CIT 594 Module 8 Programming Assignment

As we saw in this module, graphs allow us to represent relationships between pieces of data that we want to store in a data structure; however, there is no graph implementation in the Java Collections Framework. In this assignment, you will modify a graph implementation and write methods used for analyzing the graph.

Learning Objectives

In completing this assignment, you will:

- Become more familiar with the "adjacency list" representation of a graph
- Apply what you have learned about how to traverse a graph
- Demonstrate that you can use graphs to solve common problems in Computer Science

Getting Started

Begin by downloading **the starter code** for this assignment.

This zip file includes the following **.java** files: **Graph**, **UndirectedGraph**, **DirectedGraph**, and **Edge**. These files include the implementations for the adjacency list representation of a graph (undirected and directed) along with breadth-first search and depth-first search implementations that we saw in the lessons.

We have also provided **GraphBuilder.java**, which includes static methods for generating directed and undirected graphs from an input file. These methods assume that the input file is formatted as follows:

- each line of the file consists of the values/labels of two nodes in the graph, separated by a single whitespace
- there is an edge in the graph between the two nodes; if the graph is directed, the edge is directed from the first node to the second

For instance, if the input file looked like this:

```
cat dog
dog platypus
```

and the graph were directed, there would be an edge from “cat” to “dog,” and from “dog” to “platypus.”

We have provided a file **graph_builder_test.txt** that you can use as input for testing the methods that you will implement in this assignment. You are, of course, free to create your own input files, but this is the file we will use to assess your solution. If you are using Eclipse, be sure to put this file in the root directory of your project.

Last, **GraphUtils.java** contains the unimplemented methods for the code that you will write in this assignment.

Activity

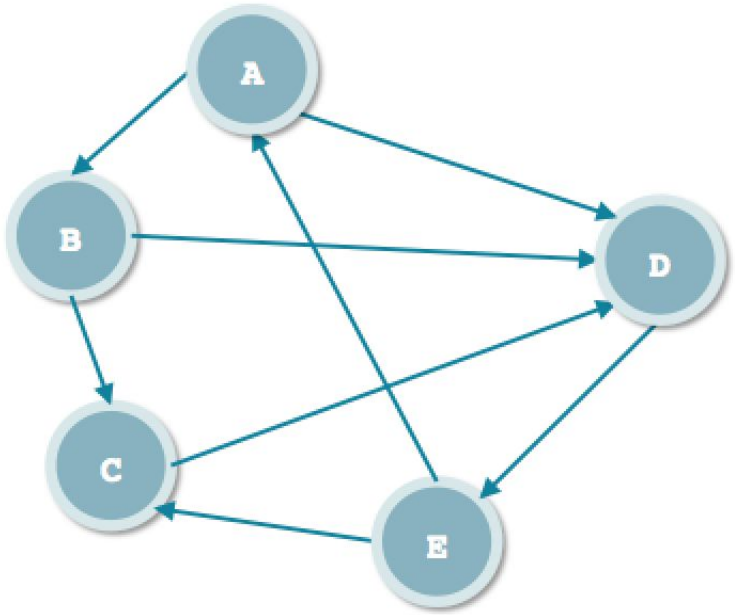
Implement the following methods in the **GraphUtils.java** file:

minDistance: Given a Graph, this method returns the shortest distance (in terms of number of edges) from the node labeled *src* to the node labeled *dest*. The method should return -1 for any invalid inputs, including: null values for the Graph, *src*, or *dest*; there is no node labeled *src* or *dest* in the graph; there is no path from *src* to *dest*. Keep in mind that this method does not just return the distance of **any** path from *src* to *dest*, it must be the **shortest** path.

nodesWithinDistance: Given a Graph, this method returns a Set of the values of all nodes within the specified distance (in terms of number of edges) of the node labeled *src*, i.e. for which the minimum number of edges from *src* to that node is less than or equal to the specified distance. The value of the node itself should **not** be in the Set, even if there is an edge from the node to itself. The method should return null for any invalid inputs, including: null values for the Graph or *src*; there is no node labeled *src* in the graph; distance less than 1. However, if distance is greater than or equal to 1 and there are no nodes within that distance (meaning: *src* is the only node in the graph), the method should return an empty Set.

