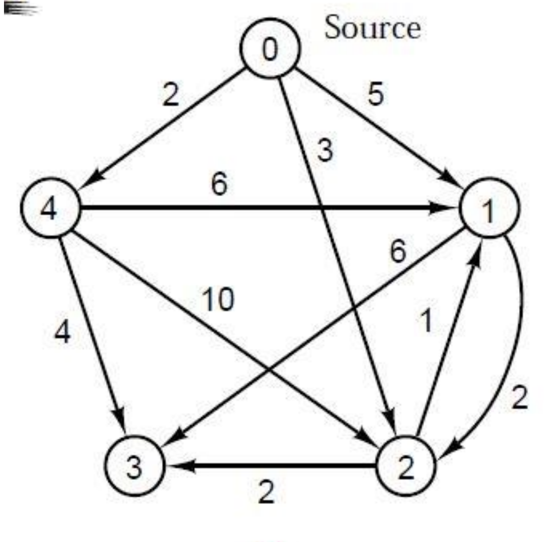
ESE 344 Software Techniques for Engineers,

Electrical and Computer Engg., Stony Brook University, Prof. Murali Subbarao

Project 4: GRAPHS: Adjacency Lists, Depth-First-Traversal, Breadth-first-traversal and Shortest Paths

A directed graph/network is given to be *connected* and it is specified as follows.
The first item is the number of vertices N and the second item is the number of edges NE. The vertices are labeled as 0,1,2,3,…, N-1. The subsequent items specify the edges, one edge data on each line. Each edge has the format "Vi Vj W" where Vi is the first vertex number, Vj is the second vertex number, and W is the weight of the edge or "distance from Vi to Vj". The edge direction is from Vi to Vj. Assume that there is no error in the input file.



(a)

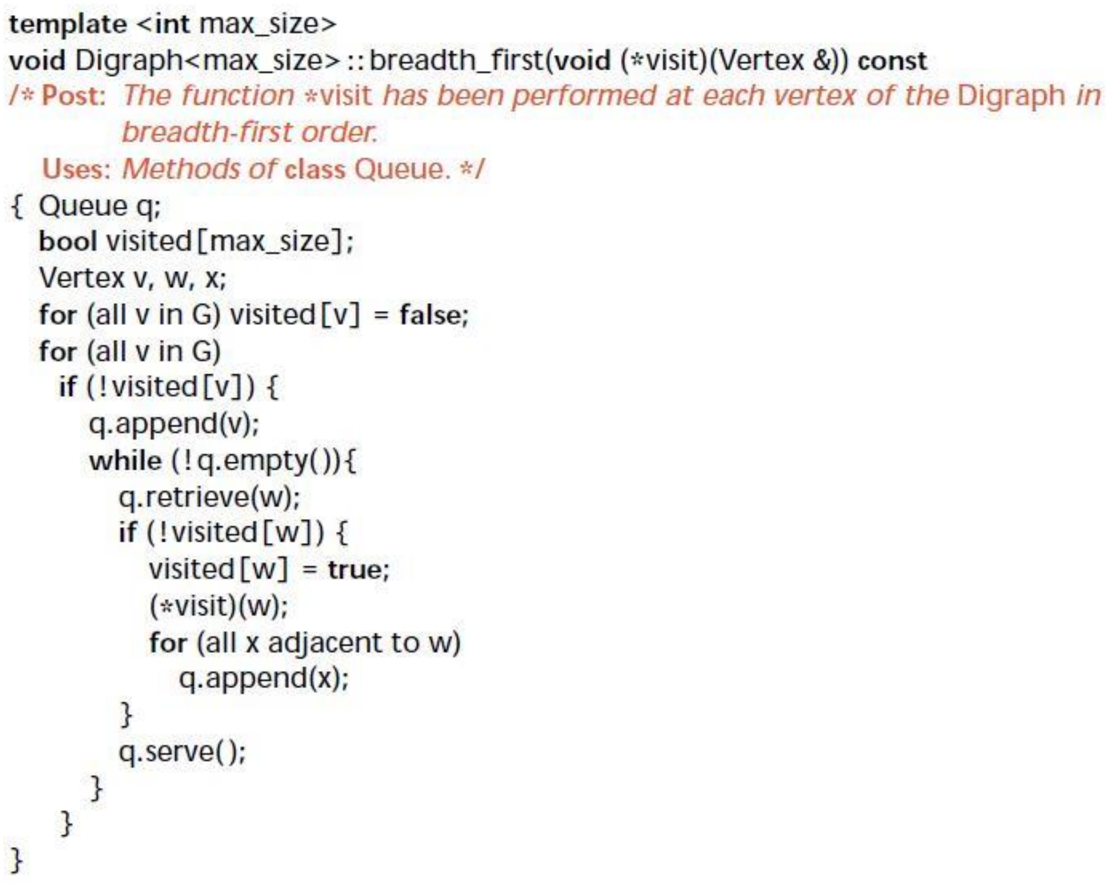
For example, the graph shown above (taken from Fig. 12.10(a) on page 585 in the Kruse & Ryba reference book) is specified by the input file:

```
5
10
0   4   2.0
0   2   3.0
0   1   5.0
4   1   6.0
4   3   4.0
4   2   10.0
2   3   2.0
2   1  1.0
1   3   6.0
1   2   2.0
```

