CS 6301.002. Implementation of advanced data structures and algorithms
Spring 2016
Short Project 2
Wed, Jan 27, 2016

Ver 1.0: Initial description (Jan 27, 9:00 AM).

Due: 1:00 PM, Thu, Feb 4.

Submission procedure:
Create a folder whose name starts with the name of the group (e.g., "G18_SP2"). Place all files you are submitting in that folder. There is no need to submit binary files created by your IDE (such as class files). Do not submit any data files bigger than 10 KB. Make sure there is a "readme" file that explains the contents of the files being submitted. Zip the contents into a single zip or rar file. Upload that file on elearning. Submission can be revised before the deadline. Only the final submission before the deadline will be graded. Only one member of each group needs to submit project.

Solve at least one problem from the following list. First solution will be graded out of 10. Each additional problem will be considered for 1 extra point.

No sample data will be provided for short projects. Create your own input data sets for testing.

Topic: Graphs

Download the following java files and use/modify them as needed:
http://www.utdallas.edu/~rbk/teach/2016s/java/code/Edge.java
http://www.utdallas.edu/~rbk/teach/2016s/java/code/Vertex.java
http://www.utdallas.edu/~rbk/teach/2016s/java/code/Graph.java

a. Topological ordering of a DAG.
Implement two algorithms for ordering the nodes of a DAG topologically. Both algoritms should return null if the given graph is not a DAG.

```
List<Vertex> toplogicalOrder1(Graph g) {
   /* Algorithm 1. Remove vertices with no incoming edges, one at a
      time, along with their incident edges, and add them to a list.
    */
}

Stack<Vertex> toplogicalOrder2(Graph g) {
   /* Algorithm 2. Run DFS on g and push nodes to a stack in the
      order in which they finish. Write code without using global variables.
    */
}
```

b. Diameter of a tree.
In this problem, you are given an unrooted tree as input. Since the tree may not be a binary tree, we will represent it with an adjacency list (i.e., it is a graph that happens to be a tree). The following algorithm can be used to find its diameter:

1. Run BFS from an arbitrary node as root.
2. Select a node Z with maximum distance from the first BFS.
3. Run a second BFS with Z as root.
4. The diameter of the tree is the maximum distance to any node from Z in BFS 2.

```
int diameter(Graph t) { ... }
```
If the input graph is not a tree, return -1.

c. Strongly connected components of a directed graph.

```
   Implement the algorithm for finding strongly connected components of a
   directed graph (see page 617 of Cormen et al, Introduction to algorithms, 3rd ed.).
   Run DFS on G and push the nodes into a stack as they complete DFSVisit().
   Find G^T, the graph obtained by reversing all edges of G.
   Run DFS on G^T, but using the stack order in DFS outer loop.
   Each DSF tree in the second DFS is a strongly connected component.

   int stronglyConnectedComponents(Graph g) { ... }
   Each node is marked with a component number, and the function returns
   the number of strongly connected components of G.


d. Finding an odd-length cycle in a non-bipartite graph.
   Given a graph, find an odd-length cycle and return it.
   If the graph is bipartite, return null.

   Algorithm: run BFS.  If no edge of G connects two nodes at the same
   level, then the graph is bipartite and has no odd-length cycle.
   If two nodes u and v at the same level are connected by edge (u,v),
   then an odd-length cycle can be found by combining the edge (u,v)
   with the paths from u and v to their least common ancestor in the BFS tree.
   If g is not connected, this is repeated in each component.

   List<Vertex> oddLengthCycle(Graph g) { ... }


e. Is a given graph Eulerian?

   A graph G is called Eulerian if it is connected and the degree of
   every vertex is an even number.  It is known that such graphs have a
   cycle (not simple) that goes through every edge of the graph
   exactly once.  A connected graph that has exactly 2 vertices of odd
   degree has an Eulerian path.  Write a function that outputs one of
   the messages that applies to the given graph.

   void testEulerian(Graph g) { ... }

   Possible outputs:

   Graph is Eulerian.
   Graph has an Eulerian Path between vertices ?? and ??.
   Graph is not connected.
   Graph is not Eulerian.  It has ?? vertices of odd degree.
```