数据结构与算法课程设计报告
Report

尹超

中国科学院大学，北京 100049

Carter Yin

University of Chinese Academy of Sciences, Beijing 100049, China

2025.6

序言

本文为笔者数据结构与算法的课程设计报告。

望老师批评指正。

目录

迷宫生成与求解桌面应用程序设计

1.1 问题分析

本项目旨在设计并实现一个能够生成和求解迷宫的桌面应用程序。根据题目要求，一个迷宫是由 $m \times n$ 个格子组成的矩形，迷宫内部的每个格子有 4 个方向，每个方向或者有障碍（如墙）不能通过，或者无障碍而能通过。迷宫在相对的两条边分别包含一个入口和一个出口。

1.1.1 基本要求

程序需要满足以下基本要求：

- 设计一个迷宫及其障碍的表示方式，并能够随机生成迷宫。
- 设计并实现迷宫路径寻找算法，自动找到从入口到出口的一条路径。
- 如有多条路径，设计并实现一个算法找到步数最少的路径。

1.1.2 进阶项

程序实现了以下进阶功能：

- 通过图形界面显示迷宫和路径。
- 实现非矩形迷宫（本项目中为三角形迷宫）的生成和路径寻找，其出口可以在迷宫的“内部”或边界。本项目重点实现了非矩形迷宫（三角形）的生成与求解，未涉及“闯入名画”的蒙德里安几何迷宫布局。

1.1.3 程序功能概述

基于上述要求，本程序实现了以下核心功能：

(1) 迷宫类型支持：程序支持生成和显示两种类型的迷宫：
- **矩形迷宫**：由用户指定的行数和列数定义的传统网格结构。入口和出口通常位于相对的边界。
- **三角形迷宫**：由等边三角形单元组成的三角网格结构，提供了一种非矩形的迷宫体验。其起点和终点可以根据结构特点设定。

(2) 迷宫生成：采用随机深度优先搜索（Randomized Depth-First Search）算法生成迷宫。该算法能确保迷宫的所有单元格都是连通的。

(3) 迷宫求解：提供两种经典的路径搜索算法：
- **广度优先搜索 (BFS)**：用于找到从起点到终点的最短路径（以经过的单元格数量计）。
- **深度优先搜索 (DFS)**：用于找到从起点到终点的一条路径（不一定是最短的）。

(4) 图形用户界面 (GUI)：
- 使用 Python 的 Tkinter 库构建用户友好的图形界面。
- 允许用户选择迷宫类型（矩形或三角形）并输入相应的尺寸参数。
- 提供操作按钮，用于触发迷宫生成、选择求解算法 (BFS/DFS) 进行路径搜索以及清除已显示的路径。
- 在画布 (Canvas) 组件上直观地显示迷宫的墙壁、单元格。起点、终点和求解路径以不同颜色高亮显示。
- 包含一个状态栏，用于向用户反馈当前操作信息或结果。

1.2 数据结构设计与实现

为了有效地表示和操作迷宫，我们设计了以下核心数据结构，主要通过 `Maze` 类和 `MazeApp` 类实现。

1.2.1 `Maze` 类

`Maze` 类封装了迷宫的逻辑结构、生成算法和求解算法。

核心属性

- `self.type` (str): 存储迷宫的类型，例如 `"rectangular"` 或 `"triangular"`。
- `self.cells` (dict): 这是最核心的数据结构，一个字典，用于存储迷宫中所有单元格的详细信息。
  - **键 (key)**: `cell_id`，一个元组，作为单元格的唯一标识符。例如，矩形迷宫的单元格 ID 为 `(row, col)`，三角形迷宫的单元格 ID 为 `(row, index_in_row)`。
  - **值 (value)**: 另一个字典，包含该单元格的具体属性（详见 2.1.2 单元格数据结构）。
- `self.start_node` (tuple): 存储起点单元格的 `cell_id`。
- `self.end_node` (tuple): 存储终点单元格的 `cell_id`。
- **特定类型属性**:
  - 对于矩形迷宫: `self.rows` (int), `self.cols` (int)。
  - 对于三角形迷宫: `self.num_triangle_rows` (int)。

单元格数据结构

`self.cells` 字典中的每个值（代表一个单元格）本身也是一个字典，包含以下关键字段:

- `'walls'` (dict): 表示当前单元格与其相邻单元格之间的墙壁状态。
  - **键 (key)**: 相邻单元格的 `cell_id`。
  - **值 (value)**: 布尔值。`True` 表示该方向存在墙壁，`False` 表示墙壁已被打通（即存在路径）。
- `'visited_gen'` (bool): 在迷宫生成阶段（随机 DFS）使用，标记该单元格是否已被访问。
- `'visited_solve'` (bool): 在路径求解阶段（BFS/DFS）使用，标记该单元格是否已被访问。
- `'parent'` (tuple/None): 在路径求解算法执行后，存储路径上前一个单元格的 `cell_id`，用于从终点回溯以重建路径。
- **特定类型的几何/显示数据**:
  - 矩形迷宫: `'rect_coords'`: `(col, row)`，主要用于绘图时的坐标计算。
  - 三角形迷宫:
    * `'is_up'` (bool): 指示三角形单元是尖端向上还是尖端向下。
    * `'vertices'` (list): 包含三个 `(x, y)` 坐标元组的列表，表示三角形单元的顶点坐标，用于在 Canvas 上绘图。
    * `'center_coords'` (tuple): `(x, y)` 坐标元组，表示单元格的几何中心，用于绘制路径时连接各点。

实现细节

- **迷宫初始化**:
  - `_initialize_rectangular_grid()`: 对于矩形迷宫，此方法遍历所有行列组合，创建每个单元格对象，并初始化其所有四个方向的墙壁状态为 `True` (即默认存在墙壁)。

  - `_initialize_triangular_grid()`: 对于三角形迷宫，根据总行数和每行的单元格数量（奇数个，交替朝上和朝下）创建单元格。同样，初始化所有相邻单元格间的墙壁状态为 `True`。
- **墙壁表示**: 墙壁的存在与否通过每个单元格的 `'walls'` 字典来表示。如果单元格 A 与单元格 B 之间的墙壁被打通，则 `cells[cell_A]['walls'][cell_B]` 和 `cells[cell_B]['walls'][cell_A]` 都会被设置为 `False`。
- **辅助数据结构**:
  - `collections.deque`: 在随机 DFS 生成算法和 DFS 求解算法中用作栈（Stack LIFO），在 BFS 求解算法中用作队列（Queue FIFO）。

1.2.2 MazeApp 类

`MazeApp` 类负责构建图形用户界面 (GUI)、处理用户交互事件，并调用 `Maze` 对象的相应方法来执行迷宫的生成、求解和显示。

UI 元素与状态管理

- **UI 组件**: 包含多种 Tkinter 控件，如 `tk.Frame` (用于布局), `tk.Radiobutton` (选择迷宫类型), `tk.Entry` (输入尺寸), `tk.Button` (执行操作), `tk.Canvas` (绘制迷宫), `tk.Label` (显示状态)。
- **状态管理属性**:
  - `self.maze` (Maze object): 指向当前活动（已生成或正在操作）的 `Maze` 实例。
  - `self.current_path` (list/None): 存储当前已找到并显示的求解路径（一个 `cell_id` 列表），如果没有路径或路径已清除，则为 `None`。
  - `self.maze_type` (str): 存储用户当前选中的迷宫类型。

