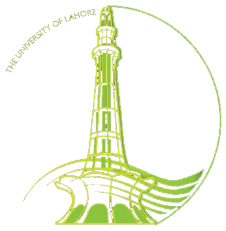

THE UNIVERSITY OF LAHORE
ISLAMABAD CAMPUS

Data Structure and Algorithms

Lab Report

Name:	Waqar Nawaz Khan
Registration #:	CSU-F16-115
Lab Report #:	08
Dated:	28-04-2018
Submitted To:	Mr. Usman Ahmed

The University of Lahore, Islamabad Campus
Department of Computer Science & Information Technology

Experiment # 08
Weighted And Directed Graph

Objective
The objectives of this lab session are to understand the concept of weighted and directed graph.

Software Tool
1. Dev C++

1 Theory

Weighted Graph: Weighted graph representation using STL is discussed. The implementation is for adjacency list representation of weighted graph. We use two STL containers to represent graph:

vector : A sequence container. Here we use it to store adjacency lists of all vertices. We use vertex number as index in this vector.
pair : A simple container to store pair of elements. Here we use it to store adjacent vertex number and weight of edge connecting to the adjacent.

2 Program

```
#include <bits/stdc++.h>
using namespace std;

void addEdge(vector <pair<int, int> > adj[], int u,
                                        int v, int wt)
{
    adj[u].push_back(make_pair(v, wt));
    adj[v].push_back(make_pair(u, wt));
}
```

```
void printGraph(vector<pair<int,int> > adj[], int V)
{
    int v, w;
    for (int u = 0; u < V; u++)
    {
        cout << "Node " << u << " makes an edge with \n";
        for (auto it = adj[u].begin(); it!=adj[u].end(); it++)
        {
            v = it->first;
            w = it->second;
            cout << "\tNode " << v << " with edge weight ="
                 << w << "\n";
        }
        cout << "\n";
    }
}

int main()
{
    int V = 5;
    vector<pair<int, int> > adj[V];
    addEdge(adj, 0, 1, 10);
    addEdge(adj, 0, 4, 20);
    addEdge(adj, 1, 2, 30);
    addEdge(adj, 1, 3, 40);
    addEdge(adj, 1, 4, 50);
    addEdge(adj, 2, 3, 60);
    addEdge(adj, 3, 4, 70);
    printGraph(adj, V);
    return 0;
}
```

Direested Graph: Depth First Traversal can be used to detect cycle in a Graph. DFS for a connected graph produces a tree. There is a cycle in a graph only if there is a back edge present in the graph. A back edge is an edge that is from a node to itself (selfloop) or one of its ancestor in the tree produced by DFS. In the following graph, there are 3 back edges, marked with cross sign.
For a disconnected graph, we get the DFS forrest as output. To detect cycle,

we can check for cycle in individual trees by checking back edges.

To detect a back edge, we can keep track of vertices currently in recursion stack of function for DFS traversal. If we reach a vertex that is already in the recursion stack, then there is a cycle in the tree. The edge that connects current vertex to the vertex in the recursion stack is back edge. We have used recStack[] array to keep track of vertices in the recursion stack.

3 Program

```
#include<iostream>
#include <list>
#include <limits.h>

using namespace std;

class Graph
{
    int V;
    list<int> *adj;
    bool isCyclicUtil(int v, bool visited[], bool *rs);
public:
    Graph(int V);
    void addEdge(int v, int w);
    bool isCyclic();
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v s list.
}
```

```
bool Graph::isCyclicUtil(int v, bool visited[], bool *recStack)
{
    if(visited[v] == false)
    {
        visited[v] = true;
        recStack[v] = true;
        list<int>::iterator i;
        for(i = adj[v].begin(); i != adj[v].end(); ++i)
        {
            if ( !visited[*i] && isCyclicUtil(*i, visited, recStack) )
                return true;
            else if (recStack[*i])
                return true;
        }

    }
    recStack[v] = false;  // remove the vertex from recursion stack
    return false;
}
bool Graph::isCyclic()
{
    bool *visited = new bool[V];
    bool *recStack = new bool[V];
    for(int i = 0; i < V; i++)
    {
        visited[i] = false;
        recStack[i] = false;
    }

    for(int i = 0; i < V; i++)
        if (isCyclicUtil(i, visited, recStack))
            return true;

    return false;
}

int main()
{
    Graph g(4);
    g.addEdge(0, 1);
```

```
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    if(g.isCyclic())
        cout << "Graph contains cycle";
    else
        cout << "Graph doesn't contain cycle";
    return 0;
}
```