Maze Solver Exercise

Introduction

In this exercise, you will be implementing a maze-solving algorithm. The maze consists of walls, an entry point labeled as A, and an exit point labeled as B. Your task is to navigate from the entry to the exit point, if possible. The final result can be run in the terminal as 'python maze.py maze1.txt'.

Background

There are many ways to solve a maze, but we'll focus on two common search strategies:

1. **Depth-First Search (DFS)**: This strategy explores as far as possible along each branch before backtracking. It uses a stack data structure. This means it will go deep into a path before exploring other options.
2. **Breadth-First Search (BFS)**: This strategy explores all the neighboring states before moving on to the neighbors of those states. It uses a queue data structure, ensuring it explores all options at the current depth level before moving deeper.

Instructions

1. **Node Structure**:
   - Each position in the maze will be represented by a node.
   - Implement the `create_node` function to create a node. A node should have the following properties:
     a. `state`: The current position in the maze.
     b. `parent`: The node from which this current node is reached.
     c. `action`: The action taken to reach the current state from its parent.
2. **Frontier**:
   - The frontier is a collection of nodes that are waiting to be explored.
   - For DFS, we'll use a stack as our frontier. For BFS, we'll use a queue.
   - Implement the following functions:
     - `create_stack_frontier`: Initializes the stack frontier.
     - `add_to_frontier`: Adds a node to the frontier.
     - `contains_state`: Checks if a given state is in the frontier.
     - `frontier_is_empty`: Checks if the frontier has no nodes left to explore.

- `remove_from_stack_frontier` and `remove_from_queue_frontier` Removes and returns the next node to be explored from the stack frontier.

3. **Solving the Maze**:
   - Implement the `solve_maze` function.
   - Initialize the frontier with the starting point of the maze.
   - Use a loop to explore nodes until you find the exit or exhaust all possibilities.
   - In each iteration of the loop, remove a node from the frontier, and add its neighbors to the frontier if they haven't been explored yet.
   - If you reach the exit, backtrack using the `parent` property of nodes to find the path taken.
   - Why not keep track of the explored set, so that you don't revisit previous states?
4. **Challenge**:
   - Once you've implemented the maze solver using the DFS strategy, try modifying it to use BFS. Think about what needs to change in the frontier 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?