Problem Set 9 Solution

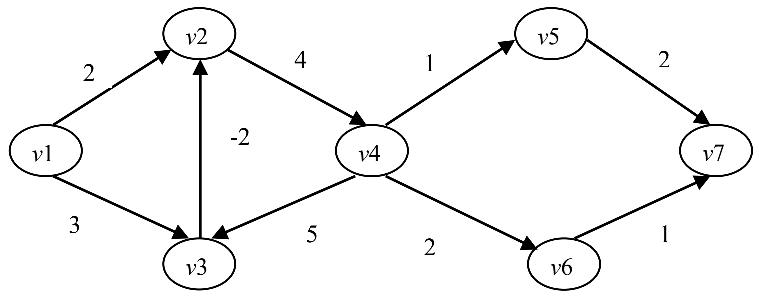
Q1 Show how to modify Dijkstra's algorithm to not only output the distance from v to each vertex in G, but also to output a tree T rooted at v such that the path in T from v to a vertex u is a shortest path in G from v to u.

Solution: For each vertex u, we introduce a variable closest[u] that stores its adjacent vertex in the cloud. In order to construct the tree, we make the following two changes to Dijkstra's algorithm:

1. Whenever a vertex u (u≠v) with the minimum D(u) is removed from the priority queue Q, add the edge (closest[u], u) to the tree T.
2. if (D[u] + w((u,z)) < D[z]) holds, set closest[z] to u.

```
Algorithm DijkstraDistances(G, v)
  Input: A simple undirected weighted graph G with nonnegative edge weights,
 and a distinguished vertex s.
  Output: A label D[u], for each vertex u, such that D[u] is the length of a
 shortest path from v to u in G.
 { for each  u є G  do
       if  ( u = v )
            D[u] = 0;
       else
            D[u] = +∞;
     Create a priority queue Q containing all the vertices of G using the D labels
 as keys;
     Create an empty tree T;
     while  Q is not empty do
        {
          u = Q.removeMin();
          if ( u!=v )
              add the edge (closest[u], u) to T;
         for each  z є Q such that  z is adjacent to u do
              if  ( D[u] + w((u,z)) < D[z] )  // relax edge e
                 {  D[z] = D[u] + w((u,z));
                    Change to D[z]  the key of vertex z in Q;
                    closest[z]=u;
                 }
        }
     return T and the label D[u] of each vertex u;
 }
```

Q2 Simulate the execution of Dijkstra's algorithm to find the shortest path from *v*1 to *v*7 in the directed graph shown below.



Is there something going wrong? Explain.

Solution: Let's simulate the execution of Dijkstra's algorithm in the given graph with a negative weight:

Step 0: D(*v*1) = 0, d(*v*2) = d(*v*3) = . . .= d(*v*7) = +∞. cloud ={}, Q={*v*1, *v*2, …, *v*7}
Step 1: Remove *v*1 from Q. cloud ={*v*1}, Q={*v*2, *v*3, …, *v*7}, d(*v*2) = 2, d(*v*3) = 3, d(*v*4) = d(*v*5) = . . .= d(*v*7) = +∞.
Step 2: Remove *v*2 from Q. cloud ={*v*1, *v*2}, Q={*v*3, *v*4, …, *v*7}, d(*v*3) = 3, d(*v*4) = 6, d(*v*5) = d(*v*6) = d(*v*7) = +∞.
Step 3: Remove *v*3 from Q. cloud ={*v*1, *v*2, *v*3}, Q={*v*4, *v*5, …, *v*7}, d(*v*4) = 6, d(*v*5) = d(*v*6) = d(*v*7) = +∞.
Now Dijkstra's algorithm goes wrong here. In step 3, since *v*2 is already in the cloud, d(*v*4) is not modified. As a result, the shortest path *v*1->*v*3->*v*2 is missed out. The shortest path *v*1->*v*2->*v*4->*v*5->*v*7 found by Dijkstra's algorithm is wrong. The real shortest path is *v*1-> *v*3->*v*2->*v*4->*v*5->*v*7.

Q3 Bob loves foreign languages and wants to plan his course schedule for the following years. He is interested in the following nine language courses: LA15, LA16, LA22, LA31, LA32, LA126, LA127, LA141, and LA169. The course prerequisites are:

- LA15: (none)
- LA16: LA15
- LA22: (none)
- LA31: LA15
- LA32: LA16, LA31
- LA126: LA22, LA32
- LA127: LA16
- LA141: LA22, LA16
- LA169: LA32

Find the sequence of courses that allows Bob to satisfy all the prerequisites. Describe your method briefly.

