Mavaldi Rizqy Hazdi

5025211086

DAA (E)

Random Maze Generator and Solver with DFS Algorithm in Python Programming Language

This program generates a random maze and solves it using the Depth-First Search (DFS) algorithm. It uses the Pygame library to display the maze and its solution. Let's break down the code into its components:

```python
import pygame
import random

# Maze dimensions
WIDTH = 25
HEIGHT = 20
CELL_SIZE = 50
WINDOW_SIZE = (WIDTH * CELL_SIZE, HEIGHT * CELL_SIZE)

# Colors
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
GREEN = (0, 255, 0)
RED = (255, 0, 0)
```

Import libraries and define constants:

- Import the required libraries: pygame for creating the graphical interface and random for generating random numbers.
- Define the maze dimensions (WIDTH, HEIGHT, CELL_SIZE) and window size (WINDOW_SIZE).
- Define color constants (WHITE, BLACK, GREEN, RED) for drawing the maze and solution.

```
class Cell:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.visited = False
        self.walls = {'top': True, 'right': True, 'bottom': True, 'left': True}

    def draw(self, screen):
        x, y = self.x * CELL_SIZE, self.y * CELL_SIZE
        if self.visited:
            pygame.draw.rect(screen, WHITE, (x, y, CELL_SIZE, CELL_SIZE))

        if self.walls['top']:
            pygame.draw.line(screen, BLACK, (x, y), (x + CELL_SIZE, y))
        if self.walls['right']:
            pygame.draw.line(screen, BLACK, (x + CELL_SIZE, y), (x + CELL_SIZE, y + CELL_SIZE))
        if self.walls['bottom']:
            pygame.draw.line(screen, BLACK, (x, y + CELL_SIZE), (x + CELL_SIZE, y + CELL_SIZE))
        if self.walls['left']:
            pygame.draw.line(screen, BLACK, (x, y), (x, y + CELL_SIZE))
```

Create a Cell class that represents a single cell in the maze. Each cell has an x and y coordinate, a visited flag, and a dictionary of walls (top, right, bottom, left). The draw method is used to draw the cell on the screen.

```
    def draw(self, screen):
        x, y = self.x * CELL_SIZE, self.y * CELL_SIZE
        if self.visited:
            pygame.draw.rect(screen, WHITE, (x, y, CELL_SIZE, CELL_SIZE))

        if self.walls['top']:
            pygame.draw.line(screen, BLACK, (x, y), (x + CELL_SIZE, y))
        if self.walls['right']:
            pygame.draw.line(screen, BLACK, (x + CELL_SIZE, y), (x + CELL_SIZE, y + CELL_SIZE))
        if self.walls['bottom']:
            pygame.draw.line(screen, BLACK, (x, y + CELL_SIZE), (x + CELL_SIZE, y + CELL_SIZE))
        if self.walls['left']:
            pygame.draw.line(screen, BLACK, (x, y), (x, y + CELL_SIZE))
```

This part of the code defines the draw method of the Cell class, which is responsible for drawing the cell and its walls on the screen.

The draw method takes two arguments:selfandscreen. selfrefers to the current instance of theCellclass, andscreen` is the Pygame surface on which the cell will be drawn.

The method first calculates the position of the cell on the screen based on its x and y coordinates and the CELL_SIZE constant. It then checks if the cell has been visited and, if so, draws a white rectangle on the screen to represent the cell.

Next, the method checks each wall of the cell (top, right, bottom, left) and, if the wall is present, draws a black line on the screen to represent the wall. The pygame.draw.line function takes four

arguments: the Pygame surface on which to draw the line (screen), the color of the line (BLACK), the starting point of the line (the x and y coordinates of the cell), and the ending point of the line (the x and y coordinates of the adjacent cell).

In summary, the draw method of the Cell class is responsible for drawing the cell and its walls on the screen using Pygame's pygame.draw.rect and pygame.draw.line functions.

```
def generate_maze(width, height):
    stack = []
    maze = [[Cell(x, y) for y in range(height)] for x in range(width)]
    current = maze[0][0]
    current.visited = True
    stack.append(current)

    while stack:
        neighbors = []
        x, y = current.x, current.y

        if x > 0 and not maze[x - 1][y].visited:
            neighbors.append(('left', maze[x - 1][y]))
        if x < width - 1 and not maze[x + 1][y].visited:
            neighbors.append(('right', maze[x + 1][y]))
        if y > 0 and not maze[x][y - 1].visited:
            neighbors.append(('top', maze[x][y - 1]))
        if y < height - 1 and not maze[x][y + 1].visited:
            neighbors.append(('bottom', maze[x][y + 1]))

        if neighbors:
            direction, next_cell = random.choice(neighbors)
            current.walls[direction] = False
            next_cell.walls[{'left': 'right', 'right': 'left', 'top': 'bottom', 'bottom': 'top'}[direction]] = False
            next_cell.visited = True
            stack.append(next_cell)
            current = next_cell
        else:
            current = stack.pop()

    return maze
```