isHamiltonianPath: Given a Graph, this method indicates whether the List of node values represents a Hamiltonian Path. A Hamiltonian Path is a valid path through the graph in which every node in the graph is visited exactly once, except for the start and end nodes, which are the same, so that it is a cycle. If the values in the input List represent a Hamiltonian Path given the order in which they appear in the List, the method should return true, but the method should return false otherwise, e.g. if the path is not a cycle, if some nodes are not visited, if some nodes are visited more than once, if some values do not have corresponding nodes in the graph, if the input is not a valid path (i.e., there is a sequence of nodes in the List that are not connected by an edge), etc. The method should also return false if the input Graph or List is null.

For example, consider the following directed graph:



The *isHamilitonianPath* method should return true for an input List consisting of {A, B, C, D, E, A} in that order, since:

- It is a valid path
- It covers every node in the graph
- It is a cycle

The following are examples of input Lists that should cause *isHamiltonianPath* to return false:

Input List	**Reason**
{ D, E, A, B, D }	Does not cover node C
{ A, B, D, E, C, A }	There is no edge from C to A (the last pair of nodes in the List)
{ A, B, C, D, E }	This is not a cycle because the first and last nodes are not the same.

For all of the above methods, be sure to test your implementation with **both** directed **and** undirected graphs.

Please do not change the signature of any of the three methods, and please do not create any additional .java files for your solution; if you need additional classes, you can define them in *GraphUtils.java*. You may modify the code we distributed, but do not create any other files. Last, please be sure that all code is in the default package, i.e. there is no "package" declaration at the top of the source code.

Helpful Hints

Resist the urge to look online for solutions! We are not asking you to derive any new algorithms or to "discover" something not covered in the lessons. Rather, you should try to apply the graph representations and traversal techniques that you have already learned.

You are free to use or even modify the BFS and DFS implementations in the *Graph* class as needed for this assignment. You may find that you can simply use one implementation or the other without modification, or that there may be some implementation that you don't need to use at all. These are just references for you to consider while writing these methods.

Before You Submit

Please be sure that:

- all classes are in the default package, i.e. there is no "package" declaration at the top of the source code
- your classes compile and you have not changed the signature of any of the three methods in the *GraphUtils* class
- you have not created any additional .java files

How to Submit

After you have finished implementing the *GraphUtils* class, go to the "Module 8 Programming Assignment Submission" item and click the "Open Tool" button to go to the Codio platform.

Once you are logged into Codio, read the submission instructions in the README file. Be sure you upload your code to the "submit" folder.

To test your code before submitting, click the "Run Test Cases" button in the Codio toolbar.

As in the previous assignment, **this will run some but not all of the tests that are used to grade this assignment.** That is, there **are** "hidden tests" on this assignment!

The test cases we provide here are "sanity check" tests to make sure that you have the basic functionality working correctly, but **it is up to you to ensure that your code satisfies all of the requirements described in this document.** Just because your code passes all the tests when you click "Run Test Cases" doesn't mean you'd get 100% if you submit the code for grading!

When you click “Run Test Cases,” you’ll see quite a bit of output, even if all tests pass, but at the bottom of the output you will see the number of successful test cases and the number of failed test cases.

You can see the name and error messages of any failing test cases by scrolling up a little to the “Failures” section.

Assessment

This assignment is scored out of a total of 82 points.

The **minDistance** method is worth a total of 27 points, based on whether it correctly calculates the number of edges between the two nodes, and whether it correctly handles errors.

The **nodesWithinDistance** method is worth a total of 19 points, based on whether it correctly returns a Set of nodes within the specified distance, and whether it correctly handles errors.

The **isHamiltonianPath** method is worth a total of 36 points, based on whether it correctly determines whether a path is Hamiltonian in both directed and undirected graphs, and whether it correctly handles errors.