Maze Generator and Solver Project

This project is a Python-based Maze Generator and Solver implemented using the Tkinter library.

The maze is generated using a recursive backtracking algorithm, and the solution is found using the Breadth-First Search (BFS) algorithm.

The application provides a graphical user interface for visualizing the maze generation and solution process.

Features:

1. Generates mazes of customizable dimensions.

2. Solves the maze using BFS, tracing the shortest path from start to end.

3. Visualizes both the maze generation and the solving process.

How it Works:

- Maze Generation: Uses a recursive backtracking algorithm to generate a perfect maze.

- Maze Solving: Uses Breadth-First Search to find the shortest path from the top-left corner to the bottom-right corner.

- Visualization: Each step of the maze generation and solving process is animated in real-time.

Python Code:

```
import tkinter as tk
import random
from collections import deque

class MazeApp:
    def __init__(self, root, rows, cols, cell_size):
        self.root = root
        self.rows = rows
        self.cols = cols
        self.cell_size = cell_size
         self.canvas = tk.Canvas(root, width=cols * cell_size, height=rows * cell_size,
bg="white")
```

```
        self.canvas.pack()
        self.grid = [[0 for _ in range(cols)] for _ in range(rows)]
        self.visited = [[False for _ in range(cols)] for _ in range(rows)]
        self.directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]  # Right, Down, Left, Up

        self.generate_maze(0, 0)
        self.draw_maze()

                        self.start_button   =   tk.Button(root,   text="Start   Solving",
command=self.solve_maze)
        self.start_button.pack()

    def generate_maze(self, x, y):
        self.visited[x][y] = True
        random.shuffle(self.directions)
        for dx, dy in self.directions:
            nx, ny = x + dx, y + dy
            if 0 <= nx < self.rows and 0 <= ny < self.cols and not self.visited[nx][ny]:
                self.grid[x][y] |= self.get_bit(dx, dy)
                self.grid[nx][ny] |= self.get_bit(-dx, -dy)
                self.generate_maze(nx, ny)

    def get_bit(self, dx, dy):
        if dx == 0 and dy == 1:  # Right
            return 1
        elif dx == 1 and dy == 0:  # Down
            return 2
        elif dx == 0 and dy == -1:  # Left
            return 4
        elif dx == -1 and dy == 0:  # Up
            return 8

    def draw_maze(self):
        for x in range(self.rows):
            for y in range(self.cols):
                cell_x = y * self.cell_size
                cell_y = x * self.cell_size

                if not (self.grid[x][y] & 1):  # Right wall
                        self.canvas.create_line(cell_x + self.cell_size, cell_y, cell_x +
self.cell_size, cell_y + self.cell_size)
                if not (self.grid[x][y] & 2):  # Down wall
                        self.canvas.create_line(cell_x, cell_y + self.cell_size, cell_x +
self.cell_size, cell_y + self.cell_size)
                if not (self.grid[x][y] & 4):  # Left wall
                                self.canvas.create_line(cell_x,  cell_y,  cell_x,  cell_y +
self.cell_size)
                if not (self.grid[x][y] & 8):  # Up wall
                        self.canvas.create_line(cell_x, cell_y, cell_x + self.cell_size,
cell_y)

    def solve_maze(self):
```

```
        self.bfs_solve(0, 0)

    def bfs_solve(self, start_x, start_y):
        queue = deque([(start_x, start_y)])
        came_from = {(start_x, start_y): None}

        while queue:
            x, y = queue.popleft()

            if (x, y) == (self.rows - 1, self.cols - 1):
                self.trace_path(came_from, x, y)
                return

            for dx, dy in self.directions:
                nx, ny = x + dx, y + dy
                    if 0 <= nx < self.rows and 0 <= ny < self.cols and (nx, ny) not in
came_from:
                     if self.grid[x][y] & self.get_bit(dx, dy):
                         queue.append((nx, ny))
                         came_from[(nx, ny)] = (x, y)

    def trace_path(self, came_from, x, y):
        while (x, y) is not None:
                  self.canvas.create_oval(y * self.cell_size + self.cell_size // 4, x *
self.cell_size + self.cell_size // 4,
                                          y * self.cell_size + 3 * self.cell_size // 4, x *
self.cell_size + 3 * self.cell_size // 4,
                                       fill="blue")
            self.root.update()
            self.root.after(50)  # Delay for animation
            if came_from[(x, y)] is None:
                break  # Reached the start of the maze
            x, y = came_from[(x, y)]

if __name__ == "__main__":
    rows, cols = 20, 20  # Maze dimensions
    cell_size = 20       # Size of each cell

    root = tk.Tk()
    root.title("Maze Generator and Solver")
    app = MazeApp(root, rows, cols, cell_size)

    root.mainloop()
```

