Problem 3: Maze Solver [65 points]

Given a two-dimensional maze with a starting position and a goal position, your task is to write a Java program called `FindPath.java` that assists a robot in solving the maze by finding a path from the start node to the goal node (as described below) if one exists. The program must accept command line arguments in the following format:

```
$ java FindPath maze search-method
```

The first argument, `maze`, is the path to a text file containing the input maze as described below, and the second argument, `search-method`, can be either "`bfs`" or "`astar`" indicating whether the search method to be used is breadth-first search (BFS) or A* search, respectively.

The Maze

A maze will be given in a text file as a matrix in which the start position is indicated by "`S`", the goal position is indicated by "`G`", walls are indicated by "`%`", and empty positions where the robot can move are indicated by " ". The outer border of the maze, i.e., the entire first row, last row, first column and last column will *always* contain "`%`" characters. A robot is allowed to move only horizontally or vertically, not diagonally.

The Algorithms

For both BFS and A* search, explore the surrounding positions in the following order: move-Left (L), move-Down (D), move-Right (R), and move-Up (U). In BFS, add the successors in that order to the queue that implements the *Frontier* set for this search method. In this way, moves will be visited in the same order as insertion, i.e., L, D, R, U. Assume all moves have cost 1. Repeated state checking should be done by maintaining both *Frontier* and *Explored* sets. If a newly generated node, *n*, does *not* have the same state as any node already in *Frontier* or *Explored*, then add *n* to *Frontier*; otherwise, throw away node *n*.

For A* search, use the heuristic function, *h*, defined as the Euclidean distance from the current position to the goal position. That is, if the current (row#, column#) position is (*u*, *v*) and the goal position is (*p*, *q*), the Euclidean distance is $\sqrt{(u-p)^2+(v-q)^2}$. Add moves in the order L, D, R, U to the priority queue that implements the *Frontier* set for A* search. Assume all moves have cost 1. For A* search, repeated state checking should be done by maintaining both *Frontier* and *Explored* sets as described in the Graph-Search algorithm in Figure 3.14 in the textbook. That is,

- If a newly generated node, *n*, does *not* have the same state as any node already in *Frontier* or *Explored*, then add *n* to *Frontier*.
- If a newly generated node, *n*, has the *same* state as another node, *m*, that is already in *Frontier*, you must compare the *g* values of *n* and *m*:
  - If $g(n) \geq g(m)$, then throw node *n* away (i.e., do *not* put it on *Frontier*).
  - If $g(n) < g(m)$, then remove *m* from *Frontier* and insert *n* in *Frontier*.
- If new node, *n*, has the *same* state as previous node, *m*, that is in *Explored*, then, because our heuristic function, *h*, is consistent (aka monotonic), we know that the optimal path to the state is guaranteed to have already been found; therefore, node *n* can be thrown away. So, in the provided code, *Explored* is implemented as a Boolean array indicating whether or not each square has been expanded or not, and the *g* values for expanded nodes are not stored.

Output

After a solution is found, print out on separate lines:

1. the maze with a "." in each square that is part of the solution path
2. the length of the solution path
3. the number of nodes expanded (i.e., the number of nodes removed from *Frontier*, including the goal node)
4. the maximum depth searched
5. the maximum size of *Frontier* at any point during the search.

If the goal position is *not* reachable from the start position, the standard output should contain the line "`No Solution`" and nothing else.

Code

You must use the code skeleton provided. You are to complete the code by implementing `search()` methods in the `AStarSearcher` and `BreadthFirstSearcher` classes and the `getSuccessors()` method of the `State` class. **You are permitted to add or modify the classes and methods, but we require you to keep the IO class as is for automatic grading.** The `FindPath` class contains the main function.

Compile and run your code using an IDE such as Eclipse. To use Eclipse, first create a new, empty Java project and then do `File` → `Import` → `File System` to import all of the supplied java files into your project.

You can also compile and run with the following commands in a terminal window:

```
javac *.java
java FindPath input.txt bfs
```

Testing

Test both of your search algorithms on the sample test input file: `input.txt` and compare your results with the two output files: `output_astar.txt` and `output_bfs.txt`. Make sure the results are correct on CSL machines.

Deliverables

Put *all* `.java` files needed to run your program, including ones you wrote, modified or were given and are unchanged, into a folder called `<wiscNetID>-HW1-P3` Compress this folder to create `<wiscNetID>-HW1-P3.zip` and upload it to Canvas. For example, for someone with UW NetID crdyer@wisc.edu the file name must be: `crdyer-HW1-P3.zip`