This part of the code defines the generate_maze function, which generates a random maze using the Depth-First Search (DFS) algorithm. Here's a detailed explanation of the function:

1. Initialize variables:
    - Create an empty stack to keep track of visited cells.
    - Create a 2D array of Cell objects to represent the maze.
    - Set the starting cell (top-left corner) as the current cell and mark it as visited.
    - Add the current cell to the stack.
2. While the stack is not empty, do the following:

- Find all unvisited neighbors of the current cell and add them to a list of neighbors.
- For each neighbor, determine the direction to the neighbor and add it to the neighbors list as a tuple of the direction and the neighbor cell.
- If there are any unvisited neighbors, randomly select one and remove the wall between the current cell and the neighbor cell in the chosen direction.
- Mark the neighbor cell as visited, add it to the stack, and set it as the new current cell.
- If there are no unvisited neighbors, backtrack by popping the last cell from the stack and setting it as the new current cell.

3. Return the maze.

In summary, the generate_maze function generates a random maze using DFS by selecting a random unvisited neighbor of the current cell, removing the wall between the current cell and the neighbor cell, and repeating the process until all cells have been visited. The function returns a 2D array of Cell objects representing the maze.

```python
def solve_maze(maze, start, end):
    stack = [(start, [])]
    visited = set()

    while stack:
        current, path = stack.pop()
        if current == end:
            return path + [current]

        visited.add(current)
        x, y = current.x, current.y

        for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            nx, ny = x + dx, y + dy
            if 0 <= nx < WIDTH and 0 <= ny < HEIGHT:
                neighbor = maze[nx][ny]
                if neighbor not in visited and not current.walls[{'-1,0': 'left', '1,0': 'right', '0,-1': 'top', '0,1': 'bottom'}[f'{dx},{dy}']]:
                    stack.append((neighbor, path + [current]))
```

This part of the code defines the solve_maze function, which finds a path from the start cell to the end cell in the given maze. Here's a detailed explanation of the function:

1. Initialize:
    - Create a stack to keep track of visited cells and their paths.
    - Create a set to keep track of visited cells.
    - Add the start cell to the stack with an empty path.
2. While the stack is not empty, do the following:

- Pop the last cell and its path from the stack.
- If the current cell is the end cell, return the path to the end cell.
- Add the current cell to the set of visited cells.
- For each neighbor of the current cell, check if it has not been visited and there is no wall between the current cell and the neighbor cell in the direction of the neighbor cell.
- If the neighbor cell meets these conditions, add it to the stack with the path to the current cell appended to its path.

3. If the end cell is not found, return None.

In summary, the solve_maze function finds a path from the start cell to the end cell in the given maze by using a stack to keep track of visited cells and their paths. The function checks each neighbor of the current cell to see if it has not been visited and there is no wall between the current cell and the neighbor cell in the direction of the neighbor cell. If a neighbor cell meets these conditions, it is added to the stack with the path to the current cell appended to its path. The function returns the path to the end cell if it is found, or None if it is not found.

```
def draw_solution(screen, solution):
    for cell in solution:
        x, y = cell.x * CELL_SIZE, cell.y * CELL_SIZE
        pygame.draw.rect(screen, GREEN, (x + CELL_SIZE // 4, y + CELL_SIZE // 4, CELL_SIZE // 2, CELL_SIZE // 2))
```

This part of the code defines the draw_solution function, which draws the solution path on the screen. Here's a detailed explanation of the function:

1. The function takes two arguments: screen, which is the Pygame surface on which to draw the solution, and solution, which is a list of Cell objects representing the solution path.
2. For each cell in the solution path, do the following:
    - Calculate the position of the cell on the screen based on its x and y coordinates and the CELL_SIZE constant.
    - Draw a green rectangle on the screen to represent the cell. The rectangle is centered on the cell and has a width and height of CELL_SIZE // 2.