绘图逻辑

- `_draw_rectangular_maze()` 和 `_draw_triangular_maze()` 方法: 这些方法负责根据 `self.maze` 对象中存储的单元格数据（包括墙壁状态、起点、终点）在 `Canvas` 上绘制迷宫的视觉表示。
- `_draw_path_on_canvas()` 方法: 如果 `self.current_path` 不为空，此方法会在已绘制的迷宫上用特定颜色（蓝色）和线宽绘制出路径，并在路径的最后一个线段末端显示箭头以指示方向。
- `_update_canvas_size_and_coords()` 方法: 在生成新迷宫或更改参数后，此方法会根据迷宫类型和尺寸动态调整 `Canvas` 的大小，并计算每个单元格在画布上的精确绘图坐标（如矩形单元的左上角和右下角坐标，三角形单元的顶点坐标和中心点坐标）。

1.3 算法复杂度分析

设 V 为迷宫中单元格的总数量，E 为单元格之间潜在边的数量（即可能的通道或墙壁）。对于本项目中实现的网格状迷宫（矩形、三角形），E 与 V 成正比关系，例如，在矩形迷宫中 $E \approx 2V$。因此，时间复杂度中的 $O(V+E)$ 通常可以简化为 $O(V)$。

1.3.1 迷宫生成 (随机深度优先搜索 - `generate_maze_randomly`)

- **过程描述**: 算法从一个随机选择（或预设）的起始单元格开始，将其标记为已访问，并将其压入一个栈中。当栈不为空时，查看栈顶单元格。如果它有未被访问的邻居单元格，则随机选择一个未访问的邻居，打通当前单元格与该邻居之间的墙壁，将该邻居标记为已访问，并将其压入栈中。如果

栈顶单元格没有未访问的邻居（即陷入死胡同），则从栈中弹出一个单元格（回溯），继续处理新的栈顶单元格。此过程持续直到栈为空，此时所有可达单元格均已访问。

- **时间复杂度**: $O(V)$。每个单元格最多被访问一次并压入栈一次。每条被打通的边（即墙壁被移除）被处理一次。
- **空间复杂度**: $O(V)$。主要用于存储 `self.cells` 数据结构本身。此外，算法执行过程中使用的栈（通过 `collections.deque` 实现）在最坏情况下（例如，迷宫形成一条非常长的蜿蜒路径）其深度可能达到 V。

1.3.2 迷宫求解 (广度优先搜索 - `solve_bfs`)

- **过程描述**: BFS 算法从起点单元格开始，将其加入一个队列并标记为已访问。当队列不为空时，算法从队列中取出一个单元格（队首），然后检查其所有相邻且之间没有墙壁（即路径通畅）并且尚未被访问的邻居单元格。对于每个这样的邻居，将其标记为已访问，记录其父节点（即从哪个单元格到达此邻居，用于后续路径回溯），并将其加入队列。此过程重复进行，直到找到终点单元格或队列为空（表示没有路径）。由于 BFS 按层级扩展搜索，它找到的第一条路径即为最短路径（以经过的单元格数量衡量）。
- **时间复杂度**: $O(V)$。每个单元格和每条通道（无墙的边）最多被访问和处理一次。
- **空间复杂度**: $O(V)$。主要用于存储 `self.cells`（包括在求解过程中更新的 `'visited_solve'` 和 `'parent'` 标记）以及 BFS 算法使用的队列。在最坏情况下，队列中可能包含接近所有 V 个单元格（例如，在一个星形或非常开阔的迷宫中）。

1.3.3 迷宫求解 (深度优先搜索 - `solve_dfs`)

- **过程描述**: DFS 算法同样从起点单元格开始，将其压入一个栈并标记为已访问。当栈不为空时，算法查看栈顶单元格。如果栈顶单元格是终点，则路径找到，算法结束。否则，算法查找栈顶单元格的一个未被访问且之间没有墙壁的邻居。如果找到这样的邻居，则将该邻居标记为已访问，记录其来源（父节点），并将其压入栈中，然后在新栈顶上继续搜索。如果栈顶单元格没有符合条件的未访问邻居（即当前路径走到尽头），则从栈中弹出该单元格（回溯），尝试从前一个单元格的其他分支继续搜索。
- **时间复杂度**: $O(V)$。与 BFS 类似，每个单元格和每条通道在搜索过程中最多被访问一次（前提是正确地使用 `'visited_solve'` 标记来避免重复访问和无限循环）。
- **空间复杂度**: $O(V)$。用于存储 `self.cells` 数据以及 DFS 算法使用的栈。栈的深度在最坏情况下（例如，迷宫是一条长链）可能达到 V。

1.4 结果分析

1.4.1 功能实现与正确性

- **迷宫生成**: 程序能够根据用户选择的类型（矩形、三角形）和输入的尺寸参数成功生成迷宫。采用的随机深度优先搜索算法确保了生成的迷宫是“完美”的（perfect maze），即迷宫中的任意两个单元格之间都存在唯一的一条路径，并且所有单元格都是连通的。
- **路径求解**:
  - **BFS 算法**能够正确地找到从起点到终点的最短路径（以路径中包含的单元格数量为标准）。

  - **DFS 算法**能够正确地找到一条从起点到终点的有效路径，但这条路径不一定是最短的，其具体形态取决于邻居选择的随机性和迷宫结构。
- **GUI 交互**:
  - 用户可以通过图形界面直观地选择迷宫类型、设置相关参数（如行数、列数、三角形迷宫的行数等）。
  - “Generate Maze”、“Solve (BFS - Shortest)”、“Solve (DFS)”以及“Clear Path”按钮的功能均按预期工作，响应用户操作。
  - 程序对用户输入的参数进行了基本的有效性检查（例如，是否为整数，是否在合理范围内），并通过 Tkinter 的 `messagebox` 模块向用户显示错误或警告信息。
- **可视化效果**:
  - 迷宫的结构，包括单元格和墙壁，都能在画布上清晰地显示出来。
  - 起点单元格以浅绿色高亮显示，终点单元格以粉红色（salmon）高亮显示，易于辨认。
  - 求解出的路径以醒目的蓝色线条在迷宫上绘制，并且在路径的最后一个线段末端会显示一个箭头，清晰地指示了从起点到终点的行进方向。
  - 界面底部的状态栏能够提供实时的操作反馈，例如“Generated rectangular maze. Start: (X, Y), End: (A, B)”，或“Path found using BFS (Shortest Path) with Z steps.”，或“No path found...”。
- **进阶项实现**: 程序成功实现了非矩形迷宫（三角形迷宫）的生成和路径寻找功能。三角形迷宫的起点通常设在顶端，终点设在底部某单元格，符合“出口可在内部（或边界）”的描述。

1.4.2 用户体验

- 图形用户界面的布局相对直观，用户可以比较容易地理解各项功能并进行操作。
- 迷宫和路径的实时可视化使得算法的执行结果一目了然，增强了程序的交互性和趣味性。
- 必要的错误提示和状态反馈有助于用户了解程序的当前状态和操作结果。

