CSC 233 – Foundations of Computation
Programming Project #4 – Graphs

Our next out-of-class task will be a programming project on graphs. I encourage you to do this assignment in your pairs.

In this project you'll be programming the Bellman-Ford algorithm for the single-source shortest paths problem. I'll break this project into steps.

We'll be using Java. Just to keep things simple, let's please plan on using BlueJ as our IDE.

This is not a trivial assignment! Please start early so that you have plenty of time to work through bugs and ask me questions.

Turning In This Project

To turn in this code, please follow the instructions below:

- Create a zip file of the entire BlueJ project folder. NOT just the package.bluej file, but rather the entire folder, containing all the .java files, .class files, etc.
- Name the zip file *lastname*.zip or *lastname1-lastname2*.zip, when working alone or with a partner, respectively.
- Upload the entire project, as a zip, to Moodle.
- If you have any doubts about getting Moodle to work, please also email your code to me. Do not let the due time pass without turning in your code.

Java for This Project

Why are we using Java, and not Scala, for this project? My reasoning is that, in our study of functional programming, our consistent emphasis was functional programming. Scala was just the vehicle we used to study it. There are many matters specific to Scala that we didn't study, instead focusing on the universally-applicable functional programming ideas. So I don't want to then turn around and ask you to learn lots of new material in Scala for this project. We'll stick with a Java implementation, so that you can focus on the graph algorithm rather than particulars of Scala.

---

1) Understanding provided code

I've given you some starting code on the I: drive, as a BlueJ project. It's a basic graph implementation, using adjacency lists. Look at the Node class first. Read through the comments and the code carefully.

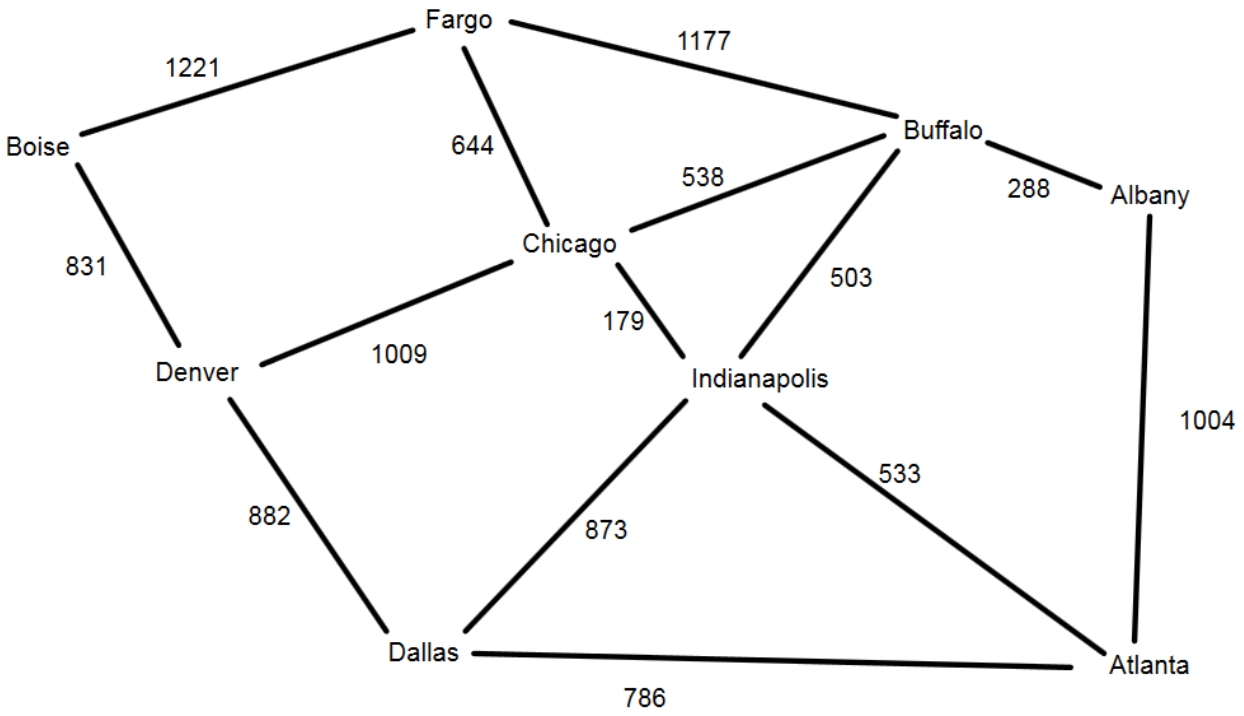
There's one thing in the Node class that you might not be as familiar with, depending on what other courses you've taken: the <E> notation. This means that Node is a "generic" class. It's a similar idea to what you've already seen with ArrayLists. Recall that when you make an ArrayList, you specify the type it will hold, in < >. So for example, ArrayList<Double> is an ArrayList of doubles, while ArrayList<Integer> is

an ArrayList of ints. If you feel rusty on the use of ArrayLists, check out your old CS1 notes, or something like:
http://examples.javacodegeeks.com/core-java/util/arraylist/arraylist-in-java-example-how-to-use-arraylist/
before moving on.