In summary, the draw_solution function draws the solution path on the screen by iterating through the list of Cell objects representing the solution path and drawing a green rectangle for each cell. The rectangles are centered on the cells and have a width and height of CELL_SIZE // 2.

```python
def main():
    pygame.init()
    screen = pygame.display.set_mode(WINDOW_SIZE)
    pygame.display.set_caption('Maze Generator and Solver')
    clock = pygame.time.Clock()

    maze = generate_maze(WIDTH, HEIGHT)
    solution = solve_maze(maze, maze[0][0], maze[WIDTH - 1][HEIGHT - 1])

    running = True
    while running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False

        screen.fill(WHITE)
        for row in maze:
            for cell in row:
                cell.draw(screen)

        draw_solution(screen, solution)
        pygame.display.flip()
        clock.tick(60)

    pygame.quit()

if __name__ == '__main__':
    main()
```

This part of the code defines the main function, which is the entry point of the program. Here's a detailed explanation of the function:

1. Initialize Pygame:
    - Call pygame.init() to initialize Pygame.
    - Create a Pygame surface with the size specified by the WINDOW_SIZE constant.
    - Set the caption of the window to "Maze Generator and Solver".
    - Create a Pygame clock object to control the frame rate.
2. Generate the maze and find the solution:
    - Call the generate_maze function to generate a maze with the specified WIDTH and HEIGHT.
    - Call the solve_maze function to find a solution path from the top-left cell to the bottom-right cell of the maze.
3. Enter the main loop:
    - While the program is running, do the following:
        - Check for Pygame events, such as quitting the program.

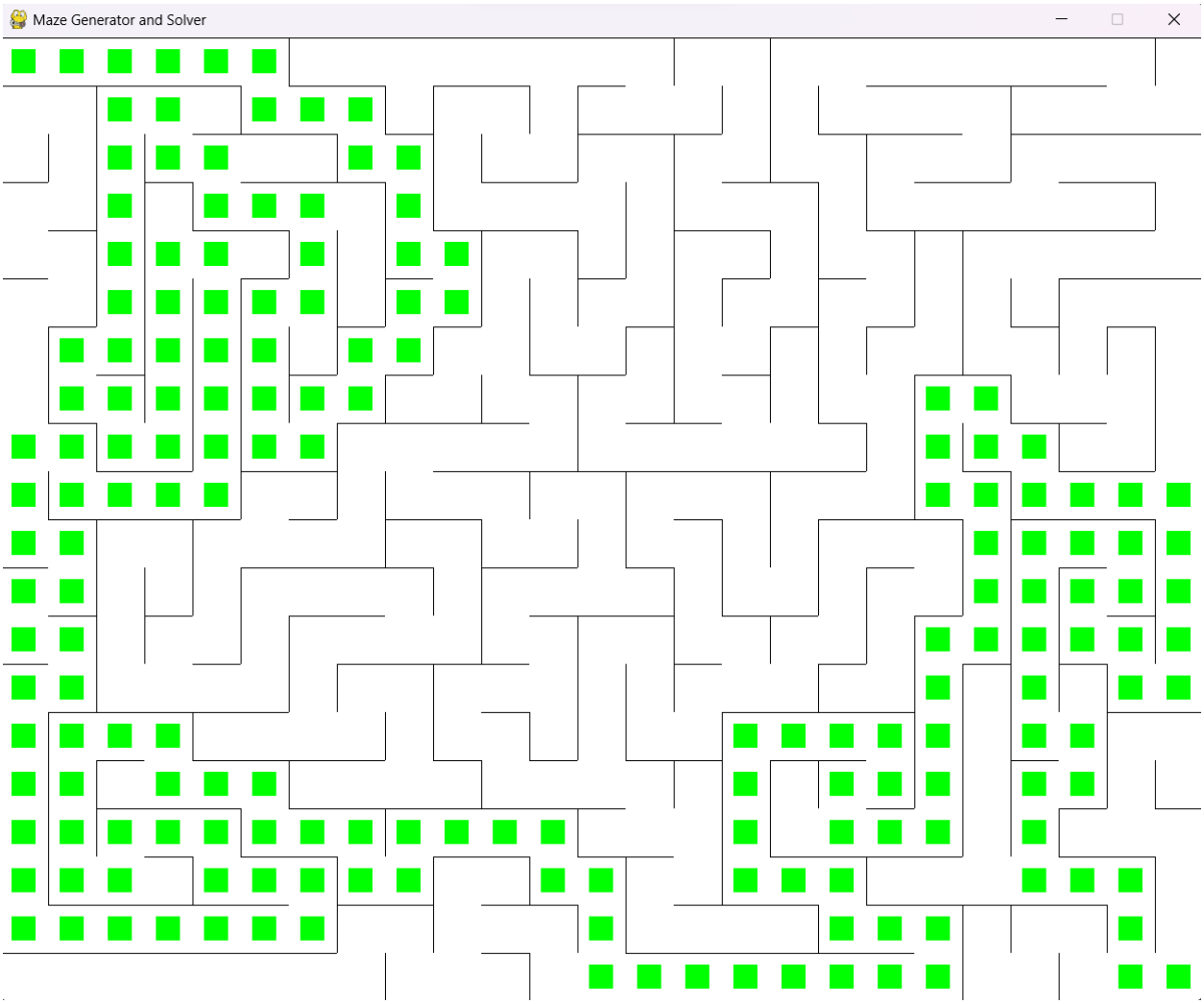
- Fill the screen with white.
- Draw each cell of the maze on the screen using the draw method of the Cell class.
- Draw the solution path on the screen using the draw_solution function.
- Update the display.
- Control the frame rate by calling clock.tick(60) to limit the frame rate to 60 FPS.