A program that reads the input in the above format and builds an adjacency list representation of the graph is given to you below. You are also given the program for printing the graph and Depth-

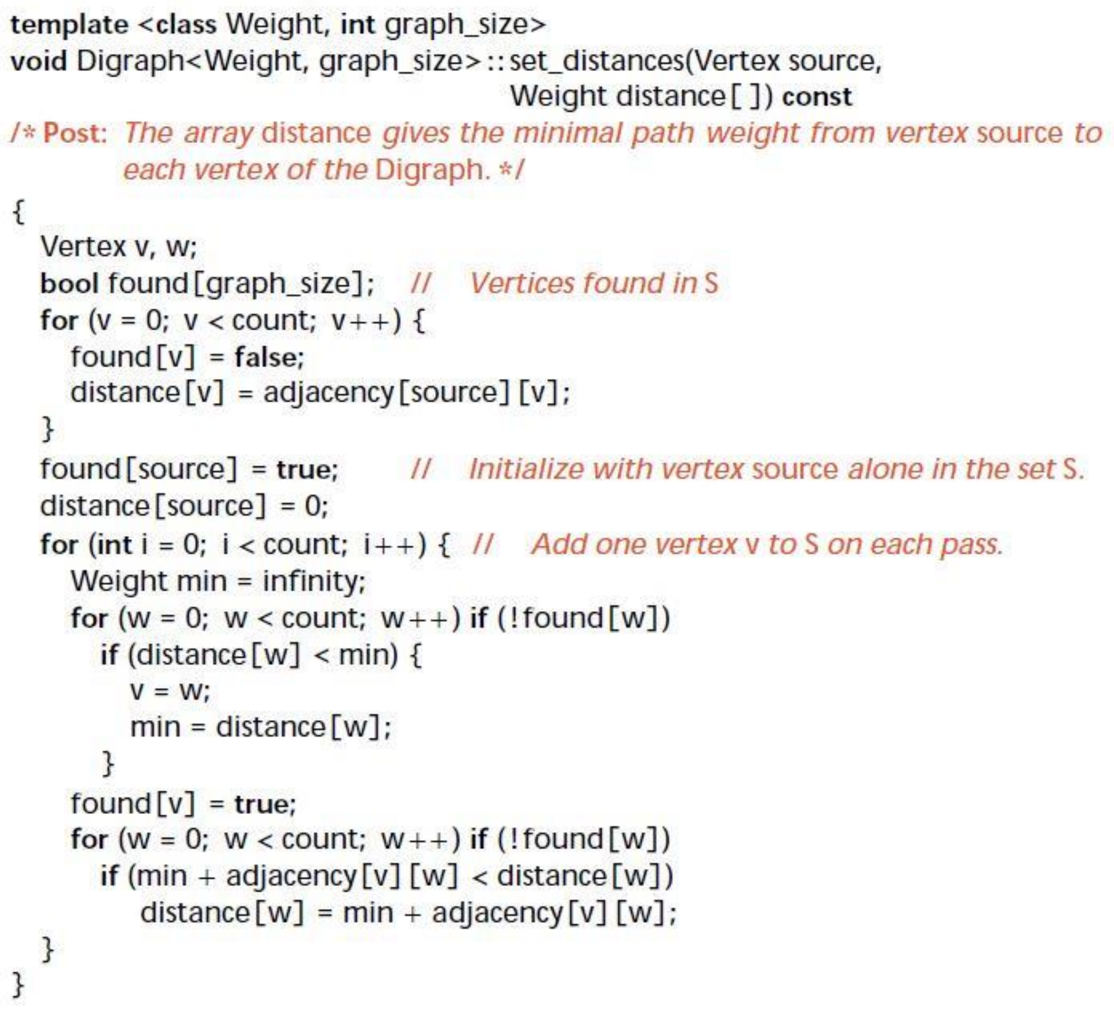
First-Traversal of the directed graph. Write C++ programs to complete the following. You are not permitted to use any class library from the Standard Template Library (STL) in C++ (except the vector class if you wish, but not the List or Queue or Stack, or …).

