Assignment 6: Grid World Navigation

Data Structures, CSCI-UA 102, Section 7
Instructor: Max Sklar

Successfully navigate through a grid from the upper-left corner to the lower-right corner while avoiding blocked grid cells.

Overview

Given a rectangular grid with a fixed number of rows and columns (some of which may be blocked), find the shortest path from the top-left position `(0, 0)` to the bottom-right position `(numRows-1, numCols-1)`. Express your answer as a sequence of moves (either UP, DOWN, LEFT, or RIGHT).

The path-finding algorithm must:

- Avoid blocked cells.
- Move only in cardinal directions (UP, DOWN, LEFT, or RIGHT).
- Find the shortest possible path to the goal.
- Return null if no valid path exists.

If a solution exists, it can be found using a breadth-first search (BFS).

Given Classes

Your task is to complete `PathFinder.java` by implementing the `findShortestPath()` method. You may find it helpful to create a few private helper methods as well. `findShortestPath()` will return a list of moves that can be made from position $(0, 0)$ in order to traverse the grid. Sometimes this isn't possible because the starting cell is completely blocked from the end goal, in which case `findShortestPath()` should return null.

The pathfinder will also have access to a `GridWorld` object, which contains the information you will need about the grid.

The class `GridIndex` represents a grid cell location. Its members are `row` and `col`.

`Move` is an enumerated type consisting of four values: UP, DOWN, LEFT, and RIGHT.

The following methods from `GridWorld` are needed to complete `PathFinder`:

- `getRowCount()` - returns the number of rows in the grid.
- `getColCount()` - returns the number of columns in the grid.

- `getStartingPoint()` - returns the GridIndex that is the starting point of the puzzle. This is really just `new GridIndex(0, 0)`
- `getEndingPoint()` - returns the GridIndex that is the ending point of the puzzle.
- `tryMove(index, move)` - attempts to move in a particular direction and returns the new `GridIndex`. This will return null if the move pushes the location off the board (for example, if `index` is in the top row and `Move.UP` is passed as the move).
- `tryUndoMove(index, move)` - same as `tryMove`, but moves in the opposite direction.
- `isStartingPoint(index)` - if the index is the starting point (same as checking row and col are both 0).
- `display()` - displays the grid using `AnsiColor`.
- `validateSolution(moves, print)` - validates a given solution, and prints out a visual representation of the path.

Puzzle Examples

Several puzzle files are given to you in the puzzles folder. Running them through `GridWorld` will display the puzzle and possibly solve it. You can run `GridWorld` to display the puzzles, even though it won't be solved until the assignment is completed. Each test will report "No solution found" until `PathFinder` is implemented correctly.

Open cells are displayed in cyan and blocked cells in red. When `PathFinder` finds a solution, that path is displayed in yellow.

```
javac GridWorld.java
cat puzzles/maze_like.txt | java GridWorld
cat puzzles/all_open.txt | java GridWorld
cat puzzles/navigable_obstacles.txt | java GridWorld
cat puzzles/dense_navigable.txt | java GridWorld
cat puzzles/no_path.txt | java GridWorld
```

Algorithm

In order to do a breadth-first search, `PathFinder` should keep track of its own grid of `Move` values.

```
private final Move[][] previousMoveData;
```

This `previousMoveData` matrix starts out null, and when filled in, represents the last move to reach any given location in the shortest path. If a cell is unreachable, its value in `previousMoveData` will always remain null. `previousMoveData` should be filled in until all the reachable locations have been reached, or the ending point has been reached (the bottom right of the grid). Here is a sketch of how to do this using BFS:

1. Create a queue containing `GridIndex` objects, and enqueue the starting point.
2. Loop until the queue is empty:
   (a) Get the next `GridIndex` from the queue (dequeue it).
   (b) Attempt to move in all four directions (DOWN, RIGHT, UP, and LEFT).
   (c) If the move results in a valid grid position (still on the grid), the cell is not blocked in `GridWorld`, and it hasn't been visited before (i.e., `previousMoveData` at that position is null), then fill in `previousMoveData` appropriately and enqueue that location.
   (d) If previousMoveData for the ending point is no longer null (after the four directions have been tried), then the puzzle can be solved, so break out of the loop
3. If previousMoveData for the ending point is null, then that location is not reachable in the puzzle, so return null.
4. Otherwise, follow the moves from the ending point to the starting point, building up a `SinglyLinkedList<Move>` as you go, stopping when you reach the starting point.
5. Return the linked list of moves.

Testing

There are two modes of testing. The first is to run `GridWorldTest` which will automatically test your code against the five sample worlds in the puzzles folder.

```
javac GridWorldTest.java
java GridWorldTest
```

You can also use `RandomPuzzleGenerator` to generate random puzzles and test your `PathFinder`. You can set the number of rows, columns, and density of blocks for the random puzzle generator.

```
javac RandomPuzzleGenerator.java
java RandomPuzzleGenerator 20 20 0.2 | java GridWorld
```

Your puzzle should be tested on enormous worlds to ensure that your solution scales. It will be tested on worlds with over 100,000 rows.

```
java RandomPuzzleGenerator 100000 20 0.2 | java GridWorld
```

Also try changing the block density. Lower density worlds are more likely to have solutions than higher density worlds.

Submission

The entire submission should be a zip file containing all relevant files and folders and should be uploaded to Brightspace by the due date. Submit a zip file named with your NetID, containing all relevant files and folders. This includes:

- Your Java code, which should include one or more Java files and may or may not be organized into packages.
- Your `readme.txt` file (described below).

README

Included in the `readme.txt` file should be the following:

NOTES: Include the difficulties and roadblocks encountered while working on this project, including notable bugs that needed to be overcome. Are there any quirks in your solution that we should be aware of? Was the outcome what you expected?

RESOURCES AND ACKNOWLEDGEMENTS: Disclose the sources you consulted for help on this project. Possible human sources include: NYU Staff (the instructor, TA, tutor), other students, and anyone who helped you with the project. Sources that you consulted, including code that you found online, in the book, and output from LLMs. If you used an LLM, briefly describe how you used it.

There will never be a grading penalty for a complete and accurate accounting of resources and acknowledgments.