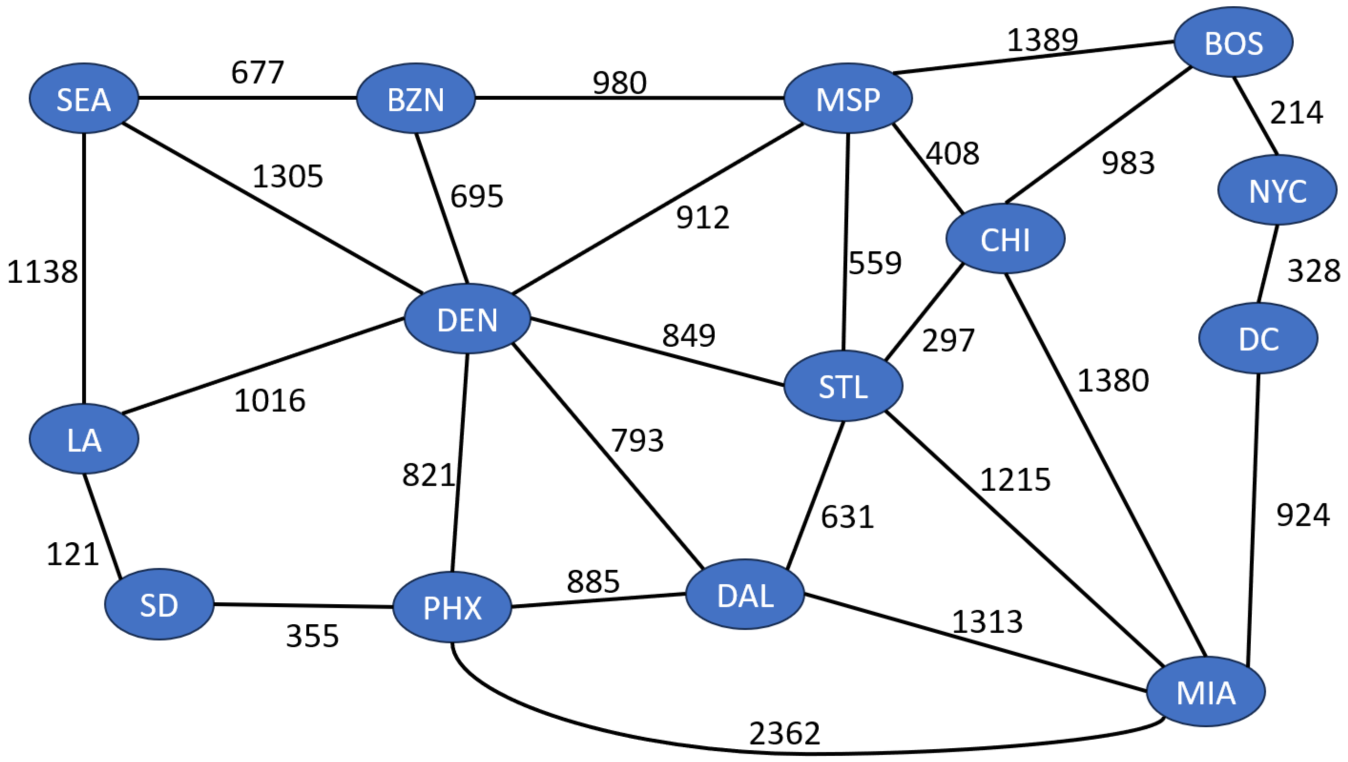
CSCI 232 Program 3

Due Monday June 19th @ 11:59 PM. Please submit this assignment (.java files) to the appropriate dropbox on D2L.

This is a pretty beefy assignment, because you will be given quite a bit of code, and you will be required to write quite a bit of code. A lot of the code given to you is identical to the code we've been writing in class, so nothing should surprise you. We will be doing Parts 1, 2, and 5 during lecture on Wednesday, June 14th, and you will be able to copy and paste the code from lecture. Still, make sure you give yourself enough time to complete this assignment.

