CMSC 132

Sections 020X/030X

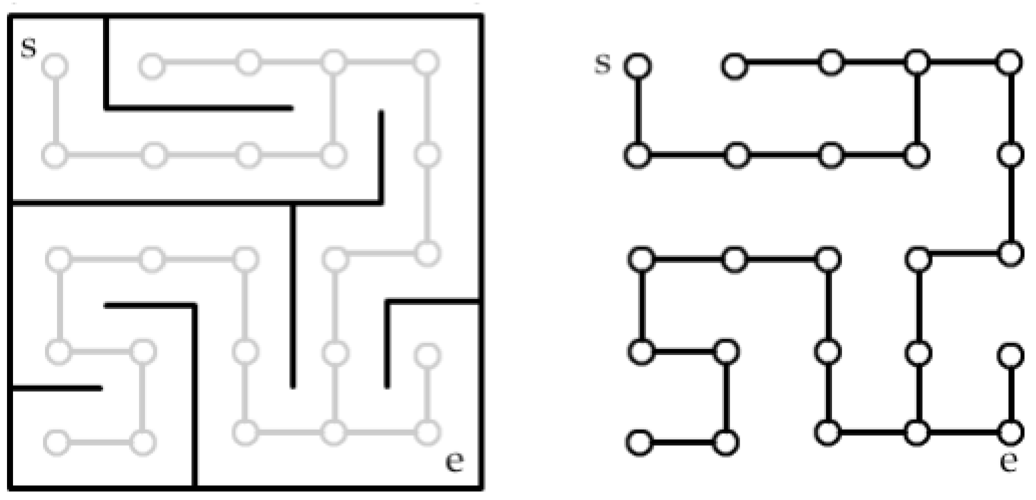
DEPARTMENT OF COMPUTER SCIENCE

Assignment: Project #7

Due Date: Thursday 4/28, 11:00PM

Open/Closed policy: CLOSED

Graphs and Mazes



Introduction

A maze can be viewed as a graph, if we consider each juncture (intersection) in the maze to be a vertex, and we add edges to the graph between adjacent junctures that are not blocked by a wall. **Please study the picture above until you understand how a maze can be converted into a graph.**

Our mazes will always have a starting point at the upper left corner and an ending point at the lower right corner.

Our mazes will have random weights (from 1 to 9) specified between any two adjacent junctures. (Sometimes these weights are not displayed on the screen, but they are always there.) These values can be thought of as the "cost" of traveling from one juncture to an adjacent one. The weights will be used for running Dijkstra's algorithm.

Our mazes can be made with varying degrees of "density" from very sparse to "100% dense". The more dense the maze, the more walls it has. A very sparse maze has very few walls at all; instead it has a lot of open space so that there are many different paths leading from one juncture to another juncture that is far away. On the other end of the spectrum, a maze with "100% density" contains lots more walls and has the interesting property that there is always **exactly one** path from one juncture to any other juncture in the maze.

The project will solve the mazes three different ways, by first converting the maze into a graph, and then applying one of the standard graph algorithms:

1. Depth-First-Search
2. Breadth-First-Search
3. Dijsktra's Algorithm (to find the path from "start" to "end" that uses edges with a minimal total cost). This is much more interesting with a sparse graph than with a "100% dense" graph, which only has ONE path from start to finish.

Project Implementation

Begin by downloading the project zip file and importing it into Eclipse, as usual.

Part I: WeightedGraph

You will begin the project by writing a class called "WeightedGraph". This class is very general and could be used for any purpose. (It has nothing to do with mazes.) The WeightedGraph class represents a weighted, directed graph. The implementation will include methods for Breadth-First-Search, Depth-First-Search, and Dijkstra's algorithm. You may use whatever data structure(s) you choose to implement your WeightedGraph. See the Javadoc for specific instructions about the methods you must implement.

Part II: MazeGraph

We have written a class called "Maze", which represents a Maze, and we have written a class called MazeGUI, which can display the maze in various ways. **When you are finished implementing the project, you should try running the main method in the MazeGUI class to see your hard work put to good use!** The GUI has buttons that will allow the user to re-draw the Maze as he/she likes, and to solve the maze in various ways. The MazeGUI class works in conjunction with a class you will write (described below) called MazeGraph. The MazeGraph class can convert a Maze into a graph. Before you begin working on the MazeGraph class, you should familiarize yourself with the API for the Maze class by reading the Javadoc.

The MazeGraph class is an extension of WeightedGraph<Juncture>, which is the class you wrote in Part I, where we choose to use "Juncture" objects as vertices. ("Juncture" is a simple class that we have provided.)

The constructor of the MazeGraph class will convert a Maze into a graph in the manner depicted in the picture at the top of this project description. If you view the maze as a rectangular grid of "junctures", then each vertex of the graph you will construct consists of one juncture from the maze. Two adjacent junctures will be connected by an edge if there is no wall separating them in the maze.

You only need to write the constructor for the MazeGraph class, which takes a Maze as a parameter and constructs the MazeGraph from it. The vertices of the MazeGraph will be Juncture objects. (Juncture is a simple class that we have provided.) Here are specific details about how to write the constructor:

1. For every juncture in the Maze, add a Juncture object as a vertex in the MazeGraph. You must instantiate the Juncture objects yourself, using nested loops to process all of the X and Y coordinates from the maze.
   - As usual, the juncture at the top-left of the maze has X and Y coordinates equal to 0; the X coordinates get larger as you move to the right, and the Y coordinates get larger as you move down. E.g.: Use **new Juncture(0, 0)** to create the Juncture that represents the top left corner of the maze, **new Juncture(1, 0)** for the one just to its right, etc.
   - Notice that the parameters for the Juncture constructor represent X and Y coordinates, **not** row and column, which would be backwards!
   - Also note that the Maze class has methods that you can use to access its height and width so that you know how many junctures to add as vertices to your MazeGraph.
2. Now you must add weighted edges. For each pair of adjacent junctures A and B in the maze (that are not separated by a wall) you must add two edges to the graph: One from A to B and the other from B to A. The Maze has methods that you can use to ask if there are walls above, below, to the left, and to the right of any particular Juncture, so you can check whether or not two adjacent junctures are separated by a wall. (Only add an edge if there is no wall between.) The Maze also has methods that you can use to ask about the weight between the current juncture and the one above, below, to the left, or to the right -- use these values for the weights of the edges in the graph.

Running the GUI

When you're finished coding, try running the main method in the MazeGUI class. It's pretty cool!

Grading

The public tests, and release tests together will comprise 90% of your grade on the project. (There are no secret tests this time.) The remaining 10% will be "style grading", done by visual inspection. Be sure to use good variable names, proper indentation and use of braces, etc. Also be careful to avoid redundant code.