TCSS 342A - Data Structures
Group Project 3 - Graphs and shortest paths

Group membership: due March 3, 2017 (Friday) 9:30am via Canvas
Presentation and demo: due March 10, 2017 (Friday) 9:30am via Canvas
Code and documentation: due March 11, 2017 (Saturday) 9:30am via Canvas

This project is to be completed in a group of 1, 2 or 3 members. You are highly encouraged, but not required to work in groups. One submission per group. **Please submit your group membership via canvas by March 3.** State the official names (as appear in myUW) of all your group members clearly in your submission. This group project consists of in-class presentation/demo, programming and written work.

1. **In-class presentation/demo**: Each group will submit a ppt, pptx or pdf file **before 9:30am on March 10, 2017** via Canvas. Each presentation is expected to be 5 minutes. The order of presentation will be determined randomly. Depending on the random number you draw, your group may not have a chance to present. If your group does not have a chance to present, I will award points based on your submitted presentation. Therefore, your presentation must be **self-explanatory** and may include screen shots or even videos to demonstrate the working of your code.

2. The **code and documentation** will be due on **March 11 @ 9:30am** via Canvas. Solutions should be a complete working Java program including your original work. These files should be compressed in a .zip file for submission through the Canvas link.

This project will contribute 10% towards your final grade, with the following breakdown of points: 2% (presentation/demo) + 3% (code compiles and pass tests) + 2% (testing) + 3% (documentation and analysis) + 2% (bonus points)

Project Description

For this assignment, you will develop a graph representation and use it to implement Dijkstra's algorithm for finding shortest paths. Unlike previous assignments, you will use some classes in the Java standard libraries, gaining valuable experience reading documentation and understanding provided APIs.

You may use anything in the Java standard collections (or anything else in the standard library) for any part of this assignment. Take a look at the Java API (https://docs.oracle.com/javase/8/docs/api/) as you are thinking about your solutions. At the very least, look at the Collection and List interfaces to see what operations are allowable on them and what classes implement those interfaces.

Provided Files

Download these files into a new directory:

- Graph.java -- Graph interface. *Do **not** modify.*
- Vertex.java -- Vertex class
- Edge.java -- Edge class
- MyGraph.java -- Implementation of the Graph interface: you will need to fill in code here
- Path.java -- Class with two fields for returning the result of a shortest-path computation. *Do **not** modify.*
- FindPaths.java-- A client of the graph interface: Needs small additions

vertex.txt and edge.txt -- an example graph in the correct input format

Part 1: Graph representation

In this part of the assignment, you will implement a graph representation that you will use in Part 2. Add code to the provided-but-incomplete `MyGraph` class to implement the Graph interface. Do not change the arguments to the constructor of `MyGraph` and do not add other constructors. Otherwise, you are free to add things to the `Vertex, Edge, and MyGraph` classes, but please do not remove code already there and do not modify `Graph.java`. You may also create other classes if you find it helpful.

As always, your code should be correct (implement a graph) and efficient (in particular, good asymptotic complexity for the requested operations), so choose a good graph representation for computing shortest paths in Part 2.

We will also grade your graph representation on how well it protects its abstraction from bad clients. In particular this means:

- The constructor should check that the arguments make sense and throw an appropriate exception otherwise. Here is what we mean by make sense:
    - The edges should involve only vertices with labels that are in the vertices of the graph. That is, there should be no edge from or to a vertex labeled A if there is no vertex with label A.
    - Edge weights should not be negative.
    - Do *not* throw an exception if the collection of vertices has repeats in it: If two vertices in the collection have the same label, just ignore the second one encountered as redundant information.
    - *Do* throw an exception if the collection of edges has the same *directed* edge more than once with a different weight. Remember in a directed graph an edge from A to B is *not* the same as an edge from B to A. Do *not* throw an exception if an edge appears redundantly with the same weight; just ignore the redundant edge information.

Other useful information:

- The `Vertex` and `Edge` classes have already defined an appropriate **equals** method. If you need to decide if two `Vertex` objects are "the same", you probably want to use the equals method and not ==.
- You will likely want some sort of Map (https://docs.oracle.com/javase/8/docs/api/java/util/Map.html) in your program so you can easily and efficiently look up information stored about some Vertex. (This would be much more efficient than, for example, having a `Vertex[]` and iterating through it every time you needed to look for a particular Vertex.)
- As you are debugging your program, you may find it useful to print out your data structures. There are `toString` methods for Edge and Vertex. Remember that things like `ArrayLists` and `Sets` can also be printed.

Part 2: Dijkstra's algorithm

In this part of the assignment, you will use your graph from Part 1 to compute shortest paths. The `MyGraph` class has a method `shortestPath` you should implement to return the lowest-cost path from its first argument to its second argument. Return a Path object as follows:

- If there is no path, return null.

- If the start and end vertex are equal, return a path containing one vertex and a cost of 0.
- Otherwise, the path will contain at least two vertices -- the start and end vertices and any other vertices along the lowest-cost path. The vertices should be in the order they appear on the path.