1.4.3 局限性与可改进之处

- **性能考量**: 对于生成非常大规模的迷宫（例如，单元格数量远超过几千个），Tkinter 的画布绘图性能可能会成为瓶颈，导致界面响应速度下降。
- **迷宫生成算法单一**: 目前仅实现了随机深度优先搜索（Randomized DFS）作为迷宫生成算法。未来可以考虑引入其他经典的生成算法，如 Prim 算法或 Kruskal 算法（修改版），这些算法可能会生成不同风格和特征的迷宫。
- **起点/终点固定性**: 当前版本的程序中，起点和终点是根据迷宫类型和尺寸通过预设逻辑确定的。可以考虑增加允许用户在生成的迷宫上通过鼠标点击等方式自定义选择起点和终点的功能。
- **视觉效果与定制性**: 迷宫的墙壁颜色、粗细，路径的颜色、粗细等视觉元素的定制性有限。可以提供更多用户自定义选项，以增强程序的个性化体验。
- **功能扩展潜力**:
  - 可以增加保存当前生成的迷宫状态到文件以及从文件加载迷宫的功能。
  - 可以考虑加入迷宫生成过程或路径寻找过程的逐步动画演示，使用户能更直观地理解算法的执行流程。

1.4.4 总结

本项目成功地设计并实现了一个具有图形用户界面的迷宫生成与求解程序。该程序支持矩形和三角形两种不同类型的迷宫，并提供了广度优先搜索（BFS）和深度优先搜索（DFS）两种经典的路径求解算法。通过采用面向对象的设计思想，将迷宫的逻辑结构 (`Maze` 类) 与用户界面及交互逻辑 (`MazeApp` 类) 分离，使得代码结构较为清晰。

程序在功能完整性、算法正确性以及用户交互方面均达到了预期的设计目标。特别地，非矩形迷宫（三角形）的实现满足了进阶要求，为用户提供了更多样化的迷宫体验。尽管在性能优化、功能丰富度等方面仍有提升空间，但作为一项课程设计，本项目较好地完成了核心任务，并为后续的进一步开发和功能扩展奠定了坚实的基础。