1. (5 points) Modify the main() function and write a function for breadth-first traversal of the graph. Implement the breadth-first-traversal algorithm by making changes and adding to the algorithm shown below (taken from page 578 in the Kruse & Ryba book). Print the vertices in breadth-first order starting from every vertex 0 to N-1. You should implement the queue data structure that you need using the List class provided to you. Run your algorithm on the given input graphs.



```
template <int max_size>
void Digraph<max_size> :: breadth_first(void (*visit)(Vertex &)) const
/* Post: The function *visit has been performed at each vertex of the Digraph in
         breadth-first order.
   Uses: Methods of class Queue. */
{ Queue q;
  bool visited[max_size];
  Vertex v, w, x;
  for (all v in G) visited[v] = false;
  for (all v in G)
    if (!visited[v]) {
      q.append(v);
      while (!q.empty()){
        q.retrieve(w);
        if (!visited[w]) {
          visited[w] = true;
          (*visit)(w);
          for (all x adjacent to w)
            q.append(x);
        }
        q.serve();
      }
    }
}
```

2. (10 points) Implement the Shortest Paths algorithm by making changes and adding to the algorithm on page 584-587 in the Kruse & Ryba book (shown below).



```
template <class Weight, int graph_size>
void Digraph<Weight, graph_size>::set_distances(Vertex source,
                                                 Weight distance[ ]) const
/* Post: The array distance gives the minimal path weight from vertex source to
         each vertex of the Digraph. */
{
   Vertex v, w;
   bool found[graph_size];     //   Vertices found in S
   for (v = 0; v < count; v++) {
      found[v] = false;
      distance[v] = adjacency[source][v];
   }
   found[source] = true;       //   Initialize with vertex source alone in the set S.
   distance[source] = 0;
   for (int i = 0; i < count; i++) {  //   Add one vertex v to S on each pass.
      Weight min = infinity;
      for (w = 0; w < count; w++) if (!found[w])
         if (distance[w] < min) {
            v = w;
            min = distance[w];
         }
      found[v] = true;
      for (w = 0; w < count; w++) if (!found[w])
         if (min + adjacency[v][w] < distance[w])
             distance[w] = min + adjacency[v][w];
   }
}
```

A. Print the shortest paths from vertex 0 to all other vertices. Note that, you should use the adjacency list implementation provided to you. You are not permitted to use the adjacency matrix representation that is used in the Kruse & Ryba book. Run your implementation on the given graph. For each vertex, you must print the length of the shortest path (i.e. distance[v] for vertex v), as well as the shortest path from the source vertex to that vertex; it is acceptable to print the path in reverse order.
B. Your implementation should be computationally efficient by possibly using some additional memory. For example, the code in the Kruse and Ryba book has two for loops that scan through all vertices to check if they are already found or not with the code

   "**for** (w = 0**;** w < count**;** w++) **if** (!found[w])"

   but that is not efficient (as explained in class, because the graph is given to be connected with just one component). Make appropriate changes to improve the computational efficiency. Run your implementation on the given graph starting from every vertex 0 to N-1 once and print the output. That is, you need to print the shortest path as well as its length from every vertex to every other vertex in the graph.
C. Basic implementation without computational efficiency: 6 points out of 10.

