CS 61B Homework 9
Due 3pm Friday, April 22, 2005

Part I (8 points)

Edit the file Maze.java and complete the implementation of the Maze constructor. Use our disjoint sets data structure to create 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.

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). 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.)

The Maze constructor currently creates a "maze" 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 DisjointSets class (described in the Lecture 33 notes), which is in the set package we've provided.

(2) Visit every interior wall of the maze (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.) Scramble the walls by reodering them into a random permutation. Each possible permutation (ordering) of walls should be equally likely.

Here's how to do it. Put all the walls into the array. The idea is to randomly choose (from all the walls) the wall that will be at the end of the array. Swap it to the end, then never move it again. From the remaining walls, choose the wall that will come second-last. Swap it to its final position, then never move it again. Repeat until you've chosen a wall for each slot in the array.

Here's an algorithmic rephrasing of what I just said. 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 (at index w - 1). This permanently establishes the randomly chosen wall as the wth wall. Then decrease w by one. Repeat this operation until w is one.

Next, visit the walls in the (random) order they appear in in the array. For each wall you visit:

(i) Determine which cell is on each side of the wall.
(ii) 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.
(iii) 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 disjoint sets data structure.

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

Be forewarned that the DisjointSets class has no error checking, and will fail catastrophically if you union() vertices that are not roots of their respective sets, or if you union() a set with itself. You may want to add error checking to DisjointSets.java to help you find your bugs. This error checking can help you with Project 3 as well.

All the other methods you need, including test methods, are provided for you.

toString() converts the maze to a string so you can print it.
randInt(c) generates a random number from 0 to c - 1, and is provided to help you write the Maze() constructor. To keep the mazes interesting, it generates a different sequence of random numbers each time you run the program.
diagnose() tests your maze for cycles or unreachable cells using depth-first search. DON'T CHANGE IT. YOUR CODE MUST WORK WITH _OUR_ COPY OF THIS METHOD.
main() generates a maze (using your constructor), prints it, and tests it.

diagnose() depends on the following two methods, so don't make changes that will prevent these from working:

horizontalWall(x, y) determines whether a horizontal wall is intact.
verticalWall(x, y) determines whether a vertical wall is intact.

You may see how you're doing by compiling and running Maze.java. To look at a 30 x 10 maze, run:

```
java Maze 30 10
```

The default dimensions, if you don't specify any on the command line, are 39 x 15.

```
Part II  (2 points)
-------------------
You have probably noticed the similarity between your maze and a graph data
structure.  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.
Our diagnose() method uses depth-first search to test your maze.

If the depthFirstSearch() method ever examines an "edge" and discovers a cell
that has already been visited, then there is a cycle in the maze.  (The depth-
first search implementation used here never walks back over an edge it's just
traversed, so it won't look back and mistakenly diagnose a cycle.)  If some
cell is not visited at all, then it is not reachable from the cell where the
search started.  Hence, depth-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.  (You may want to look at the diagnose() and
depthFirstSearch() methods to see how this is done.)

In your GRADER file, suggest (in simple English) how you could use depth-first
search to generate a random maze (or more importantly, lots of different random
mazes), without using disjoint sets at all.

(a)  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)?

(b)  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 depth-first search method?

These questions can be answered with just a few sentences.

Submitting your solution
------------------------
Change (cd) to your hw9 directory, which should contain GRADER, Maze.java, any
other files your solution needs, and the set directory.  The set directory
should contain DisjointSets.java.  You must submit the latter because you're
allowed to change it.

Include your name, login, lab section, lab TA's name, and answer to Part II in
GRADER.  Make sure your homework compiles and runs on the _lab_ machines just
before you submit.

From your hw9 directory, type "submit hw9".  (Note that "submit" will not work
if you are inside the set directory!)  After submitting, if you realize your
solution is flawed, you may fix it and submit again.  You may submit as often
as you like.  Only the last version you submit before the deadline will be
graded.
```