Programming Assignment 7 (Assignment 6 Cont)
Maze Solver

1 Overview

Stacks and Queues are two basic abstract data types that are useful in a wide variety of applications. Despite their conceptual simplicity, stacks and queues are powerful tools for storing and accessing data. In particular, these data structures are at the heart of two of the most important algorithms in computer science: depth first search (DFS) and breadth first search (BFS). Priority Queues are slightly more complicated conceptually, but ubiquitous in areas ranging from operating systems to scheduling surgeries at hospitals. They are also used to implement the heuristic-based best-first search.

1.1 Depth First Search

A DFS of a maze involves following some path through the maze as far as we can go, until a dead end or previously visited location is met. When this occurs, the search backtracks to its most recent choice and tries a different path instead.

1.2 Breadth First Search

BFS proceeds differently: it visits the locations in order of their distance from the starting point of the search. First it visits all locations one step away, then it visits all locations that are two steps away, and so on, until the exit is found. Because of this, BFS has the nice property that it will naturally discover the shortest route through a maze.

1.3 Best First Search

"Best"-First search uses a heuristic function, which provides an approximate judgment on the quality of a node. Given this function, it explores the most promising leads in the maze first, leaving poor prospects for later.

1.4 How does it work

The algorithms for DFS, BFS, and BestFS all work very similarly. Each one is based on a different one of the three data structures we discussed above: a stack, a queue, and a priority queue. If we "line up" the operations of these three data structures-push, enqueue, and insert together and pop, dequeue, and deleteMax (or Min) together-and call them abstract "add" and "remove" operations on a "todo list", we can describe all three algorithms at once.

All three algorithms begin by adding the entrance location onto the todo list. Then, until the todo list is empty or the exit is found, they remove a location from the todo list. If that

location has not already been visited (i.e., previously removed from the todo list), they find the location's children and add each child location onto the todo list.

2 Algorithms Implementation

```
Begin
S: empty Data Structure of cells
T = (the start tile 'S')
S.add(T)
Mark T as visited
While S is not empty
    T = S.remove()
    Mark T as visited
    if T is target cell
        Mark that goal is reached
        break
    for each valid unvisited neighbour N of T
        S.add(N)
End While
```

The above implementation is valid for both DFS and BFS. The difference is based on the order of visiting cells as the remove() function will change when you're using a stack or a queue. To retrieve the path you used to reach the target cell, you shall add more variable to each cell, which means who is the parent cell who made that one reachable. When you reach the target cell, you can find backward who is it's parent and similarly until reaching the starting cell.

3 Application

3.1 Maps

Your application should read the description of a map from file. The file will be structured as follows:

- In the first line, you will be given the size of the map which is represented as a $2D$ grid of size $N * M$.
- In the following N lines, each one will contain M characters representing the $i - th$ row in the map.
- Each character can be 'S' denoting the start cell, 'E' denoting the target end cell, '.' empty cell where you can move freely, '#' denoting a wall cell which means that you can't move over this cell.

An example can look like :

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

BFS: {0, 4}, {1, 4}, {2, 4}, {3, 4}, {3, 3}, {3, 2}, {3, 1}, {3, 0}
DFS: {0, 4}, {1, 4}, {2, 4}, {3, 4}, {3, 3}, {3, 2}, {3, 1}, {4, 1}, {4, 0}, {3, 0}

3.2 Path Finding

You should be able to find the path from the start cell to the goal cell using the path finding algorithms. The user can specify how he would like to solve the maze using DFS or BFS algorithm and you should output the path he should follows.

Organize your code under package with name

`eg.edu.alexu.csd.datastructure.maze.cs<xx>_cs<yy>_cs<zz>`
`where xx, yy and zz are your and your partners two digits class number.`

and you need to implement the following interface. If there is any errors in the file, or in the passed parameter, then you should throw an exception. Coordinates indexes are zero based.

```
package eg.edu.alexu.csd.datastructure.maze;
public interface IMazeSolver {

    /**
     * Read the maze file, and solve it using Breadth First Search
     * @param maze maze file
     * @return the coordinates of the found path from point 'S'
     *         to point 'E' inclusive, or null if no path found.
     */
    public int[][] solveBFS(java.io.File maze);

    /**
     * Read the maze file, and solve it using Depth First Search
     * @param maze maze file
     * @return the coordinates of the found path from point 'S'
     *         to point 'E' inclusive, or null if no path found.
     */
    public int[][] solveDFS(java.io.File maze);
}
```

4 Deliverables

- This assignment will be delivered along side the previous assignment.
- You should work in the same team of the previous assignment.
- You are only required to implement DFS and BFS algorithms. However, try to implement your own priority queue and implement the Best-FS. Check this link.
- You should use your own data structures which were implemented in the previous assignments. Don't use any built-in data structure.
- By now, you should organize your code well in classes. Use functions and make each function related to exactly one and only one responsibility.
- Try out your code using http://onlinetester.tk/
- Late submission is accepted for only one week.
- Delivering a copy will be severely penalized for both parties, so delivering nothing is so much better than delivering a copy.

Good Luck :)