D. (2 points) Maintain a contiguous list L[w] implemented with a vector such that, after the shortest paths have been found for the first k vertices, found[L[w]]==true for 0<=w<k, and found[L[iw]]==false for k<=i<count. Then change the first for-loop: "**for** (w = 0**;** w < count**;** w++) **if** (!found[w])" to something like "**for** (w = k**;** w < count**;** w++) {// scan vertices L[w] instead of w since found[L[w]]==false.

E. (2 points) At any iteration, if S is the set of vertices for which the shortest paths have been found, and R is the set of remaining vertices for which the shortest paths have not been found yet. Then, for any vertex v in R that is connected to at least one vertex w in S, the distance[v] will be finite. You can maintain a list of all such vertices v with finite distance in a sorted list LS[k] such that distance[LS[k]]<=distance[LS[k+1]] for k=0 to LL-1 where LL is the length of the list LS[k]. First item LS[0] will have the minimum distance[LS[0]] and it can be found without a search. However, this list should be updated appropriately when the distance[] array is updated after LS[0] is added to the set S. Implement this scheme. Note that the distance array is updated for only those vertices w that are adjacent to v and found[w]==false, Therefore the original for-loop below

```
for (w = 0; w < count; w++) {
    if (!found[w]) {
        if (min+ adjacency[v][w] < distance[w]) {
            distance[w] = min+adjacency[v][w];
        }
    }
}
```

Is changed to something like:

```
for all w adjacent to v {
    if (!found[w]) {
        if (min+ adjacency[v][w] < distance[w]) {
            //delete w from LS[k] if it is present
            distance[w] = min+adjacency[v][w];
            // add w to LS[k] based on the updated distance[w]
        }
    }
}
```

The code above can be made even more efficient by first sorting all the vertices w adjacent to v whose distances are updated above, sorting based on the updated distance, and merging that list with LS[k]. This part is optional.

Hint for storing the shortest path: you can maintain a spanning tree formed by the shortest paths as follows. Create an integer array of parent[i] of size N for N vertices. Initialize each entry of the array to be the starting vertex s, i.e. parent[i]= -1 for all i= 0 to N-1.

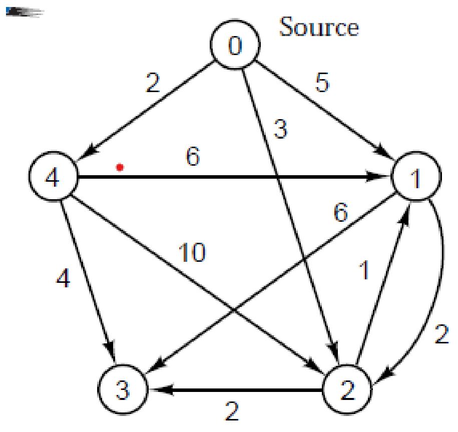
Then modify the loop in the Kruse & Ryba book as follows:

```
for (w = 0; w < count; w++) {
    if (!found[w]) {
        if (min+ adjacency[v][w] < distance[w]) {
            distance[w] = min+adjacency[v][w];
            parent[w]=v;
        }
    }
}
```

Now the path from each vertex v to starting vertex s can be traced backwards as follows

```
int k;
for(v=0;v<N;v++) {
   cout<<distance[v]<<endl;
   cout<<v<<”  “ ;
   k=v;
   while((k!=s) && (k != -1))  {
           cout<< parent[k]<<”  “;
          k=parent[k];
         } ;
if (k== -1) { cout<< “There is no path from” << v << “to” << s <<endl;
  cout<<endl;
}
```