附录：代码

完整代码与注释

本节提供了本项目的完整代码，包括迷宫生成与求解的核心逻辑以及图形用户界面的实现。代码中包含了详细的注释，以帮助理解每个部分的功能和实现细节。

```python
import tkinter as tk  # Import the Tkinter library for GUI creation
from tkinter import messagebox  # Import messagebox for displaying alerts
import random  # Import random for shuffling and random choices
from collections import deque  # Import deque for efficient queue (BFS) and stack (DFS) operations
import math  # Import math for calculations like sqrt (used in triangular maze geometry)

class Maze:
    """
    Represents the logical structure of a maze.
    This class handles the initialization of different maze types (rectangular, triangular),
    maze generation using Randomized Depth-First Search, and pathfinding algorithms (BFS, DFS).
    """
    def __init__(self, type="rectangular", rows=10, cols=10,
                 num_triangle_rows=7):
        """
        Initializes a new Maze object.

        Args:
            type (str): The type of maze to create ("rectangular" or "triangular").
            rows (int): Number of rows for a rectangular maze.
            cols (int): Number of columns for a rectangular maze.
            num_triangle_rows (int): Number of rows for a triangular maze.
        """
        self.type = type  # Store the maze type
        self.cells = {}  # Dictionary to store cell data, keyed by cell_id (e.g., (row, col))
        self.start_node = None  # Stores the cell_id of the starting cell
        self.end_node = None  # Stores the cell_id of the ending cell

        if self.type == "rectangular":
            self.rows = rows
            self.cols = cols
            self._initialize_rectangular_grid()  # Initialize the grid structure for a rectangular maze
            # Set default start and end nodes for rectangular mazes
            # Start is typically middle of the left edge, end is middle of the right edge
            self.start_node = (self.rows // 2, 0) if self.rows > 0 and self.cols > 0 else (0,0)
```

```python
            self.end_node = (self.rows // 2, self.cols - 1) if self.rows > 0 and self.cols > 0 else (0,0)
            # Handle edge cases for very small mazes
            if self.rows == 1 and self.cols == 1: self.start_node = self.end_node = (0,0)
            elif self.cols == 1 and self.rows > 1:
                self.start_node = (0,0); self.end_node = (self.rows-1,0)
            elif self.rows == 1 and self.cols > 1:
                self.start_node = (0,0); self.end_node = (0, self.cols-1)


        elif self.type == "triangular":
            self.num_triangle_rows = num_triangle_rows
            self._initialize_triangular_grid()  # Initialize the grid structure for a triangular maze
            # Set default start and end nodes for triangular mazes
            # Start is typically the top-most cell (0,0)
            self.start_node = (0,0)
            if self.num_triangle_rows > 0:
                base_row_idx = self.num_triangle_rows - 1  # Index of the last row
                # End is typically the middle cell of the base row
                middle_idx_in_base = base_row_idx # In a triangular grid, the middle cell index in a row 'r' is 'r'
                self.end_node = (base_row_idx, middle_idx_in_base)
                # Validate that the determined start/end nodes actually exist in the generated cells
                if not self._is_valid_cell_id(self.start_node): self.start_node = None
                if not self._is_valid_cell_id(self.end_node): self.end_node = None
            else:
                # If no rows, no start or end node
                self.start_node = self.end_node = None

    def _initialize_rectangular_grid(self):
        """
        Initializes the `self.cells` dictionary for a rectangular maze.
        Each cell is represented by a tuple (row, col) and stores its walls,
        visited status for generation/solving, parent for path reconstruction,
        and rectangular coordinates for drawing.
        All walls are initially set to True (closed).
        """
        for r in range(self.rows):
            for c in range(self.cols):
                cell_id = (r, c)
                self.cells[cell_id] = {
                    'walls': {},  # Key: neighbor_id, Value: True (wall exists) or False (no wall)
                    'visited_gen': False,  # Visited status for maze generation algorithm
                    'visited_solve': False,  # Visited status for path solving
```

```
algorithm
                    'parent': None,  # Parent cell in the solved path (for BFS/DFS reconstruction)
                    'rect_coords': (c, r)  # Store (col, row) for easier access in drawing
                }
                # Initialize walls with potential neighbors
                if r > 0: self.cells[cell_id]['walls'][(r - 1, c)] = True  # Wall to the North
                if c < self.cols - 1: self.cells[cell_id]['walls'][(r, c + 1)] = True  # Wall to the East
                if r < self.rows - 1: self.cells[cell_id]['walls'][(r + 1, c)] = True  # Wall to the South
                if c > 0: self.cells[cell_id]['walls'][(r, c - 1)] = True  # Wall to the West


    def _initialize_triangular_grid(self):
        """
        Initializes the `self.cells` dictionary for a triangular maze.
        Each cell is represented by (row, index_in_row).
        Cells alternate between pointing up and pointing down.
        Stores walls, visited status, parent, 'is_up' status, and drawing vertices/center.
        All walls are initially set to True (closed).
        """
        if self.num_triangle_rows == 0: return
        # First pass: create all cells and their basic properties
        for r in range(self.num_triangle_rows):
            num_cells_in_row = 2 * r + 1  # Number of triangles in row 'r'
            for i in range(num_cells_in_row):
                cell_id = (r, i)
                is_up = (i % 2 == 0)  # Cells at even indices in a row point up, odd indices point down
                self.cells[cell_id] = {
                    'walls': {},
                    'visited_gen': False,
                    'visited_solve': False,
                    'parent': None,
                    'is_up': is_up  # True if triangle points up, False if it points down
                    # 'vertices' and 'center_coords' will be added by MazeApp._update_canvas_size_and_coords
                }

                # Define potential neighbors and set walls to True (closed)
                # This defines one-way walls initially; the second pass makes them two-way.
                if is_up: # Triangle points up
```

```
                    # Horizontal neighbors (left and right)
                    if i + 1 < num_cells_in_row: self.cells[cell_id]['walls'][(r, i + 1)] = True # Right neighbor (down-pointing)
                    if i - 1 >= 0: self.cells[cell_id]['walls'][(r, i - 1)] = True # Left neighbor (down-pointing)
                    # Neighbor below (down-pointing triangle in the next row)
                    if r + 1 < self.num_triangle_rows and (i + 1) < (2 * (r + 1) + 1): # Check bounds for the cell below
                             self.cells[cell_id]['walls'][(r + 1, i + 1)] = True # Cell (i+1) in row (r+1) is below an up-pointing cell (r,i)
                else: # Triangle points down
                    # Horizontal neighbors (left and right)
                    if i + 1 < num_cells_in_row: self.cells[cell_id]['walls'][(r, i + 1)] = True # Right neighbor (up-pointing)
                    if i - 1 >= 0: self.cells[cell_id]['walls'][(r, i - 1)] = True # Left neighbor (up-pointing)
                    # Neighbor above (up-pointing triangle in the previous row)
                    if r - 1 >= 0 and (i - 1) >= 0 and (i-1) < (2*(r-1)+1): # Check bounds for the cell above
                            self.cells[cell_id]['walls'][(r - 1, i - 1)] = True # Cell (i-1) in row (r-1) is above a down-pointing cell (r,i)

        # Second pass: ensure walls are bidirectional and remove invalid wall entries
        all_cell_ids = list(self.cells.keys()) # Iterate over a copy of keys if modifying dict
        for cell_id_iter in all_cell_ids:
            current_walls = dict(self.cells[cell_id_iter]['walls']) # Iterate over a copy of this cell's walls
            for neighbor_id, wall_exists in current_walls.items():
                if neighbor_id in self.cells: # If the defined neighbor actually exists
                    # Ensure the wall is bidirectional
                    if cell_id_iter not in self.cells[neighbor_id]['walls']:
                         self.cells[neighbor_id]['walls'][cell_id_iter] = True
                else: # If the defined neighbor doesn't exist (e.g., due to boundary conditions in initial setup)
                    # Remove this invalid wall entry from the current cell
                    if neighbor_id in self.cells[cell_id_iter]['walls']:
                        del self.cells[cell_id_iter]['walls'][neighbor_id]

    def _is_valid_cell_id(self, cell_id):
        """
        Checks if a given cell_id is valid (exists within the maze's cells).

        Args:
            cell_id (tuple): The cell identifier to check.

        Returns:
            bool: True if the cell_id is valid, False otherwise.
        """
```

```python
        return cell_id and isinstance(cell_id, tuple) and len(cell_id) == 2 and cell_id in self.cells

    def _get_unvisited_neighbors_gen(self, cell_id):
        """
        Gets a list of unvisited neighboring cells for maze generation (Randomized DFS).
        Neighbors are shuffled to ensure randomness in maze generation.

        Args:
            cell_id (tuple): The cell_id of the current cell.

        Returns:
            list: A list of cell_ids of unvisited neighbors.
        """
        neighbors = []
        if not self._is_valid_cell_id(cell_id) or 'walls' not in self.cells[cell_id]:
            return neighbors # Return empty list if current cell is invalid or has no wall data

        # Get all potential neighbors defined by the 'walls' dictionary keys
        potential_neighbors = list(self.cells[cell_id]['walls'].keys())
        random.shuffle(potential_neighbors) # Shuffle for randomness in choosing the next cell

        for neighbor_id in potential_neighbors:
            # A neighbor is valid for generation if it exists and hasn't been visited yet by the generation algorithm
            if self._is_valid_cell_id(neighbor_id) and not self.cells[neighbor_id]['visited_gen']:
                neighbors.append(neighbor_id)
        return neighbors

    def generate_maze_randomly(self, start_cell_id_param=None):
        """
        Generates the maze structure using a Randomized Depth-First Search (DFS) algorithm.
        This algorithm carves paths by removing walls between cells.

        Args:
            start_cell_id_param (tuple, optional): An optional starting cell for generation.
                                                   If None, uses self.start_node or a default.
        """
        if not self.cells: return # Cannot generate if no cells are initialized

        # Reset visited status and ensure all walls are initially closed (True)
        for cell_id_key in self.cells:
```

```python
        self.cells[cell_id_key]['visited_gen'] = False
        self.cells[cell_id_key]['visited_solve'] = False # Also reset solve state
        self.cells[cell_id_key]['parent'] = None # Also reset parent
        # Ensure all walls are set to True before generation
        if 'walls' in self.cells[cell_id_key]:
             for neighbor in self.cells[cell_id_key]['walls']:
                self.cells[cell_id_key]['walls'][neighbor] = True
                # Ensure bidirectional wall reset if the neighbor also exists and has a wall entry
                if self._is_valid_cell_id(neighbor) and cell_id_key in self.cells[neighbor]['walls']:
                    self.cells[neighbor]['walls'][cell_id_key] = True

    # Determine the starting cell for the generation algorithm
    start_gen_id = None
    if self.start_node and self._is_valid_cell_id(self.start_node): # Prefer the maze's defined start_node
        start_gen_id = self.start_node
    elif start_cell_id_param and self._is_valid_cell_id(start_cell_id_param): # Use parameter if provided and valid
        start_gen_id = start_cell_id_param

    if not start_gen_id and self.cells: # Fallback to the first available cell if no other start defined
        start_gen_id = list(self.cells.keys())[0]

    # If still no valid start_gen_id, cannot proceed
    if not start_gen_id or not self._is_valid_cell_id(start_gen_id):
         if not self.cells: return
         # Ultimate fallback if specific logic fails but cells exist
         start_gen_id = list(self.cells.keys())[0] # Try again with the first cell
         if not self._is_valid_cell_id(start_gen_id): return # Give up if still invalid


    stack = deque()  # Use deque as a stack for the DFS algorithm
    self.cells[start_gen_id]['visited_gen'] = True  # Mark the starting cell as visited
    stack.append(start_gen_id)  # Push the starting cell onto the stack

    while stack:  # Loop as long as there are cells in the stack
        current_cell_id = stack[-1]  # Get the cell at the top of the stack (peek)
        unvisited_neighbors = self._get_unvisited_neighbors_gen(current_cell_id)

        if unvisited_neighbors:
            # If there are unvisited neighbors, choose one randomly
            chosen_neighbor_id = unvisited_neighbors[0] # Already shuffled, so pick the first

