CS29003 ALGORITHMS LABORATORY
Assignment 12
Last Date of Submission: 11 – October – 2018

General Instruction

1. Please do not use any global variable unless you are explicitly instructed so.
2. Please use proper indentation in your code.
3. Please name your file as `<roll_no>_<assignment_no>`. For example, if your roll number is 14CS10001 and you are submitting assignment 3, then name your file as `14CS10001_3.c or 14CS10001_3.cpp` as applicable.
4. Please write your name and roll number at the beginning of your program.

There are N cities in Magicland. Some of the cities are connected by some roads. A road connects two cities and is bi-directional, i.e., we can go either way through a road. There is a path from a city i to every other city j, and it is possible to have multiple paths from one city to another city.

Once in a while, though, the cities remain closed and can not be used as an intermediate stop while going from any city i to other city j. For a traveller, who wants to go from city i to city j during that time, if he has a path without using the closed city, he is willing to take that but for certain pair of cities, there may not be any path without using this city. We call a city as 'critical junction' if closing of the city implies that there is no path from a certain city to another in Magicland.

The administration thus decides to make a list of 'critical junctions' for which they need to be extra careful and avoid frequent closing of these. They need your help in preparing this list.

Let us formulate this problem in graph theoretic terms. The cities are your nodes, and the roads are your edges. You have learned how to use graph traversals to find the number of connected components in a graph. Now, one possible algorithm to list the 'critical junctions' is to remove each of the node (city) from the graph one by one, check the number of connected components. If the number of components is more than one, this is a 'critical junction'. This algorithms though, is time consuming, and has complexity of $O(n(n + e))$, where n and e denote the number of vertices and edges in the graph, respectively. We want to come up with a faster algorithm. The idea is as follows which makes use of a single DFS and uses the 'back edges'.

First we run DFS over the graph G starting from a start vertex s. If G is connected, which is the case in the given scenario, the DFS tree thus obtained covers the entire set of vertices in G.

Now, if the root s has more than one child, this implies that there is no path for going from one child to another other than via the root. Thus, root is a 'critical junction'.

For any other internal node v, however, if there is a subtree rooted at a child of v that does not have a back edge to an ancestor of v, then removing v will increase the number of connected components, and v is a 'critical junction'.

Implement this idea using two arrays, $discovery[]$ and $low[]$. While $discovery[v]$ stores the discovery time of a vertex v (1 for the start vertex, and so on, in the order of discovery), $low[v]$ stores the minimum of the discovery times of any vertex reachable from v (note that you can not traverse tree edges in the opposite direction) via atmost one back edge. It is easy now to specify a condition for v being a 'critical junction'. This algorithm takes $O(n + e)$ time.

The following figures show a graph along with a DFS tree obtained while traversing from the vertex a. In the DFS tree, the tree edges are shown as directed, and the back edges are shown as undirected. For each node, we also show the two numbers, discover and low, for that vertex. For instance, for vertex c, $discovery[c] = 4$, which is the order in which it is discovered, and $low[c] = 1$, because this is the minimum discovery time of a vertex reachable (a) from c. Vertices d and g are the 'critical junctions'.

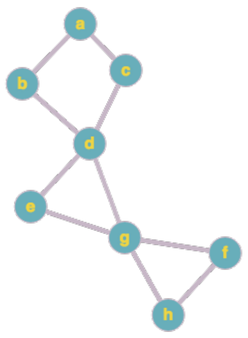

Figure 1: An example graph

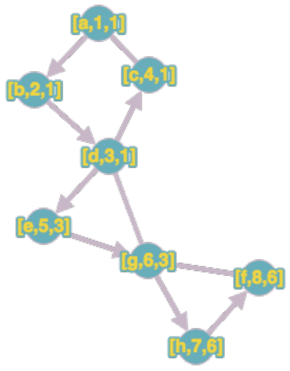

Figure 2: DFS tree of the graph

Part 1. Read the graph from the user

In this part, you will read the number n of vertices and e of edges from the user. The vertices are then numbered as $0, 1, \ldots, n-1$ and you will read the list of neighbors for each of the n vertices in that order. Store this information in the adjacency-list format. Print the adjacency list of the graph.

