COMP 2150 – Spring 2014
Project 2: A Snappy Mapping App
(Posted Mar. 26 – Due Apr. 18 by 11:59 pm)

Note: You have the option of working in teams of two for this assignment. If you choose to do so, please include both your name and your teammate's name on all your source code files. Only one person needs to submit the final product to eCourseware, but please include a note with the submission that lists both of your names. If an honors student and a regular student choose to work together, both agree to be graded on the honors scale.

Submission: Please zip your source code into a single file (you can zip the entire project folder if you're using BlueJ) and upload it to the proper folder in the eCourseware dropbox at https://elearn.memphis.edu. The dropbox will cut off all submissions after the indicated deadline, so please don't wait until the very last minute to submit your work!

Grading: This assignment is worth a maximum of 80/75 points for regular students (75/75 for honors students). The assignment will be graded by the TA Vincent Nkawu (venkawu@memphis.edu). If you have questions or concerns about your grade, please contact him first. I'll be happy to look over your assignment myself if he's not able to resolve the situation to your satisfaction. Also remember that as stated on the syllabus, all submissions MUST compile and run to receive credit. The TA does not have time to find and correct your syntax errors!

Coding Style: Be sure to follow the good coding practices that were discussed in COMP 1900:

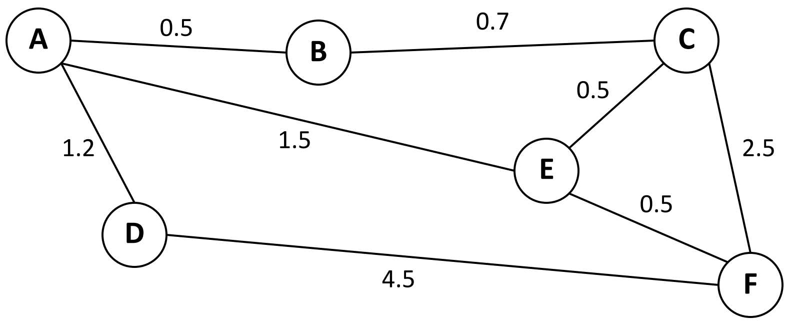
- Use descriptive variable and method names. Avoid using single-character names unless it's very obvious what the variable's being used for (like a loop counter).
- Follow standard Java programming conventions for **variableAndMethodNames**, **ClassNames**, **CONSTANT_NAMES**.
- Consistently indent your code.
- Use comments judiciously. Every source code file should include a comment block at the top that lists your name and the assignment number. Every non-trivial method should have a comment preceding it that specifies what the method does, what its parameters are, and what it returns. For simple methods like accessors or mutators this isn't necessary, but it doesn't hurt to put in a simple comment like `// accessor methods`. I also expect to see comments throughout your code explaining what actions are being taken!

Background Information

Have you ever wondered how websites like Google Maps can give you directions from point A to point B? One solution involves the use of graph theory and a well-known procedure known as *Dijkstra's algorithm*.

First, some basics! As you may have learned from your math classes, a *graph* is just a collection of *vertices* (also known as *nodes*) that may be connected by *edges*. In a *weighted graph*, these edges are associated with numerical values. Think of the vertices as representing particular locations, and the edges as representing the distance (or more generally, "cost" of traveling) between two vertices.

Here's an example of what a simple graph might look like (not drawn to scale):



This graph contains six vertices (labeled A-F) and eight edges. As you can see, there are multiple ways to get from any vertex to any other vertex. For example, to get from A to E, you could take the direct path A-E (at a cost of 1.5) or take the path A-B-C-E (at a cost of 0.5 + 0.7 + 0.5 = 1.7).

