# **Final Programming Assignment**

### Introduction

The final exam for this course is a programming assignment. It is based on Exercise 8.17 from the text, "Data Structures and Algorithm Analysis in Java, third edition", by Mark Allen Weiss. If you don't own a copy of the book, a pdf of it can be found on the Internet. The specific section of the book that provides information necessary of solving the exercise is section "8.7 An Application".  paraphrased as follows:

A simple algorithm to generate a maze is to start with walls everywhere (except for the entrance and exit). Continually choose a wall randomly, and know it down if the cells that the wall separates are not already connected to each other. Repeat this process until the starting and ending cells are connected. It is actually better to continue knocking down walls until every cell is reachable from every other cell (this generates more false leads in the maze).

### Specifics

(8.17) Write a Java program that generates mazes of arbitrary size, up to a maximum size of 50x80. Use Swing to generate a maze similar to that in Figure 8.25.

The cells of the maze should be implemented as elements in a Disjoint Set. Initially, each cell should be represented as a single element its own set. As walls are removed from the maze, the cells it separates should be merged into a larger set. When the starting cell and ending cell are contained within the same set, the algorithm to generate a random maze can terminate. However, in the words of the author, "It is actually better to continue knocking down walls until every cell is reachable from every other cell (this generates more false leads in the maze)."

Implement a java class named Maze. It should have as a minimum the following public interface:

***public Maze(int width, int height) { }*** - this constructor takes two integer arguments, ***height*** specifies the number of cells in a maze column; ***width*** specifies the number of "cells" in a maze row.

***public void drawWalls( ) { }*** - when called, this public method draws the current maze on a Swing canvas.

***public void decimateWalls( int beyond ) { }*** - randomly remove walls from the maze until the start cell and end cell are contained in the same set in the Disjointset. The argument, "beyond" specifies how many more times to continue randomly removing walls. The value of beyond must be between 0 and half the number of total cells in the maze (i.e., 0.5 \* width \* height).

***public DisjointSet getMaze( ) { }*** - return a reference to the current DisjointSet.

Use the DisjointSet provided on the Canvas module for this assignment. I will be testing your code on mazes that are matrices of  4x6, 10x10, and 50x80 cells. Your code will have to scale the size of the cells to fit within a single size Swing canvas (the drawing surface in Swing). In other words the drawing surface for the maze must accommodate a 4x6 matrix of cells as well as one that's 50x80 on a canvas of constant size. The following code sets up a 1000 column by 850 row drawing surface in Swing. You are free to decide on the actual dimensions of the drawing surface, but what ever the size, it's the same size regardless of the number of cells you specify in the maze.

// like toString, but instead of returning a String, it uses Swing calls to   
// draw the maze  
  
final private int CanvasMargin = 100;  
  
public void drawWalls() {  
        JFrame window = new JFrame();  
       window.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);  
       window.setBounds(CanvasMargin, CanvasMargin, 1000, 850);   
       window.getContentPane().add(new MazeCanvas());  
       window.setVisible(true);  
   }

When you have finished, upload a single java file named, Maze.java. a

### Extra Credit

Write an algorithm that graphically displays the solution to the maze. Start with the first cell and then advance through the solution one cell at a time with a delay between each step.