CS 251 Summer 2020
Project 4: Graphs
Due date:

General Guidelines

The APIs given below describe the required methods in each class that will be tested. You may need additional methods or classes that will not be directly tested but may be necessary to complete the assignment. Keep in mind that anything that does not *need* to be public should generally be kept private (instance variables, helper methods, etc.).

Unless otherwise stated in this handout, you are welcome (and encouraged) to add to/alter the provided java files as well as create new java files as needed.

In general, you are not allowed to import any additional classes in your code without explicit permission from your instructor! For this project, it is acceptable to use ArrayLists. For any other imported classes, you should ask.

Note on academic dishonesty: Please note that it is considered academic dishonesty to read anyone else's solution code to this problem, whether it is another student's code, code from a textbook, or something you found online. You MUST do your own work! You are allowed to use resources to help you, but those resources should not be code, so be careful what you look up online!

Note on implementation details: Note that even if your solution passes all test cases, if a visual inspection of your code shows that you did not follow the guidelines in the project, you will receive a 0.

Note on grading and provided tests: The provided tests are to help you as you implement the project. The Vocareum tests will be similar but not exactly the same. Keep in mind that the points you see when you run the local tests are only an indicator of how you are doing. The final grade will be determined by your results on Vocareum.

Project Overview

This project uses several graph algorithms to build a cost-effective flight network for a new airline. The airline must determine which flights to include in its regional networks and in a global network.

What is provided: Several classes are provided for you. You may alter these as you choose, but your submitted code must still work with the test cases. You are strongly encouraged to review these classes thoroughly before starting your solution. The following classes are provided:

- *Graph.java*: This is a representation for a graph and includes a lot of methods that may be useful to you as you write your solution.
- *DistQueue.java*: This class is provided to help with implementations of Dijkstra's Algorithm.
- *UnionFind.java*: This class is provided to help with the implementation of Kruskal's Algorithm.
- *Edge.java*: This class represents an edge.
- *EdgeSort.java*: This class is used to sort the edges by edge weight.

What you must submit: The following classes are provided as skeleton code and the methods listed are required. You should not change the method signatures for the required methods, but you may add to these classes as necessary.

- *RegNet.java*: This should contain your solution to Part A.
- *GlobalNet.java:* This should contain your solution to Part B.
- *Graph.java, DistQueue.java, UnionFind.java, Edge.java, EdgeSort.java*
- Any other *.java* files that are necessary for your solution

Part A. Building a Regional Network

In this part, you will build a regional network. Your solution should be in *RegNet.java*. The required method is described below. The idea here is to build a maximal regional network by prioritizing certain things: minimize total distance (the assumption is that this will minimize overall cost) while also maximizing connectivity between airports and maximizing the number of flights that are offered, while also minimizing the number of flights with a lot of connections. This is explained with a couple examples below.

`public static Graph run(Graph G, int max)`

`Input:`

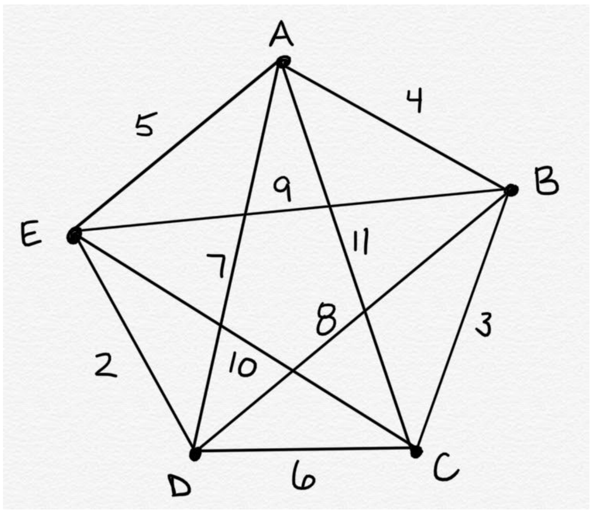
- *`G`*`: a complete, simple, undirected graph with positive edge weights. Vertices are airports edges represent flight distances between airports.`
- *`max`*`: a positive integer value representing the airline's budget for this region`

```
Output:
     A new graph connecting as many airports in G as possible while
keeping the total distance within the budget. Furthermore, this graph
should try to include as many flights as possible and try to minimize
the number of flights with a lot of connections.
```

At this point, you're very likely confused. Hopefully, these examples will help.

Example 1

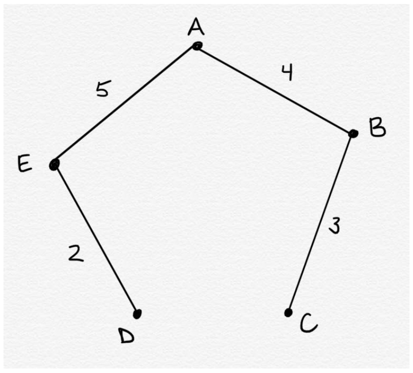
The input is a complete graph containing 5 vertices (airports).



Let the budget be 10.