4. Quit Pygame:
    - Call pygame.quit() to quit Pygame.

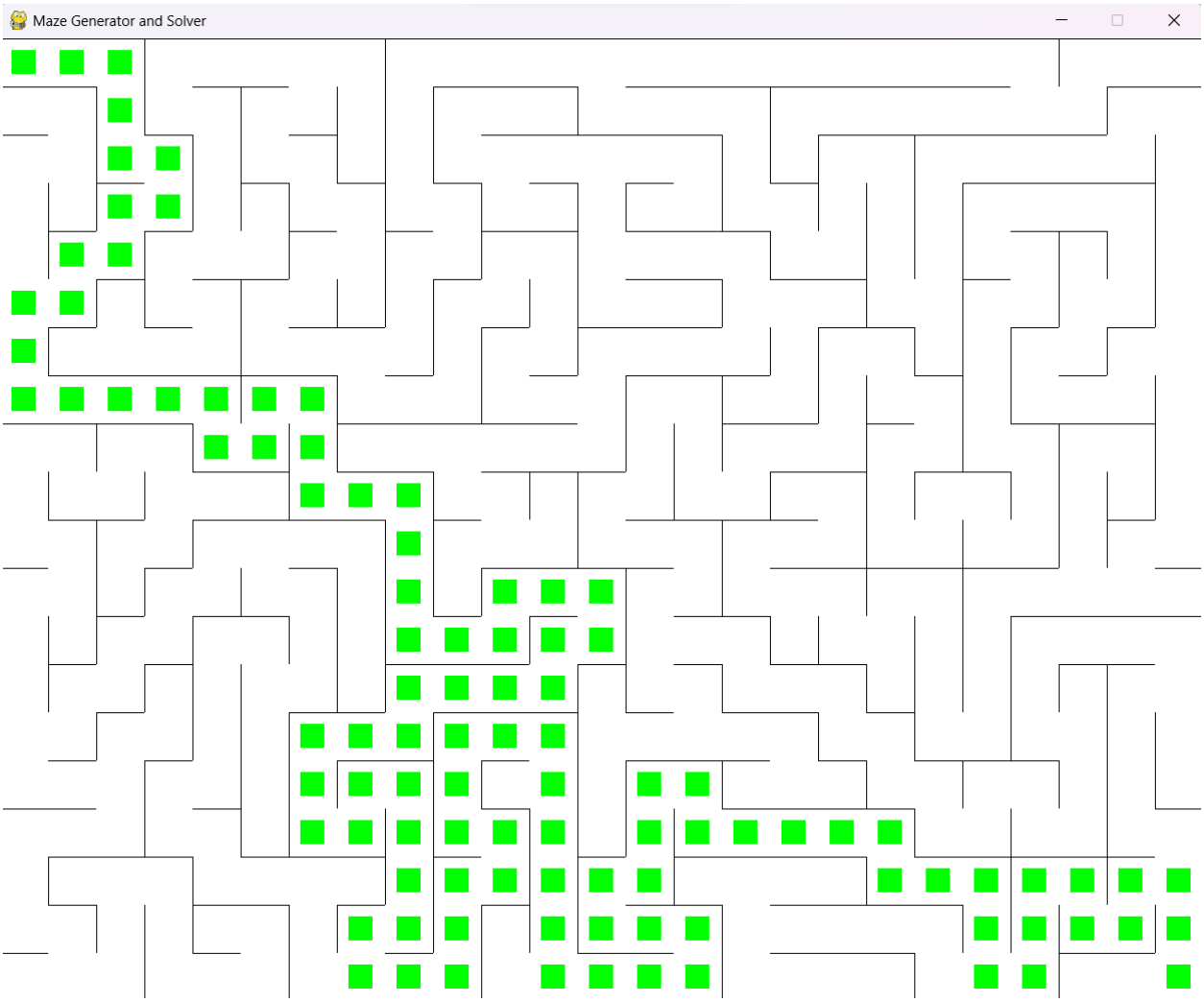
In summary, the main function initializes Pygame, generates a maze and finds a solution path, enters the main loop to draw the maze and solution on the screen, and quits Pygame when the program is finished.