Node is defined to be a generic class, so that means it works kind of like ArrayList, in that it's able to operate with any type. More specifically, this class can be used to make a Node that contains String data, or a Node that contains Integer data, or a Node that contains Person data (for some defined class named Person), etc. So it's a very flexible class. E represents the type. So E might be String, or Integer, or Person, for example.

Next check out the Graph class. It includes some methods for creating a graph, setting up the edges, and printing out information about it. Note that it's a generic class too, able to work with Nodes of any type E. Its only field is an ArrayList of Nodes. Observe how this is declared: ArrayList<Node<E>>, since it's an ArrayList that contains Nodes that have data of type E. In the Graph class, if you want to declare a local variable that is a node, use the type Node<E>. Read through the comments and code carefully in the Graph class.

Finally, check out the GraphTester class. When I declare the variable g to be of type Graph<String>, I'm setting the type parameter E to a String. So you can imagine that everywhere E appears in the Node and Graph classes, it's actually String, for this particular Graph g. The code then goes on to create the nodes and edges. If we wanted to take the time to make something more sophisticated, we'd probably want to write code that reads graph data from a file and automatically generates the corresponding graph, but this is sufficient for now. The graph created is the one shown below, of U.S. cities and the distance in miles between them:



It is absolutely essential that you fully understand the provided code in Node, Graph, and GraphTester before moving on to step 2 of the project. Any time you might save by skimming something here will lead to a much larger loss of time later as you wrestle with implementation. If you'd like to talk with me more about how this provided code works, please let me know.

Please note: In your solution to this project, do not change anything in the Node or GraphTester classes, and do not modify existing code in the Graph class. Your solution should consist entirely of defining the required methods in the Graph class (along with any private helper methods in the Graph class that you want to create).

---

2) Getting familiar with our graph data structure implementation: printOneEdgeAtATime

The next step in this project is to write the method printOneEdgeAtATime, in the Graph class. The method header is provided already. When you're done, you should be able to run the test1 method in the GraphTester class, and the following should print out as a result of calling printOneEdgeAtATime:

```
Edge from 0 Albany, NY to 1 Atlanta, GA with weight 1004.0
Edge from 0 Albany, NY to 3 Buffalo, NY with weight 288.0
Edge from 1 Atlanta, GA to 0 Albany, NY with weight 1004.0
Edge from 1 Atlanta, GA to 8 Indianapolis, IN with weight 533.0
Edge from 1 Atlanta, GA to 5 Dallas, TX with weight 786.0
Edge from 2 Boise, ID to 6 Denver, CO with weight 831.0
Edge from 2 Boise, ID to 7 Fargo, ND with weight 1221.0
Edge from 3 Buffalo, NY to 0 Albany, NY with weight 288.0
Edge from 3 Buffalo, NY to 4 Chicago, IL with weight 538.0
Edge from 3 Buffalo, NY to 7 Fargo, ND with weight 1177.0
Edge from 3 Buffalo, NY to 8 Indianapolis, IN with weight 503.0
Edge from 4 Chicago, IL to 3 Buffalo, NY with weight 538.0
Edge from 4 Chicago, IL to 6 Denver, CO with weight 1009.0
Edge from 4 Chicago, IL to 7 Fargo, ND with weight 644.0
Edge from 4 Chicago, IL to 8 Indianapolis, IN with weight 179.0
Edge from 5 Dallas, TX to 1 Atlanta, GA with weight 786.0
Edge from 5 Dallas, TX to 6 Denver, CO with weight 882.0
Edge from 5 Dallas, TX to 8 Indianapolis, IN with weight 873.0
Edge from 6 Denver, CO to 2 Boise, ID with weight 831.0
Edge from 6 Denver, CO to 4 Chicago, IL with weight 1009.0
Edge from 6 Denver, CO to 5 Dallas, TX with weight 882.0
Edge from 7 Fargo, ND to 2 Boise, ID with weight 1221.0
Edge from 7 Fargo, ND to 3 Buffalo, NY with weight 1177.0
Edge from 7 Fargo, ND to 4 Chicago, IL with weight 644.0
Edge from 8 Indianapolis, IN to 1 Atlanta, GA with weight 533.0
Edge from 8 Indianapolis, IN to 3 Buffalo, NY with weight 503.0
Edge from 8 Indianapolis, IN to 4 Chicago, IL with weight 179.0
Edge from 8 Indianapolis, IN to 5 Dallas, TX with weight 873.0
```

Of course, printOneEdgeAtATime should print the appropriate information for whatever Graph object on which it is called. Note in the output above that 28 edges are shown, while the graph image in step 1 shows 14 edges. This is because the method printed each edge twice, once for each "direction": For example, it prints the edge from Albany to Atlanta, and the (same) edge from Atlanta to Albany. It'd be

easy enough to adapt this method to print each edge in only one direction, but this would complicate your code later in the project, so we just won't worry about it.