Input GRAPH:



Sample input:

```
5
10
0    4    2.0
0    2    3.0
0    1    5.0
4    1    6.0
4    3    4.0
4    2    10.0
2    3    2.0
2    1   1.0
1    3    6.0
1    2    2.0
```

Sample output

```
Input:  nv = 5 , ne =  10

Graph
0 :  Edge:(0, 4, 2) Edge:(0, 2, 3) Edge:(0, 1, 5)
1 :  Edge:(1, 3, 6) Edge:(1, 2, 2)
2 :  Edge:(2, 3, 2) Edge:(2, 1, 1)
3 :
4 :  Edge:(4, 1, 6) Edge:(4, 3, 4) Edge:(4, 2, 10)

Depth First traversal
Order of vertices visited :    0   4   1   3   2
```

/****************** Edge.h ****************************/

```
#pragma once
#include <iostream>
#include <vector>
#include <string>
using namespace std;

class Edge
{
public:
	Edge(int v1 = 0, int v2 = 1, double w = 0.0) : fv{ v1 }, tv{ v2 }, weight{ w }
	{ }

	void print(ostream& out = cout) const
	{
		out << " Edge:(" << fv << ", " << tv << ", " << weight << ")";
	}

public:
	int fv; // from-vertex
	int tv; // to-vertex
	double weight;
};

// Define an output operator for Square
ostream& operator<< (ostream& out, const Edge& rhs)
{
	rhs.print(out);
	return out;
}
```

