ECE250 – Lab 3 Help
Graphs and Kruskal's MST

Tiuley Alguindigue (Lab Instructor)

Fall 2020

Before you start implementing project 3

Please review and make sure you have a clear understanding of the topics in the following lectures by Prof. Ladan:

- Lecture #24 – Graph-Part A – Adjacency Matrix and Adjacency List
- Lecture #26 - MST - 26
- Lecture #27 – Disjoint Sets

Outline: Graphs and Kruskal's MST

- The driver program
- Representing the graph
  - Adjacency matrix or adjacency list
- Implementing graph operations:
  - Constructor, destructor, insert, clear, delete, edge_count, degree
- Calculate minimum spanning tree using Kruskal's algorithm
- Exception handling
- Disjoint Sets class

Guideline for your P3 driver program:

Let the driver program take care of **all input/output**. The driver must also catch all the exceptions.

In order to achieve this your class functions must be designed to return an indication of success/failure. The functions must throw exceptions where indicated by requirements.

Graph as matrix in C++

- For project 3, you will need to represent a weighted undirected graph.

- We consider the *n* nodes in the graph to be numbered from *0* to *n – 1*. This means a graph with 4 nodes has nodes named *0*, *1*, *2* and *3*. Each edge has a weight (a positive number of double type) associated with it.

- You can represent the graph as an adjacency matrix or an adjacency list. In this presentation, you will see some tips about implementing the graph as an **adjacency matrix**.

Graph as Adjacency Matrix in C++

- Class variables
- Constructor
- Accessing the element at position (i,j)
- Destructor

Adjacency Matrix in C++

Class variables

```
class graph {
 private:

        double** adj; // array 2d
        int n_nodes;
        int n_edges;
         // any other variables you need
 public:
       // class functions ..
}
```

Adjacency Matrix in C++

Constructor

```
graph::graph(int n) {
        // allocate memory for pointers to each row
        n_nodes = n;
        adj = new double * [n_nodes];
        // allocate memory for each row
        for ( int i = 0; i < n_nodes; ++i ) {
                adj[i] = new double[n_nodes];
        }

        init_empty_graph();

}
```

Adjacency Matrix in C++

Constructor

```
void graph::init_empty_graph(){

  // initialize empty graph -  all elements to infinity except diagonal is 0
    for ( int i = 0; i < n_nodes; ++i ) {
            for ( int j = 0; j < n_nodes; ++j ) {
            adj[i][j] = INF;
            }

            adj[i][i] = 0.0;
   }
   // any other initializations of class variables needed
}
```

Notice the notation to access position (i,j)

Note: INF is a class constant declared as:

```
const double Weighted_graph::INF std::numeric_limits<double>::infinity();
You will need to add to your class an  include for the library limits
#include <limits>
```

Adjacency Matrix in C++

Destructor

```
graph::~graph() {

    // delete memory for each row
    for ( int i = 0; i < n_nodes; ++i ) {
        delete [] adj[i];
    }

    // delete memory for pointers to each row
    delete      [] adj;

}
```

Exception handling

- A class for the exception
- Exception throw
- Exception catch

Exception handling

A class for the exception

```
class invalid_argument {
    // empty class
};
```

Exception handling

Throwing exception

```
int graph::degree(int u) {
    if ( u < 0 || u >= n_nodes)
        throw invalid_argument();

    // return degree for node u
}
```

Exception handling

Catching exception in the driver program:

```
int n = 4;
graph g(n);
// read command for insert
 // for this example, we try to insert and edge for   u = 0, v = -5 , w=10.0

try {
   g.insert(u,v,w );
}
catch (invalid_argument) {
    cout << “invalid argument" << endl ;
}
catch(...){
         std::cout << “Unknown exception " << std::endl;
}
```

Kruskal's MST algorithm

See Prof. Ladan's lectures 26 and 27 for to get full understanding of this algorithm.

Disjoint Sets class

You must write a *Disjoint sets* class for implementing the MST.

Disjoint sets is a well-known data structure for grouping *n* elements (nodes) into a collection of disjoint sets (connected components). You can read more information on disjoint sets from Chapter 21 of CLRS book.

We recommend that you write the *disjoint sets* class using linked lists.

Use of classes C++ libraries

You are **not allowed** to use classes from the C++ STL library for the implementation of the Disjoint Set class.

You can use the vector class for the graph representation and the operations on the graph (insert , update, clear,mst etc)

You can use functions from the standard C++ library, such as **`sort`**

Resources

- Prof Ladan's lectures:
  - Lecture #24 – Graph-Part A – Adjacency Matrix and Adjacency List
  - Lecture #26 - MST - 26
  - Lecture #27 – Disjoint Sets

- For the implementation of the graph - https://ece.uwaterloo.ca/~dwharder/aads/Lecture_materials/#graph-algorithms (slides 1-32)

References

- https://ece.uwaterloo.ca/~dwharder/aads/Lecture_materials/#graph-algorithms (slides 1-32)