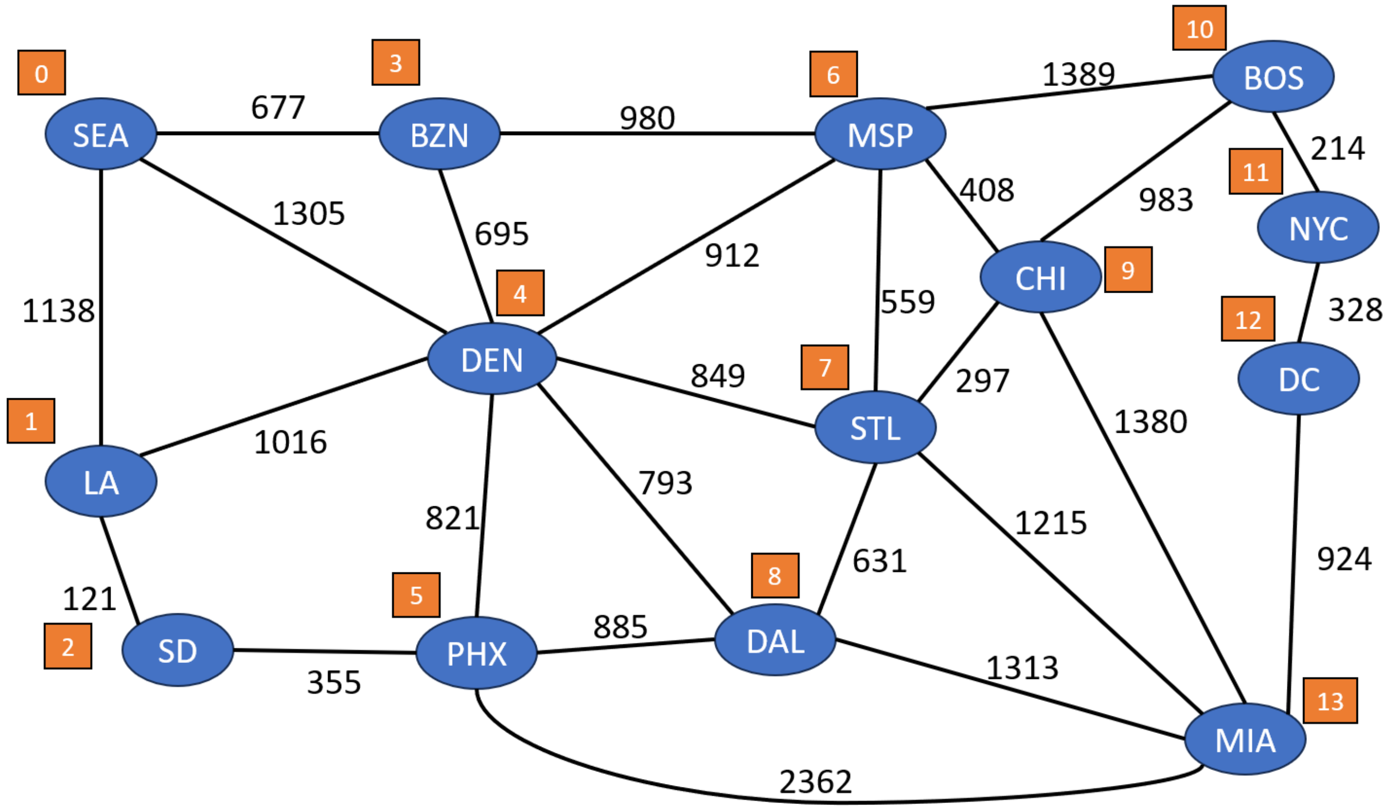
Background and Instructions

In this assignment, you are going to write a program that computes interesting information about an undirected, weighted graph. The graph represents airports in the United States, and the cost of a plane ticket to travel to other airports. The graph you are analyzing in this assignment looks something like this:



For example, the cost of flying from Seattle (SEA) to Denver (DEN) is 1305.