/************************** List.h ***************************************/

```cpp
#pragma once
#ifndef LIST_H
#define LIST_H

#include <algorithm>
using namespace std;

template <typename Object>
class List
{
private:
    // The basic doubly linked list node.
    // Nested inside of List, can be public
    // because the Node is itself private
    struct Node
    {
        Object  data;
        Node* prev;
        Node* next;

        Node(const Object& d = Object{ }, Node* p = nullptr, Node* n = nullptr)
            : data{ d }, prev{ p }, next{ n } { }

        Node(Object&& d, Node* p = nullptr, Node* n = nullptr)
            : data{ std::move(d) }, prev{ p }, next{ n } { }
    };

public:
    class const_iterator
    {
    public:

        // Public constructor for const_iterator.
        const_iterator() : current{ nullptr }
        { }

        // Return the object stored at the current position.
        // For const_iterator, this is an accessor with a
        // const reference return type.
        const Object& operator* () const
        {
            return retrieve();
        }

        const_iterator& operator++ ()
        {
            current = current->next;
            return *this;
        }

        const_iterator operator++ (int)
        {
            const_iterator old = *this;
            ++(*this);
            return old;
        }

        const_iterator& operator-- ()
        {
            current = current->prev;
            return *this;
        }

        const_iterator operator-- (int)
        {
            const_iterator old = *this;
            --(*this);
            return old;
        }

        bool operator== (const const_iterator& rhs) const
        {
            return current == rhs.current;
        }

        bool operator!= (const const_iterator& rhs) const
        {
            return !(*this == rhs);
        }

    protected:
        Node* current;

        // Protected helper in const_iterator that returns the object
        // stored at the current position. Can be called by all
        // three versions of operator* without any type conversions.
        Object& retrieve() const
        {
            return current->data;
        }

        // Protected constructor for const_iterator.
        // Expects a pointer that represents the current position.
        const_iterator(Node* p) : current{ p }
        { }

        friend class List<Object>;
    };

    class iterator : public const_iterator
    {
    public:

        // Public constructor for iterator.
        // Calls the base-class constructor.
        // Must be provided because the private constructor
        // is written; otherwise zero-parameter constructor
        // would be disabled.
        iterator()
        { }

        Object& operator* ()
        {
                return const_iterator::retrieve();
            }

            // Return the object stored at the current position.
            // For iterator, there is an accessor with a
            // const reference return type and a mutator with
            // a reference return type. The accessor is shown first.
            const Object& operator* () const
            {
                return const_iterator::operator*();
            }

            iterator& operator++ ()
            {
                this->current = this->current->next;
                return *this;
            }

            iterator operator++ (int)
            {
                iterator old = *this;
                ++(*this);
                return old;
            }

            iterator& operator-- ()
            {
                this->current = this->current->prev;
                return *this;
            }

            iterator operator-- (int)
            {
                iterator old = *this;
                --(*this);
                return old;
            }

        protected:
            // Protected constructor for iterator.
            // Expects the current position.
            iterator(Node* p) : const_iterator{ p }
            { }

            friend class List<Object>;
        };

    public:
        List()
        {
            init();
        }

        ~List()
        {
            clear();
            delete head;
            delete tail;
        }

        List(const List& rhs)
        {
            init();
            for (auto& x : rhs)
                push_back(x);
        }

        List& operator= (const List& rhs)
        {
            List copy = rhs;
            std::swap(*this, copy);
            return *this;
        }


        List(List&& rhs)
            : theSize{ rhs.theSize }, head{ rhs.head }, tail{ rhs.tail }
        {
            rhs.theSize = 0;
            rhs.head = nullptr;
            rhs.tail = nullptr;
        }

        List& operator= (List&& rhs)
        {
            std::swap(theSize, rhs.theSize);
            std::swap(head, rhs.head);
            std::swap(tail, rhs.tail);

            return *this;
        }

        // Return iterator representing beginning of list.
        // Mutator version is first, then accessor version.
        iterator begin()
        {
            return iterator(head->next);
        }

        const_iterator begin() const
        {
            return const_iterator(head->next);
        }

        // Return iterator representing endmarker of list.
        // Mutator version is first, then accessor version.
        iterator end()
        {
            return iterator(tail);
        }

        const_iterator end() const
        {
            return const_iterator(tail);
        }
```

```
// Return number of elements currently in the list.
int size() const
{
    return theSize;
}

// Return true if the list is empty, false otherwise.
bool empty() const
{
    return size() == 0;
}

void clear()
{
    while (!empty())
        pop_front();
}

// front, back, push_front, push_back, pop_front, and pop_back
// are the basic double-ended queue operations.
Object& front()
{
    return *begin();
}

const Object& front() const
{
    return *begin();
}

Object& back()
{
    return *--end();
}

const Object& back() const
{
    return *--end();
}

void push_front(const Object& x)
{
    insert(begin(), x);
}

void push_back(const Object& x)
{
    insert(end(), x);
}

void insert_at(iterator itr, const Object& x) {
    insert(itr, x);
}

void push_front(Object&& x)
{
    insert(begin(), std::move(x));
}
```