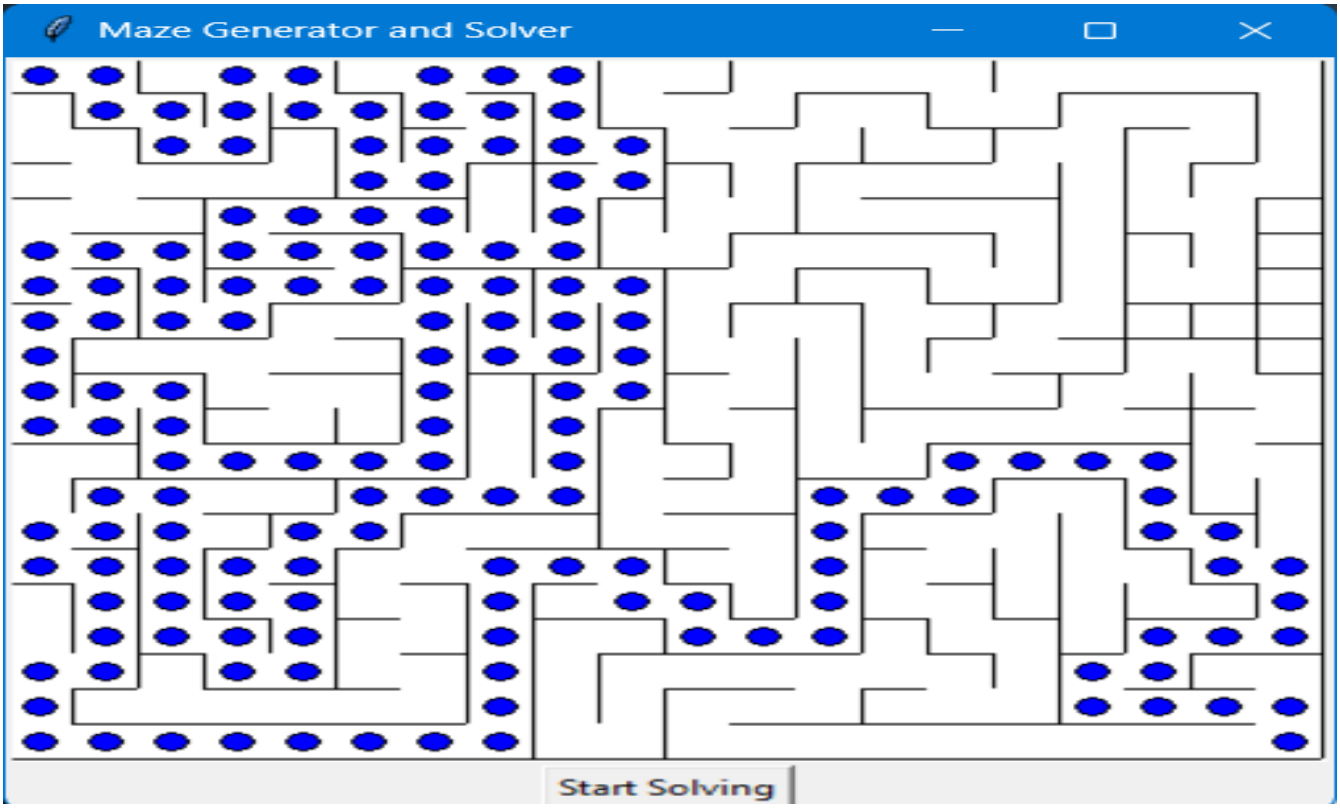

Figure: Visualization of the maze generation and solving process.