In this project, we will write the code and you will write the test cases with JUnit.

1 Graphs

You will be testing a package that implements simple graphs and operations on them\(^1\). Graphs are made up of vertices and edges connecting pairs of vertices. Your task is to write JUnit test cases to check whether a particular implementation meets our specification.

You can find out everything you need to know about graphs from the specification. There are two interfaces in our graph package: Graph and Vertex. Graphs contain a method newVertex and newEdge to create vertices and edges. Each vertex in a graph is identified by a label. Within a single graph, all vertices must have unique labels. Vertices are associated with a particular graph; directly comparing Vertex objects from different graphs is undefined.

Graph and Vertex are interfaces. Thus they cannot contain constructor methods. For Vertex this is fine, since we have a newVertex method. For making new Graphs, we use the factory design pattern. We have created a class GraphFactory with a makeEmptyGraph method that produces a new graph.

Graphs contain several standard operations, such as computing the set of all in (or predecessor) and out (or successor) edges of a vertex. Graphs also contain routines to find paths in the graph, to merge vertices in various ways, and to check whether one graph is contained in another graph. To make the latter tractable, we use vertex labels to match up nodes when checking for containment. For example, consider the following three graphs:

The vertices in these graphs are labeled A, B, C, D, E, and F, meaning, for example, that graph (a) was created with the commands

```java
Graph a = GraphFactory.makeEmptyGraph();
Vertex v1 = a.newVertex("A");
Vertex v2 = a.newVertex("B");
Vertex v3 = a.newVertex("C");
a.addEdge(v1, v2);
a.addEdge(v1, v3);
```

Containment is computed by comparing labels of vertices in different graphs. Thus, for example, graph (a) contains graph (b): All of the vertex labels in (b) are present in (a), and the edge from A to C in (b) is also present in graph (a). On the other hand, (b) does not contain (a). Furthermore, two vertices in different graphs are equivalent only if they have the same label. Thus, even though (a) and (c) have the same shape, it is not the case that (a) and (b) are equivalent.

### 1.1 Dot

Graphs may be output in the *dot* textual format (dot is part of a graph visualization package from AT&T). For example, in dot format, graph (a) could look like

```dot
digraph G {
    A -> B;
    A -> C;
}
```

In dot representation, vertex names can be enclosed in double quotes if they contain spaces (and quotes within vertex names should be escaped with a backslash). Vertices can appear multiple times, either by themselves (which just denotes that the vertex exists) or as part of an edge. Note that you don’t need to download or even use dot for this project—you just need to understand the relevant portion of their file format—but you can if you’d like.

---

Here’s a grammar of the subset of dot that we support:

```plaintext
vertex ::= identifier
graph ::= digraph identifier { stmts }
stmts ::= stmt stmts | stmt
stmt ::= vertex ; | vertex -> vertex ;
```

Here, an `identifier` is

- any string of alphabetic characters, underscores or digits, but not beginning with a digit; or
- a number; or
- any double-quoted string ("...") possibly containing escaped quotes \

2 Writing JUnit Tests

The deliverable for this assignment will be a single class `GraphTestCase` that extends `junit.framework.TestCase`. Inside this class you will write JUnit test cases to test implementations of the Graph specification. Here is a skeleton for the class you will write:

```java
import junit.framework.*;

public class GraphTestCase extends TestCase {
    public GraphTestCase() { }
    public void testCase1() { ... }
    public void testCase2() { ... }
    ...
}
```

This class will contain one no-argument, void-returning method for each test that you write, and each method’s name must begin with the word `test`; above, we have indicated this by defining methods `testCase1`, `testCase2`, etc. which are presumed to contain tests. The class constructor need not do anything.

Note that JUnit, version 3.8.1, is installed on the Linuxlab machines in `/usr/local/junit3.8.1/`. This is added to your classpath by default when you log in, so you should be to start using it immediately. If you want to write your project at home, you’ll have to download JUnit v3.8.1.

2.1 Test cases for Graph

To facilitate your testing, we’ve supplied you with a working implementation of graphs. However, remember that this is only one implementation. Your test cases should be designed to find bugs in any implementation of the specification,

not just this one. Obviously you won’t be able to find bugs in this implementation if it is correct; however, it should help you understand how things might be implemented, and therefore give you more ideas for tests. In particular, you can try to write glass box tests in the ways described in class (e.g. statement coverage, branch coverage, and/or condition coverage).

You should write tests that, as completely as possible, check the compliance of an implementation of Graph with its specification. To get you started, we will describe 10 tests that you should write. We will run your tests against implementations that have 15 different, independent bugs. Ten of these bugs will be found by a correct implementation of the tests we describe next, while the remaining bugs must be caught by other tests that you will write.

