UNIVERSITY OF WESTMINSTER

INFORMATICS INSTITUTE OF TECHNOLOGY

Informatics Institute of Technology

Algorithms: Theory, Design and Implementation

5SENG003C

Coursework Report

- Iynkaran Pavanantham
- UOW Number – w1990839
- IIT Number – 20222388
- Group Number – SE ‘M’

Table of Contents

1. Game Puzzle Brief (Problem Introduction) 3

2. Choice of Data Structure and Algorithm..... 4

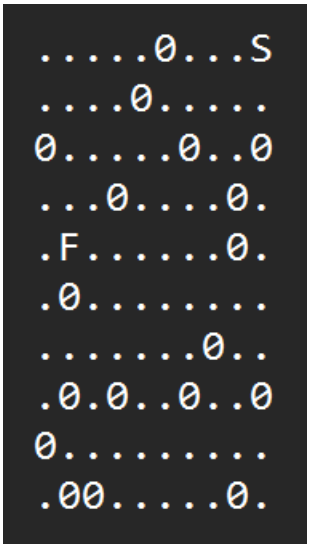
3. Choice Justification 5

4. Performance Analysis..... 5

5. Time Complexity Analysis 6

6. Testing..... 7

1. Game Puzzle Brief (Problem Introduction)



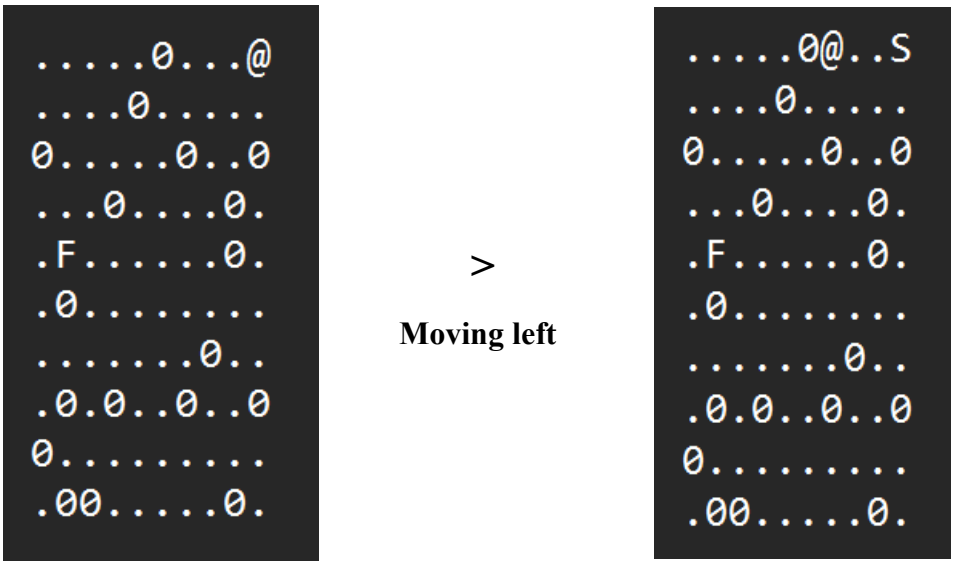
The solution is supposed to use path finding to solve a type of puzzle that occurs in many video games.

The player starts at the location labelled “S” (10,1) and wants to reach the finish, labelled “F” (2,5). Each turn they choose one of the four cardinal (up, down, left, right) directions to move. However, except for S and F the floor is covered in frictionless ice, so they will keep sliding in the chosen direction until they hit the wall surrounding the area, or one of the rocks (labelled “0”).

(Where the squares are numbered left to right, top to bottom).

For example, starting in the map given above :

The player (“@”) moving left would end up here,



So, we must implement an algorithm which finds the shortest path (between S and F) from the start to the finish in any given map.

2. Choice of Data Structure and Algorithm

Data Structures:

- **Coordinate Class**: This class represents a coordinate on the game map. It contains the following attributes:
    1. **row** and **column**: The position of the coordinate. (Y, X)
    2. **distanceFromStart**: The distance from the start position to this coordinate.
    3. **path**: A string representing the path taken from the start to this coordinate, including cardinal directions.
    4. **queue**: A queue storing the path coordinates.
- **Queue**: Have used Java's **LinkedList** as a queue to store coordinates. This is used for Breadth-First Search (BFS), allowing you to explore all possible paths from the starting point.

Algorithm:

This algorithm employs a modified BFS approach to find the shortest path from the start to the finish in the ice slider puzzle. Here's an overview of how it works:

Initialization:

- Initialize a queue (pq) with the starting coordinate.
- Initialize a boolean array (visited) to keep track of visited coordinates.

