Problem Set 8 Solutions

Problem 1 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:

The main algorithm remains the same. The breadth-first search algorithm starting with a vertex is modified as follows:

```
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);
                      }
    }
 }
```

Problem 2 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 Li;
               visited(u)=1;
             }
          i=i+1;
        }
    }
}
```

Problem 3 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, vi)
 {
   for each vertex v of G do
       Create an empty list Q(v);
   Create an empty queue L;
   L.enqueue(vi);
   setLabel(vi, 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(v)+{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).

Problem 4 A connected undirected graph is said to be biconnected if it contains no vertex whose removal would divide G into two or 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.

Time complexity analysis:
We can use an adjacency list to represent G.

1. It takes O(n) time to number all the n vertices.
2. Adding an edge from vertex i to vertex (i+1) mod n takes O(degree(i)) time, where degree(i) is the degree of vertex i. Since the total degree of all the vertices is 2m, where m is the number of edges in G. Therefore, the time complexity of adding all the edges is O(2m)=O(m).

As a result, the time complexity of this algorithm is O(n)+O(m)=O(n+m).

Problem 5 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(vi) be the topological number of vertex vi.
    for each vertex vi in G do
      TSN(vi)= TSN(vi)-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 true;
  }
```

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 $O(n+m)+O(n)+O(n^2)=O(n^2)$.

Problem 6 An Euler tour of a directed graph G with n vertices and m edges is a cycle that traverses each edge of G exactly once according to its direction. Such a tour always exists if G is connected and the in-degree equals to the out-degree for each vertex in G. Describe an O(n+m)-time algorithm for finding an Euler tour of such a directed graph.

Solution: Hierholzer proposed an efficient algorithm in 1873 which works as follows:

1. Arbitrarily select a starting vertex v.
2. Find a cycle C starting and ending at v such that C contains all the edges going into and out of v, and insert each edge of C into a doubly linked list L in the order of the cycle C.
3. Traverse L to find the first vertex v′ which has an outgoing unvisted edge. If such a vertex v′ is not found, the algorithm terminates. Otherwise, let (v′, v′′) be the next edge in L when v′ is found, and find a cycle C′ starting and ending at v such that C′ contains all the edges going into and out of v′, and insert each edge of C′ into L in the order of the cycle such that all the edges of C′ appear before (v′, v′′) in L. Repeat this step starting at the first edge of C′ in L.

This algorithm visits each vertex and each edge in O(1) time. Therefore, the time complexity is O(m+n), where m is the number of edges and n is the number of vertices in the graph. Since a directed graph with an Euler tour must be strongly connected, we have O(m+n)=O(m).

Problem 7 An independent set of an undirected graph G=(V, E) is a subset I of V such that no two vertices in I are adjacent. That is, if u and v are in I, then (u, v) is not in E. A maximal independent set is an independent set such that if any additional vertex is added to it, it will not be an independent set. Give an efficient algorithm for finding a maximal independent set for an undirected graph, and analyse its time complexity.

Solution: We can use a greedy algorithm to find a maximal independent set as follows:

1. Create an empty set maxSet.
2. Arbitrarily select a vertex v, and add v to maxSet.
3. Repeat the following until no vertex can be added to maxSet:
   - Find a vertex v′ that is in V and not adjacent to any vertex in maxSet, add v′ to maxSet.

Time complexity analysis: Assume that the adjacency matrix structure is used to represent the input graph. Adding one vertex to maxSet takes O(n) time. Notice that the maximum size of a maximal independent set is n, where n is the number of vertices in the graph. Therefore, the time complexity of this algorithm is O(n^2).

Comments: The maximum independent set problem is different from this problem, and NP-complete.