Dijkstra's algorithm, developed in 1959 by the Dutch computer scientist of the same name, gives a way of computing the "shortest" (lowest-cost) path between any two vertices of a graph. Here's how the algorithm works. Given a graph and a starting vertex:

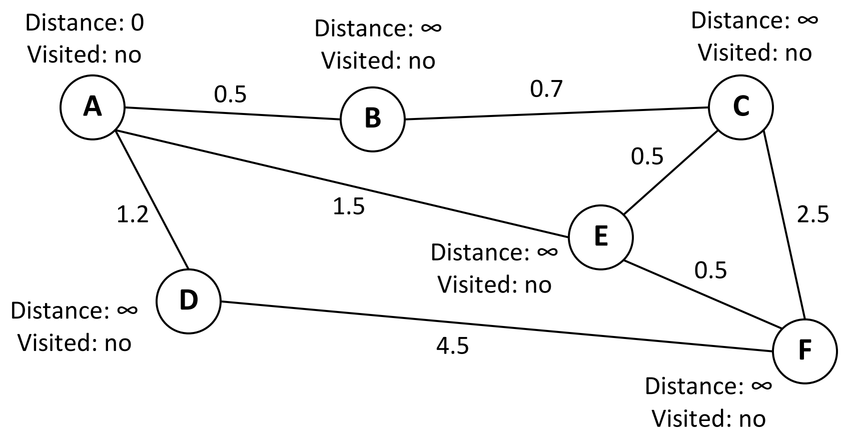
1. Assign each vertex a "distance" value. This value will represent the optimal distance between that vertex and the starting vertex. To begin with, assign the starting vertex a distance of 0 and all other vertices a distance of ∞.
2. Mark all vertices as "unvisited."
3. From the entire graph, pick the unvisited vertex with the lowest distance value. Note that the first time you do this, it'll always be the starting vertex. Call this the "current" vertex, *V*.
4. Consider each of *V*'s unvisited neighbors (i.e., vertices that are directly accessible from *V*). For each of these neighbors *N*, compute *N*'s "tentative" distance by adding *V*'s distance to the weight of the edge connecting *V* and *N*. If this tentative distance is less than *N*'s existing distance, overwrite *N*'s distance with the new one. When an overwrite is performed, also record vertex *V* as the "previous" vertex of vertex *N*.
5. Once you've checked all of *V*'s unvisited neighbors, mark *V* as visited. *V* is now "done" and will not be involved in the algorithm any more.
6. Check if all vertices in the graph have been visited. If so, the algorithm is finished. If not, return to step 3.

Once the algorithm is finished, the final distance values for each vertex represent the minimum cost required to get from the starting vertex to that vertex. The shortest path itself can be found by going to the end vertex, going to its "previous" vertex, going to that previous vertex's own "previous" vertex, and so on until you reach the starting vertex.

Aside from its obvious applications for finding routes on maps, Dijkstra's algorithm is also used in a number of network routing protocols (algorithms that determine the optimal paths to send data packets over a network).

Here's a partial example of how Dijkstra's algorithm would work to compute the lowest-cost path between vertex A and any other vertex in the graph above.

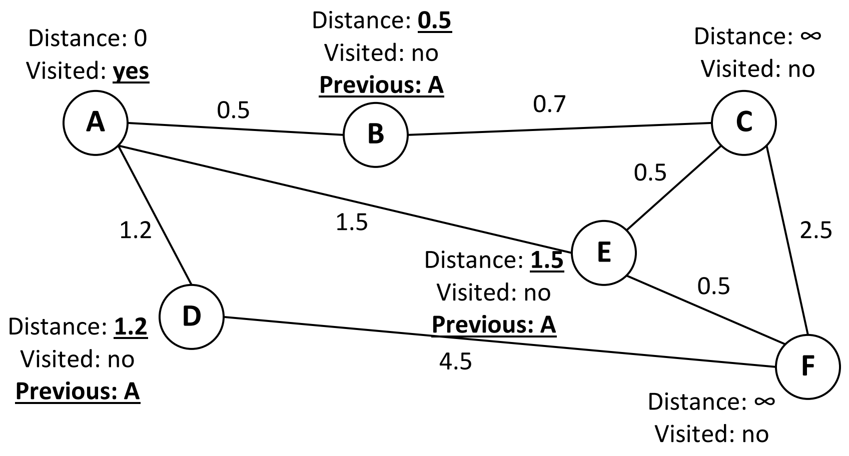
1. Start by assigning all vertices a distance value of ∞, except A itself. Vertex A gets a distance value of 0, since it's the starting vertex. Also mark all vertices as unvisited.



