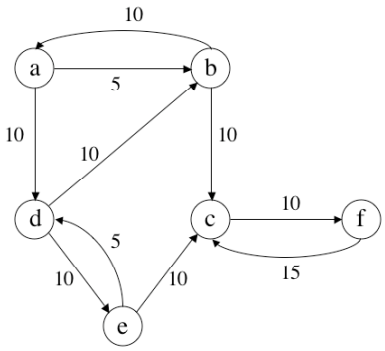
Harvey Mudd College
Computer Science 42
Fall 2009

Assignment 6
Fun with Java, Data Structures, and Directed Graphs
Due. 11:59 p.m., Wed., 4 November 2009

Implement a Java program that computes the shortest paths from the first-listed node (called the "source") to all other nodes in directed graphs with non-negative arc weights. The input format for a graph is a free-form S expression, representing a list of nodes. Each node is a list beginning with a *name*, which is a string or symbol, followed by a list of pairs of the form (<target> <weight>). By target, we mean a node to which the source node is connected and by weight we mean the direct-connection cost from the source to target. The latter cost might not be the shortest-path cost, however. An example graph is shown below with its representation:

```
(
 (a (b 5)   (d 10))
 (b (a 10)  (c 10))
 (c (f 10))
 (d (b 10)  (e 10))
 (e (c 10)  (d 5))
 (f (c 15))
)
```



In the graph shown, the set of nodes is {`a, b, c, d, e, f`}. There is a direct connection from `a` to `b` with a distance of 5, from `a` to `d` with a distance 10, etc. The direct distances are not necessarily symmetric, i.e. the distance from `a` to `b` is 5, while the direct distance from `b` to `a` is 10, in this example. In any case, the direct distance is not necessarily the shortest directed distance from the source to other nodes.

The output for the program is an S expression listing all nodes reachable from the source node, in order of decreasing distance from the first node, in the following format:

(<distance from source> <name of node> <name of predecessor on shortest path>)

For the above example, we would have the output (where the first node is `a`):

```
((25 f c) (20 e d) (15 c b) (10 d a) (5 b a) (0 a a))
```

Nodes that are not reachable from the source should not be listed. List the predecessor of the first node as itself by convention.

Suggestions and hints for solving this problem:

a. You are welcome to use my openlist package, which includes an S-expression reader. With the reader, the entire graph can be read in one statement (just as in Scheme), then you may use the structure as is (using first, rest, etc. to extract info), or convert it to some other structure. For example, you might convert the graph into an array of OpenList's, one entry per node.

b. Dijkstra's algorithm, as explained in the lecture and diagrammed on the next pages, is highly recommended. **Note**: Except for using standard Java libraries and the one provided, your implementation must be your own code, not someone else's.

c. You may read more about Dijkstra's algorithm, and see pseudo-code, here:

`http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm`

d. Dijkstra's algorithm will "retire" one node at a time, beginning with the source node. When a node is retired, its minimum distance from the source is known. (The simple justification is that, if there were a *shorter* path to the retired node coming from some other active (i.e. un-retired node) then *that* node should be the one to be retired, contradicting the choice of retiree.)

e. On each major step, the active node with the minimum distance estimate is retired and the best estimates of minimum distances to the remaining active nodes are updated for posterity, to reflect the possibility of a path from the newly-retired node to the remaining active nodes. This is the essence of Dijkstra's algorithm.

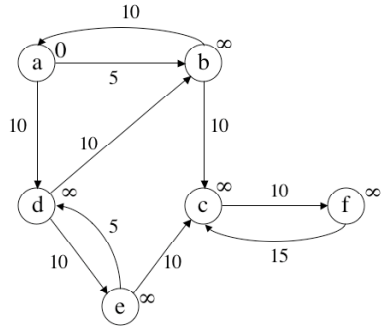
f. When a lesser distance is installed, the *predecessor* of the node is updated too, to reflect the shorter path. This will enable easy reconstruction of the paths when all nodes have been retired.

g. Assume that the test cases will consist of a large number of nodes and relatively sparse connections. So creating an n-by-n weight matrix and running the Floyd/Warshall algorithm is not to be recommended. That is an $O(n^3)$ algorithm vs. $O(n^2)$ for Dijkstra's, and it does n-times the needed work of this problem.

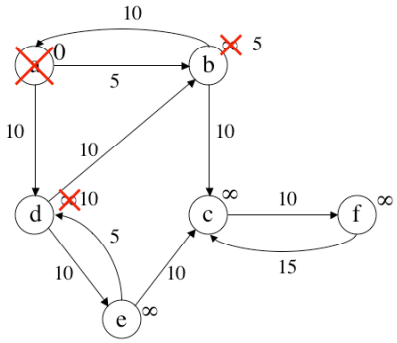
h. For a development environment, I prefer Netbeans, which is available free for all platforms. Eclipse is another one that people use. My sources will come with a Netbeans project file. If you use Eclipse, you are on your own.

Dijkstra's algorithm on the given graph example:
X'ed nodes are retired. X'ed numbers are distance updates.