The output of running the program is such as below:

- First run:



- Second run:



From both run/output we can see that the shape of the maze is always change because the maze was randomly generated based on the width and length in the program. From the image we can see that the solver will give the way/direction or solution to the start of the maze in top left to the end of the maze in bottom right and the solver give green path for the solution.

Here is the complete implementation of the program:

```python
import pygame
import random

# Maze dimensions
WIDTH = 25
HEIGHT = 20
CELL_SIZE = 50
WINDOW_SIZE = (WIDTH * CELL_SIZE, HEIGHT * CELL_SIZE)

# Colors
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
GREEN = (0, 255, 0)
RED = (255, 0, 0)

class Cell:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.visited = False
        self.walls = {'top': True, 'right': True, 'bottom': True, 'left': True}

    def draw(self, screen):
        x, y = self.x * CELL_SIZE, self.y * CELL_SIZE
        if self.visited:
            pygame.draw.rect(screen, WHITE, (x, y, CELL_SIZE, CELL_SIZE))

        if self.walls['top']:
            pygame.draw.line(screen, BLACK, (x, y), (x + CELL_SIZE, y))
        if self.walls['right']:
            pygame.draw.line(screen, BLACK, (x + CELL_SIZE, y), (x + CELL_SIZE, y + CELL_SIZE))
        if self.walls['bottom']:
            pygame.draw.line(screen, BLACK, (x, y + CELL_SIZE), (x + CELL_SIZE, y + CELL_SIZE))
        if self.walls['left']:
            pygame.draw.line(screen, BLACK, (x, y), (x, y + CELL_SIZE))

def generate_maze(width, height):
    stack = []
    maze = [[Cell(x, y) for y in range(height)] for x in range(width)]
    current = maze[0][0]
    current.visited = True
    stack.append(current)

    while stack:
        neighbors = []
        x, y = current.x, current.y

        if x > 0 and not maze[x - 1][y].visited:
            neighbors.append(('left', maze[x - 1][y]))
        if x < width - 1 and not maze[x + 1][y].visited:
            neighbors.append(('right', maze[x + 1][y]))
        if y > 0 and not maze[x][y - 1].visited:
            neighbors.append(('top', maze[x][y - 1]))
        if y < height - 1 and not maze[x][y + 1].visited:
            neighbors.append(('bottom', maze[x][y + 1]))

        if neighbors:
            direction, next_cell = random.choice(neighbors)
            current.walls[direction] = False
            next_cell.walls[{'left': 'right', 'right': 'left', 'top': 'bottom', 'bottom': 'top'}[direction]] = False
            next_cell.visited = True
            stack.append(next_cell)
            current = next_cell
        else:
            current = stack.pop()

    return maze
```

```
def solve_maze(maze, start, end):
    stack = [(start, [])]
    visited = set()

    while stack:
        current, path = stack.pop()
        if current == end:
            return path + [current]

        visited.add(current)
        x, y = current.x, current.y

        for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            nx, ny = x + dx, y + dy
            if 0 <= nx < WIDTH and 0 <= ny < HEIGHT:
                neighbor = maze[nx][ny]
                if neighbor not in visited and not current.walls[{'-1,0': 'left', '1,0': 'right', '0,-1': 'top',
'0,1': 'bottom'}[f'{dx},{dy}']]:
                    stack.append((neighbor, path + [current]))

def draw_solution(screen, solution):
    for cell in solution:
        x, y = cell.x * CELL_SIZE, cell.y * CELL_SIZE
        pygame.draw.rect(screen, GREEN, (x + CELL_SIZE // 4, y + CELL_SIZE // 4, CELL_SIZE // 2, CELL_SIZE // 2))

def main():
    pygame.init()
    screen = pygame.display.set_mode(WINDOW_SIZE)
    pygame.display.set_caption('Maze Generator and Solver')
    clock = pygame.time.Clock()

    maze = generate_maze(WIDTH, HEIGHT)
    solution = solve_maze(maze, maze[0][0], maze[WIDTH - 1][HEIGHT - 1])

    running = True
    while running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False

        screen.fill(WHITE)
        for row in maze:
            for cell in row:
                cell.draw(screen)

        draw_solution(screen, solution)
        pygame.display.flip()
        clock.tick(60)

    pygame.quit()

if __name__ == '__main__':
    main()
```