Each vertex is represented by a String (the airport/city name), **however in our program, the vertices are still going to be integers** (like we've had in class). The labels for the graph above look something like this (the orange boxes):



Later on in part 2, you will need to come up with a method the converts a the integer vertex label (such as 8) to the correct airport name (such as DAL).

Part I: Reading in vertex information (`loadAirportInfo()`)

The first step is reading the vertex information of the graph. This data is stored in a text file called **`airports.txt`**. You will need to read in from this file. In this task, you need to write the body of the **`loadAirportInfo()`** method, which is in the **`WeightedGraph`** class. There is a HashMap in the **`WeightedGraph`** class called **`vertexToAirport`**, which maps Integers (vertex labels) to Strings (city name). You will need to read from **`airports.txt`**, and fill this HashMap with key value pairs, where the keys are the vertex label (integers) and the values linked to the keys are a String (city name). Using the example above, it should look something like this (order does not matter):

```
vertexToAirport = {   0: SEA
                      1: LA
                      2: SD
                        …
                      13: MIA   }
```

Part II: Reading in Edge information(loadEdgeInfo())

Now you will read in the edge information of the graph. This information is stored in the text file edges.txt. The structure of this text file follows this format:

```
Vertex1  Vertex2  Cost
0,3,677
0,4,1305
0,1,1138
1,4,1016
```

For example, there is an edge that connects vertex 0 (SEA) and vertex 3 (BZN) with a weight of 677. Remember that in this assignment, we are working with undirected edges. The graph is represented by an adjacency list (just like we've done in class). The **Edge** class is already defined for you. You can modify the class, but it is not necessary. Note an important difference in the **Edge** class.

```java
private int vertex1;
private String vertex1Name;
private int vertex2;
private String vertex2Name;
private double weight;
```

Each edge object keeps track the vertices it connects, as well as the Airport names the edge connects (using the HashMap from Part 1).

Your task here is to write the body of the **loadEdgeInfo()** method (located in the **WeightedGraph** class), which fills the contents of your adjacency list. It should do this by reading the **edges.txt** file and add each edge from the file into the graph using the **addEdge()** method that is already defined for you. Once you think you have this working, you can call **graph.printGraph()** in the Program3Demo class, and the output should look something like if you did parts 1 and 2 correctly:

```
[(SEA,BZN): 677.0, (SEA,DEN): 1305.0, (SEA,LA): 1138.0]
[(SEA,LA): 1138.0, (LA,DEN): 1016.0, (LA,SD): 121.0]
[(LA,SD): 121.0, (SD,PHX): 355.0]
[(SEA,BZN): 677.0, (DEN,BZN): 695.0, (BZN,MSP): 980.0]
[(SEA,DEN): 1305.0, (LA,DEN): 1016.0, (PHX,DEN): 821.0, (DEN,BZN): 695.0, (DEN,MSP): 912.0, (DEN,STL): 849.0, (DEN,DAL): 793.0]
[(SD,PHX): 355.0, (PHX,DEN): 821.0, (PHX,DAL): 885.0, (PHX,MIA): 2362.0]
[(BZN,MSP): 980.0, (DEN,MSP): 912.0, (MSP,STL): 559.0, (MSP,CHI): 408.0, (MSP,BOS): 1389.0]
[(DEN,STL): 849.0, (MSP,STL): 559.0, (DAL,STL): 631.0, (STL,MIA): 1215.0, (STL,CHI): 297.0]
[(DEN,DAL): 793.0, (PHX,DAL): 885.0, (DAL,STL): 631.0, (DAL,MIA): 1313.0]
[(MSP,CHI): 408.0, (STL,CHI): 297.0, (CHI,MIA): 1380.0, (CHI,BOS): 983.0]
[(MSP,BOS): 1389.0, (CHI,BOS): 983.0, (BOS,NYC): 214.0]
[(BOS,NYC): 214.0, (NYC,DC): 328.0]
[(NYC,DC): 328.0, (DC,MIA): 924.0]
[(PHX,MIA): 2362.0, (DAL,MIA): 1313.0, (STL,MIA): 1215.0, (CHI,MIA): 1380.0, (DC,MIA): 924.0]
```

