Maze Solver Exercise

Introduction

In this exercise, you'll implement a maze-solving algorithm using object-oriented programming principles. The maze comprises walls, an entry point (A), and an exit point (B). Your objective is to navigate from the entry to the exit point. The final result can be run in the terminal as 'python maze.py maze1.txt'. If you wish to visualize the final path as an image (on top of terminal printing), install the required library ('pip install -r requirements.txt').

Background

Mazes can be solved using various search algorithms. Two primary search strategies are:

1. **Depth-First Search (DFS)**: This strategy explores as far as possible along each branch before backtracking. It uses a stack data structure. In this method, it dives deep into a path before exploring other alternatives.
2. **Breadth-First Search (BFS)**: This strategy explores all neighboring states before progressing to the neighbors of those states. It employs a queue data structure, ensuring that all options at the current depth level are explored before moving deeper.

Instructions

1. Node Class

- Each maze position will be represented by a node.
- Understand the properties of the node class:
    1. **state**: Represents the current position in the maze.
    2. **parent**: Refers to the node from which the current node was reached.
    3. **action**: Defines the action taken to transition to the current state from its parent.

2. Frontier Classes

- The frontier is a collection of nodes that await exploration.
- The **StackFrontier** class represents the DFS strategy as it uses a stack.
- The **QueueFrontier** class symbolizes the BFS strategy. It inherits from StackFrontier but overrides the removal method to act like a queue.

Your Tasks:

- Complete the methods in the StackFrontier and QueueFrontier classes:
  - `add`: Adds a node to the frontier.
  - `contains_state`: Checks if a given state exists within the frontier.
  - `empty`: Determines if the frontier is devoid of nodes.
  - `remove`: Extracts and returns the next node for exploration. (Remember, the way nodes are removed differentiates DFS from BFS!)

3. Maze Class

- The Maze class initializes the maze, provides utility functions, and contains the maze-solving algorithm.

Your Tasks:

- Complete the `neighbors` method to find possible directions (up, down, left, right) from a given state.
- Complete the `solve` method:
  - Initialize the frontier using the starting position.
  - Loop to explore nodes until you find the exit or run out of possibilities.
  - In each iteration, extract a node from the frontier, then explore its neighbors. Add unexplored neighbors to the frontier.
  - If the exit is located, backtrack using the `parent` property of nodes to find the taken path.

4. Tying DFS and BFS

- Understand how StackFrontier and QueueFrontier relate to DFS and BFS. The choice of data structure (stack vs. queue) dictates the order of node exploration, differentiating between DFS and BFS.

Challenge

- Once you've implemented the maze solver using DFS (using StackFrontier), modify it to use BFS (with QueueFrontier). Reflect on the changes needed to switch between these strategies. How does the output change? Which strategy requires the least number of steps for each maze? Is the optimal solution found?