CS112 Spring 2015: Programming Assignment 5

Friendship Graph Algorithms

Posted Fri, Apr 17. Due Fri, May 1, 11:00 PM - NO EXTENSIONS

Worth 50 points = 5% of your course grade

In this assignment, you will implement some useful algorithms that apply to friendship graphs of the Facebook kind.

You will work on this assignment in PAIRS. INDIVIDUAL SUBMISSIONS WILL NOT BE ACCEPTED.

Read DCS Academic Integrity Policy for Programming Assignments - you are responsible for abiding by the policy. In particular, note that "**All Violations of the Academic Integrity Policy will be reported by the instructor to the appropriate Dean**". (For this assignment, each *pair* of partners must write their own code.)

There will NOT be an extension pass option for this assignment.

Background

In this program, you will implement some useful algorithms for graphs that represent friendships (e.g. Facebook). A friendship graph is an undirected graph without any weights on the edges. It is a simple graph because there are no self loops (a self loop is an edge from a vertex to itself), or multiple edges (a multiple edge means more than edge between a pair of vertices).

The vertices in the graphs for this assignment represent two kinds of people: students and non-students. Each vertex will store the name of the person. If the person is a student, the name of the school will also be stored.

Here's a sample friendship graph:

```
   (sam,rutgers)---(jane,rutgers)-----(bob,rutgers)   (sergei,rutgers)
                         |                  |              |
                         |                  |              |
                   (kaitlin,rutgers)   (samir)----(aparna,rutgers)
                         |                            |
                         |                            |
(ming,penn state)----(nick,penn state)----(ricardo,penn state)
                         |
                         |
                   (heather,penn state)


                 (michele,cornell)----(rachel)
                         |
                         |
   (rich,ucla)---(tom,ucla)
```

Note that the graph may not be connected, as seen in this example in which there are two "islands" or cliques that are not connected to each other by any edge. Also see that all the vertices represent students with names of schools, except for `rachel` and `samir`, who are not students.

Algorithms

1. **Shortest path: Intro Chain**

   sam wants an intro to aparna through friends and friends of friends. There are two possible chains of intros:

   ```
   sam--jane--kaitlin--nick--ricardo--aparna

             or

   sam--jane--bob--samir--aparna
   ```

   The second chain is preferable since it is shorter.

   If sam wants to be introduced to michele through a chain of friends, he is out of luck since there is no chain that leads from sam to michele in the graph.

   Note that this algorithm does NOT have any restriction on the composition of the vertices: a vertex along the shortest chain need NOT be a student at a particular school, or even a student. So, for instance, you may have to find the shortest intro chain from nick to samir, which has the following solution:

   ```
   nick--ricardo--aparana--samir
   ```

   which consists of two penn state students, one rutgers student, and one non-student.

2. **Cliques**

   Students tend to form cliques with their friends, which creates islands that do not connect with each other. If these cliques could be identified, particularly in the student population at a particular school, introductions could be made between people in different cliques to build larger networks of friendships at that school.

   In the sample graph, there are two island cliques for students at rutgers:

   ```
        (sam,rutgers)---(jane,rutgers)-----(bob,rutgers)    (sergei,rutgers)
                              |                                 |
                              |                                 |
                        (kaitlin,rutgers)             (aparna,rutgers)
   ```

   If we were to look at students at penn state, instead, there is a single clique:

   ```
     (ming,penn state)----(nick,penn state)----(ricardo,penn state)
                            |
                            |
                      (heather,penn state)
   ```

   And again, a single clique for students at ucla:

   ```
        (rich,ucla)---(tom,ucla)
   ```

   And one for students at cornell:

   ```
                (michele,cornell)
   ```

   From these examples, it should be clear that if there is at least one student in the graph that goes to a particular school, then there must be at least one island clique in the graph for students at that school.

3. **Connectors: Friends who keep friends together**

   If jane were to leave rutgers, sam would no longer be able to connect with anyone else--jane was the "connector" who could pull sam in to hang out with her other friends. Similarly, aparna is a connector, since without her, sergei would not be able to "reach" anyone else. (And there are more connectors in the graph...)

   On the other hand, samir is not a connector. Even if he were to leave, everyone else could still "reach" whoever they could when samir was there, even though they may have to go through a longer chain.

   Defintion: In an undirected graph, vertex v is a connector if there are at least two other vertices x and w for which *every* path between x and w goes through v.

For example, `v=jane`, `x=sam`, and `w=bob`.

Finding all connectors in an undirected graph can be done using DFS (depth-first search), by keeping track of two additional quantities for every vertex `v`. These are:

- `dfsnum(v)`: This is the dfs number, assigned when a vertex is visited, dealt out in increasing order.
- `back(v)`: This is a number that is initially assigned when a vertex is visited, and is equal to `dfsnum`, but can be changed later as follows:
  - When the DFS backs up from a neighbor, `w`, to `v`, if `dfsnum(v)` > `back(w)`, then `back(v)` is set to min(`back(v)`,`back(w)`)
  - If a neighbor, `w`, is already visited then `back(v)` is set to min(`back(v)`,`dfsnum(w)`)

When the DFS backs up from a neighbor, `w`, to `v`, if `dfsnum(v)` ≤ `back(w)`, then `v` is identified as a connector, IF `v` is NOT the starting point for the DFS.