Solution: Topological sorting algorithm can help us solve this problem.

- Build a digraph to represent the course prerequisite requirements. The nine courses are vertices in the digraph. There will be edges from the prerequisites vertices to a course vertex according to the given information.
- Apply the topological sorting algorithm on this digraph. The result is one possible sequence of courses for Bob.

For example, one possible solution is LA15 -> LA16 -> LA22 -> LA31 -> LA32 -> LA126 -> LA127 -> LA141 -> LA169. Another solution is LA15 -> LA16 -> LA127 -> LA31 -> LA32-> LA169 -> LA22 -> LA126 -> LA141. The solution is not unique.

Q4 Tamarindo University and many other schools worldwide are doing a joint project on multimedia. A computer network is built to connect these schools using communication links that form a tree. The schools decide to install a file server at one of them to share data. Since the transmission time on a link is dominated by the link setup and synchronization, the cost of a data transfer is proportional to the number of links used. Hence, it is desirable to choose a "central" location for the file server. Given a tree *T* and a node *v* of *T*, the *eccentricity* of *v* is the length of a longest path from *v* to any other node of *T*. A node of *T* with minimum eccentricity is called a *centre* of *T*.
1. Design an efficient algorithm that given an *n*-node tree *T*, computes a centre of *T*.
2. Is the centre unique? If not, how many distinct centres can a tree have?
Hint: Consider a tree *T* and the tree *T0* produced by pruning the leaves of *T*. What is the relation between the centres of *T* and *T0*?

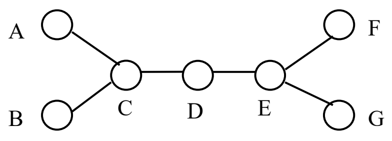
Solution:

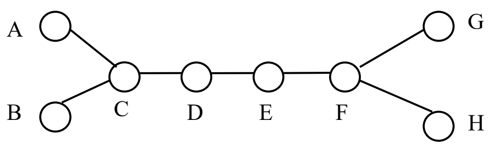
```
1. Make a copy T′ of T.
2. $k$=0.
3. Repeat the following until T′ has only one node or two nodes:
    a. Let S be a set of all the leaf nodes of T′.
    b. Remove each leaf node in S and its incident edge.
    c. $k$=$k$+1.
4. If T′ has only one node, that is the centre of T. The eccentricity of the centre
   node is $k$. If T′ has two nodes, either can be the centre of T. The eccentricity of
   the centre node is $k + 1$.
```

Time complexity analysis: Assume that the tree is represented in an adjacency list structure. In a tree, we have m=n-1, where m and n are the number of edges and the number of nodes of the tree, respectively. The time complexity of each step is shown as follows:
1. O(n)
2. O(1)
3. O(n)
4. O(1)

Therefore, the time complexity of this algorithm is O(n).

2. No! Not always unique. It's possible that the remaining tree has two nodes. We don't like to remove the leaves of a two-node tree (there will be nothing left!). You can try the following two trees. The centre of the first tree is *D* with *eccentricity* 2. The centre of the second tree is either *D* or *E* with *eccentricity* 3.



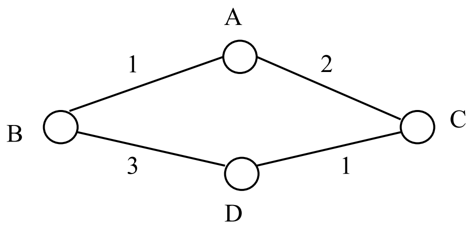


Q5 Consider the following greedy strategy for finding a shortest path from vertex *start* to vertex *goal* in a given connected graph.

1. Initialize *path* to *start*.
2. Initialize *VisitedVertices* to *{start}*.
3. If *start=goal*, return *path* and exit. Otherwise, continue.
4. Find the edge (*start, v*) of minimum weight such that *v* is adjacent to *start* and *v* is not in *VisitedVertices*.
5. Add *v* to *path*.
6. Add *v* to *VisitedVertices*.
7. Set *start* equal to *v* and go to step 3.

Does this greedy strategy always find a shortest path from *start* to *goal*? Either explain intuitively why it works, or give a counter-example.