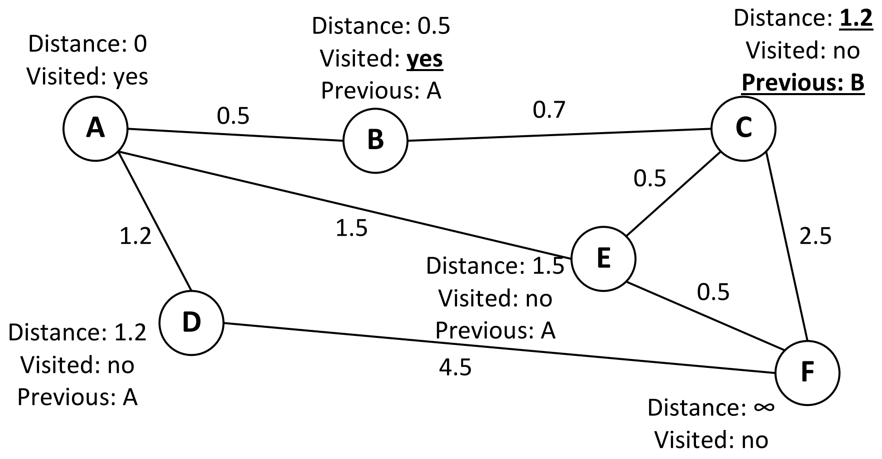
2. Pick the lowest-distance unvisited vertex from the entire graph. In this case, that's vertex A. For each of A's unvisited neighbors (B, D, and E), compute their tentative distances and replace the existing distances if necessary:

   a. For B, its tentative distance is 0 + 0.5 = 0.5. Because 0.5 < ∞, replace B's distance with 0.5. Also note B's "previous" vertex as A.
   b. For D, its tentative distance is 0 + 1.2 = 1.2. Because 1.2 < ∞, replace D's distance with 1.2. Also note D's "previous" vertex as A.
   c. For E, its tentative distance is 0 + 1.5 = 1.5. Because 1.5 < ∞, replace E's distance with 1.5. Also note E's "previous" vertex as A.

   Once all of A's neighbors are considered, mark A as visited.



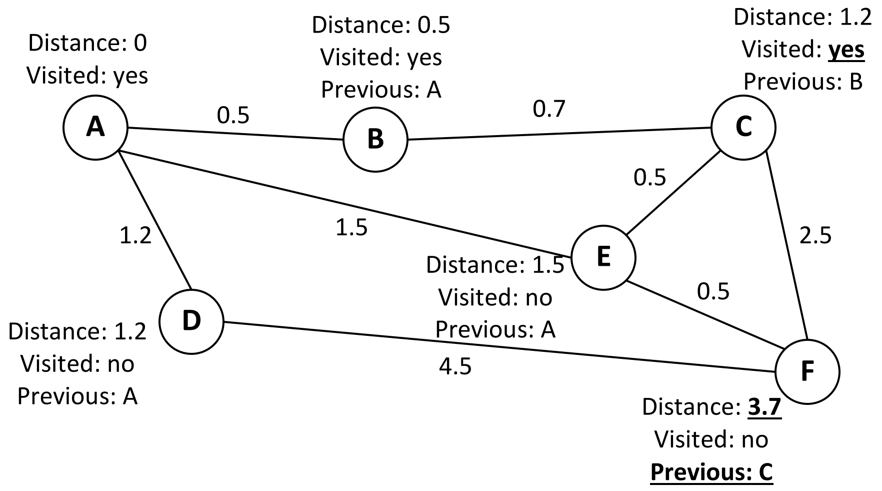
3. Again, pick the lowest-distance unvisited vertex from the entire graph. In this case, that's vertex B. B has only one unvisited neighbor, vertex C. C's tentative distance is 0.5 + 0.7 = 1.2. Because 1.2 < ∞, replace C's distance with 1.2. Also note C's "previous" vertex as B. Mark B as visited.



