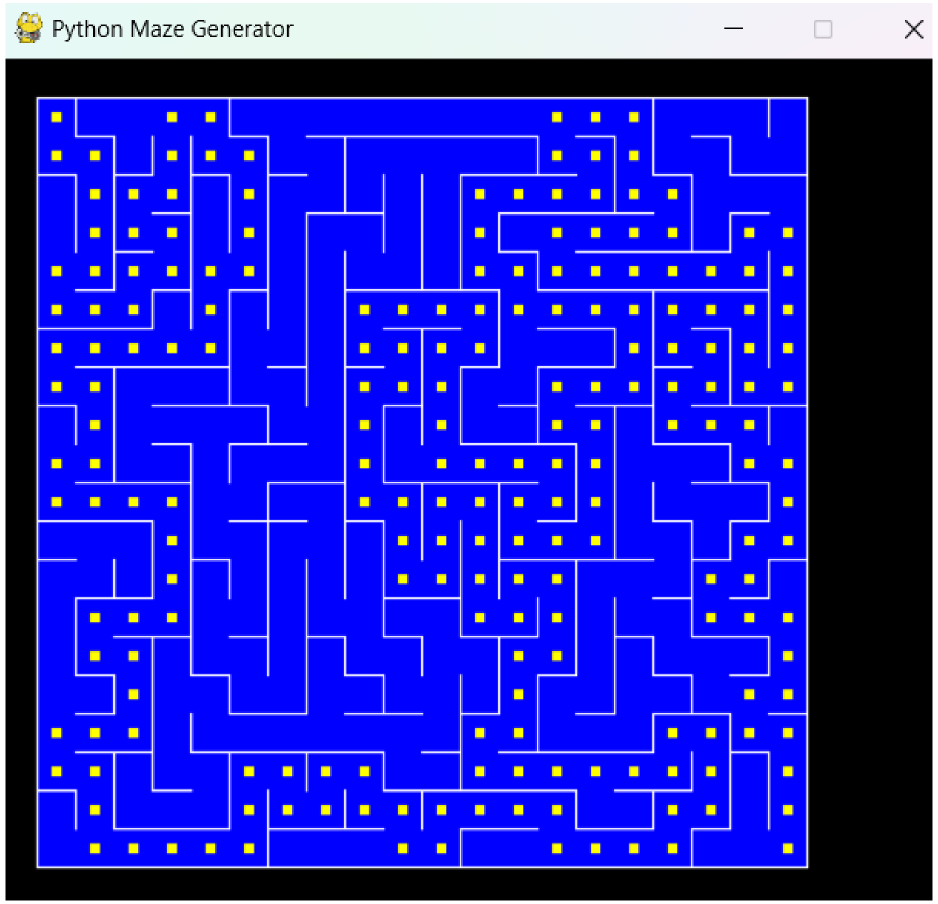
•**Blue Cells**: Indicate backtracking, where the algorithm revisits cells after reaching a dead end.

•**Yellow Dots**: Highlight the solution path from the maze's endpoint back to the starting point.

RESULT:

•A randomly generated maze with a unique solution path is displayed.

•The output reflects the maze layout, and users can observe both the path creation and backtracking phases in real time.



7. CONCLUSION

- The **Maze Generator And Solver Project** successfully demonstrates the use of **Depth-First Search (DFS)** with **backtracking** to create a visually engaging and algorithm-driven maze.

- By leveraging **Python** and **Pygame**, the project visually illustrates **recursion, stack-based backtracking and graph traversal concepts**, all crucial for understanding **pathfinding and navigation systems**.

- This approach to maze generation is foundational for applications in **game development, AI navigation, and simulations**.

8. FUTURE SCOPE

1.**Algorithm Exploration**: Implement alternative maze generation algorithms, such as Prim's or Kruskal's, to compare efficiency and maze complexity.

2.**Scalability**: Adapt the code for larger maze grids and optimize memory usage, allowing for more extensive mazes on various devices.

3.**3D Maze Generation**: Expand the project to generate and visualize 3D mazes, creating an immersive experience suitable for VR applications.

4.**AI Pathfinding Integration**: Add AI algorithms like A* or Dijkstra's to enable autonomous navigation within the maze, which could be used in game development or robotics.

5.**User Interface**: Develop an interactive web or desktop interface where users can generate custom mazes, adjust difficulty, and select maze dimensions.

These enhancements will improve the project’s **utility, adaptability, and user engagement**, making it a comprehensive tool for **learning and experimentation** in algorithmic concepts and game design.

THANK YOU