CIT 594 Module 8 Programming Assignment
Graphs

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 are provided an implementation of directed graphs and simple undirected graphs, both of which implement a uniform interface provided in an abstract *Graph* class. You will write methods for analyzing this graph implementation using the traversal techniques you have learned.

Learning Objectives

In completing this assignment, you will:

- Become more familiar with the "adjacency set" 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 zip file for this assignment.

The zip file includes the following files inside the **src** folder:

- **Graph.java**, **UndirectedGraph.java**, and **DirectedGraph.java**: the implementations for the adjacency set representation of a graph (undirected and directed) along with breadth-first search and depth-first search implementations that we saw in the lessons
- **GraphUtils.java**: contains the unimplemented methods for the code that you will write in this assignment
- **GraphBuilder.java**: includes static methods for generating directed and undirected graphs from an input file

Outside the **src** folder, you will find **student_graph_test.txt**, which is a sample graph file that you can use as input for testing the methods that you will implement in this assignment. You are encouraged to create your own test input files; see the FAQ for how to do this. The sample file is the same as what is used in the visible test cases on Codio. The hidden test cases contain hidden graph input files.

Activity

Implement the following specifications in the **GraphUtils.java** file.

Your implementation must work for both directed *and* undirected graphs. Do not change the signature of any of the three methods, and do not create any additional .java files for your solution; if you need additional classes or methods, you must define them in **GraphUtils.java**. You may use portions of the provided code to help write your solution and helper methods. Last, be sure that all code is in the default package, i.e. there is no "package" declaration at the top of the source code.

You may not mutate any graph or list that is given to you as input. The test suite will throw an `UnsupportedOperationException` if you attempt to update, add, or remove elements from any graph or list.

`static int minDistance(Graph graph, String src, String dest)`

Given a *graph*, this method returns the smallest number of edges from the *src* node to the *dest* node, or 0 when $src = dest$, or -1 for any invalid input. Invalid inputs are defined as: any of *graph*, *src*, or *dest* is null; no path exists from *src* to *dest*; any of *src* or *dest* do not exist in *graph*.

`static Set<String> nodesWithinDistance(Graph graph, String src, int distance)`

Given a *graph*, a *src* node contained in *graph*, and a *distance* of at least 1, this method returns the set of all nodes, excluding *src*, for which the smallest number of edges from *src* to each node is less than or equal to *distance*; null is returned if there is any invalid input. Invalid inputs are defined as: any of *graph* or *src* is null; *src* is not in *graph*; *distance* is less than 1.

`static boolean isHamiltonianCycle(Graph g, List<String> values)`

This method returns true only if:

- The graph *g* is non-null
- *g* has at least three nodes
- *values* is non-null
- *values* represents a Hamiltonian cycle through *g*
- *values* is given as a sequence of vertices ending in the starting node of the cycle

Otherwise, return false. See the definitions below, as well as the FAQ for any further clarifications if needed.

Related definitions

For defining this problem, we borrow these definitions from CIT 592/596:

- A walk of a graph is a non-empty sequence of vertices consecutively linked by edges.
- A path is a walk of a graph in which all vertices are distinct.
- A cycle is a path through a graph in which all edges are distinct but the first vertex is also the last vertex.
- A simple cycle is a cycle where no vertex is repeated except the first, which is only repeated once.

- A Hamiltonian path is a path through a graph that visits each vertex in the graph exactly once.
- **A Hamiltonian cycle is a Hamiltonian path that is a simple cycle.**

Before You Submit

Please be sure that:

- your *GraphUtils* class is in the default package, i.e. there is no "package" declaration at the top of the source code
- your *GraphUtils* class compiles and you have not changed the signatures of the three required methods
- you did not overload any exposed methods
- you have not created any additional .java files
- you have filled out the required academic integrity signature in the comment block at the top of your submission file

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 <u>some but not all</u> 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.

You must manually submit when you are done. Your code will not be automatically submitted at the deadline.

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

Optional Challenges

Extra tasks for your understanding, discussion, practice and a little fun. These **will not be graded**.

Re-write BFS and DFS

Re-write BFS to use a recursive style. Re-write DFS to use an iterative style. Hint: these functions can be written to look identical except for one line of code.

Remember from m6pa's FAQ that iteration and recursion are "theoretically equivalent", however in practice, iteration is preferable in languages like Java and Python since their implementations lack support for tail recursion.

Big-O runtime

For each method you implement, add a Javadoc comment containing the asymptotic runtime (in Big-O) of your implementation and whether or not you think it's optimal. Use "n" for the number of nodes and "m" for the number of edges as appropriate.

Minimal Hamiltonian cycle for weighted graph

Add a comment to **isHamiltonianCycle** to indicate what you think would be the complexity of an optimal **isMinimalHamiltonianCycle** implementation for a weighted graph. Would such a function be useful in this assignment given that the graph implementation does not support weights?

