Dijkstra's Algorithm

Given a graph G with positive weights on the edges, and a particular vertex A, suppose we want to find the shortest paths (more precisely, the *least weight* paths) from A to all other vertices of G (note that if G is not connected, some vertices may not be reachable from A).

Dijkstra's Algorithm works by building a set **Reached** of vertices for which the shortest route from A has been found. Initially Reached is empty. On each iteration one more vertex is added to Reached until no more can be added (either because all vertices have been added or because some vertices cannot be reached from A). The algorithm also maintains a set **Candidates** of vertices for which some path from A has been found (but not necessarily the best path). Initially Candidates contains only the vertex A.

The algorithm maintains an array **Cost** where Cost[x] is the total weight of the best path seen so far that joins A to x. Initially Cost[A] = 0 and Cost[x] = ∞ for all vertices except A because we haven't found paths to any other vertices yet. We also keep an array **Pred**, where Pred[x] is the last vertex before x on the current best path from A to x. Initially, Pred[x] = "-" for all vertices x.

On each iteration, the algorithm chooses the vertex x in Candidates such that Cost[x] is minimal. (On the first iteration this will be A. On the second iteration, it will be one of A's neighbours, etc.) Vertex x is moved from Candidates to Reached (so x can never be chosen again). Then for each neighbour y of x that is not yet in Reached, the algorithm determines if the weight of the path from A to x, plus the weight of the edge from x to y, is less than Cost[y]. If it is, Cost[y] is updated to show that we have found a new and cheaper way to reach vertex y. If Cost[y] was ∞ prior to this change, y is added to Candidates.

It may not be obvious that this will find the optimal solution, but this is provable. The crux of the proof is this: when we choose vertex x because Cost[x] is $\leq$ all the other current Cost[y] values, we know that any as-yet-undiscovered path to x would have to go through one of the other candidates – but going through any of those vertices already costs more than going directly to x, so the direct connection to x must be the least-cost way to get there. Thus it is safe to move x to the Reached set.

This algorithm requires a certain amount of data management to make it efficient. On each iteration we need to do some amount of updating, and we need to choose the vertex x in Candidates with the lowest Cost[x] value. Please refer to Prim's MST algorithm from CISC-235 for a discussion of the implementation options – they are exactly the same for this algorithm.

In the pseudo-code given here, let w(x,y) be the weight on the edge joining vertices x and y. We will assume all vertices are identified by letters

```
# Cost(x) will be used to keep track of the cost of reaching
# vertex x

Cost(A) = 0
Cost(v) = ∞  for all v != A       # initially we have not found any paths
                                    #  from A to any other vertex
# Pred(x) will be used to keep track the predecessor of x
Pred(v) = ""        for all v
# create a set Reached to contain the vertices for which we have already
# found the shortest paths. This makes sure we only process each vertex once.
Reached = {}            # initially, we have found the road to nowhere
                         # (Hey!  Hey!)
# create a set Candidates to contain the vertices that are candidates for
# selection
Candidates = {A}          # Candidates can be implemented in a
                            # variety of ways (a heap is a popular choice)
while Candidates != {}  # keep going as long as there are candidates
       let x be the vertex in Candidates with minimum Cost value
       add x to Reached, and remove x from Candidates
       for each vertex y such that y is a neighbour of x
                                          and y is not in Reached:
          if Cost(x) + w(x,y) < Cost(y):    # we have now found a better
                                              # path from A to y
             if Cost(y) == ∞:          # if we haven't seen y before
                  add y to Candidates
             Cost(y) = Cost(x) + w(x,y)     # update our information about
                                              # the best path from A to y
             Pred(y) = x        # make note of the fact that on this path,
                                  #    y's predecessor is x

return Cost, Pred
```