```

```
                # "Remove" the wall between the current cell and the chosen neighbor
                self.cells[current_cell_id]['walls'][chosen_neighbor_id] = False
                self.cells[chosen_neighbor_id]['walls'][current_cell_id] = False

                # Mark the chosen neighbor as visited and push it onto the stack
                self.cells[chosen_neighbor_id]['visited_gen'] = True
                stack.append(chosen_neighbor_id)
            else:
                # If there are no unvisited neighbors, backtrack by popping the current cell from the stack
                stack.pop()

    def _get_solve_neighbors(self, cell_id):
        """
        Gets a list of neighboring cells that are accessible (no wall) for path solving.

        Args:
            cell_id (tuple): The cell_id of the current cell.

        Returns:
            list: A list of cell_ids of accessible neighbors.
        """
        neighbors = []
        if not self._is_valid_cell_id(cell_id) or 'walls' not in self.cells[cell_id]: return neighbors

        # Iterate through all potential neighbors defined in the 'walls' dictionary
        for neighbor_id, wall_exists in self.cells[cell_id]['walls'].items():
            # A neighbor is accessible if it's a valid cell and the wall_exists is False (meaning no wall)
            if self._is_valid_cell_id(neighbor_id) and not wall_exists:
                neighbors.append(neighbor_id)
        return neighbors

    def _reset_solve_state(self):
        """
        Resets the 'visited_solve' status and 'parent' pointers for all cells.
        Called before starting a new pathfinding attempt (BFS or DFS).
        """
        for cell_id in self.cells:
            self.cells[cell_id]['visited_solve'] = False
            self.cells[cell_id]['parent'] = None

    def solve_bfs(self):
        """
        Solves the maze using Breadth-First Search (BFS) to find the shortest path
        from self.start_node to self.end_node.

```

```python
        Returns:
            list: A list of cell_ids representing the shortest path, or None if no path is found.
        """
        # Ensure start and end nodes are valid before attempting to solve
        if not self.start_node or not self.end_node or \
           not self._is_valid_cell_id(self.start_node) or \
           not self._is_valid_cell_id(self.end_node):
            return None # Cannot solve if start/end nodes are invalid

        self._reset_solve_state()  # Clear previous solving states
        q = deque()  # Use deque as a queue for BFS

        q.append(self.start_node)  # Add the start node to the queue
        self.cells[self.start_node]['visited_solve'] = True  # Mark start node as visited

        path_found = False
        while q:  # Loop as long as the queue is not empty
            r_id = q.popleft()  # Dequeue the current cell
            if r_id == self.end_node:  # Check if the current cell is the end node
                path_found = True
                break  # Path found, exit loop

            # Explore accessible, unvisited neighbors
            for nr_id in self._get_solve_neighbors(r_id):
                if not self.cells[nr_id]['visited_solve']:
                    self.cells[nr_id]['visited_solve'] = True  # Mark neighbor as visited
                    self.cells[nr_id]['parent'] = r_id  # Set current cell as parent (for path reconstruction)
                    q.append(nr_id)  # Enqueue the neighbor

        if path_found:
            # Reconstruct the path from end_node back to start_node using parent pointers
            path = []
            curr = self.end_node
            while curr is not None:
                path.append(curr)
                if curr == self.start_node: break  # Reached the start of the path
                parent_of_curr = self.cells[curr]['parent']
                # Safety break for malformed parent links (should not happen in correct BFS)
                if curr == parent_of_curr : break
                curr = parent_of_curr
                # Safety break for excessively long paths (longer than total number of cells)
                if len(path) > len(self.cells) + 5 : break
```

```
            return path[::-1]  # Reverse the path to get it from start to end
        return None # No path found

    def solve_dfs(self):
        """
        Solves the maze using Depth-First Search (DFS) to find a path
        from self.start_node to self.end_node. (Not necessarily the shortest).

        Returns:
            list: A list of cell_ids representing a path, or None if no path is found.
        """
        # Ensure start and end nodes are valid
        if not self.start_node or not self.end_node or \
           not self._is_valid_cell_id(self.start_node) or \
           not self._is_valid_cell_id(self.end_node):
            return None

        self._reset_solve_state()  # Clear previous solving states
        stack = deque()  # Use deque as a stack for DFS
        # path_map stores parent pointers for path reconstruction, similar to BFS's self.cells[id]['parent']
        path_map = {self.start_node: None}

        stack.append(self.start_node)  # Push the start node onto the stack
        self.cells[self.start_node]['visited_solve'] = True # Mark start node as visited when pushed

        while stack:  # Loop as long as the stack is not empty
            r_id = stack[-1] # Peek at the top of the stack (current cell)

            if r_id == self.end_node:  # Check if the current cell is the end node
                # Path found, reconstruct it using path_map
                path = []
                curr = self.end_node
                while curr is not None:
                    path.append(curr)
                    curr = path_map.get(curr) # Get parent from path_map
                return path[::-1]  # Reverse to get path from start to end

            found_next_move = False
            neighbors = self._get_solve_neighbors(r_id) # Get accessible neighbors
            random.shuffle(neighbors) # Shuffle to explore different paths on subsequent runs (DFS specific)

            for nr_id in neighbors:
                if not self.cells[nr_id]['visited_solve']:
                    self.cells[nr_id]['visited_solve'] = True # Mark neighbor as visited when pushed
                    path_map[nr_id] = r_id # Record parent
```

```
                    stack.append(nr_id) # Push neighbor onto stack to explore next
                    found_next_move = True
                    break # Move to the new top of the stack

            if not found_next_move: # If no unvisited accessible neighbor was found
                stack.pop() # Backtrack
        return None # No path found

    def open_wall(self, cell1_id, cell2_id):
        """
        Manually opens a wall between two specified cells.
        (Not directly used by the current generation/solving logic but can be a utility).

        Args:
            cell1_id (tuple): The ID of the first cell.
            cell2_id (tuple): The ID of the second cell.
        """
        if self._is_valid_cell_id(cell1_id) and self._is_valid_cell_id(cell2_id):
            # Ensure the wall entry exists before trying to set it to False
            if cell2_id in self.cells[cell1_id]['walls']:
                self.cells[cell1_id]['walls'][cell2_id] = False
            if cell1_id in self.cells[cell2_id]['walls']:
                self.cells[cell2_id]['walls'][cell1_id] = False

class MazeApp:
    """
    Manages the GUI for the maze application using Tkinter.
    It handles user interactions, maze parameter inputs, maze display, and path visualization.
    """
    def __init__(self, root, default_rows=10, default_cols=15,
                 default_tri_rows=7, cell_size=25):
        """
        Initializes the MazeApp GUI.

        Args:
            root (tk.Tk): The main Tkinter window.
            default_rows (int): Default number of rows for rectangular mazes.
            default_cols (int): Default number of columns for rectangular mazes.
            default_tri_rows (int): Default number of rows for triangular mazes.
            cell_size (int): The size (in pixels) of each cell for drawing.
        """
        self.root = root  # The main Tkinter window
        self.cell_size = cell_size  # Size of each cell in pixels for drawing
        self.maze = None  # Will hold the current Maze object
        self.current_path = None  # Will hold the list of cell_ids for the solved path
        self.maze_type = "rectangular"  # Default maze type