Implement: `static Set<List<String>> allHamiltonianCycles(Graph g)`

This method returns the set of all Hamiltonian cycles that exist in the given graph g.

Frequently Asked Questions

May I modify the other classes in the starter code?

You will *only* be able to submit the **GraphUtils.java** file; the other files are already pre-loaded into Codio. While you can of course modify them for fun, experimentation, or testing purposes, your implementation must not rely on such modifications.

I see that `bfs` and `dfs` are provided in the Graph class. Should I use those directly?

Those are provided for reference only, so if you want to use one, you should copy it into your *GraphUtils*. 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.

Where can I find the algorithms to solve these problems?

We are not asking you to derive any new algorithms or to "discover" something not covered in the lessons. Correct and elegant implementations of the assigned methods are simple and straightforward. They rely only on a solid understanding of graph traversal. Looking online for solutions would deprive you of the opportunity to reach this understanding.

Are all values in the graph unique?

Yes. Note that this graph implementation of graphs uses a `Map` from a node to that node's set of neighbor nodes; `Map` enforces uniqueness of its keys.

Can the given graph contain self-loops?

We use the definitions of graphs that were given in CIT 592, in which (undirected) graphs are not permitted to contain self-loops, while directed graphs are.

See also the provided implementations of `addEdge` in *DirectedGraph* and *UndirectedGraph*.

`nodesWithinDistance`: what if there is a self-loop from *src* to *src* in a directed graph?

Note that we said the output should exclude *src*.

`nodesWithinDistance`: what if inputs are all valid, but no nodes found within *distance*?

You should not consider this to be a special case. You should always return a non-null `Set` when the input is valid, even if that set ends up containing nothing.

I don't understand the Hamiltonian cycle problem.

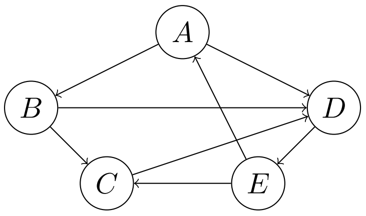
Some helpful hints:

- While the definitions speak of distinct edges, note that the input format is given as a list of <u>nodes</u>. So if you are given the list $[A, B, C, A]$, this represents the edges $A \to B$, $B \to C$, and $C \to A$. Thus the input list, if it's valid, would always have the first element repeated at the end, but all other elements would have to be unique.

- Think of the input list as a series of cities for a travel route, where you start at your home city, you want to visit all the cities in the region, and then you want to return home at the end.

  If you fail to visit any cities, or you ever *depart from* any city more than once, or if you don't make it home at the end, then it's not a Hamiltonian cycle.

As an example, consider the following directed graph:



Then `isHamilitonianCycle` should return true for an input list consisting of $[A, B, C, D, E, A]$ in that order, since:

- It is a valid path
- The path is Hamiltonian, since it covers every node in the graph
- It is a cycle

The following are examples of input lists that should cause `isHamiltonianCycle` 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 $C \to A$ (the last pair of nodes in the list)
$[A, B, C, D, E]$	This does not use our input format, because the final edge $E \to A$ is missing
$[A, B, A]$	We require that the graph have at least three nodes, so an input list with only 2 nodes can't be correct

Why do you keep saying "adjacency set" rather than "adjacency list"?

Graph theory typically calls them adjacency lists, but using a set to represent the adjacency list offers better expected performance since set membership is expected $O(1)$ for all practical purposes.

How can I create my own graph test files?

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, the construction of a directed graph from this input file:

```
cat dog
dog platypus
```

would produce a graph with two edges; one from “cat” to “dog,” and one from “dog” to “platypus.”

How can I create my own graphs in code for testing purposes?

Here’s how you could create an undirected graph then add one node to it:

```java
Graph g = new UndirectedGraph(); // or DirectedGraph
g.addNode("a");
```

More commonly though you would use `addEdge(String src, String dest)`, which implicitly creates both nodes if they don’t already exist.

Recall that we require that your methods do not modify the input data in any way. If you want to write unit tests that exhibit the same behavior as the autograder – which is to throw an exception when your code incorrectly modifies the input data) – for graph g declared above you could write: `g.makeUnmodifiable();`

If you want to create unmodifiable string lists for testing `isHamiltonianCycle`, you can use `List.of` (Java 9+ only) which takes any number of arguments and returns an *unmodifiable* list:

```java
List<String> myList = List.of("hello", "world");
```

Attempting to call `myList.add("!")` would throw `UnsupportedOperationException`.

How can I use my classes in jshell?

On the commandline terminal, after navigating to the same folder where your files are located, you can write:

```
jshell Graph.java DirectedGraph.java UndirectedGraph.java GraphBuilder.java
    GraphUtils.java
```

The reason it has to be written that way is that the java files are processed in the order they're given, so a class like Graph has to come first since other classes reference it.

If you've changed your java files on disk but you don't want to lose your place in the jshell session, you can quickly import all your changes by entering: `/reload`