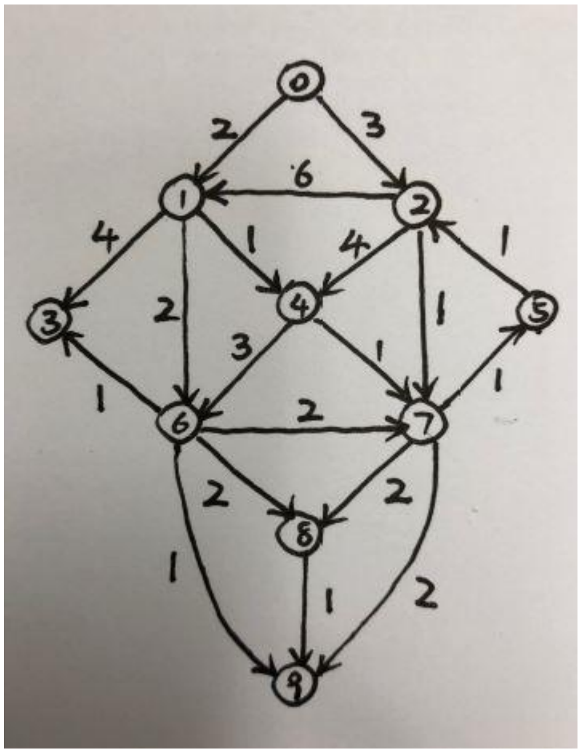
```
    void push_back(Object&& x)
    {
        insert(end(), std::move(x));
    }

    void pop_front()
    {
        erase(begin());
    }

    void pop_back()
    {
        erase(--end());
    }

    // Insert x before itr.
    iterator insert(iterator itr, const Object& x)
    {
        Node* p = itr.current;
        ++theSize;
        return iterator(p->prev = p->prev->next = new Node{ x, p->prev, p });
    }

    // Insert x before itr.
    iterator insert(iterator itr, Object&& x)
    {
        Node* p = itr.current;
        ++theSize;
        return iterator(p->prev = p->prev->next = new Node{ std::move(x), p->prev, p
});
    }

    // Erase item at itr.
    iterator erase(iterator itr)
    {
        Node* p = itr.current;
        iterator retVal(p->next);
        p->prev->next = p->next;
        p->next->prev = p->prev;
        delete p;
        --theSize;

        return retVal;
    }

    iterator erase(iterator from, iterator to)
    {
        for (iterator itr = from; itr != to; )
            itr = erase(itr);

        return to;
    }


    void printList() { // added 4/9/21 for integers
        for (iterator itr = this->begin(); itr != this->end(); itr++) {
  //          (*itr).print();
```

```
            (*itr).print(cout);
        }
        cout << endl;
    }
private:
    int   theSize;
    Node* head;
    Node* tail;

    void init()
    {
        theSize = 0;
        head = new Node;
        tail = new Node;
        head->next = tail;
        tail->prev = head;
    }
};

#endif
```

