Harvey Mudd College
Computer Science 65
Fall 2008

Assignment 3
Tail Recursion, Dijkstra's Algorithm
Due. 11:59 p.m., Wed., 23 Sept. 2008

All problems in this assignment are to be done in a purely functional style using the Scheme language. The problems total 80, and an additional 10 points are available for style and documentation and 10 points for efficiency (not doing anything totally wasteful computationally, such as re-traversing lists unnecessarily). Submit a single file named hw03.scm containing all solutions. As always, document your functions clearly. **Be sure to spell each function name exactly as given**, otherwise the automatic grading script might not give you credit for the function.

1. [10 points] Construct a *tail-recursive* (as defined in the lecture) version of a Fibonacci number generator **`my-fib`** that, with argument N, outputs the N*th* element in the Fibonacci sequence: 1, 1, 2, 3, 5, 8, 13, 21, ... in which the first two elements are 1, and each successive term is the sum of the previous two. Test sample:

```
(test (map my-fib '(0 1 2 3 4 5  6  7  8  9 10))
                  '(1 1 2 3 5 8 13 21 34 55 89))
```

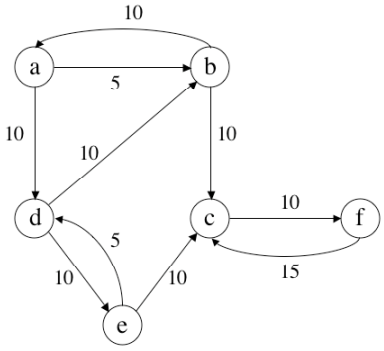
2. [10 points] Construct a *tail-recursive* version of the function **`find-max`**, which expects a list of reals as an argument. It returns a list consisting of the largest number in the argument list, followed by the index at which that number occurs. If the largest number occurs more than once, the lowest index is what is returned. If the list is empty, the pair `(-inf.0 -1)` is returned. (`-inf.0` is the Scheme representation for negative infinity floating point.) Test samples:

```
(test (find-max '(1.5))                    '(1.5 0))
(test (find-max '(2 4 6 7 3 1 5))          '(7 3))
(test (find-max '(1.5 2.7 -3 4.9 1.9))     '(4.9 3))
(test (find-max '(1.5 2.7 -3 4.9 1.9 4.9)) '(4.9 3))
(test (find-max ())                        '(-inf.0 -1))
```

3. [60 points] For this problem, you are not required to implement a purely-functional solution, although that is still probably the easiest approach. If you elect not to, you may use Scheme *vectors*, for example, which are modifiable. You can read about them in the Scheme manual.

   Implement function **`shortest-paths`** that computes the shortest paths from a specified source node to each node in a directed graph having non-negative distances associated with each arc. For this problem, a graph is a list of nodes. Each node is a list beginning with a *name*, which is a symbol. The rest of the list is an a*ssociation list* that specifies the nodes to which this node is directly (by a single arc) connected, together with a *direct distance* for that connection. An example graph is shown below:

```
(define graph1 '(
  (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, as in this example. In any case, the direct distance is not necessarily the shortest directed distance from the source to other nodes. There may be paths that go through several nodes that are shorter than the given direct distance.

The function **`shortest-paths`** has two arguments, a source node and a graph, such as the one above. It returns items consisting of the distances to each reachable node, followed by the node itself and the predecessor on the shortest path from the source to that node, if there is one:

(*total length of path* *final node on path* *predecessor on path)*

If there is no predecessor, then by convention, the final node itself is used, to keep things uniform. Note that the entire path can be constructed iteratively from the result information, if desired. Here is an example for the graph shown:

