CS146 Data Structures and Algorithms

Spring 2016, Instructor: K. Potika SJSU CS Department

Programming Project 3

*Important (NEW): Do in groups of 2-3 students (submit in canvas the groups **by 5/3/16**), each group has to turn in one project.*

Maze or Labyrinth Generator and Solver

(see greek mythology: Ariadne's thread)

The goal of this project is to write a program that will automatically generate and solve mazes. Each time you run the program, it will generate and print a new random maze and the solution. You will use depth-first search (DFS) and breadth-first search (BFS). Submit your code, and also JUnit test cases for it.

Generating a Maze:

To generate a maze, first start with a grid of rooms with walls between them. The grid contains r rows and r columns for a total of r*r rooms. For example, Figure 1 is a 4x4 grid of 16 rooms. The missing walls at the top left and bottom right of the maze border indicate the starting and finishing rooms of the maze.

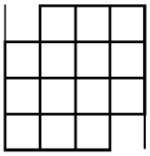


Figure 1

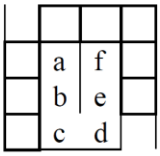


Figure 2

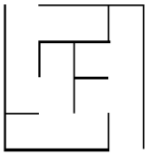


Figure 3

Our objective here is to create a *perfect* maze (see Figure 3), the simplest type of maze for a computer to generate and solve. A perfect maze is defined as a maze which has one and only one path from any point in the maze to any other point. This means that the maze has no inaccessible sections, no circular paths, no open areas.

Now, begin removing interior walls to connect adjacent rooms. The difficultly in generating a perfect maze is in choosing which walls to remove. Walls should be removed to achieve the following maze characteristics:

1. Randomized: To generate unpredictable different mazes, walls should be selected randomly as candidates for removal.
2. Single solution: There should be only one path between the starting room and the finishing room (for simplicity always use as starting and finishing rooms the ones in Figure 1, i.e. , starting the upper left room and finishing the lower right room) . Unnecessarily removing too many walls will make the maze too easy to solve. Therefore, a wall should not be removed if the two rooms on either side of the wall are already connected by some other path. For example, in Figure 2, the wall between *a* and *f* should not be removed because walls have previously been removed that create *a* path between *a* and *f* through *b, c, d, e*.

3. Fully connected: Enough walls must be removed so that every room (therefore also the finishing room) is reachable from the starting room. There must be no rooms or areas that are completely blocked off from the rest of the maze.

We now give a simple maze generation algorithm and that uses Depth First Search. Here's the DFS algorithm written as pseudocode:

```
create a CellStack (LIFO) to hold a list of cell locations
set TotalCells= number of cells in grid
choose the starting cell and call it CurrentCell
set VisitedCells = 1

while VisitedCells < TotalCells
    find all neighbors of CurrentCell with all walls intact
    if one or more found choose one at random
            knock down the wall between it and CurrentCell
            push CurrentCell location on the CellStack
            make the new cell CurrentCell
            add 1 to VisitedCells
    else
            pop the most recent cell entry off the CellStack
            make it CurrentCell
```

Note that we can eliminate recursion by the use of a stack (this DFS implementation differs from the one we have seen in class).

When the while loop terminates, the algorithm is completed. Every cell has been visited and thus no cell is inaccessible. Also, since we test each possible cell to see if we've already been there, the algorithm prevents the creation of any open areas, or paths that circle back on themselves.

Model a maze:

Represent the maze as a graph data structure, where rooms (cells) are vertices and removed walls are edges between vertices. For help you can use the book "Data Structures and Algorithms in Java" (Chapter 13).

Solving the Maze:

After generating a maze, your program should solve the maze (find a path from the starting room to the finishing room)

1. using DFS and
2. using BFS.

Each search algorithm will begin at the starting room and search for the finishing room by traversing wall openings. The search should terminate as soon as the finishing room is found. For each search algorithm, you will output the order in which rooms where visited and indicate the shortest solution path from starting to finishing room.

Input: The program should accept the number of rows and columns r of the maze (use only r=4, 5, 6, 7, 8, 10).

Output: The program should print the maze, then the DFS solution, then the BFS solution. The maze is printed in ASCII using the vertical bar '|' and dash '-' characters to represent walls, '+' for corners, and space character for rooms and removed walls. The starting and finishing rooms should have exterior openings as shown.

```
+ +-+-+-+
|     | |
+ +-+-+ +
| | |   |
+ + +-+ +
|   |   |
+-+ + + +
|     | |
+-+-+-+ +
```

For the DFS and BFS solutions, the maze should be output twice, one for each algorithm. The first maze output for each algorithm should show the order that the rooms were visited by the algorithm. The maze should be printed exactly as above except that rooms should be printed with the low-order digit of the visitation order number. The starting room is '0'. Unvisited rooms should remain a space character. The second maze output for each algorithm should show the shortest solution path, using hash '#' character for rooms and wall openings on the solution path.

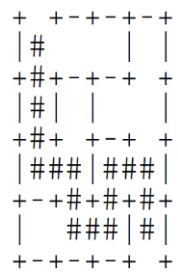
You will need to view the output in a fixed-width font.

Following is sample output for the maze in Figure 3:

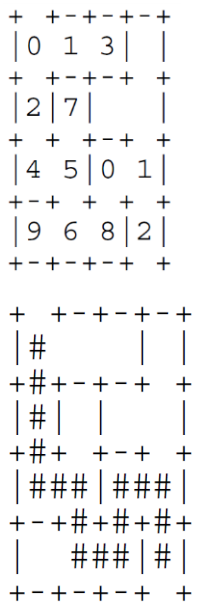
```
+ +-+-+-+
|     | |
+ +-+-+ +
| | |   |
+ + +-+ +
|   |   |
+-+ + + +
|     | |
+-+-+-+ +
```

DFS:

```
+ +-+-+-+
|0 1 2| |
+ +-+-+ +
|3| |   |
+ + +-+ +
|4 5|8 9|
+-+ + + +
|  6 7|0|
+-+-+-+ +
```



BFS:



Programming Standards:

- Your header comment must describe what your program does.
- You must include a comment explaining the purpose of every variable or named constant you use in your program.
- You must use meaningful identifier names that suggest the meaning or purpose of the constant, variable, function, etc.
- Precede every major block of your code with a comment explaining its purpose. You don't have to describe how it works unless you do something tricky.
- You must use indentation and blank lines to make control structures more readable.

Deliverables:

- ✓ Your main grade will be based on (a) how well your tests cover your own code, (b) how well your code does on your tests (create for all non-trivial methods), and (c) how well your code does on my tests (which you have to add to your test file). For JUnit tests check canvas.
- ✓ Use sjsu.<lastname>.cs146.project2 as your package, and Test classes should be your main java file, along with your JUnit java tests.
- ✓ Zip up the directory with your entire project (source code and report). Turn the zip file by uploading it to canvas.

- ✓ All projects need to compile. If your program does not compile you will receive 0 points on this project.
- ✓ Do not use any fancy libraries. We should be able to compile it under standard installs. Include a readme file on how to you compile the project.

Extra Help for JUnit tests: Create the 'same' pseudo-random numbers by using a specific seed (e.g. 0):

Random generator (once)

in the class

```
public class Graph {
        …
        private Random myRandGen;
        …
        double myrandom() {
                return myRandGen.nextDouble(); //random in 0-1
        }
```

And in the constructor

```
public Graph(int dimension_in) {
    myRandGen=new java.util.Random(0); //seed is 0
}
```

Usage

```
(int)(myrandom() * neighbors.size())
```