CSCI 338—Algorithm Design and Analysis
Programming Assignment 3: Depth-first search of a graph

Due: Friday, February 24, 2017

This is an individual assignment.

This assignment is to implement the depth-first search algorithm using an adjacency-matrix graph representation on an undirected graph.

Language

You may use Java, C, C++, or Haskell for this assignment. If you wish to use a different language, you must first get permission from the instructor. Use a suitable built-in list implementation for your language.

A note on 2D arrays in Java

The adjacency-matrix representation uses an $n \times n$ matrix of integers, where n is the number of vertices in the graph. In Java, you should use a 2D *int* array for the matrix. You can allocate the array with one step as follows:

```
int [][]adjMat;
…
adjMat = new int[n][n];
```

Once you have created this array, you can use *adjMat.length* to get the value on *n*, so you don't need to create a separate class for the graph representations, although you may if you want to.

Building random graphs

Create a class or suite of functions that has the following methods/functions to represent graphs:

- Create and return a new $n \times n$ array with all the entries initialized to zero.
- Create and return a new $n \times n$ array representing an undirected graph with k randomly-chosen edges, $0 \le k \le n(n-1)/2$, not necessarily a connected graph. (Ideas on how to do this below)
- Create and return a new $n \times n$ array representing a *connected* undirected graph with k randomly-chosen edges, $n-1 \le k \le n(n-1)/2$. (Ideas on how to do this below)
- Read a text file containing a graph description in the following format and return a corresponding adjacency-matrix array: the first line of the input contains the value *n*; there follow *n* lines each containing *n* zeros and ones separated by spaces, representing the matrix. Use an open *Scanner* as the parameter if you are programming in Java, and have the calling program handle the work of opening the file and creating the Scanner.
- Write the entries in a matrix array into a readable form, using the same format as the previous method/function. If you are programming in Java, use a *PrintWriter* for the parameter.

To create the (not necessarily connected) graph with k random edges, do the following: For each of the k desired edges, randomly select two vertex numbers (0 … n-1); if the two numbers are the same (no self-edges allowed) or if the edge is already in the graph, discard the two numbers and select again; otherwise, put the edge (both directions) into the matrix by putting a one in the appropriate two cells. Continue until the desired number of edges are selected.

To create a *connected* graph with *k* random edges, first select *n*-1 edges that connect the graph (without any cycles): add the edge (0, 1) to the matrix, then for each vertex *i* from 2 to *n*-1, randomly choose a second vertex in the range (0 … *i*-1) and add the edge (both directions) to the graph. For the remaining vertices, proceed as above. (Note that this process doesn't generate every possible connected graph, but it does generate graphs isomorphic (the same shape as) every possible graph.)

Use these algorithms to generate several small (5-10 vertex) graphs and verifying that they have the appropriate properties. It will probably be helpful to diagram the graphs. You can generate an unconnected graph by using the first generating algorithm with fewer than *n*-1 edges.

The algorithms

Using the code you wrote above, implement the *explore* algorithm in Chapter 3 of the textbook, replacing the first statement with the statement: `visited[v] ← compNum`, where *compNum* is a global integer variable accessible in both *explore* and *dfs2*. Use the versions of *previsit* and *postvisit* described in Section 3.2.4 to collect the pre-order and post-order numbers of the vertices. Finally, implement the *dfs* algorithm as follows:

```
procedure dfs2(G)

Input: a graph G = (V, E)
Output: an array visited[0 … n-1] of integers containing the connected
      component number of each vertex.

for all v ∈ V: visited[v] ← -1
compNum ←0
for all v ∈ V:
      if visited[v] < 0:
            explore(v)
            increment compNum
```

Testing the algorithms

Using the same or similar graphs to the ones you used for testing in the first section, run the *dfs2* algorithm on these graphs and print out the *pre*, *visited*, and *post* arrays for each graph in a form that allows you to see the values for each vertex together. Verify that your results are correct by comparing these values to the graph itself, either in its adjacency matrix representation or in its diagram.