```

```
        # Store default dimensions for UI fields
        self.default_rows = default_rows
        self.default_cols = default_cols
        self.default_tri_rows = default_tri_rows


        self.root.title("Labyrinth Solver")  # Set the window title

        # --- Controls Frame Setup ---
        self.controls_frame = tk.Frame(root)  # Frame to hold all control widgets
        self.controls_frame.pack(side=tk.TOP, pady=10, padx=10) # Pack it at the top

        # Maze type selection (Radiobuttons)
        self.maze_type_var = tk.StringVar(value=self.maze_type) # Tkinter string variable for radiobuttons
        tk.Radiobutton(self.controls_frame, text="Rectangular", variable=self.maze_type_var, value="rectangular", command=self.on_maze_type_change).grid(row=0, column=0)
        tk.Radiobutton(self.controls_frame, text="Triangular", variable=self.maze_type_var, value="triangular", command=self.on_maze_type_change).grid(row=0, column=1) # Adjusted column

        # Frame for dynamic parameter inputs (rows/cols or tri_rows)
        self.param_frame = tk.Frame(self.controls_frame)
        self.param_frame.grid(row=1, column=0, columnspan=4, pady=5) # Adjusted columnspan
        self._build_param_inputs() # Initial call to build inputs for the default maze type

        # Action Buttons
        self.generate_btn = tk.Button(self.controls_frame, text="Generate Maze", command=self.generate_maze_action)
        self.generate_btn.grid(row=2, column=0, padx=5, pady=5)

        self.solve_bfs_btn = tk.Button(self.controls_frame, text="Solve (BFS - Shortest)", command=lambda: self.solve_maze_action('bfs'))
        self.solve_bfs_btn.grid(row=2, column=1, padx=5)

        self.solve_dfs_btn = tk.Button(self.controls_frame, text="Solve (DFS)", command=lambda: self.solve_maze_action('dfs'))
        self.solve_dfs_btn.grid(row=2, column=2, padx=5)

        self.clear_path_btn = tk.Button(self.controls_frame, text="Clear Path", command=self.clear_path_display_action)
        self.clear_path_btn.grid(row=2, column=3, padx=5)

        # --- Canvas Setup ---
        self.canvas_width = 400  # Initial canvas width
        self.canvas_height = 400 # Initial canvas height
```

```python
        self.canvas = tk.Canvas(root, width=self.canvas_width, height=self.canvas_height, bg='ivory', highlightthickness=1, highlightbackground="black")
        self.canvas.pack(pady=10, padx=10, expand=True, fill=tk.BOTH) # Pack canvas to fill available space

        # --- Status Label Setup ---
        self.status_label = tk.Label(root, text="Welcome! Select type, adjust size, and generate.", relief=tk.SUNKEN, anchor="w")
        self.status_label.pack(side=tk.BOTTOM, fill=tk.X, padx=10, pady=5) # Pack status label at the bottom

        self.on_maze_type_change() # Initial call to set up UI based on default maze type

    def on_maze_type_change(self):
        """
        Callback function executed when the maze type (Radiobutton) is changed.
        It updates the `self.maze_type` and rebuilds the parameter input fields.
        """
        self.maze_type = self.maze_type_var.get() # Get the newly selected maze type
        self._build_param_inputs() # Rebuild the input fields specific to this maze type

    def _build_param_inputs(self):
        """
        Dynamically builds the input fields (Entry widgets) for maze parameters
        based on the currently selected `self.maze_type`.
        Clears any existing widgets in `self.param_frame` before adding new ones.
        """
        # Clear existing widgets from the parameter frame
        for widget in self.param_frame.winfo_children():
            widget.destroy()

        if self.maze_type == "rectangular":
            tk.Label(self.param_frame, text="Rows:").grid(row=0, column=0, sticky="w")
            self.rows_entry = tk.Entry(self.param_frame, width=5)
            self.rows_entry.insert(0, str(self.default_rows)) # Pre-fill with default
            self.rows_entry.grid(row=0, column=1, padx=(0,10))

            tk.Label(self.param_frame, text="Cols:").grid(row=0, column=2, sticky="w")
            self.cols_entry = tk.Entry(self.param_frame, width=5)
            self.cols_entry.insert(0, str(self.default_cols)) # Pre-fill with default
            self.cols_entry.grid(row=0, column=3, padx=(0,10))


        elif self.maze_type == "triangular":
            tk.Label(self.param_frame, text="Triangle Rows:").grid(row=0, column=0, sticky="w")
            self.tri_rows_entry = tk.Entry(self.param_frame, width=5)
```

```
            self.tri_rows_entry.insert(0, str(self.default_tri_rows)) # Pre-fill with default
            self.tri_rows_entry.grid(row=0, column=1, padx=(0,10))

    def _update_canvas_size_and_coords(self):
        """
        Updates the canvas dimensions and calculates drawing coordinates for each cell
        based on the current maze type, size, and `self.cell_size`.
        Stores calculated coordinates (e.g., 'vertices', 'center_coords') in `self.maze.cells`.
        """
        if not self.maze: return # Do nothing if no maze object exists

        if self.maze.type == "rectangular":
            # Calculate canvas dimensions based on number of cells and cell size
            self.canvas_width = self.maze.cols * self.cell_size
            self.canvas_height = self.maze.rows * self.cell_size
            # Note: For rectangular, 'rect_coords' (col, row) stored in Maze init is sufficient.
            # Path drawing will use these and cell_size to find centers.


        elif self.maze.type == "triangular":
            s = self.cell_size  # Side length of each equilateral triangle cell
            h_small = s * math.sqrt(3) / 2  # Height of each equilateral triangle cell

            if self.maze.num_triangle_rows == 0:
                self.canvas_width = s; self.canvas_height = h_small
            else:
                # Calculate overall canvas dimensions
                self.canvas_width = self.maze.num_triangle_rows * s + s # Max width approx.
                self.canvas_height = self.maze.num_triangle_rows * h_small + h_small # Max height approx.

            # Define an origin point for drawing the triangular grid (e.g., top-center)
            canvas_origin_x = self.canvas_width / 2
            canvas_origin_y = s / 2 # Small offset from the top

            # Calculate and store vertices and center coordinates for each triangular cell
            for r in range(self.maze.num_triangle_rows):
                for i in range(2 * r + 1): # Number of cells in row 'r'
                    cell_id = (r, i)
                    if not self.maze._is_valid_cell_id(cell_id): continue
                    cell_data = self.maze.cells[cell_id]
                    is_up = cell_data['is_up'] # Is this triangle pointing up or down?

                    if is_up: # Triangle points up
```

```python
                        # Calculate peak (top vertex) coordinates
                        peak_x = canvas_origin_x + (i/2.0) * s - r * s / 2.0
                        peak_y = canvas_origin_y + r * h_small
                        # Define the three vertices of the up-pointing triangle
                        v1 = (peak_x, peak_y)
                        v2 = (peak_x - s / 2.0, peak_y + h_small)
                        v3 = (peak_x + s / 2.0, peak_y + h_small)
                        cell_data['vertices'] = [v1, v2, v3]
                        # Calculate center for path drawing (centroid of a triangle)
                        cell_data['center_coords'] = (peak_x, peak_y + h_small * (2.0/3.0))
                    else: # Triangle points down
                        # Calculate base-left vertex coordinates
                        base_left_x = canvas_origin_x + ((i-1)/2.0) * s - r * s / 2.0
                        base_y = canvas_origin_y + r * h_small
                        # Define the three vertices of the down-pointing triangle
                        v1 = (base_left_x, base_y)
                        v2 = (base_left_x + s, base_y)
                        v3 = (base_left_x + s / 2.0, base_y + h_small)