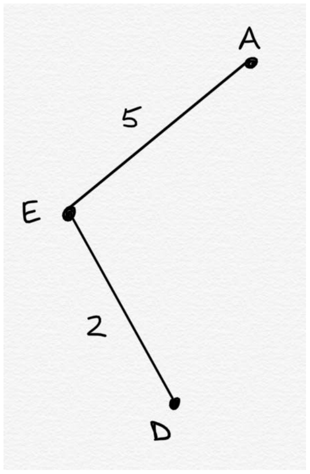
Step 1: Create an MST that fits within the budget and includes as many airports as possible.

(a) Start by running Kruskal's Algorithm on the graph to find an MST. You can assume that all the test graphs will have a unique MST.



(b) If the total weight of the MST is larger than the budget, we need to remove some edges, but we want to remove larger edges first AND make sure we still have a connected graph.

In this case, the total weight is 14. We start by considering the longest edge in the MST, which is 5. This would fix the budget problem, but it would disconnect the graph. So we go on to the next largest, which is 4. Same problem. Then we look at the next one, which is 3. We remove it, which means C cannot be in the region. Now we start over: 5 will still disconnect the graph. But 4 will not, so we remove that. Now this is our graph.



Step 2: Taking the MST, we will now see if we can add in any more flights and still be within the budget. Note that only the current airports are considered. To determine which flights to consider, we first calculate the number of stops between each pair of airports because we want to prioritize adding direct flights between airports that currently do not have a direct flight between them. **Note that it is up to you to figure out how to calculate these values in a reasonably efficient way.**

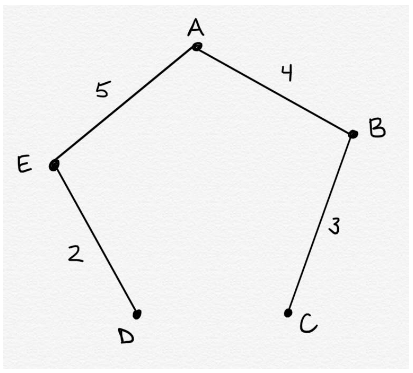
Here, we have:

A to E: 0 stops
A to D: 1 stop
E to D: 0 stops

Note that any pair that has 0 stops between them does not need to be considered. Otherwise, we consider them in order of greatest number of stops. *Here that just leaves A to D*. To get the distance of such a direct flight, we would check the original graph. *The distance from A to D is 7, which will unfortunately not fit within the budget. So we are done.*

Example 2
Here we will look at the same input graph but with a larger budget. **Let the budget be 30.**
Step 1:

(a) MST:



(b) *In this case, the MST fits well within the budget, so we move on to Step 2.*

Step 2:

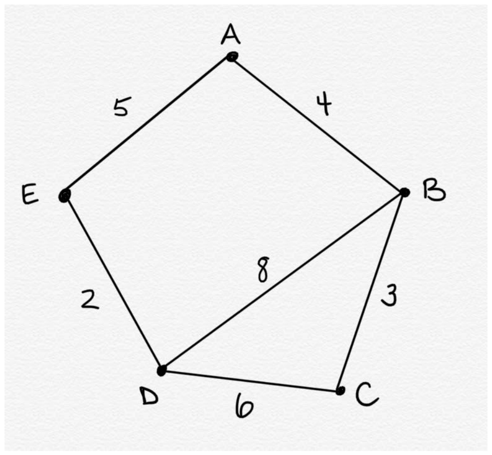
Here, we have:

A to E: 0 stops
A to B: 0 stops
A to C: 1 stop
A to D: 1 stop
B to C: 0 stops
B to D: 2 stops
B to E: 1 stop
C to D: 3 stops
C to E: 2 stops
D to E: 0 stops

Putting these in order of greatest to least and disregarding the 0 stops, we get the following list of direct flights to consider. Notice that for pairs with an equal number of stops between them, we prioritize shorter flights first.

C to D: 3 stops (d = 6)
B to D: 2 stops (d = 8)
C to E: 2 stops (d = 10)
A to D: 1 stop (d = 7)
B to E: 1 stop (d = 9)
A to C: 1 stop (d = 11)

The MST has a total weight of 14. We can add (C, D), making the total weight 20. Note that after adding this edge, some of the values we just calculated will change. ***You do not have to recalculate these. Just use the values that you originally calculated.*** *We then check the pairs with 2 stops between them, prioritizing shorter flights over longer ones. We can add (B, D), making the total weight 28. We check (C, E), but that will exceed the* *budget. Then we check the 1-stop pairs. The smallest one will exceed the budget, so we are done and our network is:*



Part B. Building a Global Network

In this part, the goal is to combine several regional networks into a global network by adding the flights related to the shortest path from one region to each of the other regions. The shortest path from one region to another means *the shortest path from any airport in the source region to any airport in the target region.* You need to do this for each pair of regions. You will be given a starting graph with all the possible paths between regions. These may be direct flights or they may have airports in between that don't belong to any of the regions. It also may not be a complete graph as some flights may not be possible. As you do this, you will also designate certain airports in each region as inter-regional airports. There is a boolean array in *Graph.java* that you can use for this. It will be useful in Part C. The following is a description of the general process you will use to solve this problem. Your solution should be in *GlobalNet.java*, implementing the following required method. (You are encouraged, however, to use private methods and possibly other classes to put the code into more manageable pieces.)

```
public static Graph run(Graph O, Graph[] regions)
```

