CSCI241 Fall 2020: Lab 6
Due Sunday, November 15th, 9:59pm

1 Overview

In this lab, you will write a program to read an undirected, unweighted graph from a text file into an adjacency matrix and output the number of connected components in the graph.

2 Git and submission for Lab 6

The Github Classroom link for Lab 6 is available in the Lab 6 assignment on Canvas. There is no skeleton code, so your repository will start out empty. You will begin by initializing a gradle project in your repo, then you will implement a program to count connected components. Whether or not you finish during lab, make sure to commit and push whatever code you have at the end of the lab period to receive credit for attending lab.

3 Creating a Gradle project from scratch

In past assignments and labs, we've given you a skeleton repository with gradle all set up to run and test the code. In this lab, you'll set up a gradle project from scratch.

1. Start by cloning your (empty) repository.
2. Change into the repo directory and run `gradle init`. You will be prompted to answer some questions about the project you'd like to create:
   - Select `application` for project type. This will make a project with a `run` task that executes some class's `main` method.
   - Select `Java` for implementation language. This tells Gradle that this is a Java project.
   - Select `groovy` for DSL; specifies that the configuration file `build.gradle` will be written in the Groovy language, which is the default and what we've been using this quarter.
   - Select `JUnit 4` for test framework. Although you won't be required to write any tests, this sets up a `test` task that compiles and runs JUnit tests stored in the test source directory.
   - Name your project `lab6`
   - Use `lab6` also for the source package; this means your code lives in the lab6 package (i.e., has a `package lab6;` declaration at the top of each code file.

   Take a look around your project directory - the init task has created a bunch of stuff that should look familiar by now. To finish setting up your project like past labs and assignments we need to do a few more things.

3. For Java Application projects, gradle defaults to creating a class called App, stored in src/main/java/lab6/App.java. Rename this file to be called Components.java. Edit the file and replace the App class name with Components (and change the main method to create an instance of Components instead of App).

4. Likewise, Gradle auto-generates an JUnit test class called src/test/java/lab6/AppTest.java. We won't be asking you to write any JUnit tests here, so you can delete this file if you'd like. If you wanted to write some unit tests, you'd want to rename the file to ComponentsTest.java and write test cases in there.

5. The last thing we need to get something running is to tell Gradle that the name of the class whose main method we want to run with the `run` task is Components instead of App. Edit the last line of `build.gradle` to set `mainClassName` to `lab6.Components`.

6. At this point, try the `gradle run` task. You should see Hello world printed under the :run task.

7. When running programs with command-line input or output, it's often annoying to have gradle's progress bars and output get in the way. To make some of this go away, create a file called `gradle.properties` and include the following line:

```
org.gradle.console=plain
```

8. You can also control the amount of output (the log level) using command-line flags, such as `-q` for quiet (no output), `-i` for info (lots of output), and `-d` for debug (more output than you know what to do with).

9. Now seems like a great time to commit your changes to git! Because the repo started out empty, we need to add all the files to our repository. Be sure you `git add` the following to your repo:

   - `build.gradle`
   - `gradle.properties`
   - `settings.gradle`
   - `.gitignore` (this file tells git that you don't want to add the build directory and the hidden .gradle directory where gradle caches local state)
   - `src/main/java/lab6/Components.java`

   Gradle init also generated a *wrapper*, which allows your gradle tasks to be run on systems that don't have gradle installed. On linux, you use the `gradlew` script and on Windows you'd use `gradlew.bat`. These scripts make use of the .jar file in the `gradle` directory to run tasks without help from a local gradle installation. It's standard practice to include the necessary wrapper files in your git repository, so also `git add` the following:

   - `gradlew`
   - `gradlew.bat`
   - `gradle/wrapper/gradle-wrapper.jar`
   - `gradle/wrapper/gradle-wrapper.properties`

10. Here are a couple of other noteworthy changes that we've used in `build.gradle` for past assignments that are not relevant to this lab, but you may find helpful to know about:

    - If you want your program to accept user input from the command line, you need to tell gradle about this by adding something like the following to build.gradle:

      ```
      run {
        standardInput = System.in
      }
      ```

    - The following lines change the output when running tests to display anything printed by your program or test code and print the full stack trace for each test failure:

      ```
      test {
        testLogging {
          showStandardStreams = true;
          exceptionFormat 'full'
        }
      }
      ```

    - A great deal of additional customization is possible by specifying your preferences in `build.gradle`. You can learn more from the gradle documentation at `http://docs.gradle.org`.

4 Problem Specification

Implement your program in the Components class. The program takes a single command line argument specifying a filename. The given file adheres to the format specified below. The program should: read the file and create an adjacency matrix representation of the graph, then compute and print the number of connected components in the graph.

You are **not** required to implement error checking, e.g., for malformed input files or bad command line arguments.

4.1 Input File Specification

Nodes in this undirected graph are numbered sequentially from $0..|V|$. The first line of the file contains a single integer, specifying the number of $|V|$, and each remaining line in the file contains two integers indicating that an edge exists between the two nodes.

4.2 Example

Here's an example of the program's behavior. The graph drawing is only for illustrative purposes - your program does not need to parse or produce drawings.

```
Graph drawing:      input.txt:       Sample invocation:
  0       4         5                 $ gradle run -q --args "input.txt"
  |       |         0 1               2
  1 - 2   3         3 4               $
                    1 2
```

Note that "$" denotes the shell prompt, indicating that the program has printed "2" followed by a newline and then terminated.

4.3 Implementation Hints and Guidelines

In this lab, you are given no skeleton code and less specific guidance than usual. First focus on parsing input correctly; then get a correct implementation of the core algorithm; then use the algorithm to find connected components. Leave efficiency considerations until last: 8/10 points are based on correctness. Please comment your code well: if we cannot understand your implementation, we can't give full credit for it.

5 Submission

Be sure that all the requisite files listed in Section 3 are included in your repository. Then, simply commit your final changes and push to GitHub as usual.

Rubric

Setup: 2 points for correct gradle project setup
Correctness: 6 points for a correctly working program.
Efficiency: 2 points if runtime is $O(v^2)$.

Possible deductions:

- Up to -5: Code does not compile or generates runtime errors when given valid input.
- Up to -5: Inadequate commenting.
- Up to -2 per violation: Other coding style issues