                        cell_data['vertices'] = [v1, v2, v3]
                        # Calculate center for path drawing
                        cell_data['center_coords'] = (base_left_x + s/2.0, base_y + h_small * (1.0/3.0))

        # Apply the calculated dimensions to the canvas widget
        self.canvas.config(width=self.canvas_width, height=self.canvas_height)

    def generate_maze_action(self):
        """
        Action performed when the "Generate Maze" button is clicked.
        It reads parameters from input fields, creates a new Maze object,
        generates the maze structure, updates canvas size, and draws the maze.
        """
        self.maze_type = self.maze_type_var.get() # Get current maze type

        try: # Error handling for user input (e.g., non-integer values)
            if self.maze_type == "rectangular":
                rows = int(self.rows_entry.get())
                cols = int(self.cols_entry.get())
                # Basic validation for rows and columns
                if not (1 <= rows <= 100 and 1 <= cols <= 100):
                    messagebox.showerror("Invalid Input", "Rows/Cols must be between 1 and 100.")
                    return
                self.maze = Maze(type="rectangular", rows=rows, cols=cols)


            elif self.maze_type == "triangular":
                tri_rows = int(self.tri_rows_entry.get())
```

```
                # Basic validation for triangle rows
                if not (1 <= tri_rows <= 30):
                    messagebox.showerror("Invalid Input", "Triangle Rows must be between 1 and 30.")
                    return
                self.maze = Maze(type="triangular", num_triangle_rows=tri_rows)

        except ValueError: # Catch error if input cannot be converted to int
            messagebox.showerror("Invalid Input", "Parameters must be integers.")
            return
        except Exception as e: # Catch any other unexpected errors during maze creation
            messagebox.showerror("Error", f"Could not generate maze: {e}")
            return

        # Check if maze object and its cells were successfully created
        if not self.maze or not self.maze.cells:
            messagebox.showerror("Error", "Maze generation failed (no cells were created).")
            return
        # Warn if start or end nodes are not properly set (should be handled by Maze init)
        if not self.maze.start_node or not self.maze.end_node:
             messagebox.showwarning("Maze Warning", "Maze generated, but start or end node is invalid. Pathfinding may fail.")

        self.maze.generate_maze_randomly()  # Call the maze generation algorithm
        self._update_canvas_size_and_coords()  # Update canvas and cell coordinates for drawing
        self.current_path = None  # Clear any previous path
        self.draw_maze()  # Draw the newly generated maze

        # Update status label
        if self.maze and self.maze.start_node and self.maze.end_node:
            self.status_label.config(text=f"Generated {self.maze.type} maze. Start: {self.maze.start_node}, End: {self.maze.end_node}")
        elif self.maze: # If maze exists but start/end might be problematic
            self.status_label.config(text=f"Generated {self.maze.type} maze. Start/End: {self.maze.start_node}/{self.maze.end_node} (may be invalid).")
        else: # Should not be reached if previous checks pass
             self.status_label.config(text="Maze generation failed.")

    def draw_maze(self):
        """
        Clears the canvas and redraws the current maze.
        Dispatches to the appropriate drawing method based on `self.maze.type`.
        If `self.current_path` exists, it also draws the path.
        """
        self.canvas.delete("all")  # Clear everything from the canvas
        if not self.maze or not self.maze.cells: return # Do nothing if no maze or
```

```
    cells

        # Call the specific drawing function based on maze type
        if self.maze.type == "rectangular":
            self._draw_rectangular_maze()
        elif self.maze.type == "triangular":
            self._draw_triangular_maze()

        # If a path has been solved, draw it on top of the maze
        if self.current_path:
            self._draw_path_on_canvas(self.current_path)

    def _draw_rectangular_maze(self):
        """
        Draws a rectangular maze on the canvas.
        Iterates through cells, draws cell backgrounds (highlighting start/end),
        and then draws walls based on `cell_data['walls']`.
        """
        cs = self.cell_size  # Cell size
        wall_color = 'black'
        wall_width = max(1, cs // 15 if cs > 15 else 1) # Adaptive wall width

        for r_idx in range(self.maze.rows):
            for c_idx in range(self.maze.cols):
                cell_id = (r_idx, c_idx)
                if not self.maze._is_valid_cell_id(cell_id): continue # Skip if somehow invalid

                # Calculate top-left (x0,y0) and bottom-right (x1,y1) coordinates of the cell
                x0, y0 = c_idx * cs, r_idx * cs
                x1, y1 = (c_idx + 1) * cs, (r_idx + 1) * cs

                # Determine fill color for the cell (default, start, or end)
                fill_color = 'ivory'
                if cell_id == self.maze.start_node: fill_color = 'lightgreen'
                elif cell_id == self.maze.end_node: fill_color = 'salmon'
                # Draw the cell background (as a rectangle with no outline, walls will form the outline)
                self.canvas.create_rectangle(x0, y0, x1, y1, fill=fill_color, outline='')

                cell_data = self.maze.cells[cell_id]

                # Draw walls if they exist (wall_exists is True)
                # Top wall (North)
                if cell_data['walls'].get((r_idx - 1, c_idx), True):
                    self.canvas.create_line(x0, y0, x1, y0, fill=wall_color, width=wall_width)
```

```
                # Right wall (East)
                if cell_data['walls'].get((r_idx, c_idx + 1), True):
                    self.canvas.create_line(x1, y0, x1, y1, fill=wall_color, width=wall_width)
                # Bottom wall (South)
                if cell_data['walls'].get((r_idx + 1, c_idx), True):
                    self.canvas.create_line(x0, y1, x1, y1, fill=wall_color, width=wall_width)
                # Left wall (West)
                if cell_data['walls'].get((r_idx, c_idx - 1), True):
                    self.canvas.create_line(x0, y0, x0, y1, fill=wall_color, width=wall_width)

        # Draw an outer border for the entire maze if it has dimensions
        if self.maze.rows > 0 and self.maze.cols > 0:
            self.canvas.create_rectangle(0,0, self.maze.cols*cs, self.maze.rows*cs, outline='black', width=wall_width)


    def _draw_triangular_maze(self):
        """
        Draws a triangular maze on the canvas.
        Iterates through cells, draws cell backgrounds (as polygons, highlighting start/end),
        and then draws walls based on `cell_data['walls']` and cell geometry.
        """
        if not self.maze or not hasattr(self.maze, 'num_triangle_rows') or self.maze.num_triangle_rows == 0: return
        s = self.cell_size # Side length of triangle
        wall_color = 'black'
        wall_width = max(1, s // 20 if s > 20 else 1) # Adaptive wall width

        for cell_id, cell_data in self.maze.cells.items():
            # Ensure cell is valid and has vertex data (calculated in _update_canvas_size_and_coords)
            if not self.maze._is_valid_cell_id(cell_id) or 'vertices' not in cell_data: continue

            r, i = cell_id # Unpack cell row and index
            vertices = cell_data['vertices'] # Get pre-calculated vertices

            # Determine fill color
            fill_color = 'ivory'
            if cell_id == self.maze.start_node: fill_color = 'lightgreen'
            elif cell_id == self.maze.end_node: fill_color = 'salmon'

            # Draw the triangle cell background
            self.canvas.create_polygon(vertices, fill=fill_color, outline='')

