Maze Solver Report

Prepared by :
Sarah Eldafrawy (29)

Data Structures

```
/** n & m maze array dimensions.*/
private int n, m;
/** k & l coordinates of start point.*/
private int k, l;
/** locationsArray is the path steps stored in a list.*/
private ArrayList<int[][]> locationsArray;
/** visited array to mark visited cells.*/
private boolean[][] visitedArray;
/** maze array is where the maze cells are stored.*/
private String[] mazeArray;
/** parents array is where the maze cells'parents are
stored.*/
private Point[][] parents;
/** foundGoal flag to stop the recursion when Exit is
found.*/
private boolean foundGoal;
/** foundS flag to check if the maze has no start.*/
private boolean foundS;
/** foundE flag to check if the maze has no Exit.*/
private boolean foundE;
/** used in the BFS to set the list of cells that must be
visited next by the BFS algorithm.*/
private QueueList queue;
/** used in the DFS to set the list of cells that must be
visited next by the DFS algorithm.*/
private Stack stack;
```

Algorithms

The same is used in the same two functions solveDFS/solveBFS except the search part. It starts by calling the function **`readFromFile()`** that take the file as a parameter and set the variables in the constructor.

`readFromFile()` the function read the maze from the file using a loop to read each line. While looping if the Exit or the Start wasn't found a flag is set in **`solveDFS/BFS()`** to throw an exception.

`stringArrayToIntArray()` this function is used in **`readFromFile()`** to read the dimensions and return them as int array to be set in the constructor later.

After wards the parent cell 'Start' is set to (-1,-1) in the parents array of points, pushed in the stack/queue, then the DFS/BFS is called.

After the DFS/BFS functions we add the start point in the locations list that is converted to 2D int array by looping starting by the end of the list.

DFS is the same as the BFS algorithm except the use of the data structure used to store the cells to be called next.

DFS/BFS path finder starts with one cell pushed in the stack/queue then entering a loop that stops when the stack/queue is empty or the exit is found. In the loop, we skip if the cell is '#' and pushed/enqueued if it is '.', when the exit is found, the loops are broken and then `findingPath()` starts.

`findingPath()`, its use is to find the path starting from the exit back to the start point using the parents array that stores each cell's parent.

Extra Work

There is a DFS recursion implementation that isn't used in solveDFS or called in any function still it works well.

Sample Runs

```
5 5
###.S
..E#.
.##..
.....
..###
```

```
DFS Path: [0, 4],
[1, 4], [2, 4], [3, 4],
[3, 3], [3, 2], [3, 1],
[3, 0], [2, 0], [1, 0],
[1, 1], [1, 2]
```

```
5 5
###.S
...#.
.##..
.....
E.###
```

```
DFS Path: [0, 4],
[1, 4], [2, 4], [3, 4],
[3, 3], [3, 2], [3, 1],
[4, 1], [4, 0]
```

```
5 5
###.S
.....
.##..
..E..
..###
```

```
BFS Path: [0, 4],
[1, 4], [2, 4], [3, 4],
[3, 3], [3, 2]
```

```
5 5
S##.E
.....
.##..
.....
..###
```

```
BFS Path: [0, 0],
[1, 0], [1, 1], [1, 2],
[1, 3], [1, 4], [0, 4]
```

Comparison between both

The DFS algorithm works by visiting the children of the node before visiting its adjacent cells unlike the BFS algorithm that visit all adjacent cells before visiting children.

BFS finds the shortest path unlike DFS; however DFS uses less memory and is more practical if we are going to search the entire tree.