INFORMATICS INSTITUTE OF TECHNOLOGY

In collaboration with

UNIVERSITY OF WESTMINSTER

ALGORITHM COURSEWORK REPORT

Module: 5SENG003C.2

Algorithms: Theory, Design and Implementation

Student Name : W.H.Sachini Wewalwala

Student Id : 20212030

UOW Id : 18714849

Tutorial Group : Group 2

a) A short explanation of your choice of data structure and algorithm

Data Structure

1.ArrayList<String>:The ArrayList<String> is used to store each line of the map file as a string.This is ideal for storing map file lines due to dynamic resizing, enabling automatic adjustment for new lines. Its methods, like get(index), simplify accessing specific lines during parsing, making it efficient for processing maps without manual resizing or complex access handling.

2.2D char array (map):The 2D char array (char[][] map) is created based on the height and width of the map. It mirrors the map's grid structure, with each character representing a grid cell. This array simplifies operations like checking cell contents and navigation during gameplay or parsing algorithms, as cell positions in the array directly correspond to positions in the grid.

3.PriorityQueue<Integer>: Used to store nodes and prioritize them based on their estimated total distances from the start node in the A* algorithm. This data structure ensures that nodes with lower estimated distances are processed first, leading to an efficient pathfinding process.

4.HashMap<Integer, LinkedList<Integer>> adjacencyList: Used in the Puzzle class to represent the graph structure of the grid. This adjacency list allows efficient storage and retrieval of neighbouring nodes for each node in the grid, supporting pathfinding operations.

Algorithm

The A* algorithm, applied in the AStar class's findShortestPath method, efficiently navigates weighted graphs by prioritising nodes in a priority queue based on estimated total distances. It iteratively explores neighbours, updating distances and paths while considering a heuristic function that guides the search toward the goal. The algorithm constructs and returns the shortest path, ideal for navigating complex environments.

b) A run of your algorithm on a small benchmark example. This should include the supporting information as described in Task 4.

Benchmark example - puzzle_10.txt

Input

puzzle_10.txt

```
.0.0...0..
0...0.0.0.
.....0...0
0.........
.0..0....0
...0.0..0.
....0..0..
.S......0.
...0.....0
..F.0...0.
.....0.0..
```

Output

```
*****************************************
               Sliding Puzzles
*****************************************

Enter the file name:
puzzle_10.txt

*****************************************
                 Path
*****************************************

1. Start at (2, 8)
2. Move up to (2,6)
3. Move right to (3,6)
4. Move down to (3,10)
5. Done!

Process finished with exit code 0
```

c) A performance analysis of your algorithmic design and implementation

Theoretical considerations

The provided A* algorithm implementation in Java aims to find the shortest path on an icy grid from a start point to a finish point, considering obstacles and sliding behaviour. Let's analyse its performance in terms of time and space complexities.

Time Complexity Analysis:

- Heuristic Calculation (heuristic method): O(1)
- Sliding on Ice (slideOnIce method): O(max(W, H)) where W is the width and H is the height of the grid.
- Priority Queue Operations: O(log C) per operation, where C is the number of nodes.
- Overall the main loop runs until the priority queue is empty, which can be O(C log C) in the worst case, assuming all nodes are reachable.

Space Complexity Analysis:

- Distances and prev Arrays: O(C) where C is the number of nodes.
- Priority Queue: O(C) in the worst case.
- Overall the space complexity is O(C)

The algorithm's time complexity primarily depends on the number of nodes and the grid's dimensions. The space complexity is mainly determined by the number of nodes.
In practical terms, the algorithm performs efficiently for moderately sized grids and a reasonable number of nodes. However, for very large grids or a high number of nodes, the time and space complexities can become significant, impacting performance.

Order-of-growth classification (Big-O notation)

The algorithm's time complexity can be classified as O(C log C) for moderately sized grids and a reasonable number of nodes, where C is the number of nodes. This classification reflects the logarithmic growth of operations on the priority queue. However, for very large grids or a high number of nodes, the time complexity may approach O(C^2) , impacting performance significantly.Retaining the space complexity at linear O(C) shows effective memory use.