**Shortest-Path**

**Assignment**

**Introduction**

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.

**Homework Questions**

Submit a Microsoft Word document, answering the following questions:

1. 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.
2. Describe how you tested your code.
3. If you did any above-and-beyond, describe what you did.

**Programming**

**From Blackboard**

Download these files from Blackboard 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

**Implementation**

**Part 1**

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. You can define your own exceptions by making classes like the exception classes we have provided you in previous homeworks. 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 (and, therefore, they also define hashCode appropriately). 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](http://docs.oracle.com/javase/7/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**

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 LAX ORD JFK 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 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**

Similar to the homework assignment Union-Find, be sure to 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**

**What to Submit**

You should submit the following files:

* Vertex.java
* Edge.java
* MyGraph.java
* FindPaths.java
* Any additional Java files needed, if any.
* TestGraph.java
* A Microsoft Word document containing answers to the Homework Questions.
* Any additional files for the extra credit in a zip file named extracredit.zip. Please make sure that this zip file decompresses its contents into a folder called extracredit and not into a bunch of individual files.

Do not turn in Graph.java, Path.java, vertex.txt and edge.txt. You must not change the Graph.java and Path.java. Your implementations must work with the code as provided to you.

**Above and Beyond**

* Find an interesting real-world data set and convert it into the right format for your program. Describe in your write-up questions what the data is and what a shortest path means. Turn in your data set in the right format as two additional files.
* Improve your implementation of Dijkstra's algorithm by using a priority queue. Note that for Dijkstra's algorithm, we need to find items in the priority queue and update their priorities (the decreaseKey operation). We would like to find items in constant time (and then logarithmic time for changing the priority). There are various ways to do this, including keeping a back-pointer from each vertex to its entry in the priority queue. Note: If you implement this above and beyond, you are not required to also implement Dijkstra's without a priority queue. You may submit the priority queue version as your only submission, but be sure to indicate in your write up that you did this.
* Extend MyGraph with a method for computing minimum spanning trees. Also write a driver program that reads in a graph and prints a minimum spanning tree. This driver will be much like FindPaths, but make a separate file and do not prompt the user for vertices or have a loop, just print one minimum spanning tree. Explain in your write up the format of what you print.