Once you verified you have this part working, you can proceed to part 3.

Part III: Computing Airport with most connections (`findMostConnections()`)

At this point, your graph should be built correctly, and we have a way to concert integer vertex labels to city names (the HashMap from part 1).

In this task, you need to write the body of the **`findMostConnections()`** method, which is in the **`WeightedGraph`** class. This method will compute the vertex with the highest degree. To put this in the context of the problem, this method computes the city that has the most direct connections with other cities. In the example above, the city/airport with the most connections is Denver (DEN). However, you need to compute this so that it would work for any given graph.

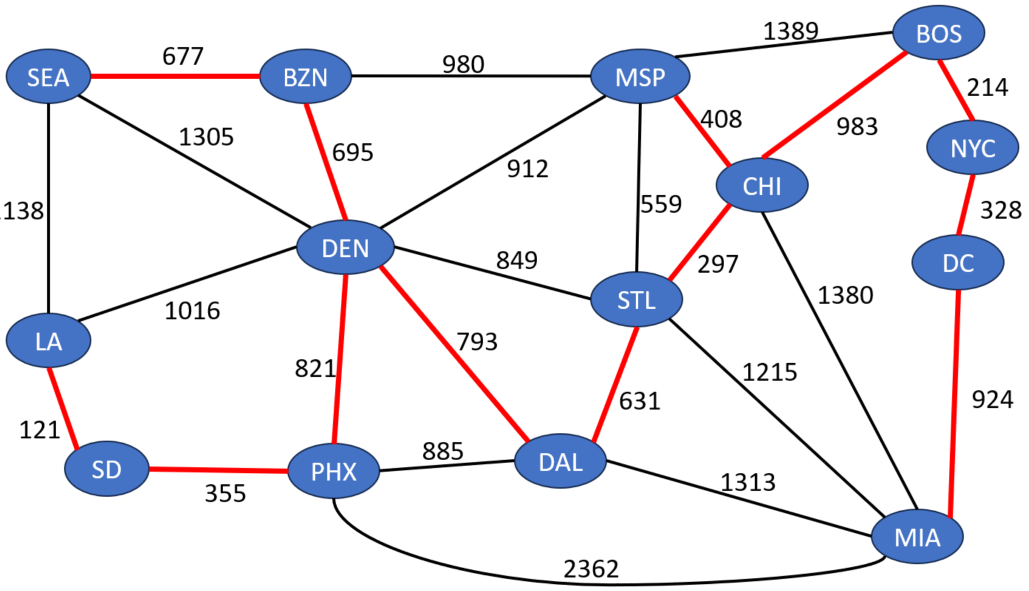
Part IV: Computing MST of graph (`computeMST()`)

Suppose that someone wanted to visit every single airport, and spend the least money possible (for this scenario to make sense, let's suppose they can teleport back to already-visited airports at no cost). In this task, you will now compute the Minimum Spanning Tree (MST) of our airport graph. You must use **Kruskal's Algorithm** to generate the MST. You should be able to reuse the code we wrote in class on Monday, June 12th.