Exploration:

- While the queue is not empty, dequeue a coordinate.
- Check if the dequeued coordinate is the finish point. If yes, return to the path.
- For each direction (up, down, left, right), attempt to slide until a wall or obstacle is encountered.
- Roll back one step if necessary to find the correct position.
- Enqueue the new coordinate if it hasn't been visited and continue exploring.

Termination:

- If no path is found, it will return "No path found!".

3. Choice Justification

- **Breadth-First Search (BFS)**: BFS is suitable for finding the shortest path in a graph when all edges have the same weight. In this case, each movement from one cell to another has a uniform cost (distance of 1). BFS guarantees the shortest path to the goal.
- **Queue for BFS**: Using a queue allows you to explore all possible moves from the current position in a systematic order (level by level), ensuring that the shortest path is found first.
- **Coordinate Class**: This class encapsulates all necessary information about a coordinate, making it convenient to manage and pass around during the search process. Storing the path in each coordinate simplifies tracking the solution path once the goal is reached.

4. Performance Analysis

- **Time Complexity:** The time complexity of the algorithm is primarily determined by the number of cells in the grid and the number of edges in the graph formed by possible moves. It scales linearly with the size of the input grid.
- **Space Complexity:** The space complexity of the algorithm also scales linearly with the number of cells in the grid. It requires additional space for storing coordinates and the queue, but the overall space usage is reasonable and proportional to the input size.

5. Time Complexity Analysis

- **Exploration Step (BFS)**:
    1. In the worst case, every cell of the grid needs to be explored.
    2. At each cell, you potentially explore four adjacent cells (up, down, left, right).
    3. Therefore, the time complexity of the exploration step is O(V + E), where V is the number of vertices (cells), and E is the number of edges (possible moves).
    4. In a worst-case scenario, V is the number of cells in the grid, and E is at most 4 times the number of cells.
- **Enqueue and Dequeue Operations**:
    1. Enqueue and dequeue operations on the queue occur for each cell visited.
    2. Each operation on the queue takes O(1) time complexity.
    3. Therefore, the overall time complexity for queue operations is O(V).
- **Total Time Complexity**:
    1. Combining the above, the total time complexity of the algorithm is O(V + E) + O(V), which simplifies to O(V + E). (**We can omit O(V) when <<<< O(V + E)**)

Conclusion:

- The time complexity of the algorithm scales linearly with the number of cells in the grid.
- It efficiently explores all possible paths using BFS, ensuring that the shortest path is found in a reasonable time frame.
- The algorithm's performance is primarily determined by the size of the input grid, making it suitable for grids of moderate size.

6. Testing

1. Text File (maze_10_4.txt)

Output

```
finding the shortest path

Total distance: 44
 START:  (9, 1)
RIGHT (10, 1)
DOWN (10, 10)
LEFT (6, 10)
UP (6, 9)
LEFT (1, 9)
DOWN (1, 10)
RIGHT (4, 10)
UP (4, 7)
LEFT (3, 7)
UP (3, 5)
LEFT (2, 5)
UP (2, 1)
RIGHT (7, 1)
DOWN (7, 3)
LEFT (6, 3)
DOWN (6, 4)


Time elapsed: 20 milliseconds
Ice blocks Height 10
```

2. Text File (maze_20_2.txt)

```
............0.......
0...0.....0.........
..........0000......
.....0......0.....0.
0.......0..0..0..F0.
..0..0.00......0.0..
0....0......0.......
......0.............
....0.....00........
0..........0....0...
....................
...0.........0....0.
....0...0.....00..0.
.000.0..0..0........
0.....00...0.....0.0
..0.................
..............0.....
....................
..00..0.......0.....
0...0.....0...S0..0.
```

Output

```
 START:  (15, 20)
LEFT (12, 20)
UP (12, 16)
LEFT (4, 16)
UP (4, 15)
RIGHT (6, 15)
DOWN (6, 20)
RIGHT (10, 20)
UP (10, 1)
RIGHT (12, 1)
DOWN (12, 2)
RIGHT (20, 2)
DOWN (20, 14)
LEFT (13, 14)
UP (13, 8)
LEFT (8, 8)
UP (8, 7)
RIGHT (12, 7)
UP (12, 6)
RIGHT (15, 6)
DOWN (15, 12)
RIGHT (18, 12)
UP (18, 7)
LEFT (14, 7)
UP (14, 4)
RIGHT (18, 4)
DOWN (18, 5)
```

Like these, for all example text files it will print the shortest path between S and F.

***