4. Again, pick the lowest-distance unvisited vertex from the entire graph. In this case, both vertices C and D have distances of 1.2 and are unvisited, so pick either one. Let's pick C for this example. For each of C's unvisited neighbors (E and F), compute their tentative distances and replace the existing distances if necessary:

    a. For E, its tentative distance is 1.2 + 0.5 = 1.7. Because this is not less than E's existing distance of 1.5, do nothing.
    b. For F, its tentative distance is 1.2 + 2.5 = 3.7. Because 3.7 < ∞, replace F's distance with 3.7. Also note F's "previous" vertex as C.

    Once all of C's neighbors are considered, mark C as visited.



5. Keep repeating this process until all vertices are marked as visited. Even at this point, however, you can stop and get the lowest-cost path from A to any vertex already marked as visited. For example, the lowest-cost path from A to C has a cost of 1.2 and is found by going from A-B-C. Note that you can construct this path by starting at the end vertex (C) and following its "previous" record to B, then following B's "previous" record to A.

The Assignment

Write a program that allows the user to read graph information from a text file that s/he specifies. Once the file is loaded, the user should be able to view all the vertices and edges (including weights) of the graph. (No need to do this graphically – although you're welcome to do so if you want!) The user should also be able to select any two vertices from the graph and see the cost of the optimal path between them, as well as the optimal path itself.

Assume that the data file just lists the edges in the graph and their weights. For the example graph discussed previously, the file would look like this:

```
A B 0.5
A E 1.5
A D 1.2
B A 0.5
B C 0.7
C B 0.7
C E 0.5
C F 2.5
D A 1.2
D F 4.5
E A 1.5
E C 0.5
E F 0.5
F C 2.5
F D 4.5
F E 0.5
```

Use the following class design for the project:

- **Vertex** class: This should include an instance variable for the vertex's name, as well as instance variables for the various quantities needed in Dijkstra's algorithm (distance, visited status, and previous vertex). You can assume that all vertices in the graph will have unique names.
- **Edge** class: This should include instance variables for the start and end vertices, as well as the edge's weight.
- **Graph** class: This should include a list of **Vertex** objects and a list of **Edge** objects as instance variables. You can use **java.util.ArrayList** or **java.util.LinkedList** for these lists. This class should also include the following methods:
    - A method that loads information from a data file
    - A method that runs Dijkstra's algorithm using a specified start and end vertex
    - A toString method that includes information about the graph's vertices and edges
- **MappingApp** class: This is the client program where all user input will be made. It should contain an instance of **Graph** and allow the user to load a data file, view the currently loaded graph, or find the shortest path between two vertices in the currently loaded graph. As mentioned before, display both the optimal cost as well as the path itself.

Implement error checking on all user inputs! This includes exception handling: possible crashes due to exceptions like **InputMismatchException** or **FileNotFoundException** should be gracefully handled. Your program should never crash due to user input. (However, you can assume that the input file will be formatted correctly.)

Some helpful tips:

- You can use the constant **Double.POSITIVE_INFINITY** for the value of ∞.
- The static method **Double.parseDouble(String s)** is useful to read a string as a **double** value. The method returns the **double** value that was read. For example, calling **Double.parseDouble("3.14")** returns the **double** value 3.14.

Need Help?

If you choose to work with a teammate for this assignment, the two of you can jointly write your code. I don't mind if you consult with students outside your team, but the code you submit should be written exclusively by you and your teammate (or just you, if you opt to fly solo). If you find yourself stuck, please feel free to contact me (Top) anytime. The Computer Science Learning Center in Dunn Hall 208 is also open, where you can get help from graduate students. Hours are posted at www.cs.memphis.edu/cslc.

Just as with Project 1, you want to get started on this project early!