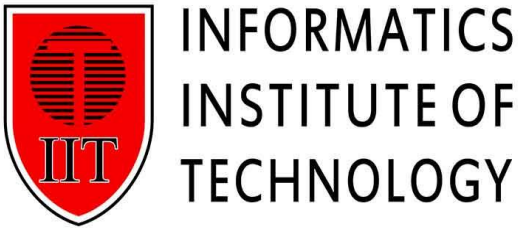


INFORMATICS INSTITUTE OF TECHNOLOGY

Module	: 5SENG003C.2 Algorithms: Theory, Design and Implementation
Module Leader	: Klaus Draeger
Student ID	: 20200696/ W1833520
Student Name:	: Kaliyugavarathan Sean Jesuharun
Tutorial Group	: Group J

a. Algorithm Approach

To find the shortest path in “Sliding Puzzles” I choose the “Breadth-First Search” algorithm because I must move in every possible direction (LEFT, RIGHT, UP, DOWN only if possible) from a specific node (Point) to look for the “F” character for the very first time from the puzzles. The Breadth-First Search algorithm starts at a specific node (Point) and explores the neighbor nodes (Points), before moving to the next level neighbor.

I am using a 2D Array called “slidingPuzzle2DArray” of type integer in order to represent the “sliding Puzzles”. In that 2D array “.” & S (Movable points, Starting point) are represented by integer “0“ and the 0 (Rocks) are represented by integer “1” and the F (Finishing point) represented by the integer “2”.

I am using another 2D Array called “trackPath2DArray” of type “Point” to avoid visiting the same node (Point) multiple times and also in order to track the path order if the algorithm finds the “F” character from the puzzle for the very first time. Here the elements of the 2D Array will have the Points where the Points will hold the reference for the location [coordinates (x & y)] for the previous node (Point) and the other elements remain null which means those are the unvisited nodes (Points).

I am using a queue Data Structure in a linked list called “queue” in order to track which node (Point) to visit next. Upon reaching the new node (Point) the algorithm adds it to the queue to visit it later.

And finally using another linked list called “shortPathOrderPoints” in order to contain all the Points for the shortest path from Start to finish [“S” to “F”].

b. A run of the Algorithm (How the Algorithm works)

(input => test.txt)

```
S..0.
....0
.00..
.0..0
..F0.
```

This is the format of the input files which contains the sliding puzzles. Then the 2D array called “slidingPuzzle2DArray” will be created. In that 2D array the elements will be populated using this format [“.” & S =>0, “0” =>1, “F” => 2]. While reading the text file the start point and the finish point coordinates [x & y] will be marked.

I am using a “queue” Data Structure in order to track which node (Point) to visit next. Upon reaching the new node (Point) the algorithm adds it to the queue to visit it later.

First the startPoint[1,1] will be added to the queue and later the first element in the queue will be retrieve to the currPosition variable and it will be removed from the queue once it retrieved. So now the currPosition is [1,1]. Then from this currPosition[1,1] we should slide through all the possible direction until hit by the wall, hit by the rock[1] or reaches the “2” point.

First it will check whether we can move in the left direction from the currPosition[1,1], in this case can’t so it will return null. Secondly it will check whether we can move in the right direction from currPosition[1,1] In this case yes can, so now we should check whether the next point is “2” or not if it’s “2” we reached the destination if not it will move one point and checks the other until it finds a rock[1], wall or the “2”. Here once it reached [3,1] it will stopped because the point next to that [4,1]

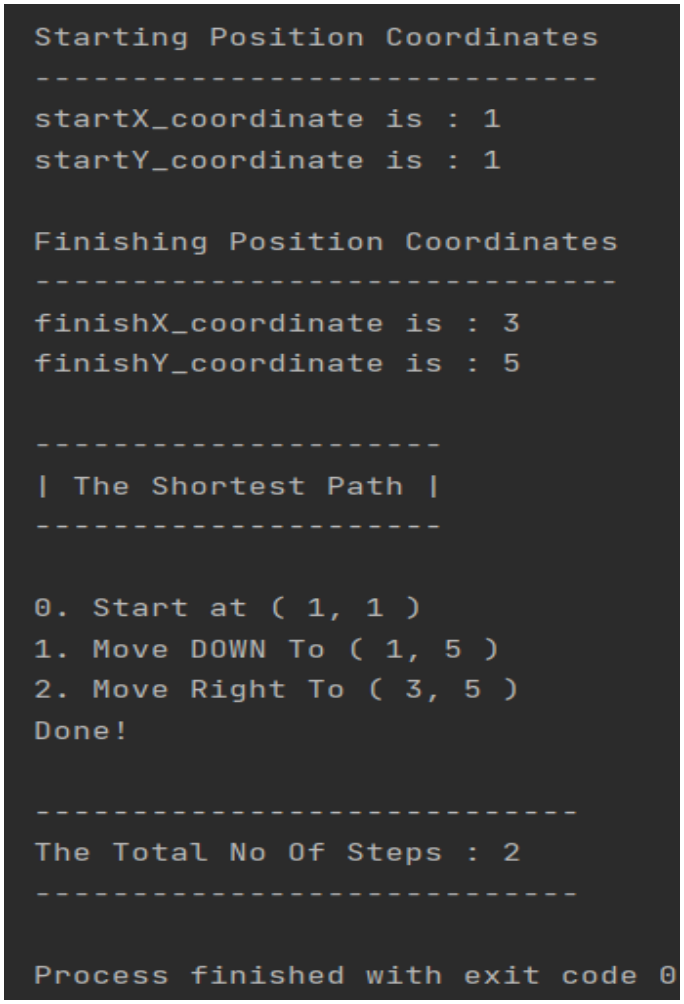
is a rock and it will return the point[3,1] and now this point[3,1] will be added to the queue. Thirdly it will check whether it can move in the up direction, in this case can't so it will return null. And finally it will check whether we can move in the down side direction in this case yes can so will move and reached the wall and return the point[1,5] and this point[1,5] will be added to the queue.

After going through all the possible directions now we will take the next point from the queue which is[3,1] and now the currPosition become [3,1]. Now like in the previous run we should go to all the possible direction that it can move. But in this case, we can move left and reach the same starting position[1,1] again as well. In order to avoid visiting the same positions again we will maintain another 2D array called "trackPath2DArray" to check whether a specific point is visited or not if its visited then it will return null if not if not it will return the position. Here from currPosition[3,1] it can't move left, right, up it can only move in the down side and it will return [3,2] and this point will be added to the queue.

Then the next point[1,5] will be retrieved from the queue and the currPosition now becomes [1,5]. And now from the currPosition[1,5] can't go up because that point is already visited can't move left or down side either only can move in right side. It will move and as soon as it finds the "2" in the position [3,5] and it will return that point and after returning that it will check whether the Finishing coordinates matches the returning point[3,5] if it matched then "BINGO" we found the shortest path to reach the "F" from "S".

And another important thing is "trackPath2DArray" will hold the reference for the previous point every times it finds a possible point to move. So, with that we can retrieve the shortest path order in reverse which is "F" to "S". So, with the help of "shortPathOrderPoints" linked list we can retrieve the order in the correct format of "S" to "F".

(output)



c. Performance analysis of the Algorithm design and implementation

Puzzle Name (.txt)	Time(ms)
maze10_1	2
maze15_2	2
maze20_3	2
maze25_4	2
maze30_5	2
puzzle_40	3
puzzle_80	4
puzzle_160	5
puzzle_320	9
puzzle_640	22
puzzle_1280	55
puzzle_2560	95