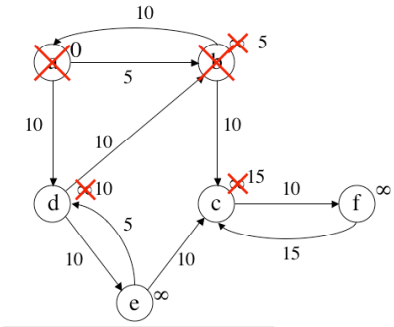
1.
Original node distance assignments (source is node a)



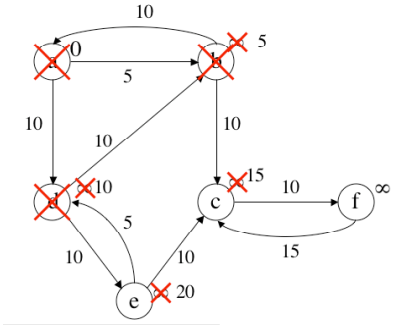
2.
Retiring minimum distance active node: a



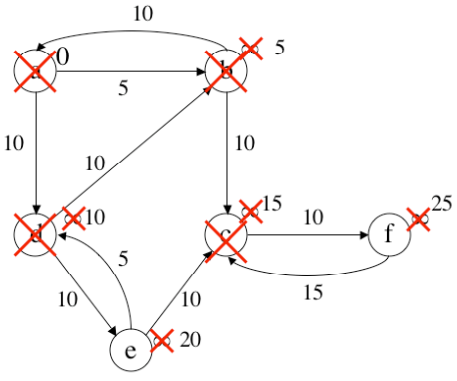
3.
Retiring minimum distance active node: b



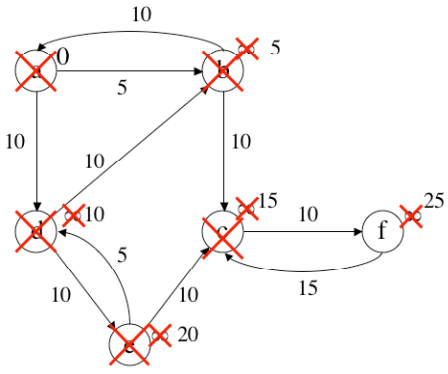
4.
Retiring minimum distance active node: d



5.
Retiring minimum distance active node: c



6.
Retiring minimum distance active node: e

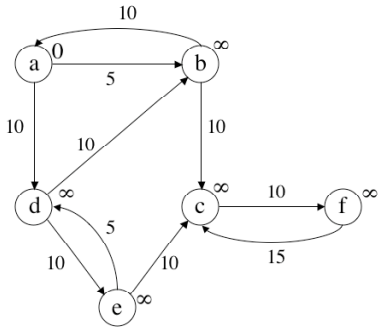


The final step is not shown, for reasons of brevity.

Dijkstra's algorithm showing predecessor references (dashed arrows):

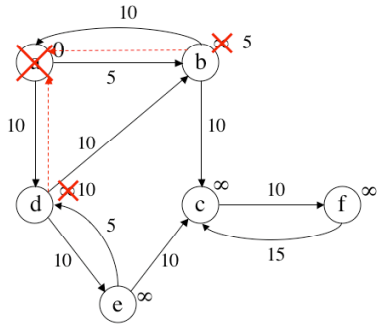
1.

Original node distance assignments (source is node a)



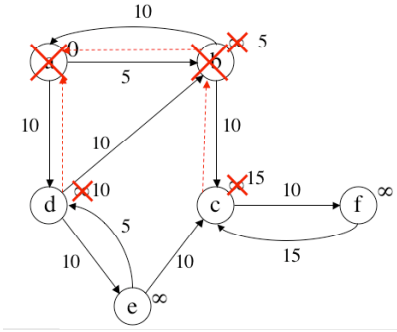
2.

Retiring minimum distance active node: a



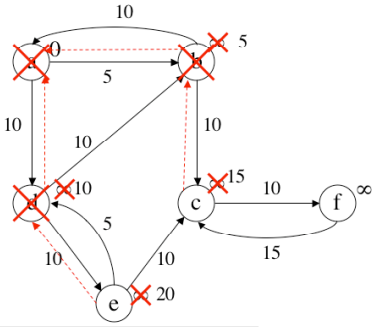
3.

Retiring minimum distance active node: b



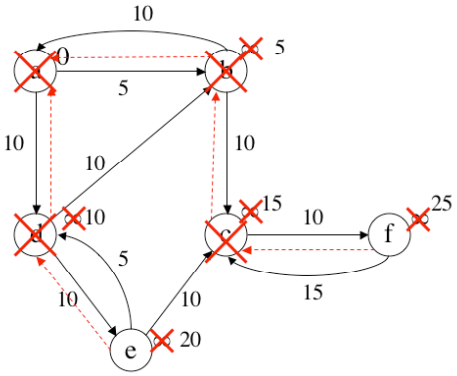
4.

Retiring minimum distance active node: d



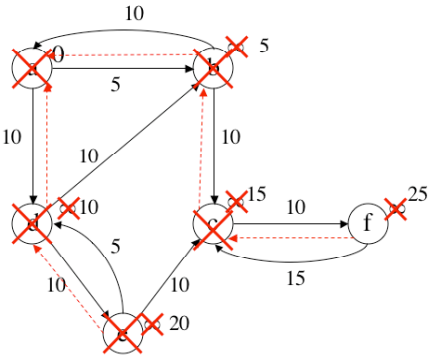
5.

Retiring minimum distance active node: c



6.

Retiring minimum distance active node: e



The final step is not shown, for reasons of brevity.