Due : December 16th, 2017 at 13:00

Overview

This assignment is to implement a weighted, undirected graph.

General Notes

- Do not change any method or class signatures. You should only edit inside of the functions. If your code changes any class or method names or signatures, you will receive an automatic 0. You should not implement any other functions or instance variables besides the ones that are provided, unless explicitly allowed.
- If you are using Eclipse, be sure to remove the line declaring your code a package when you submit your code. Failure to do so could result in you receiving a 0 if the autograder fails.
- Make sure your code compiles. Non-compiling code will automatically receive a 0. If you have a problem that is causing you to not be able to compile, it may be better to just comment out the incorrect code and return a dummy value (something like null or -1) so the rest can compile.
- Make sure that your code does not print out anything (there should be no System.out.println in your code). You will receive an automatic 0 if your code outputs something to STDOUT during the tests.
- You will be graded on correctness and efficiency. You may receive partial credit for a correct but sub-optimal (within reason) solution.

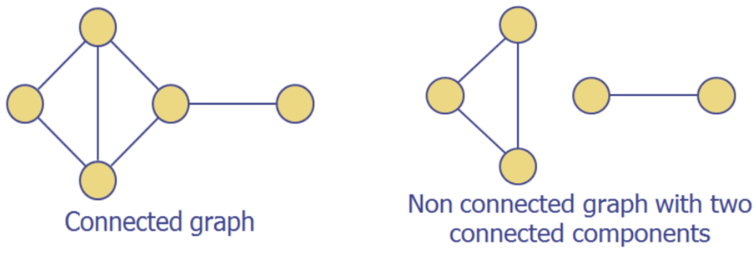
Graphs

In this assignment, you will implement an undirected, weighted graph and several methods related to graphs. In addition to the basic methods for the graph (adding nodes and edges, getting a list of nodes, etc.), you will implement the following methods:

- `minSpanningTree` - return the list of edges in a minimum spanning tree of the graph. If there are multiple minimum spanning trees, you may return any of them.
- `minSpanningTreeWeight` - return the weight of a minimum spanning tree of the graph
- `areUVConnected` - check if there is a path between nodes u and v
- `isConnected` - check if the graph is connected (meaning there is only one connected component)
- `numConnectedComponents` - return the number of connected components of the graph
- `dijkstra` - run Dijkstra's algorithm on the graph with the specified source node
- `shortestPathLength` - return the shortest path between the specified nodes

Definition

As you learned in the lecture, a connected graph is a graph where every pair of nodes has a path between them. Note that an isolated node (a node with no adjacent nodes) is a connected component by itself.



Implementation Notes

- We will always use valid graphs for each problem, so, in general, you do not have to worry about edge cases from invalid graphs.
- We will only be adding nodes and edges to the graph, so you do not have to worry about removing and relabeling the nodes.
- You do not have to worry about edge cases where the graph is empty.
- We will only have non-negative (≥ 0) integers as edge weights, so you can always run Dijkstra's algorithm.
- You do not need to consider the case of adding an edge with an endpoint that does not exist in the graph.
- You can use the instance variable `INFINITY` to represent ∞. We will not have test cases where the calculation overflows Java's `int`.
- Be sure to follow the guidelines on labeling and arranging the nodes in the `Graph.nodes` instance variable (see the `addNode` docstring).

- The adjacency list is of type `HashMap<Node, HashMap<Node, Edge>>`, which means if you call `adjacencyList.get(u)`, it will return an object of type `HashMap<Node, Edge>`. To get the edge between nodes u and v (if it exists), you can write `adjacencyList.get(u).get(v)`.
- We imported `HashMap, PriorityQueue, ArrayList, HashSet` and `Iterator` for you to use. You may not import anything else.
  - ArrayList: https://beginnersbook.com/2013/12/java-arraylist/
  - HashMap: https://beginnersbook.com/2013/12/hashmap-in-java-with-example/
  - Priority Queue: https://www.tutorialspoint.com/java/util/priorityqueue_super.htm
  - HashSet: https://beginnersbook.com/2013/12/hashset-class-in-java-with-example/
  - Iterating over a HashSet: https://beginnersbook.com/2014/08/how-to-iterate-over-a-sethashset/
- We have already implemented `Node` and `Edge` for you (and `Edge`'s comparison method is already done), so you do not need to worry about those. You can insert `Edges` into Priority Queues without modifying anything.

Directions

- Write your name and student ID number in the comment at the top of the Graph.java file.
- Implement all of the required functions in the Graph class.
- You are allowed to add any instance variables and methods that you want. However, you may not change any existing ones.
- Pay careful attention to the required return types.

General Directions

- Write your name and student ID number in the comment at the top of the java files.
- Implement all of the required functions in the java files.
- You should not import anything that is not already included in the file.
- Pay careful attention to the required return types and edge cases.

Submission Procedure

To submit your homework, simply drag and drop each file individually into the attachment box on the YSCEC assignment page. You should **NOT** zip them or place them in any folder. For this assignment, you should submit only the following files:

- Graph.java
- ⟨student_id⟩.txt

Inside of the ⟨student_id⟩.txt file, you should include the following text, and write your name at the bottom.

In completing this assignment, I pledge that I have not given nor received any unauthorized assistance.

If this file is missing, you will get a 0 on the assignment. It should be named *exactly* your student id, with no other text. For example, `2017123456.txt` is correct while something like `2017123456_hw1.txt` will receive a 0.

Testing

We have provided a small test suite for you to check your code. You can test your code by compiling and running the tester program. You will always get a notification if you fail any of the tests, but you may not get a notification if you passed.

Note that the test suite we use to grade your code will be much more rigorous than the one provided here (and not necessarily a superset of the provided tests). You should consider making your own test cases to check your code more thoroughly.