COSC-211: DATA STRUCTURES
HW9: MAZE SOLVING

Due Thursday, April 19, 11:59pm

Reminder regarding intellectual responsibility: This is an individual assignment, and the work you submit should be your own. Do not look at anyone else's code, and do not show anyone your code (except for me and the course TAs).

1 Introduction

In this assignment, you will write a program to solve mazes in the style and format of the mazes that you generated in HW8. In doing so, you will implement a graph data structure using one of the representations we've discussed in class, and you will implement either Breadth-First Search or Depth-First Search. The goals of this assignment are for you to think about which representation and search process best fit the needs of the problem you are solving, and, as usual, for you to get more practice and experience implementing data structures (and debugging!).

1.1 Setup

Begin by downloading some starter code:

```
wget -nv -i https://goo.gl/H8GZXn
```

You should now have three Java files:

- `Node.java`: represents a square in the maze grid (this is what we called a `Cell` in HW8; I've renamed it `Node` for this assignment because it contains different variables and methods than `Cell.java` from HW8). Each node contains:
  - An `index` that uniquely identifies it; in a graph with n nodes the nodes will have indices 0 through $(n-1)$. The nodes in a maze are organized in a grid and are indexed row by row. For example, in a 3×3 grid there are 9 nodes indexed as follows:

0	1	2
3	4	5
6	7	8

  - Two booleans, `visited` and `inSolution`, which are meant to help you keep track of which nodes you have searched so far, and which nodes end up in the solution path from entrance to exit.
  - A method called `toString()` that converts a `Node` to a `String`, where the value of the `String` depends on whether the `Node` has been visited and whether it is marked as being part of the solution. This is used by the `printMaze()` method in `MazeSolver` (see below).
  - A constructor that sets the index of the node.

- `Graph.java`: some very basic starter code to implement a graph. You will fill in most of this as part of the assignment. Right now all it contains is an array of `Node`s and a constructor that initializes each `Node` in this array with an index corresponding to the array position. There is also a `addEdge()` method that doesn't do anything; this is included so that the code compiles as is, and you will need to make the method do something more meaningful.
- `MazeSolver.java`: some starter code to help you solve mazes. Specifically, the code that I have given you does the following:
  1. Calls the `parse()` method to read the contents of a file* and extract information about the size of the maze and the walls in the maze.
  2. Calls the `buildGraph()` method to generate a graph based on the maze information extracted by `parse()`. This method calls the `addEdge()` method in `Graph`.
  3. Calls the `printMaze()` method to print the maze. Right now if you compile and run the program an empty maze will print that exactly matches the contents of your input file. If you update the booleans in each node as you're solving the maze, it should print out something more interesting when you're done solving.

  *Here the name of the file containing the maze is an input parameter to the program. For example, if you run the program using the command `java MazeSolver maze.txt`, this will solve the maze contained in `maze.txt`. If you don't specify an input parameter, the program will not run.

Now run the following command:

```
wget -nv -i https://goo.gl/L3PnGr
```

This will download several mazes of different sizes on which you can test your solving program. If you submitted HW8 on time, one of these mazes should look familiar to you—these are the mazes you generated in HW8!

2 Your Tasks

Your job in this assignment is to complete the program so that it solves mazes. Specifically, you will do the following:

1. Finish implementing the `Graph` class.
   (a) The first step is to choose a representation: adjacency matrix or adjacency list. **In a comment at the top of `MazeSolver.java`, answer the question:** Which representation do you think is more appropriate for this problem? Why?
   (b) Fill in the `addEdge()` method. The inputs to this method are two `int`s, where each input represents an index of a `Node`. For example, calling `addEdge(3,6)` should add an edge between the node with index 3 and the node with index 6.

   (c) You will need to make other changes to `Graph` and/or `Node` to complete your implementation. What changes you make will depend on which representation you chose to use. You are free to add other fields and methods to `Graph` and `Node` (for example, you likely will want to include an `edgeExists()` method), and you can even add other classes if you wish.

2. In `MazeSolver`, write a method called `solve()` that takes a `Graph` as input and returns an array of `Nodes` containing, in order, the `Nodes` included on the path from the maze entrance to the exit. Your `solve()` method should use either Breadth-First Search or Depth-First Search (hint: if you need a stack or a queue, you have one from way back in HW2!).

   (a) **In a comment at the top of `MazeSolver.java`, answer the question:** Which search algorithm did you choose, and why?

3 Submit your work

Make sure you **test your code thoroughly** before submitting it. Code that does not compile will not receive credit.

Submit all of your Java files using either the submission web site or (from `remus/romulus`) the command line:

```
$ cssubmit *.java
```

This assignment is due on Thursday, April 19, 11:59pm.