BFS Wizards, maze

By Ivan Malishevskyi & Oleksii Savelich
All the code can be found here

Problem Statement

Given:

- A 2D maze, where `#` represents walls, `.` corridors, and `>` therefore the exit.
- Three starting points for the wizards.
- A file listing each wizard's speed (corridors per minute). "Minute" here is just a unit of time measurement, I suppose it's conventional unit, so it doesn't matter what to use.

Task:

- Use Breadth-First Search (BFS) *exactly once* to compute all necessary shortest-path distances.
- Determine which wizard reaches the exit first by comparing
  `arrivalTime = distance_to_exit / speed`

Algorithm Design

Instead of running BFS from each wizard (which would be three traversals), we invert the search:

1. **Run BFS once from the exit**
   - Seed a queue with the exit cell (`>`), set its distance to 0.
   - Expand in the four cardinal directions, assigning each reachable cell its minimum corridor-count to the exit.
   - Store results in a 2D map:
     `distanceMap[row, col] = shortest corridors to exit
2. Compute each wizard's arrival time

`arrivalTime = distanceMap[startRow, startCol] / speed`

3. **Select the winner**
   - The wizard with the smallest arrivalTime wins.

This guarantees only one full BFS while yielding every start position's distance.

Maze Representation and Movement

- **Data structure:** `char[,] mazeGrid`
  - `'#'` → wall (impassable)
  - `'.'` → open corridor
  - `'>'` → exit (record location, then treat as `'.'` because it can be passed)
- **Wizards**: parsed positions on the grid but considered walkable for BFS.
- **Allowed moves:** Up, Down, Left, Right (no diagonals because I'm lazy to implement that and the task doesn't explicitly asks for that).

Sample Output

```
Wizard #1: distance=8  speed=1.5  arrival=5.33 minutes
Wizard #2: distance=2  speed=2  arrival=1.00 minutes
Wizard #3: distance=8  speed=1.2  arrival=6.67 minutes
Winner: Wizard #2 (arrival: 1.00 minutes)
```

Code Structure

- `DataTypes/Point.cs`
  Defines `Point` struct with `int Row, Col`.
- `DataTypes/Wizard.cs`
  Defines `Wizard` struct with `Point Position` and `double Speed`.
- `Maze/MazeParser.cs`
  Reads the file into `char[,] mazeGrid`, locates exit and wizards.
- `Maze/DistanceMap.cs`
  Implements `BuildDistanceMap(grid, exit)` via one BFS.
- `Maze/MazeSolver.cs`
  Implements `DetermineWinner(wizards, distanceMap)` by computing arrival times.
- `Program.cs`
  Parsing, distance-map construction, all the methods invocation etc.

Complexity Analysis

Time Complexity

- **BFS** from exit: O(R × C) for an R×C grid.
- **Arrival lookups**: O(W), with W = 3 wizards, obviously.
- **Total:** O(R × C).

Space Complexity

- O(R × C) for the maze grid and distance map.
- O(W) for wizard data.

Potential Limitations

- Wizards can not omit walls, once they stuck and there is no way for escape, they shall forever wander across maze...
- No diagonal or teleport moves allowed. (even though they wizards I won't allow any cheaters here!)
- Single-exit assumption; multiple exits would need small adjustments (and I don't think it will be possible to use BFS only once).

Conclusion

A reverse BFS from the exit satisfies the single-BFS constraint and efficiently computes all wizard distances. Converting corridor counts to minutes is a clear prediction of the fastest wizard. This solution is optimal, robust, and scales linearly with maze size.

Tools used: >

- Obsidian *, to generate this fancy report and export it into pdf*
- C# & .NET *, the language of choice and environment to implement algorithms*

- Grokking Algorithms exercises code repository *, as a reference material*