While you might try to complete this method by modifying the toString methods in the Node and Graph classes, this won't help you in preparation for step (3), which is the whole point of step (2). So instead, work out your solution to step (2) by setting up loops as needed, in the printOneEdgeAtATime method.

---

3) Implementing Bellman-Ford

Write the method called bellmanFord, in the Graph class. It should be an implementation of the Bellman-Ford algorithm. My advice is to follow the pseudocode from class extremely closely. As much as possible, even use the same variable names. I also recommend you start by just copying your code for printOneEdgeAtATime into the bellmanFord method. After all, a key part of Bellman-Ford is to loop through every edge. That's why it will be helpful that you've already done the printOneEdgeAtATime method to practice this.

When you're done, you should be able to run the test1 method and get output like this, for two calls to bellmanFord (with a source of Albany, and then a source of Dallas):

```
Source: Albany
Albany, NY: 0.0, by way of N/A
Atlanta, GA: 1004.0, by way of Albany, NY
Boise, ID: 2666.0, by way of Denver, CO
Buffalo, NY: 288.0, by way of Albany, NY
Chicago, IL: 826.0, by way of Buffalo, NY
Dallas, TX: 1664.0, by way of Indianapolis, IN
Denver, CO: 1835.0, by way of Chicago, IL
Fargo, ND: 1465.0, by way of Buffalo, NY
Indianapolis, IN: 791.0, by way of Buffalo, NY

==========================================
==========================================
==========================================
==========================================
Source: Dallas
Albany, NY: 1664.0, by way of Buffalo, NY
Atlanta, GA: 786.0, by way of Dallas, TX
Boise, ID: 1713.0, by way of Denver, CO
Buffalo, NY: 1376.0, by way of Indianapolis, IN
Chicago, IL: 1052.0, by way of Indianapolis, IN
Dallas, TX: 0.0, by way of N/A
Denver, CO: 882.0, by way of Dallas, TX
Fargo, ND: 1696.0, by way of Chicago, IL
Indianapolis, IN: 873.0, by way of Dallas, TX
```

The most important thing to note at this time is that the code in the Node and Graph classes is completely general. We're applying it to one graph of cities here, but of course we could create any graph of cities we wanted. More than that, though, we could create a graph of any kind, with nodes

containing any kind of data, using either directed or undirected edges, weighted or unweighted edges. We could then apply Bellman-Ford to this graph. This is very flexible and powerful code!

---

4) Improving the output (Optional: 2 points extra credit)

In your implementation in step 3, comment out the code that prints the results. Keep that original version saved while you work on this printing enhancement. Instead of the basic output of step 3, traverse the predecessors to come up with each complete path. For example, running test1 in GraphTester:

```
Source: Albany
From Albany, NY to Albany, NY, distance: 0.0, path: Albany, NY
From Albany, NY to Atlanta, GA, distance: 1004.0, path: Albany, NY, Atlanta, GA
From Albany, NY to Boise, ID, distance: 2666.0, path: Albany, NY, Buffalo, NY, Chicago, IL, Denver, CO, Boise, ID
From Albany, NY to Buffalo, NY, distance: 288.0, path: Albany, NY, Buffalo, NY
From Albany, NY to Chicago, IL, distance: 826.0, path: Albany, NY, Buffalo, NY, Chicago, IL
From Albany, NY to Dallas, TX, distance: 1664.0, path: Albany, NY, Buffalo, NY, Indianapolis, IN, Dallas, TX
From Albany, NY to Denver, CO, distance: 1835.0, path: Albany, NY, Buffalo, NY, Chicago, IL, Denver, CO
From Albany, NY to Fargo, ND, distance: 1465.0, path: Albany, NY, Buffalo, NY, Fargo, ND
From Albany, NY to Indianapolis, IN, distance: 791.0, path: Albany, NY, Buffalo, NY, Indianapolis, IN

==========
==========
==========
==========
Source: Dallas
From Dallas, TX to Albany, NY, distance: 1664.0, path: Dallas, TX, Indianapolis, IN, Buffalo, NY, Albany, NY
From Dallas, TX to Atlanta, GA, distance: 786.0, path: Dallas, TX, Atlanta, GA
From Dallas, TX to Boise, ID, distance: 1713.0, path: Dallas, TX, Denver, CO, Boise, ID
From Dallas, TX to Buffalo, NY, distance: 1376.0, path: Dallas, TX, Indianapolis, IN, Buffalo, NY
From Dallas, TX to Chicago, IL, distance: 1052.0, path: Dallas, TX, Indianapolis, IN, Chicago, IL
From Dallas, TX to Dallas, TX, distance: 0.0, path: Dallas, TX
From Dallas, TX to Denver, CO, distance: 882.0, path: Dallas, TX, Denver, CO
From Dallas, TX to Fargo, ND, distance: 1696.0, path: Dallas, TX, Indianapolis, IN, Chicago, IL, Fargo, ND
From Dallas, TX to Indianapolis, IN, distance: 873.0, path: Dallas, TX, Indianapolis, IN
```