Solution: The greedy algorithm presented in this problem is not guaranteed to find the shortest path between vertices in graph. This can be seen by counterexample. Consider the following weighted graph:



Suppose we wish to find the shortest path from *start* = *A* to *goal* = *D* and we make use of the proposed greedy strategy. Starting at *A* and with *path* initialized to *A*, the algorithm will next place *B* in *path* (because (*A, B*) is the minimum cost edge incident on *A*) and set *start* equal to *B*. The lightest edge incident on *start* = *B* which has not yet been traversed is (*B, D*). As a result, *D* will be appended to *path* and *start* will be assigned *D*. At this point the algorithm will terminate (since *start* = *goal* = *D*). In the end we have that *path* = *A,B,D*, which clearly is not the shortest path from *A* to *D*. (Note: in fact the algorithm is not guaranteed to find any path between two given nodes since it makes no provisions for backing up. In an appropriate situation, it will fail to halt.)

Q6 The time delay of a long distance call can be determined by multiplying a small fixed constant by the number of communication links on the telephone network between the caller and the callee. Suppose the telephone network of a company named Delstra is a tree. The engineers of Delstra want to compute the maximum possible time delay that may be experienced in a long-distance call. Given a tree, the diameter of T is the length (the number of edges) of a longest path between two nodes of T. Give an efficient algorithm for computing the diameter of T.

Solution: The algorithm is almost the same as that for Q4.

1. Make a copy T′ of T.
2. *k*=0.
3. Repeat the following until T′ has only one node or two nodes:
    d. Let S be a set of all the leaf nodes of T′.
    e. Remove each leaf node in S and its incident edge.
    f. *k*=*k*+1.
4. If T′ has only one node, the diameter of T is *2k*. If T′ has two nodes, the *eccentricity* of the centre node is 2*k* + 1.

The time complexity of this algorithm is O(n).

Q7 Design an efficient algorithm for finding a longest path from a vertex u and a vertex v of an acyclic directed graph G where each edge has a weight. Specify the graph representation used and any auxiliary data structures used. Also analyse the time complexity of your algorithm.

Solution: We can use breadth-first search to find a longest path from a vertex u to a vertex v in G. For each node, we introduce two variables: D(x) and prevVertex(x), where D(x) stores the longest path length from u to c, and prevVertex(x) stores the previous vertex of x in the longest path from u to x. prevVertex(x) is used to recover the longest path from u to v.

The breadth-first-based algorithm traverses the subgraph reachable from u in breadth-first order to compute D(x) for each vertex x reachable from u. The algorithm is shown in pseudo code as follows:

```
Algorithm longestPath(G, u, v)
Input: An edge-weighted digraph G, and two vertices u and v
Output: A longest path from u to v
   {
     for each vertex x ∈ G
        {
         setLabel(u, UNEXPLORED);
          D(x)=-∞; // D(x) stores the longest path length found so far
        }
     for all e ∈ G.edges()
         setLabel(e, UNEXPLORED);

     D(u)=0.
     Create an empty queue L;
      L.enqueue(u);
      setLabel(u, VISITED);
      x=u;
      while  ( ! L.isEmpty() and x!=v)
        {
          x = L.dequeue();
          for all  e ∈ G.ougoingEdges(x)
              if  ( getLabel(e) = UNEXPLORED )
                 { w = opposite(x,e);
```

```
                    if  ( getLabel(w) = UNEXPLORED )
                       {
                         setLabel(e, DISCOVERY);
                         setLabel(w, VISITED);
                         // Perform edge relaxation
                        if ( D(x)+w(x,w)>D(w) )
                           {
                             D(w)=D(x)+w(x,w);
                             // x is the previous vertex in the longest path
                             prevVertex(w)=x;
                      }
                         L.enqueue(w);
                       }
                    else
                       setLabel(e, CROSS);
                  }
             }
         if (x=v)  // Recover the longest path from u to v
             {
               Create an empty list L;
               while (x!u)
                  { Add x to the end of L;
                    x=prevVertex(x);
                  }
               Add u to the end of L;
               return L;
             }
        else
            return "No path from u to v";
       }
```

Time complexity: The initializations take O(m+n) time, where m and n are the number of edges and the number of vertices of the graph. The breadth-first search for computing D(x)'s for all the vertices reachable from u takes O(m+n) time. Recovering the longest path takes O(n) time. Therefore, this algorithm takes (m+n) time.