In our example above, the method should output the edges (and costs) of the MST:

```
(LA,SD): 121.0
(SEA,BZN): 677.0
(SD,PHX): 355.0
(DEN,BZN): 695.0
(PHX,DEN): 821.0
(DEN,DAL): 793.0
(MSP,CHI): 408.0
(DAL,STL): 631.0
(STL,CHI): 297.0
(CHI,BOS): 983.0
(BOS,NYC): 214.0
(NYC,DC): 328.0
(DC,MIA): 924.0
```

^ This is the output of the method



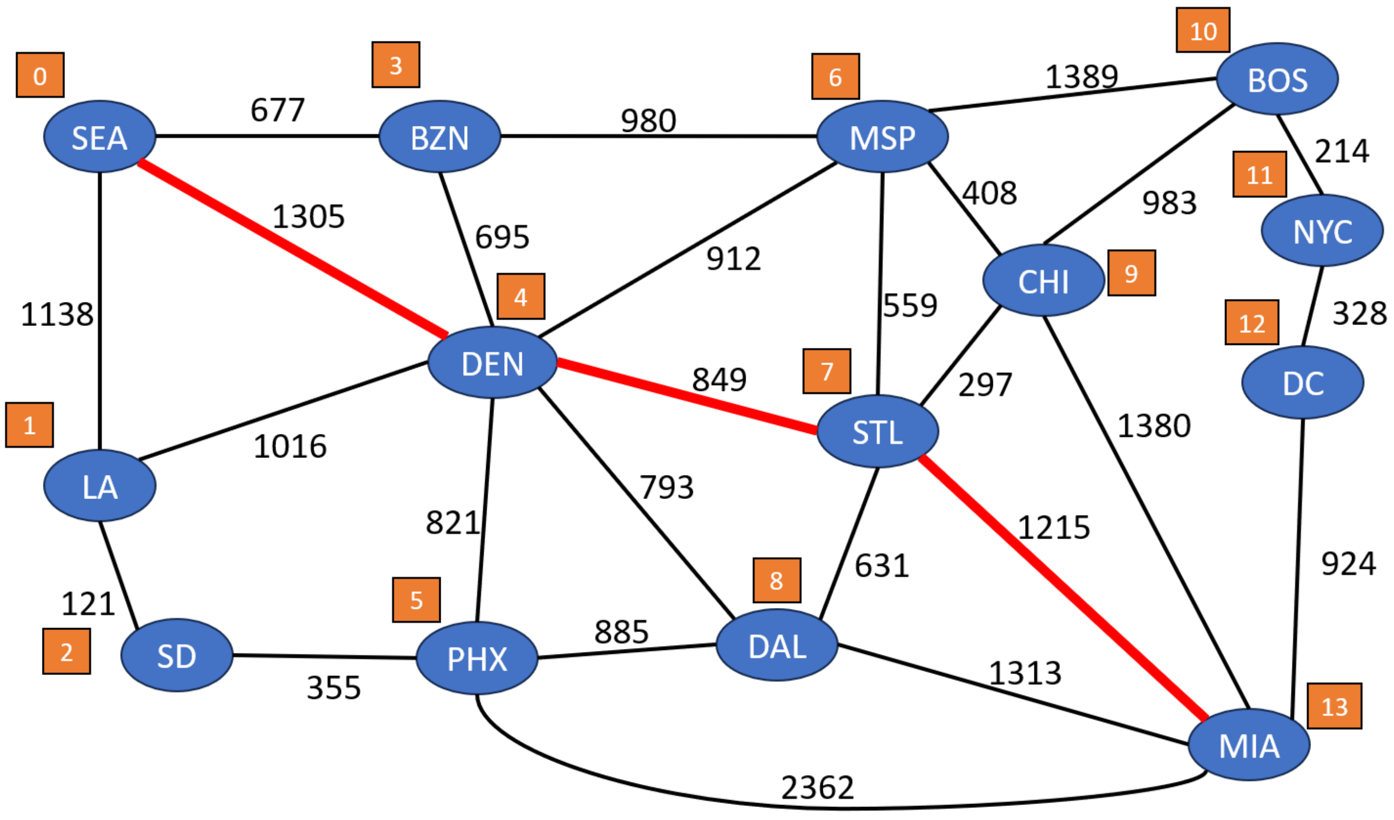
Once again, you cannot hard code this. Your MST algorithm should work for any graph. The code for Kruskal's Algorithm will go inside of the **`computeMST()`**, which is in the **`MST`** class.