Each test should be implemented in its own method; that is, each bullet item below should be a single method in your GraphTestCase class. You are free to define other methods as necessary, e.g. setUp or tearDown (make sure you spell these with the correct capitalization).

1. For some graph \( g \), verify that when a new vertex is added to the graph using \( g.\text{newVertex}(s) \), then it is contained in the graph’s vertices. Moreover, make sure that if a vertex with label \( s \) already exists in the graph, then an additional one is not added.

2. For some graph \( g \), which contains a vertex \( a \), verify that when \( a \) is removed from \( g \), none of the edges that mention \( a \) remain in \( g \), and \( a \) no longer remains in the vertices of \( g \). Try this for a vertex that has one incoming edge, for a vertex that has one outgoing edge, and for a vertex with two or more incoming or outgoing edges.

3. For some graph \( g \), make sure that the path returned by \( g.\text{findPath}(a,b) \), for some vertices \( a \) and \( b \) in \( g \), is correct. Test paths of length 0, 1, and 2.

4. For some graph \( g \), which contains two vertices \( a \) and \( b \), verify that \( g.\text{mergeVertices}(a,b,"A") \) works when "A" is the label of either \( a \) or \( b \). For input, use a graph with two vertices and no edges.

5. For some graph \( g \), verify that \( g.\text{mergeVertices}(v_1,v_2,n) \) works when \( v_1 == v_2 \). Try this on a vertex that has 0 edges and one that has 1 edge.

6. Verify that \( g.\text{contains}(g_2) \) works for some cases. Consider as input the case when \( g \) and \( g_2 \) are empty graphs. Also verify that it works when \( g \) and \( g_2 \) are the same graph object, and for a few graphs of at least two vertices and edges.

7. For some graph \( g \), verify that the set \( s \) returned by \( g.\text{reachableFrom}(a) \) is correct. Try this for graphs in which \( a \) has no successors, in which \( a \) has one or more successors immediately adjacent to it, and in which \( a \) has nodes reachable from it via paths of at least length 2.

(Hint: you can use \( \text{findPath} \) to check the correctness of \( \text{reachableFrom} \) for arbitrary input graphs. Of course, you don’t have to make use of this insight.)
8. When adding some edge \((a,b)\) to a graph \(g\) (which already contains the vertices \(a\) and \(b\)), verify that \(a\) is added to the predecessors of \(b\), and \(b\) is added to the successors of \(a\) (if not already present). Try this for cases when \(a\) and \(b\) are different vertices, when they are the same vertex, and when the edge already is present in the graph.

9. For some graph \(g\), verify that the dot representation of \(g\) output by \texttt{GraphFactory} works properly. For input, use graphs with no vertices, one vertex, and three vertices with one of the vertices having no edges.

(\textit{Hint:} an easy way to test this without having to write a textual parser for the dot documents is to use the \texttt{GraphFactory.read()} method parse in the dot document that was output, and then check that. This technique assumes that \texttt{GraphFactory.read()} works properly, which you can assume since you will be testing it elsewhere.)

10. Verify that \texttt{GraphFactory.read()} creates correct graphs. As input, create strings representing dot documents, and create a \texttt{StringReader} object for each, passing it to \texttt{GraphFactory.read()}. Try this on input graphs that are empty, on graphs with one or more vertices with no edges, and on graphs that attempt to define the same edge twice (these graphs should fail).

Now you should write test cases for functionality that the above cases do not cover. You do not have to write additional cases for \texttt{GraphFactory} methods (i.e. \texttt{read} and \texttt{write}); just focus on methods from \texttt{Graph}.

## 3 Final Notes

We will use an automated system to test your tests, so please make sure your project compiles in the context of the provided code. Also, when writing your tests, do not read input from \texttt{System.in} or write any output to \texttt{System.out}; if you want to test \texttt{GraphFactory.read()}, you should use a \texttt{StringReader} object to read input from a constant string, and you can use a \texttt{StringWriter} object to send the output of \texttt{GraphFactory.write()} to a string. Your tests should work for \texttt{any} implementation of \texttt{Graph}; therefore, make sure that your tests only refer to \texttt{Graph} objects, not \texttt{BillGraph} objects (which is the implementation we provide). Finally, make sure that you write tests to be reasonably small; use the provided tests as a guideline. This will ensure you get the most credit when grading.

To submit this project, use a command like

```
java -jar ~/Submit.jar 2 GraphTestCase.java
```

Do not submit any of the Graph files that we have provided you; just submit your testcases. If you have testing files other than \texttt{GraphTestCase.java}, you can submit these.
4 Updates to this assignment

Summary of changes:

(9/22/2003)

- Due to University closing on Thursday (9/18) and Friday (9/19), we have moved the due date to Friday, October 3, at 6pm (prior due date of October 2 was a typo—should have been Wednesday, October 1).