Lab 10

Due No Due Date **Points** 1

You are suggested to use the teaching servers burrow.soic.indiana.edu or hulk.soic.indiana.edu or tank.soic.indiana.edu for practicing C programs.

Lab 10: Depth First Search

For this lab, all students must have a good grasp on graph construction.

C++ vector

C++ standard vector works like a **dynamic** array where we can insert, delete, change value in the runtime. Vector would adjust the space it consumes automatically in the runtime, so taht we always allocate about the amount just needed. We don't need to care much about handling spacing issues that we have with traditional C array.
In C++ standard library, we can use iterator to access the values of a sequence container. A sample program is given below:

```
        #include <iostream>
        #include <vector>

        using namespace std;

        int main()
        {
                vector<int> vectorList;         // Creating a vector of int
                for (int i=1; i<10; i++)
                        vectorList.push_back(i);  // inserting values at the end of the vector

                cout << "vectorList contains:";
                for (vector<int>::iterator it = vectorList.begin() ; it != vectorList.end(); ++it) // itera
ting through the vector array
                        cout << ' ' << *it;
                cout << endl;

                return 0;
        }
```

For more information about C++ stardard vector, you can refer to the ducumentation for C++ standard library.

Representing Graph data structure with vector

Given 100 nodes and 30 edges, how can we maintain a data structure to hold the graph?
We can create a two dimensional 100x100 matrix array so that each cell holds either 1 or 0. matrix[i][j] == 1 if there is an edge between i and j, and 0 otherwise. The problem with this approach is, say there exist only 3 edges for a given node 4. To get the edges we will have loop through all 100 columns of matrix[4] to find that 3 edges which is time consuming.

Rather we can use array of vector<int> such as vector<int> graph[100], where graph[4] holds only 3 values, the 3 edges node 4 has.
For the bidirectional graph with nodes 0,1,2,...,9 and following edges below:
0-1
1-2
2-3
1-3
7-6
6-8
8-9
9-6
5-4
4-3

In this way we only store an edge when it exists. It reduces the running time when we want to find all the edges a node has provided that the gragh is not dense.

Here is a sample program that uses vector to hold all edges of the graph and shows the edges of each nodes to the output.

```
        #include <stdio.h>
        #include <stdlib.h>
        #include <string.h>
        #include <iostream>
        #include <vector>

        using namespace std;

        typedef struct edge_struct
        {
                int u;
                int v;
        } edge;

        int main()
        {
                int numEdge = 10;
                int u, v;
                vector<int> graph[50];

                edge listOfEdges[10] = {{0,1},{1,2},{2,3},{1,3},{7,6},{6,8},{8,9},{9,6},{5,4},{4,3}};

                for(int i = 0; i < numEdge; ++i)
                {
                        u = listOfEdges[i].u;
                        v = listOfEdges[i].v;
                        graph[u].push_back(v);  // For a birectional edge, we should insert both (u,v) and
 (v,u) as an edge
                        graph[v].push_back(u);
                }

                for(int i = 0; i < 50; ++i)
                {
                        if (graph[i].size() == 0)       // There is no edge for node i
                                continue;
                        cout << i << ": ";
                        for(vector<int>::iterator vi = graph[i].begin(); vi != graph[i].end(); ++vi)
                        {
                                cout << *vi << " ";
                        }
                        cout << endl;
                }

                return 0;
        }
```

Depth first search

We can use DFS for different tasks such as finding path, finding minimum path, topological sorting, finding cycle, finding whether two nodes are connected and so on. In this lab, we will use DFS to find whether two nodes of a graph are connected via a path. A sample pseudocode for such task is given below:

```
//To check whether node u and v are connected
Build Graph
Fill up array visited[100] with zeros
Call DFS(u)
Check for visited[v]

global array visited[100];
DFS(u)
{
        visited[u] = 1; // Mark this node as visited

        For all edge (u, x):
                if visited[x] == 0:  // only visit again if not visited previously
                        DFS(x)

}
```

There is also iterative implementation of DFS if you want to make the space consumption explicit instead of taking up space from the function call stack

```
DFS(u)
 {
        Let S be a stack
        S.push(u)
        visited[u] = 1
        while S is not empty:
               v = S.pop()
               For all edge (v, x):
                       if visited[x] == 0
                                visited[x] = 1
                                S.push(x)
 }
```

Build some graph and test if your algorithm gives the correct connectivity.