Part V: Computing the shortest path from one airport to another (computeShortestPath())

Lastly, you will write the code that will compute the shortest path from one vertex to another. This method takes in (1) the graph, (2) the integer starting vertex, and (3) the integer ending vertex. You will need to write the body of the **`computeShortestPath()`** method, which is located in the **`ShortestPath`** class. This method **must use Dijkstra's Algorithm** to compute the shortest path. You should be able to reuse the code that we wrote in class on Tuesday, June 13th. Your program should output the edges and costs of the shortest path, and the total cost of the path. You can assume the user will enter two valid city names, and you can assume there will be a path between the two cities.

```
What is the starting city? (ex. BZN, NYC, MSP)
SEA
What is the destination city? (ex. BZN, NYC, MSP)
MIA
(SEA,DEN): 1305.0
(DEN,STL): 849.0
(STL,MIA): 1215.0
Total cost to go from SEA to MIA: 3369.0
```

(Shortest path from SEA to MIA visualized)



Sample Output

Below is a sample transcript of running the different menu options. The format of your output does not need to match it exactly, but it should be pretty close.

```
1. Find city with most connections
2. Print MST of Graph
3. Find shortest path between cities
4. Exit
Your choice?
1
The city with most connections is: DEN
1. Find city with most connections
2. Print MST of Graph
3. Find shortest path between cities
4. Exit
Your choice?
2
(LA,SD): 121.0
(SEA,BZN): 677.0
(SD,PHX): 355.0
(DEN,BZN): 695.0
(PHX,DEN): 821.0
(DEN,DAL): 793.0
(MSP,CHI): 408.0
(DAL,STL): 631.0
(STL,CHI): 297.0
(CHI,BOS): 983.0
(BOS,NYC): 214.0
(NYC,DC): 328.0
(DC,MIA): 924.0
1. Find city with most connections
2. Print MST of Graph
3. Find shortest path between cities
4. Exit
Your choice?
3
What is the starting city? (ex. BZN, NYC, MSP)
SEA
What is the destination city? (ex. BZN, NYC, MSP)
MIA
(SEA,DEN): 1305.0
(DEN,STL): 849.0
(STL,MIA): 1215.0
Total cost to go from SEA to MIA: 3369.0
1. Find city with most connections
2. Print MST of Graph
3. Find shortest path between cities
4. Exit
Your choice?
3
What is the starting city? (ex. BZN, NYC, MSP)
BOS
What is the destination city? (ex. BZN, NYC, MSP)
DEN
(CHI,BOS): 983.0
(STL,CHI): 297.0
(DEN,STL): 849.0
Total cost to go from BOS to DEN: 2129.0
```

Starting Code:

There is a lot of starting code here, so I will give you a zip of the entire Java project. You will need to extract the zip file, and then *File→Open Project From File System →Select the extracted folder (csci232-program3-release)*

- csci232-program3-release.zip (https://www.cs.montana.edu/pearsall/classes/summer2023/232/programs/csci232-program3-release.zip)

Grading

• HashMap is loaded correctly in part 1 – **10 points**

• Edges are correctly loaded into the graph in part 2 – **10 points**

• Your program correctly computes the city with most connections (part 3) – **10 points**

• Your program correctly computes and prints out the MST of the graph (part 4) – **30 points**

• Your program correctly computes and prints the shortest path from one city to another (part 5)- **30 points**

• Each method written by the user has a comment describing what the method does- **10 points**

NOTE: If your code does not compile, correctness cannot be verified, and you won't receive any points for your code. Turn in code that compiles!