```

```
            is_up = cell_data['is_up'] # Is the current triangle pointing up?
            v = cell_data['vertices'] # Alias for vertices for convenience

            # Define which edges correspond to which neighbors for wall drawing
            # Each tuple is ((vertex1, vertex2), neighbor_cell_id)
            edges_map = []
            if is_up: # For up-pointing triangles
                edges_map = [
                    ((v[0], v[1]), (r, i - 1)),  # Left edge, connects to left neighbor (r, i-1)
                    ((v[0], v[2]), (r, i + 1)),  # Right edge, connects to right neighbor (r, i+1)
                    ((v[1], v[2]), (r + 1, i + 1)), # Bottom edge, connects to neighbor below (r+1, i+1)
                ]
            else: # For down-pointing triangles
                edges_map = [
                    ((v[0], v[2]), (r, i - 1)),  # Left edge (relative to orientation), connects to (r, i-1)
                    ((v[1], v[2]), (r, i + 1)),  # Right edge (relative to orientation), connects to (r, i+1)
                    ((v[0], v[1]), (r - 1, i - 1)), # Top edge, connects to neighbor above (r-1, i-1)
                ]

            # Iterate through the defined edges and draw walls if they exist
            for (p1, p2), neighbor_id in edges_map:
                draw_this_wall = False
                # A wall should be drawn if:
                # 1. The neighbor_id is not a valid cell (i.e., it's an outer boundary).
                # 2. Or, the wall to this neighbor_id is marked as True (closed) in cell_data.
                if not self.maze._is_valid_cell_id(neighbor_id) or \
                   cell_data['walls'].get(neighbor_id, True): # Default to True if neighbor not in walls dict
                    draw_this_wall = True

                if draw_this_wall:
                    self.canvas.create_line(p1[0], p1[1], p2[0], p2[1], fill=wall_color, width=wall_width)

    def _draw_path_on_canvas(self, path_coords):
        """
        Draws the solved path on the canvas.

        Args:
            path_coords (list): A list of cell_ids representing the path.
        """
```

```
        if not path_coords or len(path_coords) < 1 or not self.maze or not self.maze.cells: return

        path_color = 'blue'
        path_width = max(2, self.cell_size // 8 if self.cell_size >= 8 else 1) # Adaptive path width

        points_to_draw = [] # List to store (x,y) screen coordinates for path segments
        for cell_id in path_coords:
            if self.maze._is_valid_cell_id(cell_id):
                # Get the center coordinates for drawing based on maze type and available data
                if 'display_coords' in self.maze.cells[cell_id]:
                    points_to_draw.append(self.maze.cells[cell_id]['display_coords'])
                elif 'center_coords' in self.maze.cells[cell_id]: # Used by triangular
                     points_to_draw.append(self.maze.cells[cell_id]['center_coords'])
                elif self.maze.type == "rectangular" and 'rect_coords' in self.maze.cells[cell_id]:
                    # Calculate center for rectangular cells if not pre-calculated
                    c, r_coord = self.maze.cells[cell_id]['rect_coords']
                    cs = self.cell_size
                    points_to_draw.append( (c * cs + cs / 2, r_coord * cs + cs / 2) )

        if not points_to_draw: return # No valid points to draw

        if len(points_to_draw) == 1: # If path is just one cell (start=end)
            x_center, y_center = points_to_draw[0]
            radius = self.cell_size / 4.0 # Draw a small circle
            self.canvas.create_oval(x_center - radius, y_center - radius,
                                    x_center + radius, y_center + radius,
                                    fill=path_color, outline='')
        elif len(points_to_draw) > 1: # If path has multiple cells
            # Draw lines between consecutive points (cell centers)
            for i in range(len(points_to_draw) - 1):
                x1_center, y1_center = points_to_draw[i]
                x2_center, y2_center = points_to_draw[i+1]

                is_last_segment = (i == len(points_to_draw) - 2) # Is this the last segment of the path?
                # Define arrow shape parameters for the last segment
                arrow_shape_val = self.cell_size / 4.0
                arrow_s1 = max(1.0, arrow_shape_val) # base
                arrow_s2 = max(1.0, arrow_shape_val * 4.0/3.0) # length
                arrow_s3 = max(1.0, arrow_shape_val / 2.0) # width
                arrow_shape_tuple = (arrow_s1, arrow_s2, arrow_s3)

                if is_last_segment: # Add an arrowhead to the last segment
                    self.canvas.create_line(x1_center, y1_center, x2_center, y2_center,
                                            fill=path_color, width=path_width, arrow=tk
```

```
.LAST,
                                            arrowshape=arrow_shape_tuple, capstyle=tk.ROUND)
                else: # For other segments, just draw a line
                    self.canvas.create_line(x1_center, y1_center, x2_center, y2_center,
                                            fill=path_color, width=path_width, capstyle=tk.ROUND)

    def clear_path_display_action(self):
        """
        Action for the "Clear Path" button.
        Removes the current path from display and redraws the maze without it.
        """
        if not self.maze: # If no maze exists, nothing to clear from
            self.status_label.config(text="No maze to clear path from.")
            return
        self.current_path = None # Set current path to None
        self.draw_maze() # Redraw the maze (which will not draw the path if current_path is None)
        self.status_label.config(text="Path cleared. Ready for new solve or generation.")

    def solve_maze_action(self, method):
        """
        Action for the "Solve (BFS)" and "Solve (DFS)" buttons.
        Calls the appropriate solving method on the `self.maze` object
        and updates the display with the found path or a "no path" message.

        Args:
            method (str): The solving method to use ('bfs' or 'dfs').
        """
        # Pre-checks before attempting to solve
        if not self.maze or not self.maze.cells:
            messagebox.showwarning("No Maze", "Please generate a maze first.")
            return
        if not self.maze.start_node or not self.maze.end_node:
            messagebox.showwarning("Invalid Maze", "Start or End node is not set or invalid for the current maze.")
            return
        if not self.maze._is_valid_cell_id(self.maze.start_node) or \
           not self.maze._is_valid_cell_id(self.maze.end_node):
            messagebox.showwarning("Invalid Maze", f"Start ({self.maze.start_node}) or End ({self.maze.end_node}) cell ID is not valid in the current maze cells.")
            return

        # Call the selected solving algorithm
        if method == 'bfs':
            self.current_path = self.maze.solve_bfs()
```

```
                algo_name = "BFS (Shortest Path)"
            elif method == 'dfs':
                self.current_path = self.maze.solve_dfs()
                algo_name = "DFS"
            else: # Should not happen with current UI setup
                return

            self.draw_maze() # Redraw the maze (will include the path if found)

            # Update status label with the result
            if self.current_path:
                self.status_label.config(text=f"Path found using {algo_name} with {len(self
    .current_path)} steps.")
            else:
                self.status_label.config(text=f"No path found from {self.maze.start_node}
    to {self.maze.end_node} using {algo_name}.")


if __name__ == '__main__':
    """
    Main entry point of the application.
    Creates the Tkinter root window and an instance of MazeApp.
    Starts the Tkinter event loop.
    """
    main_root = tk.Tk()  # Create the main Tkinter window
    # Instantiate the MazeApp, passing the root window and default parameters
    app = MazeApp(main_root, cell_size=25, default_rows=15, default_cols=20,
    default_tri_rows=8)
    main_root.mainloop()  # Start the Tkinter event loop to run the GUI
```

Listing 2.1: 迷宫生成与求解程序代码