CS 410 Summer 2000
Homework 5
Due 11:30 AM, Monday July 31

1 Setup

For this assignment you will be creating random mazes to learn a few simple things about graphs. Each maze is an h-by-v grid of square cells (where h is the number of cells in the horizontal direction, and v is the number of cells in the vertical direction). Each cell is indicated by an ordered pair containing its column and row. Hence, the cell in the upper left corner is numbered $(0,0)$. The cell to its right is numbered $(1,0)$. The cell below the upper left cell is numbered $(0,1)$.

There are vertical walls and horizontal walls separating adjacent cells. Each interior horizontal wall has the same numbering as the cell immediately above it. Each interior vertical wall has the same numbering as the cell to its immediate left. Hence, horizontal wall (i,j) separates cell (i,j) from cell $(i,j+1)$, and vertical wall (i,j) separates cell (i,j) from cell $(i+1,j)$. Here is a depiction of a 6-by-3 grid.

```
+-----------+-----------+-----------+-----------+-----------+-----------+
|           |           |           |           |           |           |
|           |           |           |           |           |           |
|   (0,0) (0,0) (1,0) (1,0) (2,0) (2,0) (3,0) (3,0) (4,0) (4,0) (5,0)   |
|           |           |           |           |           |           |
|           |           |           |           |           |           |
+---(0,0)---+---(1,0)---+---(2,0)---+---(3,0)---+---(4,0)---+---(5,0)---+
|           |           |           |           |           |           |
|           |           |           |           |           |           |
|   (0,1) (0,1) (1,1) (1,1) (2,1) (2,1) (3,1) (3,1) (4,1) (4,1) (5,1)   |
|           |           |           |           |           |           |
|           |           |           |           |           |           |
+---(0,1)---+---(1,1)---+---(2,1)---+---(3,1)---+---(4,1)---+---(5,1)---+
|           |           |           |           |           |           |
|           |           |           |           |           |           |
|   (0,2) (0,2) (1,2) (1,2) (2,2) (2,2) (3,2) (3,2) (4,2) (4,2) (5,2)   |
|           |           |           |           |           |           |
|           |           |           |           |           |           |
+-----------+-----------+-----------+-----------+-----------+-----------+
```

Note that there is an h-by-$(v-1)$ set of horizontal walls, and an $(h-1)$-by-v set of vertical walls. In your maze, some of these walls will be present and some will be missing. The walls present are indicated by the arrays `hWalls` and `vWalls`, which are an h-by-$(v-1)$ boolean array and an $(h-1)$-by-v boolean array, respectively. (Exterior walls are numbered according to the same system, but there is no explicit storage for them, because they are presumed to always be present.)

2 What do I need to do?

First, edit the file Grid.java and complete the implementation of the createMaze method. Use a UnionFind object (which implements disjoint sets) to turn your grid into a random rectangular maze. Your random mazes should have two properties: there is a path from any given cell to any other cell, and there are no cycles (loops) – in other words, there is only one path from any given cell to any other cell.

The Grid constructor creates a grid in which every possible wall is present. To make a proper maze, you will need to eliminate walls selectively. Do so as follows.

1. Create a disjoint sets data structure in which each cell of the maze is represented as a separate item. Use the UnionFind class (described below).

2. Visit every interior wall of the grid (once) in a random order.

   One way to do this is to create an array in which every wall (horizontal and vertical) is represented. (How you represent each wall is up to you.) Maintain a counter w, initially set to the number of walls. Iterate the following procedure: select one of the first w walls in the array at random, and swap it with the wth wall in the array. (This has the effect of choosing a random wall to fill the wth place in the array.) Then decrease w by one. Repeat this operation w times, so that each element of the array, starting with the last, references a random wall.

   Next, visit the walls in the random order you just generated. For each wall you visit:

   (a) Determine which cell is on each side of the wall.

   (b) Determine whether these two cells are members of the same set in the disjoint sets data structure. If they are, then there is already a path between them, so you must leave the wall intact to avoid creating a cycle.

   (c) If the cells are members of different sets, eliminate the wall separating them (thereby creating a path from any cell in one set to any cell in the other) by setting the appropriate element of `hWalls` or `vWalls` to false. Form the union of the two sets in the UnionFind object (disjoint sets data structure).

When you have visited every wall once, you have a finished maze!

