PROJECT 3
DUE 2:00 PM MONDAY DECEMBER 8 **CS 2223 B TERM 2014**

version of *December 5, 2014 – 10 : 03*

The last project involved very little Java code, but asked you to think hard to derive algorithms for the task at hand. In this project, the key task will be to read and adapt code already written—available from the text web site—and assemble it so that it works together.

Added Dec 4: see the Appendix for some remarks that should clarify what your goals are from a running-time point of view.

Background

Suppose a minimum spanning tree T has been computed for an edge-weighted graph G. Now suppose that we change G by adding or removing an edge, or by changing the weight on an existing edge, yielding a different edge-weighted graph G'. It turns out that we can compute an MST T' for G' incrementally, that is, without starting all over again. In this project you will develop such an "adaptive" minimum spanning tree algorithm.

It is a very nice fact of life that when an edge is updated, at most one edge of T needs to be changed; see the next section.

Theory Let $G = (V,E)$ be a connected undirected weighted graph, and let $T = (V,F)$ be a minimal spanning tree for G. If we update G to a new graph G', the following describes how a minimum spanning tree T' can be constructed for G'.

1. If G' is obtained from G by **increasing** the weight of an edge $\{x,y\}$ where $\{x,y\}$ is an edge in G but **not in** T: then $T = (V,F)$ is still a minimum spanning tree for G'. (We mean that the same set of edges defining the original T still defines a minimum spanning tree for G' even though the weights are different.)

2. If G' is obtained from G by **decreasing** the weight of an edge $\{x,y\}$ where $\{x,y\}$ is an edge in G that **is in** T: then $T = (V,F)$ is still a minimum spanning tree for G'.

3. If G' is obtained from G by **decreasing** the weight of an edge $\{x,y\}$ where $\{x,y\}$ is an edge in G but **not in** T: then we consider the cycle obtained by adding $\{x,y\}$ to T and form T' by deleting the maximum-weight edge in this cycle.

4. If G' is obtained from G by **increasing** the weight of an edge $\{x,y\}$ where $\{x,y\}$ is an edge in G that **is in** T: then we consider the two trees T_1 and T_2 obtained by removing $\{x,y\}$ from T. We ask whether the minimum-weight crossing edge $\{u,v\}$ connecting T_1 and T_2 has weight less than that of $\{x,y\}$. If so we swap out $\{x,y\}$ and swap in $\{u,v\}$ to obtain T'.

5. If G' is obtained from G by adding a **new weighted edge:** it is not hard to see that this can be handled by the one of the techniques above (think about it).[1]

[1] in fact removing an edge can also be handled without any new ideas; you aren't asked to implement that here simply because it would complicate the user-input

Assignment

Write a class `AdaptiveMST` whose main method behaves as follows.

Your program will read an edge-weighted undirected graph G from a file, using the same conventions as in the constructor for EdgeWeightedGraph.java.

It will first compute a minimum spanning tree T. If the number of edges is less than 50, print the tree to standard output. It is assumed that you will just use the text's code to compute T, so there is no challenge here: we ask you to do it to make it easier to grade your updating code.

The program will next read single lines of input from standard input, each line denoting a weighted edge, using the same convention as in the constructor for EdgeWeightedGraph.java. This will represent an already-existing edge with a new weight, or it might represent a new weighted edge.

After each line of input your program should decide how to update the MST appropriately, following the advice above, to arrive at a minimum spanning tree T' of G'. The output after each line should *print the difference between T and T'*, namely the edge+weight removed from T and the edge+weight added to T' (or print "no change" if appropriate).

A blank line of input terminates the program.

Note that your program will be getting input in two different modes: (i) it will read from a a file to get the initial graph, while (ii) it will read standard input to get updated edges "on-line". The starter code provided will handle that business for you. So to call your program you would execute (if the input file were tinyEWF.txt)

```
> java AdaptiveMST tinyEWG.txt
```

Note: no indirection symbol `<` before the input file name!

An important simplification. Assume that each line of input represents an update to *the original graph G and tree T*. It is conceptually and algorithmically simple enough to have edge-updates happen sequentially but it would be annoying for you to have to have to do so, so here we keep it simple. Thus, do *not* change the original MST T as you process the user-input.