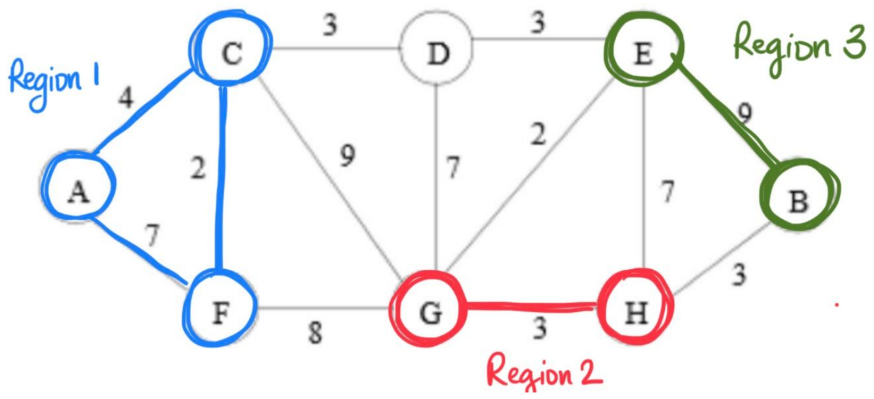
Input: A graph consisting of all the regional networks as well as all possible paths between them. An array of the Graphs representing each region.
Output: A graph consisting of all the regional networks as well as the shortest path between each pair of regions. This graph will also designate the source and target airports of each inter-regional shortest path as inter-regional airports for use in Part C.

`**Process:**` `For each pair of regions` ***`S`*** `and` ***`T`****`, determine the shortest path between`* ***`S`*** *`and`* ***`T`*** *`and add it to the graph`*`. Let` ***`s`*** `be the starting vertex and` ***`t`*** `be the ending vertex of this shortest path. Designate` *`s`* `and` *`t`* `as inter-regional airports.`

`The process for` *`determining the shortest path between`* ***`S`*** `and` ***`T`*** `is best done with a variation on Dijkstra's Algorithm. Note that the runtime (in terms of big-Oh) should not be worse than Dijkstra's Algorithm.`

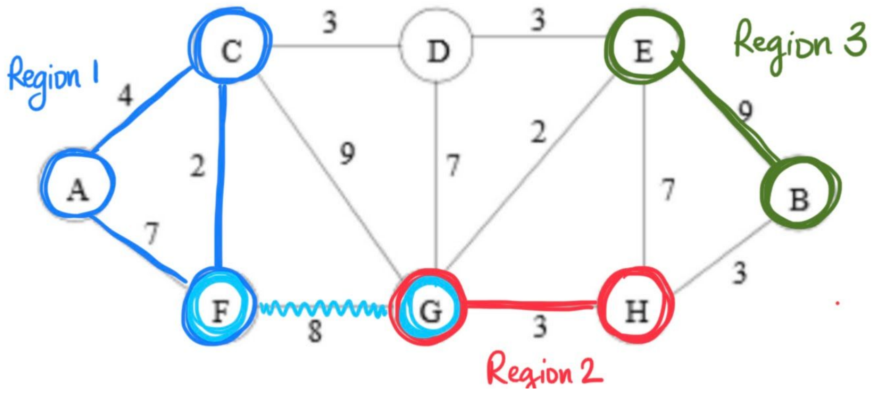
Example

The following example may help to illuminate this process.



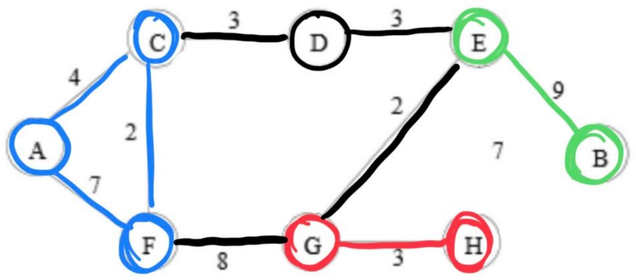
The input graph shows three regions and all the possible paths between them.

- We use a variation of Dijkstra's Algorithm to determine the shortest path from Region 1 to Region 2. That is determined to be the path from F to G, so that path is added to the graph and F and G are designated as inter-regional airports. Note that C-D-E-G is the same length, but a typical implementation of Dijkstra's Algorithm would return F-G.



- We then run the same algorithm to determine the shortest paths from Region 1 to Region 3 and from Region 2 to Region 3. Those are C-D-E and G-E respectively, so C, E, H, and

G are marked as inter-regional airports and (C, D), (D, E), and (G, E) are added to the global network. The final global network is shown below.



Grading & Testing

Part A Tests	36 points
Part B Tests	36 points
Robustness	28 points
Total	**100 points**

You are provided with test cases for each of these, and you may create your own tests as necessary. The tests provided are similar to the tests used for grading.