COEN 70: Formal Specification and Advanced Data Structures

Winter 2015

Lab 10: Maze

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 using a graph class. 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). Submit your code and test cases for it.

Generating a Maze:

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

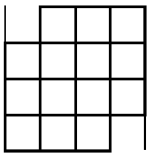
Figure 1

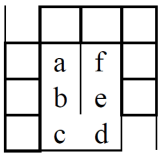

Figure 2

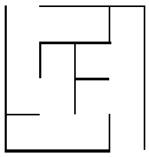
Figure 3

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 cells. The difficulty 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 cell and the finishing cell (for simplicity always use as starting and finishing cells the ones in Figure 1, i.e. , starting the upper left cell and finishing the lower right cell) . Unnecessarily removing too many walls will make the maze too easy to solve. Therefore, a wall should not be removed if the two cells 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 cell (therefore also the finishing cell) is reachable from the starting cell. There must be no cells 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 (different DFS implementations exist).

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 cells (cells) are vertices and removed walls are edges between vertices.

Solving the Maze:

After generating a maze, your program should solve the maze (find a path from the starting cell to the finishing cell) using DFS. Begin at the starting cell and search for the finishing cell by traversing wall openings. The search should terminate as soon as the finishing cell is found.

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

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

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

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

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

Following is a sample output for the maze in Figure 3:

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

DFS:

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

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