Resources Together with the starter code for `AdaptiveMST` I have collected the I/O routines from the StdIn and StdOut libraries, so that, if you wish, you can develop your code in a directory counting these files. This means that the standard javac and java commands will compile and execute your code (in case you prefer not to use the algs4-javac and algs-java commands). See the course web page.

The text web page has a code you can download and use, including infrastructure for Edge Weighted Graphs and Prim's Algorithm. It also has code for computing cycles in a graph and connected components, but you will have to adapt some of this because, as written, it applied to undirected-but-not-weighted graphs or weighted-but-directed graphs. The modifications are straightforward.

Assumptions and Conventions You may assume that the orignal input graph G is connected, has no self-loops, has no parallel edges, and has unique edge weights.

You may assume that no user input will create a graph with repeated edge weights, and no user input will add new vertices to the graph.

Sample input/output

Suppose we run the program starting with the input graph from the file tinyEWG.txt. See page 604 of the text for the graph and the MST. Now suppose the user makes the following modifications:

- decrease the weight of tree edge $\{5,7\}$ to be .25: *no change in the tree*
- increase the weight of non-tree edge $\{1,2\}$ to be .40: *no change in the tree*
- decrease the weight of non-tree edge $\{4,6\}$ to be .39: *the result is to remove tree edge* $\{2,6\}$ *and replace it by edge* $\{4,6\}$
- increase the weight of tree edge $\{0,7\}$ to be .42: *the result is to remove tree edge* $\{0,7\}$ *and replace it by edge* $\{1,3\}$
- increase the weight of tree edge $\{0,7\}$ to be .20: *no change in the tree*

Submission

Some of you were caught off-guard by the last assignment, and there were too many late submissions. And with a class this size, it seems to be too unwieldy to manage my preferred late-penalty system, in which the amount of penalty is related to the closeness-to-completion of the work done before the due date. So we will revert to the following simpler scheme:

> **Projects submitted late, but less than 24 hours late, incur a 25% grade penalty. No assignments will be accepted beyond 24 hours after the due date/time.**

All submission are to be done via Turnin (don't email code to course staff)

Grading Rubric

Since software development is not the primary subject matter of the class, the project grading will stress correctness. We continue to expect the good software development habits developed in previous classes, but in this class we will not ask you to explicitly present evidence of those habits for grading. For example, we assume that you have done thoughtful testing but showing us that test data is not part of your submission.

The project will be graded on a 20-point scale

- 4 pts documentation and structure:
  - Put the names of all students for whom the assignment is submitted at the beginning of each source code file
  - Annotate every method with a brief comment explaining what the method does.
  - Use newlines and indentation to produce clean, well-formatted code (we actually read your code, we don't just run it).

  - It is OK to use or modify code that you find at the text's web site, as it was released under the CNU General Public License, but note in the comments that you are doing so, and give proper credit. **Giving such acknowledgement is part of earning your documentation points.**

- 16 pts correctness.

  1. You will receive no credit for a solution that re-runs a standard MST computation at each step.
  2. Items 1 and 2 in the list of possible alterations to G are the easiest to handle: these will earn a small amount of credit (4 points). Do these first.
  3. The other items are the significant aspect of the project and will account for the remaining correctness points.

Appendix

What should you (and we) expect in terms of run-time for your algorithms?

For the "decrease non-tree edge" situation: you should be able to come up with an algorithm that runs in worst-case time $O(n)$ where n is the number of vertices.

For the "increase tree edge" situation: you should be able to come up with an algorithm that runs in worst-case time $O(m)$ where m is the number of edges. This is, asymptotically, not that much faster than the $O(m \lg n)$ time required to run a new MST calculation, it is true. But (i) it is still an improvement, (ii) the algorithm you should come up with is conceptually simpler, and (iii) in practice will run faster than the $O(m)$ time suggests. (And, by the way, if you use there are very sophisticated data structures you could do even better than $O(E)$ in the worst case, but they are beyond the scope of this class.)