Use the following structure to store the graph in the adjacency list format. This is dynamically allocated from the user input. *For a given vertex, keep all the neighbors sorted as per their Ids in the adjacency list.*

```
typedef struct _node {
   int vno;              /* vertex Id */
   struct _node *next;   /* Pointer to the next node in the adjacency list */
} node;

typedef struct _vertex {
   node *adjlist; /*Pointer to the adjacency list for that vertex*/
   /*Feel free to add any other information you need for the vertex*/
} vertex;

typedef vertex *graph;   /* (Dynamic) array of list headers */
```

Part 2. Find the 'critical junctions' of the graph: A simple algorithm

In this part, you will use the simple algorithm, that is, remove each of the node from the graph one by one, check the number of connected components. If the number of components is more than one, this is a 'critical junction'.

Use the following function that takes the graph G as input, removes each of the vertices of the graph one by one, computes the number of connected components in the graph, and prints the number of connected components and the vertex if this count is more than 1.

```
void findCritical (graph G)
```

For each of the iteration, while you remove a vertex, you should call a modified DFS function with the removed vertex Id as an additional input, and the function should not start from this removed vertex, and ignore this whenever encountered.

```
int DFSComp (graph G, int rem)
```

The function $\mathrm{DFSComp}(G, x)$ will make a call to the recursive function $\mathrm{DFSVisit}(G, x, v)$ with a designated start vertex $v \neq x$ until all the vertices have been visited. It returns the number of connected components. Note that the removed vertex x should be ignored whenever encountered. The complexity of the algorithm is $O(n(n+e))$.

Part 3. Find the 'critical junctions' of the graph: More efficient way

In this part you will use the efficient algorithm mentioned earlier to find the 'critical junctions' of the graph, and output those as per the specified format.

Define the following function which takes the graph G populated as per user inputs and prints the critical junctions in the graph.

```
void findCriticalFast (graph G)
```

The complexity of the algorithm is $O(n+e)$.

Main Function

main() function does the following:

1. Reads n and e from the user.
2. Reads the number and list of neighbors for each of the vertices and creates the adjacency list representation of the graph. It then prints the adjacency list of the graph.
3. Calls the function $\mathrm{findCritical}()$ and prints the critical junctions of the graph and corresponding number of connected components.
4. Now calls the function $\mathrm{findCriticalFast}()$ and prints the critical junctions of the graph.

Sample Output

```
Enter the number of vertices and edges
8 10
Enter the neighbors for each of the vertex
Degree of Vertex 0: 2
Neighbors of 0: 2 1
Degree of Vertex 1: 2
Neighbors of 1: 0 3
```

```
Degree of Vertex 2: 2
Neighbors of 2: 0 3
Degree of Vertex 3: 4
Neighbors of 3: 2 1 4 5
Degree of Vertex 4: 2
Neighbors of 4: 3 5
Degree of Vertex 5: 4
Neighbors of 5: 3 4 6 7
Degree of Vertex 6: 2
Neighbors of 6: 5 7
Degree of Vertex 7: 2
Neighbors of 7: 5 6
Adjacency list of the graph
Vertex 0: 1 2
Vertex 1: 0 3
Vertex 2: 0 3
Vertex 3: 1 2 4 5
Vertex 4: 3 5
Vertex 5: 3 4 6 7
Vertex 6: 5 7
Vertex 7: 5 6
Critical junctions using the simple algorithm:
Vertex 3: 2 components
Vertex 5: 2 components
Critical junctions using the fast algorithm:
Vertex 3 is a critical junction
Vertex 5 is a critical junction
```