/*********************** Source.cpp *******************************/

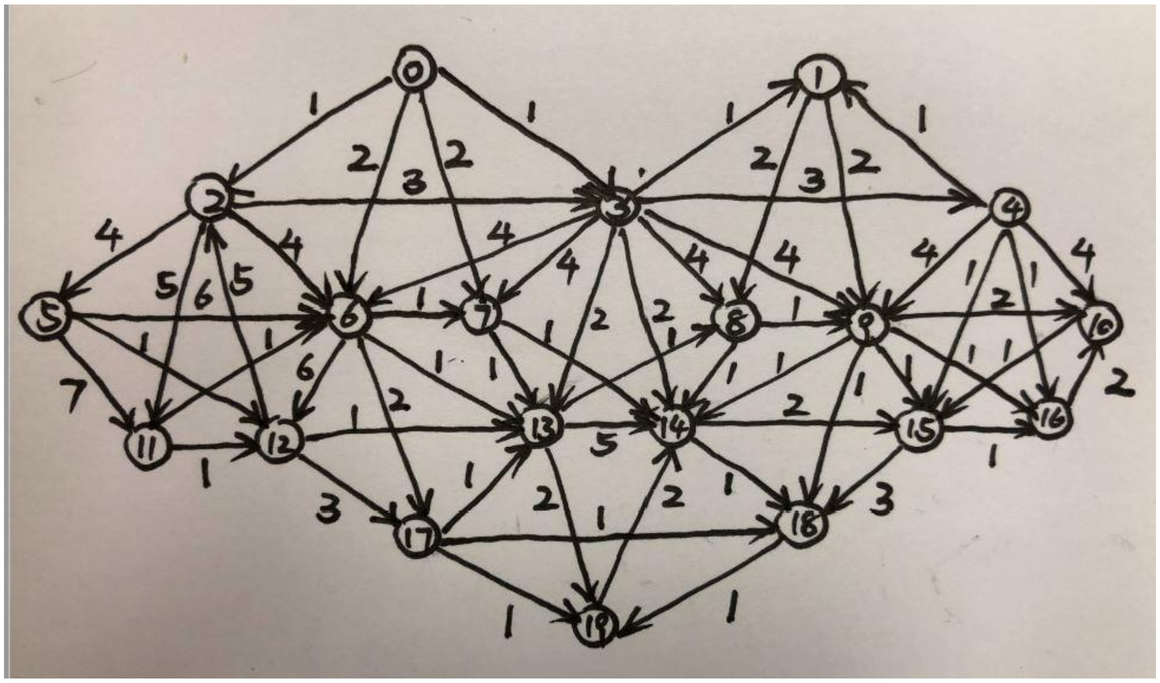
```
#include <iostream>
using namespace std;
#include <cstdlib>   // for rand(), srand()
#include <ctime>     // for time()
#include <assert.h>
#include <math.h>    // for sqrt()
#include "List.h"
#include "Edge.h"

void traverse(int v, vector<bool>& vis, const vector<List<Edge>>& g) {
	vis[v] = true;

	//cout << "Visited vertex : " << v << endl;
	cout << "    " << v;

	for (auto& w : g[v]) {
		if (!vis[w.tv])
			traverse(w.tv, vis, g);
	}
}

int main() {

	int nv = 0; // number of vertices
	int ne = 0; // number of edges
	cin >> nv >> ne; // input; assume no error in input.
	cout  << "Input:  nv = " << nv << " , ne =  " << ne  << endl;

    if( (nv<0) || (nv>10000) || (ne<0) || (ne>10000) ) {
```

```
		cerr << "Input values out of range." << endl;
		exit(1);
	}

	vector<List<Edge>> g2(nv); //graph
	int cv1 = 0, cv2 = 0; // Edge from current vertex cv1 to cv2
	double wt = 0.0; // weight

	for(int ne1=0; ne1<ne; ne1++)
	 {
		cin >> cv1 >> cv2 >> wt; // input; assume no error in input.
		if ((cv1 < 0) || (cv1 > nv) || (cv2 < 0) || (cv2 > nv) || (wt<0) || (wt
>10000.0)) {
			cerr << "Input values out of range." << endl;
			exit(1);
		}
		Edge nsq(cv1, cv2, wt);
		g2[cv1].push_back(nsq);
	}
	cout << endl << "Graph " << endl;
	for (int i = 0; i < nv; i++) {
		cout << i << " : ";
		g2[i].printList();
	//	cout << endl;
	}
	vector<bool> visited(nv, false);
	cout <<endl<< "Depth First traversal " << endl<< "Order of vertices visited :
" ;
	for (int v = 0; v < nv; v++) {
		if (!visited[v])
			traverse(v, visited,g2);
	}
	cout << endl;

	for (int i = 0; i < nv; i++) {
	//	cout << i << " : Cleared ";
		g2[i].clear();
		cout << endl;
	}

	return 0;
}



//    TEST  INPUT  1
```



```
10
19
0    1    2.0
0    2    3.0
1    3    4.0
1    6    2.0
1    4    1.0
2    1    6.0
2    4    4.0
2    7    1.0
4    6    3.0
4    7    1.0
5    2    1.0
6    3    1.0
6    7    2.0
6    8    2.0
6    9    1.0
7    5    1.0
7    8    2.0
7    9    2.0
8    9    1.0
```

```
// TEST INPUT  2
```



```
20
58
0    2    1.0
0    3    1.0
0    6    2.0
0    7    2.0
1    8    2.0
1    9    2.0
2    3    3.0
2    5    4.0
2    6    4.0
2    11   5.0
3    1    1.0
3    4    3.0
3    6    4.0
3    7    4.0
3    8    4.0
3    9    4.0
3    13   2.0
3    14   2.0
4    1    1.0
4    9    4.0
4    10   4.0
4    15   1.0
4    16   1.0
5    6    6.0
5    11   7.0
```

```
5    12   1.0
6    7    1.0
6    12   6.0
6    13   1.0
6    17   2.0
7    13   1.0
7    14   1.0
8    9    1.0
8    14   1.0
9    10   2.0
9    14   1.0
9    15   1.0
9    16   1.0
9    18   1.0
10   15   1.0
11   12   1.0
11   6    1.0
12   2    5.0
12   13   1.0
12   17   3.0
13   8    1.0
13   14   5.0
13   19   2.0
14   15   2.0
14   18   1.0
15   16   1.0
15   18   3.0
16   10   2.0
17   13   1.0
17   18   1.0
17   19   1.0
18   19   1.0
19   14   2.0
```