CS29003 ALGORITHMS LABORATORY
Assignment No: 10
Last Date of Submission: 28–Oct–2015

Let G be an undirected graph. The edges in G stand for communication links in a network. Assume that G is connected, that is, any node in the network can communicate with any other node in the network. However, it is possible that if a link e goes down, the network becomes disconnected, that is, some nodes are separated from some other nodes in the network. Links like e are the bottlenecks in the network. When you design a communication network, it is important to identify the bottleneck edges. This assignment aims at devising algorithms for this purpose. Let n denote the number of vertices in G, and m the number of edges in G. Name the vertices of G as $0, 1, 2, \ldots, n-1$.

Part 1 Write a function *getAdjList*() that converts a graph in the adjacency-matrix form to the adjacency-list form. Use a linked list (<u>not</u> a dynamically allocated array) of neighbors for each vertex.

Part 2 Write a function *lcsize*() to compute the size (that is, the number of vertices) of the largest connected component of G. Modify the DFS traversal algorithm to solve this problem. The running time of your function should be $O(n + m)$.

Part 3 Assume that G is connected. Write a function *bnefind*() to locate all bottleneck edges in G. Use the following strategy. For each edge e of G, call the function of Part 2 on the graph $G - e$. If $G - e$ contains multiple components, then e is a bottleneck edge; print e. Under the assumption that G is connected, the running time of this function is $O(m^2)$.

Part 4 Assume again that G is connected. Design an $O(m)$-time function *bnefindfast*() to identify all bottleneck edges in G. This algorithm is based on a modification of a *single* DFS traversal in the graph. Start the DFS at Vertex 0. The traversal produces a spanning tree (the DFS tree) T of G. The edges of T are called *tree edges*. Every other edge of G is a *back edge*. The back edges supply alternative connections between vertices and their proper ancestors in T. Let (u, v) be a tree edge (where v is a child of u). If the DFS subtree rooted at v contains no back edge to any proper ancestor of u, then the removal of (u, v) disconnects v from u, that is, (u, v) is a bottleneck edge.

For implementing this idea, number the vertices of G sequentially in the order they are visited during the DFS traversal. Moreover, for each node u, maintain a minimum of the sequential numbers of nodes v reachable from u along the following two types of paths: (1) v is a descendant of u, and the u-v path consists only of tree edges. (2) v is a proper ancestor of u such that for some descendant w of u (you may have $w = u$), the u-v path consists of tree edges from u to w, and a back edge from w to v. Update these minimum values at the nodes during the DFS traversal. Discover the bottleneck edges based upon these minimum values.

main()
- Read the number n of vertices in G from the user. Read and store the adjacency matrix M of G. Since G is undirected, read only the entries of M above the main diagonal. Call the function of Part 1 to convert M to the adjacency-list representation of G. Print the neighbors of each vertex using the adjacency list.
- Call the function of Part 2 to compute and print the size of the largest connected component of G. If G is not connected, exit.
- Call the function of Part 3 to locate and print all the bottleneck edges in G.
- Call the function of Part 4 to locate and print all the bottleneck edges in G.

Submit a single C/C++ source solving all the parts. Do not use any global/static variable/array.

Sample Output

```
n = 8

+++ Reading adjacency matrix
     0 0 1 0 0 0 0
       1 0 0 0 1 1
         0 0 0 1 1
           0 1 0 0
             0 1 0
               0 1
                 0

+++ Converting adjacency matrix to adjacency list

+++ Printing graph from adjacency list
    Neighbors of   0:  3
    Neighbors of   1:  2  6  7
    Neighbors of   2:  1  6  7
    Neighbors of   3:  0  5
    Neighbors of   4:  6
    Neighbors of   5:  3  7
    Neighbors of   6:  1  2  4
    Neighbors of   7:  1  2  5

+++ Finding the largest component size
    Component    1:    0    3    5    7    1    2    6    4
    The largest component has 8 nodes

+++ Finding bottleneck edges (Inefficient)
    (  0 , 3  ) is a bottleneck edge
    (  3 , 5  ) is a bottleneck edge
    (  4 , 6  ) is a bottleneck edge
    (  5 , 7  ) is a bottleneck edge

+++ Finding bottleneck edges (Efficient)
    (  6 , 4  ) is a bottleneck edge
    (  5 , 7  ) is a bottleneck edge
    (  3 , 5  ) is a bottleneck edge
    (  0 , 3  ) is a bottleneck edge
```