```
(test (shortest-path 'a graph1)
    '(( 0 a a)
      ( 5 b a)
      (10 d a)
      (15 c b)
      (20 e d)
      (25 f c)))
```

If a node is not reachable from the start, it does not appear in the list of paths.

Suggestions and hints for solving this problem:

a. Dijkstra's algorithm, as explained in the lecture and in c-e below, is highly recommended. Diagrams of Dijkstra's algorithm in action are included below.

b. The recursive implementation of 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. (This can be justified, but might not be totally obvious.)

c. Create a new list of "wrapped" nodes that also contains the shortest distance found so far, as well as other important information.

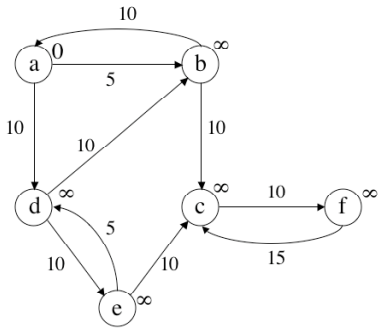
d. On each major step, the active node with the minimum distance is retired and the best estimates of minimum distances to the remaining active nodes are updated for posterity, to reflect the possibility of a path from the newly-retired node to the remaining active nodes. (Updating requires changing the distance component of the active nodes, but not the nodes themselves.) This is the essence of Dijkstra's algorithm. (When a lesser distance is installed, the predecessor of the node is updated too, to reflect the shorter path.)

e. Maintain two lists of wrapped nodes: The list of "retired" nodes and a list of "active" nodes. All nodes start on the active list. Reachable nodes move to the retired list, never to return.

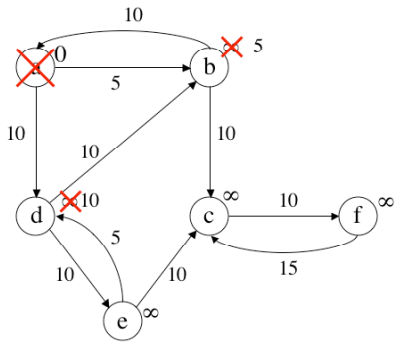
f. One way to find the minimum of a list is to sort the items and choose the first. This is not optimal, but it will be acceptable for this problem. To use the sort function built into Scheme (which is advised), you will need to specify an appropriate comparison function for wrapped nodes.

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

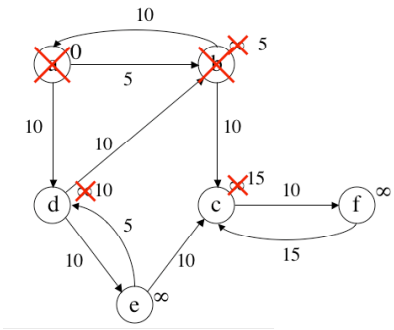
1.
Original node distance assignments (source is node a)



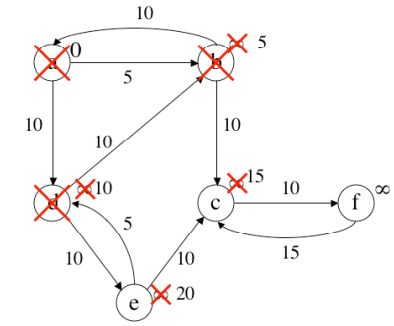
2.
Retiring minimum distance active node: a



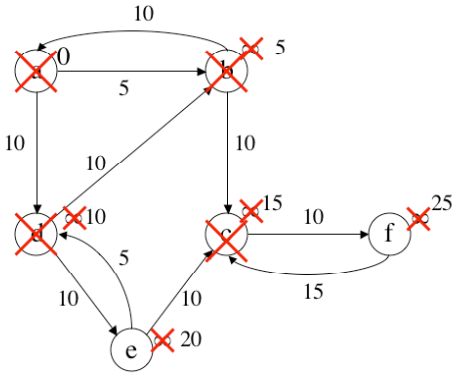
3.
Retiring minimum distance active node: b



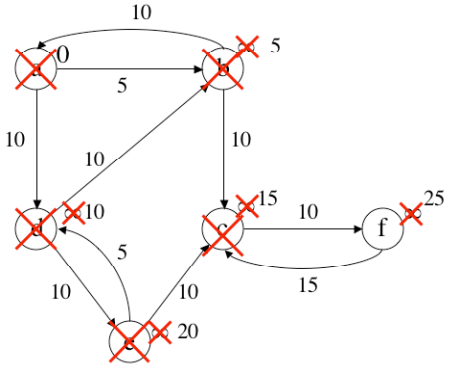
4.
Retiring minimum distance active node: d



5.
Retiring minimum distance active node: c



6.
Retiring minimum distance active node: e

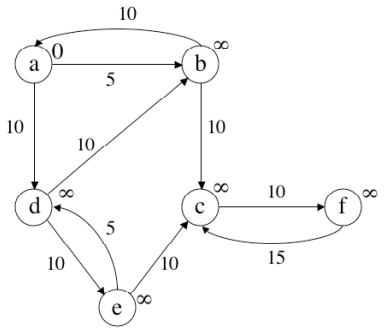


The final step is not shown, for reasons of brevity.

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

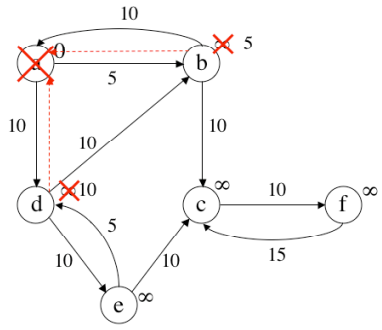
1.

Original node distance assignments (source is node a)



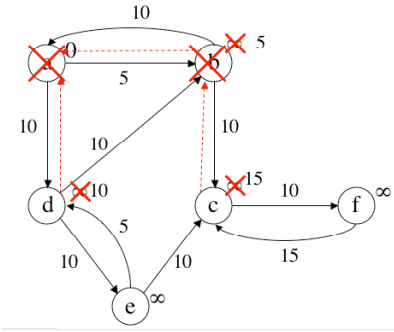
2.

Retiring minimum distance active node: a



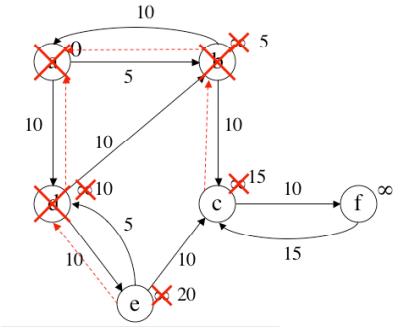
3.

Retiring minimum distance active node: b



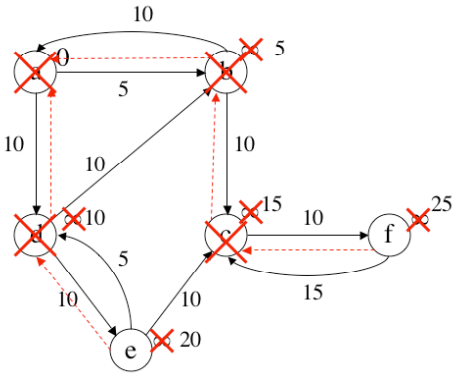
4.

Retiring minimum distance active node: d



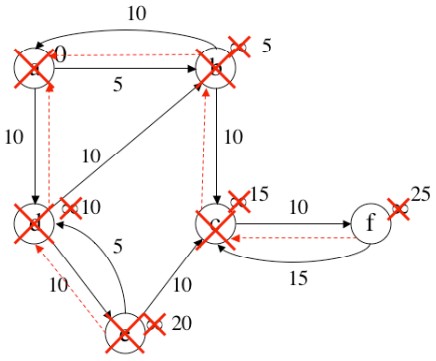
5.

Retiring minimum distance active node: c



6.

Retiring minimum distance active node: e



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