Because you know the graph contains no negative-weight edges, Dijkstra's algorithm is what you should implement. Additional implementation notes:

- One convenient way to represent infinity is with `Integer.MAX_VALUE`.
- Using a priority queue is above-and-beyond. You are not required to use a priority queue for this assignment. Feel free to use any structure you would like to keep track of distances and then search it to find the one with the smallest distance that is also unknown.
- You definitely need to be careful to use **equals** instead of == to compare `Vertex` objects. The way the `FindPaths` class works (see below) is to create multiple Vertex objects for the same graph vertex as it reads input files. Remember that equals lets us compare values (e.g. do two `Vertex` objects have the same label) as opposed to just checking if two things refer to the exact same object.

The program in `FindPaths.java` is *mostly* provided to you. When the program begins execution, it reads two data files and creates a representation of the graph. It then prints out the graph's vertices and edges, which can be helpful for debugging to help ensure that the graph has been read and stored properly. Once the graph has been built, the program loops repeatedly and allows the user to ask shortest-path questions by entering two vertex names. **The part you need to add is to take these vertex names, call `shortestPath`, and print out the result.** Your output should be as follows:

- If the start and end vertices are X and Y, first print a line
  `Shortest path from X to Y:`
- If there is no path from the start to end vertex, print exactly one more line
  `does not exist`
- Else print exactly two more lines. On the first additional line, print the path with vertices separated by spaces. For example, you might print `X Foo Bar Baz Y`. (Do not print a period, that is just ending the sentence.) On the second additional line, print the cost of the path (i.e., just a single number).

The `FindPaths` code expects two input files in a particular format. The names of the files are passed as command-line arguments. The provided files **vertex.txt** and **edge.txt** have the right format to serve as one (small) example data set where the vertices are 3-letter airport codes. Here is the file format:

- The file of vertices (the first argument to the program) has one line per vertex and each line contains a string with the name of a vertex.
- The file of edges (the second argument to the program) has three lines per directed edge (so lines 1-3 describe the first edge, lines 4-6 describe the second edge, etc.) The first line gives the source vertex. The second line gives the destination vertex. The third line is a string of digits that give the weight of the edge (this line should be converted to a number to be stored in the graph).

Note that the provided code in "FindPaths.java" assumes that you will provide two command line input files <vertex.txt> and <edge.txt>. I intend to run your code using command line on linux. See https://docs.oracle.com/javase/tutorial/essential/environment/cmdLineArgs.html for additional info on command line arguments. If you use Eclipse, you need to specify arguments when you run your

code. Specifically, you can go to run, run configurations, and then input the two text files (vertex.txt edge.txt) and then click apply.

Note data files represent **directed graphs**, so if there is an edge from A to B there may or may not be an edge from B to A. Moreover, if there is an edge from A to B and an edge from B to A, the edges may or may not have the same weight.

Testing

Test your solutions thoroughly and to turn in your testing code. Part of the grading will involve thorough testing including any difficult cases. **Name your test file as `TestGraph.java`.**

You can assume that I will test our codes with directed graphs only.

Documentation and write-up questions

In addition to this programming assignment, you will include **documentation of how you tested your code in a separate doc.pdf file**. Submit your test input files, and explain what conditions that your test input files test for. See programming guidelines on Canvas for additional guidance.

In **doc.pdf**, also describe the worst-case asymptotic running times of your methods `adjacentVertices, edgeCost, and shortestPath`. In your answers, use |E| for the number of edges and |V| for the number of vertices. *Explain and justify your answers.*

Bonus option

Find an interesting real-world data set and convert it into the right format for your program. Note that you want to find a **directed graph**. Turn in your data set in the right format as two additional files. Describe the following in a separate **bonus.pdf:**

- Where did you find this real-world data? Why did you choose this dataset?
- What the data is and what a shortest path means
- Explain how you converted this dataset into the right format, report and explain your results
- Discuss any challenge(s) that you encounter

Some potential web sites to find real-world data (you don't have to limit yourself to this list):

- DIMACS Challenge (http://www.dis.uniroma1.it/challenge9/download.shtml)
- Stanford large network dataset collection (https://snap.stanford.edu/data/)
- UC Irvine network data repository (https://networkdata.ics.uci.edu/index.php)

Submission due March 11

You will submit a .zip file containing:

`Vertex.java`
`Edge.java`
`MyGraph.java`
`FindPaths.java`
Any additional Java files needed, if any.
`TestGraph.java`
`doc.pdf`, containing answers to the Write-Up Questions.

`bonus.pdf`, containing answers to the bonus option.

You must not change the `Graph.java` and `Path.java`. Your implementations must work with the code as provided to you.

Points will be deducted if you do not follow the above file format or filenames.

Grading logistics

I will compile your code using
`javac *.java`

I will use my own “Main.java” to test your code. You are free to write your own Main.java, but your routines must work with my “Main.java”.