Second, you must implement the method `diagnoseMaze()`, which tests your maze for cycles or unreachable cells using either depth-first search or breadth-first search. Do this as follows.

1. You should think of the cells of the maze as vertices of a graph. Two adjacent cells are connected by an edge if there is no wall separating them.

2. If the search method ever examines an "edge" and discovers a cell that has already been visited, then there is a cycle in the maze. If some cell is not visited at all, then it is not reachable from the cell where the search started. Hence, depth- or breadth-first search can diagnose both potential deficiencies of a bad maze: having more than one path between two cells, or having no path between two cells.

3. Since depth-first search (DFS) and breadth-first search (BFS) are such generally useful algorithms, certain to be desired in other ways at some point, code whichever one of these you choose as a separate method which is called by `diagnoseMaze()`.

Third, you must answer the questions in the analysis section.

Fourth, you must test your program sufficiently as discussed in the testing section.

3 What's this code I've been given?

Good question. First, the easy stuff. From the file Grid.java, you are given:

- `toString()` converts the grid to a string so you can print it.
- `horizontalWall(x, y)` determines whether a horizontal wall is intact.
- `verticalWall(x, y)` determines whether a vertical wall is intact.
- `randInt(c)` generates a random number from 0 to $c - 1$, and is provided to help you write the `createMaze()` method. It generates a different sequence of random numbers each time you run the program.
- To test your code on different size mazes, you can input the dimensions of the maze on the command line (rather than changing the hard-coded size and recompiling. For a 6-by-3 maze, you would run "java Grid 6 3" on a command prompt. The J++ IDE has an option to set command line parameters, if you're using that. The default dimensions are 39×15.

Now, the harder stuff. The file UnionFind.java contains code that implements a UNION-FIND data structure, something you'll learn about in another class. For now, here's what you need to know about it. The data structure keeps track of disjoint sets as a collection of items (represented by integer identifiers from 0 to $n - 1$), and also keeps track of which set each is in. At the beginning, each element forms its own singleton set. The following operations create and modify these sets:

- `UnionFind(int numElements)` initializes a new `UnionFind` with `numElements` elements.
- `void union(int a, int b)` unions the set that a belongs to and the set b belongs to. It has no effect if a and b are already in the same set.
- `int find(int x)` returns an integer representing the set x belongs to. Two elements a and b belong to the same set if `find(a)` == `find(b)`.

4 Testing

Remember that you need to test your code for robustness, not just in the situation you feel will be most usual, but in extreme situations, as well (no, we do **not** want to see extremely large mazes). If `toString()` can handle it, so should `createMaze()`. Note that `toString()` does not handle empty grids, so you're off the hook on that one. The only thing you have control over is the shape, so we expect you to vary that. We are looking for 4 fundamentally different test cases, but will give you 5 chances to find them (i.e. you may submit 5 test cases). In the real world, you would be expected to run a randomized algorithm many times, to give more confidence that it will always work correctly, but frankly, we don't want to have to look at a lot of program output, especially when your code should be simple enough to verify by examination of the code. You would also be expected to write drivers which would deliberately break your mazes so you could adequately test your `diagnoseMaze()`, but once again, you're off the hook.

5 Analysis

Suggest (in simple English) how you could use depth- or breadth-first search to generate a random maze (or more importantly, lots of different random mazes), without using UnionFind at all.

1. How would your algorithm ensure that there is a path between every pair of cells, but no more than one path between any pair of cells (i.e., no cycles)?

2. How does your algorithm use random numbers to generate a different maze each time? Specifically, what decision should be made by random numbers at each recursive invocation of the search method?

These questions can be answered with just a few sentences.

6 What do I turn in?

1. Your modified version of Grid.java, with the methods `createMaze()` and `diagnoseMaze()` completed, and a new method for depth-first search or breadth-first search (named either `DFS()` or `BFS()`). Do **not** change the code which is already in place. In particular, this means the interfaces (i.e. the parameters, or lack thereof) for `createMaze()` and `diagnoseMaze()` are already specified, but you are free to choose your own interface for `DFS/BFS()`).

2. No more than 5 test cases, as discussed in the Testing section above.

3. Answers to the Analysis questions above.

7 Scoring

Testing: 10%
Analysis: 10%
createMaze(): 25%
diagnoseMaze(): 20%
breadth/depth-first search: 15%
Style: 20%