Homework Eleven

Q1 The BFS (Breadth-First Search) algorithm given in the lecture notes uses multiple lists. Modify the algorithm so that it uses only one queue to replace multiple lists.

Solution:

```
Algorithm BFS(G, s)
{
  Create an empty queue L;
  L.enqueue(s);
  setLabel(s, VISITED);
  while ( ! L.isEmpty() )
  {
    v = L.dequeue();
    for all  e ∈ G.incidentEdges(v)
        if ( getLabel(e) = UNEXPLORED )
        {   w = opposite(v,e);
            if ( getLabel(w) = UNEXPLORED )
            {
                setLabel(e, DISCOVERY);
                setLabel(w, VISITED);
                L.enqueue(w);
            }
            else
                setLabel(e, CROSS);
        }
  }
}
```

Q2 Describe, in pseudo code, an O(n+m)-time algorithm for computing all the connected components of an undirected graph G with n vertices and m edges.

Solution:

```
Algorithm connectedComponents(G)
  Input: An undirected graph G
  Output: All the connected components of G
 {
   for each vertex v in G do
       visited(v)=0;
   i=0;
   for each vertex v in G do
     {
       if ( visited(v)=0 )
         {
           create a new list Li ;  // Each Li stores a connected component
           perform the depth-first search or the width-first search starting with v;
```

```
            for each vertex u visited in the search do
               { add u to L_i;
                 visited(u)=1;
               }
            i=i+1;
          }
       }
   }
```

Q3 Given an undirected graph G and a vertex v_i, describe an algorithm for finding the shortest paths from v_i to all other vertices. The shortest path from a vertex v_s to a vertex v_t is a path from a vertex v_s to a vertex v_t with the minimum number of edges. What is the running time of your algorithm?

Solution: For each vertex v we introduce a list Q(v) to store the shortest path from v_i to v. We can modify the breadth–first search algorithm given in Q1 to compute the shortest path from v_i to v as follows:

```
Algorithm BFS(G, v_i)
 {
   for each vertex v of G do
       Create an empty list Q(v);
   Create an empty queue L;
   L.enqueue(v_i);
   setLabel(v_i, VISITED);
   while  ( ! L.isEmpty() )
    {
      v = L.dequeue();
       for all  e ∈ G.incidentEdges(v)
                   if  ( getLabel(e) = UNEXPLORED )
                     {  w = opposite(v,e);
                        if  ( getLabel(w) = UNEXPLORED )
                            {
                               setLabel(e, DISCOVERY);
                               setLabel(w, VISITED);
                               L.enqueue(w);
                              Q(w)=Q(s)+{v,w};
                            }
                        else
                              setLabel(e, CROSS);
                     }
    }
  }
```

The first for loop takes O(n) time, and “Q(w)=Q(s)+{v,w}” takes O(1) time. Hence, the running time of the algorithm is O(m+n).

Q4 A connected undirected graph is said to be biconnected if it contains no vertex whose removal would divide G into two pr more connected components. Give an

O(n+m)-time algorithm for adding at most n edges to a connected graph G, with n>3 vertices and m>n-1 edges, to guarantee that G is biconnected.

Solution: Number the vertices 0 to n − 1. Now add an edge from vertex i to vertex (i + 1) **mod** n, if that edge does not already exist. This connects all the vertices in a cycle, which is itself biconnected.

Q5 An n-vertex directed acyclic graph G is **compact** if there is some way of numbering the vertices of G with the integers from 0 to n-1 such that G contains the edge (i, j) if and only if i<j, for all i, j in [0, n-1]. Give an $O(n^2)$-time algorithm for detecting if G is compact.

Solution:

Algorithm compactGraphChecking(G)
  Input: A directed graph G
  Output: true if G is compact; or false
  {
    Perform topological sorting on G;
    Let TSN(v_i) be the topological number of vertex v_i.
    for each vertex v_i in G **do**
      TSN(v_i)= TSN(v_i)-1;
    Let a[0:n-1] be an array of all the vertices sorted in increasing order of their topological numbers;
    for i=0 **to** n-2 **do**
      for j=i+1 **to** n-1 **do**
        if no edge exists from a[i] to a[j]
          return false;
    return false;
  }

Running time analysis: Topological sorting on G takes O(n+m) time, where e the number of edges of G. The array a[] can be obtained by modifying the topological sorting algorithm without changing its time complexity. The first **for** loop takes O(n) time. The nested **for** loop takes $O(n^2)$ time. Therefore, the total running time is $O(n+m)+O(n)+O(n^2)=O(n^2)$.