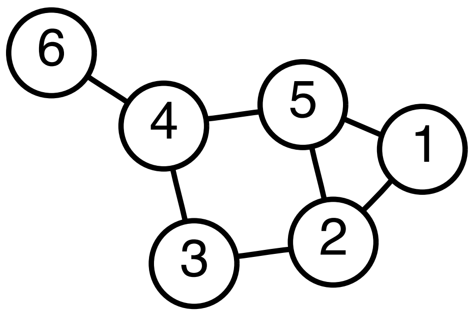
CMPSCI 187, Spring 2015 Discussion #11: Unit Testing a Graph Package: Handout

Marc Liberatore and John Ridgway

13 April 2015

Graphs are a powerful abstraction which represent a set of objects (called *vertices*, or nodes), and whether a connection (an *edge*) exists between any two vertices.

For example, here is a simple graph consisting of six vertices. Each vertex in this graph is labeled with a number, though vertex labels in general need not be numeric. Some pairs of vertices in this graph are *adjacent*, that is, an edge exist between them. For example, 6 and 4 are adjacent, but 6 and 3 are not.



For this assignment, we have provided you with a Java interface for a simple graph like this one. We have also provided you with two not-entirely-correct implementations of this interface. The code is available at `http://www-edlab.cs.umass.edu/cs187/discussions.html` in the form of an Eclipse project.

The code is missing something important: a complete set of *unit tests*. Your job is to write unit tests for several of the method of the classes, detecting errors in the implementations. Let's look at a simple example of a unit test line-by-line.

```java
import static org.junit.Assert.*;
import org.junit.Test;

public class ExampleTest {
  @Test
  public void testStringToUpperCase() {
    String x = "FooBar";
    assertEquals("FOOBAR", x.toUpperCase());
  }
}
```

`import static` `org.junit.Assert.*;` imports a large number of static (class) methods from the `Assert` class. In this example, we use only `assertEquals()`, but you can view the complete list and associated documentation at `http://junit.org/javadoc/latest/org/junit/Assert.html`. You're likely to find `assertEquals()`, `assertTrue()`, and `assertFalse()` useful, among others.

`import` `org.junit.Test;` imports the `Test` class, which is used as an *annotation* on methods.

`public class` `ExampleTest` declares the class. Like all code in Java, unit tests need to be contained in a class. By convention, classes containing only unit tests end with "Test", but that is not required.

`@Test` is the aforementioned use of an annotation. Annotations are metadata: they provide information about the program, but aren't part of the program itself. The JUnit framework requires that you mark methods that perform tests with the `@Test`. Then, when you use a tool that support JUnit (like Eclipse, or another IDE, or an automated build tool like Ant or Maven), the tests can be automatically recognized and run by your tool. This is why you can click the “Run” triangle in Eclipse when viewing a file with unit tests and have it recognize and run the tests.

The `@Test` annotation is also where you can tell JUnit that the expected result of a a test is an exception. For example, if the expected and correct result of a test is to throw a `NullPointerException`, you’d write `@Test(expected = NullPointerException.class)`, and not include any assertions in the body of the test itself. You must name the exception using its `.class` member as shown for this to work.

`public void testStringToUpper()` is a method that performs a unit test. By convention, these methods’ names start with “test” and describe what behavior or method they are testing. Each unit test should test a single behavior, not all possible behaviors of a method. And remember a common gotcha: starting a method name with “test” won’t make it a test — you must also include the `@Test` annotation.

`String x = "FooBar";` declares and initializes a string `x`. Unit tests can create objects or do anything else that can normally be done in Java. For example, test classes can declare instance variables that can be used in tests; you’ll see that in the incomplete unit tests in the Eclipse project.

`assertEquals("FOOBAR", x.toUpperCase());` is the crux of the test. Unit tests usually contain one (sometimes more) `assertX()` method calls (also known as test assertions), where `X` describes the type of testing being done (usually equality of the `.equals()` variety). The first argument to a test assertion is the expected value; the second is usually the expression being tested. Here, our assertion expects the result of `x.toUpperCase` to be equivalent to `"FOOBAR"`. Using the `assertX()` signals success or failure to the tool using JUnit in a standard way. The result of these `assertX()` calls are passed to the tool — when you run unit tests in Eclipse, these results are how it determines whether tests passed or failed.

Question 1: Write six (or more) short unit tests that demonstrate a mix of correct and incorrect behavior across the methods of `ArrayBasedGraph`. You don’t need to look at the `ArrayBasedGraph` implementation to write tests for it, though you can examine it if you wish.

In total, your tests should exercise all methods and as many distinct behaviors as you can think of. But each test should exercise as small a set of methods and behaviors as possible — the idea being that if a unit test fails, it should be easy to identify the small section of code (the unit) causing the failure. The name of each test should make clear what behavior it is testing. Indicate whether the test passes or fails with the code we provided you. (You may alter the code to fix errors you find, but do not introduce new errors for the purpose of trivializing finding failing tests!).

We’ve given you a head start in `ArrayBasedGraphTest`, which provides one passing and one failing test. Notice that the instance variable `graph` is declared. It’s initialized in the `before()` method, which is annotated with `@Before`. Something special happens with a method annotated with `@Before`: before *each* unit test (decorated with `@Test`) is run, *all* of the `@Before` methods are run (in an arbitrary order). This behavior allows us to have a single variable referring to an empty graph at the start of each test. Even if a test changes the graph, it is reset to an empty graph in the `@Before`-annotated `before` method, before the next test is run.

Question 2: Adapt your tests to `CollectionBasedGraph`. Again, you don’t need to look at or understand the implementation to write tests of the expected behaviors. Which pass and which fail?