# CS310: Advanced Data Structures and Algorithmns

# Fall 2014 Programming Assignment 6

# Due: Thu., December 4, 2014 at midnight.

## Goals

This assignment aims to help you:

- Use an open-source Graph package in a graph algorithm
- See how to implement graphs with the help of JDK classes
- Study another example of a type hierarchy at work
- Use Swing support for graph visualization Learn about backtracking and dynamic programming

## Description

We will be using the open-source Java graph package "JGraphT". See its home if you are interested. A subset of its distribution with everything we need is available at $cs310/jgrapht.zip. The graph APIs are in package org.jgrapht, most significantly Graph.java. The cs310-related sources are in package cs310.You will find javadoc for the whole package, including the cs310 additions, at javadoc. The other packages are exactly as distributed in JGraphT. Some experimental and test-related packages and the big jgraphmanual.pdf have been deleted to save on size. Of course you can download the whole thing yourself from the JGraptT site. The jgraphmanual.pdf is separately provided in $cs310, but it may not be helpful since it covers only the graph display package, not the JGraphT package we will mainly be using. To work on your home computer, download jgrapht.zip and install it the same way you did the games.zip distribution. For eclipse, set the output directory to build (not classes this time) instead of bin.

## JGraphT Graphs

The Graph API in Graph.java can be used to describe both directed and undirected graphs, by careful definition of the methods with this flexibility in mind. Concrete classes for directed and undirected graphs are provided, and interfaces DirectedGraph and UndirectedGraph, with a few methods involving degrees of vertices that can't be merged together. See cs310.LoadGraph.java for an example.

## Questions

1. **Topological Sort**. Write a generic <V, E> class TopSort.java which finds a topological sort for a given directed graph. The constructor takes the DirectedGraph<V, E> to work on. Here is the one public method it needs:

   ```
   public List<V> getTopOrder() throws HasCycleException
   ```

   Its output is a List of the vertices of the graph in topological order. Write a client TopSortTest.java to test your TopSort class without Swing visualization. Like DFSTest, TopSortTest takes an optional argument for the input file name. With an input file, say prereq.dat, in the top-level directory, you would use the command line:

   cd build
   java cs310.TopSortTest ../prereq.dat        (with no arg, uses test.dat)

Some sample input files are prereq.dat and cycle.dat described below, as well as test.dat you can find in the JGraphT package (you can ignore the weights of course). Here is the output from those two files:

Input graph

| | |
|---|---|
| 110 | 210 |
| 110 | 240 |
| 210 | 310 |
| 240 | 310 |
| 140 | 310 |
| 310 | 450 |
| 450 | 650 |

topological sort:

[110, 210, 240, 140, 310, 450, 650]

Input graph

| | |
|---|---|
| B | foo |
| foo | bar |
| bar | baz |
| baz | foo |
| B | C |
| X | C |
| bar | OK |

topological sort:

cycle: [foo, bar, baz, foo]

Note that topological sort order is not unique. The second example covers the case that topSort.getTopOrder() throws a HasCycleException when the input directed graph g contains a cycle. Your program should check if a graph has a cycle (and throw an exception if it does) but you DON'T have to determine the cycle itself. Do it only if you want to. If you do determine a cycle, provide another method to return it to the client.

Also write a client TopSortDemo.java to display the completed sort using colors as done in DFSDemo, that is blue first, etc. TopSortDemo needs no arguments on the command line and uses the random graph generator just as DFSDemo does.

Here are two ways to implement Topological Sort:

1. Use the method given in Weiss on pages 499–501.
2. Use Depth First Search postorder. See class notes. This should require just a few lines of code.

**You can choose any method you want, except using the canned JGraphT algorithms and iterators – that's too easy.**

2. **Implementing the Graph API.** We have provided a simple undirected graph implementation in $cs310/jgrapht/src/cs310/SimpleGraph1.java, following the general methodology of the JGraphT implementations, but without all the optimizations. The removeVertex and removeEdge methods are not functional yet, so you have to implement them. Modify SimpleGraphTest.java so that it thoroughly exercises your code.

3. **Dijkstra's Algorithm.** Write a class Dijkstra.java that finds (as efficiently as possible) the least-cost paths from a given node to all nodes in the given directed graph, given non-negative edge costs as weights in the given Graph. Note LoadWeightedGraph in the cs310 directory.

```
// a Dijksta HASA DirectedGraph
public Dijkstra( DirectedGraph<V,E> g)
// compute best costs from start
public Map<V,Double> dijkstra(V start);
// return least-cost path from start to target
public List<V> pathTo(V start, V target);
```

Remember the start vertex used in a call to dijkstra so that you can reuse data structures if pathTo is called with the same start vertex. Provide DijkstraTest.java so that the command (after cd build, and assuming yourgraph.dat is in the top-level directory)

```
java cs310.DijkstraTest ../yourgraph.dat
```

will output a table showing the best distances from the first node in the data to all others, and the shortest path from the first node.

A sample input file test.dat should produce output like this, in any order:

Shortest Distances and their Paths from A

```
 A 0 (A)
B 1 (A, B)
X 3 (A, B, X)
Y 5 (A, B, X, Y)
Z 6 (A, B, X, Y, Z)
C 2 (A, C)
J 5 (A, C, J)
M INFINITY ()
```

Also provide a program DijkstraDemo.java, that takes a randomly-generated graph (UnweightedDemo uses LoadGraph, do a simple modification of that to use LoadWeightedGraph) and shows the progress of dijkstra vertex by vertex, coloring the vertices of newly-determined distance in the same color sequence defined in UnweightedDemo.java. Its command line execution, after cd to build:

```
java -cp ../lib/jgraph.jar;.  cs310.DijkstraDemo
```

For UNIX execution, replace the semicolon above with a colon.

We know of one easy way to implement Dijkstra efficiently, using a trick that allows us to use an ordinary priority queue and yet change the effective priority of elements when we need to. This trick is discussed in the class notes and is coded in Weiss on page 495, but for pa6 be sure to use JGraphT.

For the priority queue, set up a PriorityQueue and Path as in Weiss, but you also need a Map (something like the Map from V to Unweighted.DistInfo in cs310.Unweighted) to handle the information held in Weisss "scratch" spot, v.scratch, as well as v.dist and v.prev. But don't use the name "scratch". This is a flag for the vertex having been processed, and deserves a name that says that. The Path class in the book is usable as is.

**memo.txt**

1. Do questions 6 and 7 in HW5. Submit in class on Thursday, Dec. 4.
2. For SimpleGraph1, what is the time complexity of methods addVertex(V) and addEdge(V,V), in terms of the number N of nodes and E of edges in the graph?
3. We are using HashMaps in our graph implementation. Could we switch over to TreeMaps by just replacing "HashMap" with "TreeMap", or would there be problems in doing this?
4. If we were using lots of little graphs, the memory overhead of all these HashMaps could be a problem. What is the default initial size of the hash table? How can you halve it?
5. What is the big-Oh performance of your Dijkstra implementation? Show its derivation and what assumptions you are using.
6. Estimate the big-Oh memory usage with N nodes and O(1) edges leaving each node.
7. If you used any late days, indicate it in your memo.txt.