If `v` is a starting point for DFS, it can be a connector, but another check must be made - see the examples below. The examples don't tell you how to identify such cases--you have to figure it out.

Here are some examples that show how this works.

- Example 1: (B is a connector)

```
A--B--C
```

The DFS starts at A. Neighbors for a vertex are stored in REVERSE alphabetical order:

```
A: B
B: C,A
C: B

dfs @ A  1/1  (dfsnum/back)
    dfs @ B 2/2
        dfs @ C 3/3
            neighbor B is already visited => C 3/2
        dfsnum(B) <= back(C) B is a CONNECTOR
        nbr A is already visited => B 2/1
    dfsnum(A) <= back(B) A is starting point of DFS, NOT connector in this case
```

- Example 2: (B is a connector)

```
A--B--C
```

The same example as the first, except DFS starts at B. This shows how even thought B is the starting point, it is still identified (correctly) as a connector. The code is not complete because it does not show HOW B is determined to be a connector in the last line - that's for you to figure out. Neighbors are stored in reverse alphabetical order as before.

```
dfs @ B  1/1
    dfs @ C 2/2
        nbr B is already visited => C 2/1
    dfsnum(B) <= back(C) B is starting point, NOT connector
    dfs @ A 3/3
        nbr B is already visited => A 3/1
    dfsnum(B) <= back(A) B is starting point, but is a CONNECTOR in this case
```

- Example 3: (B and D are connectors)

```
A---B---C
    |   |
    E---D---F
```

DFS starts at A. Neighbors stored in reverse alphabetical order again:

```
A: B
B: E,C,A
C: D,B
D: F,E,C
E: D,B
F: D

dfs @ A 1/1
    dfs @ B 2/2
        dfs @ E 3/3
            dfs @ D 4/4
                dfs @ F 5/5
                    nbr D is already visited => F 5/4
                dfsnum(D) <= back(F) => D is a CONNECTOR
                nbr E already visited => D 4/3
                dfs @ C 6/6
                    nbr D already visited => C 6/4
                    nbr B already visited => C 6/2
                dfsnum(D) > back(C) => D 4/2
            dfsnum(E) > back(D) => E 3/2
            nbr B is already visited => E 3/2
        dfsnum(B) <= back(E) => B is a CONNECTOR
        nbr C is already visited => B 2/2
        nbr A is already visited => B 2/1
    dfsnum(A) <= back(B) A is starting point, NOT a connector in this case
```

- Example 4: (B and D are connectors)

```
A---B---C
    |   |
    E---D---F
```

Same example as the previous, except DFS starts at D, and neighbors stored in alphabetical order. Connectors are still correctly identified as B and D.

```
A: B
B: A,C,E
C: B,D
D: C,E,F
E: B,D
F: D

dfs @ D 1/1
    dfs @ C 2/2
        dfs @ B 3/3
            dfs @ A 4/4
                nbr B is already visited => A 4/3
            dfsnum(B) <= back(A) => B is a CONNECTOR
            nbr C is already visited => B 3/2
            dfs @ E 5/5
                nbr B is already visited => E 5/3
                nbr D is already visited => E 5/1
            dfsnum(B) > back(E) => B 3/1
        dfsnum(C) > back(B) => C 2/1
        nbr D is already visited => C 2/1
    dfsnum(D) <= back(C) D is starting point, NOT connector
    dfs @ F 6/6
        nbr D is already visited => F 6/1
    dfsnum(D) <= back(F) D is starting point, is a CONNECTOR
```

Implementation

You will write a program called `Friends` that will read a graph file, build a graph (using the adjacency linked lists representation), and implement the ~~four~~ three algorithms described above. Details follow, including a user interface.

1. **(4 pts) User interface**

   - Your user interface should ask for the input graph file, build the graph, then spin on ~~five~~ four choices--shortest intro chain, cliques at school, connectors, and quit--until the user quits.

2. **(6 pts) Graph build**

   - Input: file that lists names of all people and edges between them. Here is the format of the input file for the sample friendship graph given in the Background section.

     ```
     15
     sam|y|rutgers
     jane|y|rutgers
     michele|y|cornell
     sergei|y|rutgers
     ricardo|y|penn state
     kaitlin|y|rutgers
     samir|n
     aparna|y|rutgers
     ming|y|penn state
     nick|y|penn state
     bob|y|rutgers
     heather|y|penn state
     rachel|n
     rich|y|ucla
     tom|y|ucla
     sam|jane
     jane|bob
     jane|kaitlin
     kaitlin|nick
     bob|samir
     sergei|aparna
     samir|aparna
     aparna|ricardo
     nick|ricardo
     ming|nick
     heather|nick
     michele|rachel
     michele|tom
     tom|rich
     ```

     The first line has the number of people in the graph (15 in this case).

     The next set of lines has information about the people in the graph, one line per person (15 lines in this example), with the '|' used to separate the fields.

     In each line, the first field is the name of the person. Names of people can have any character except '|', and are case *insensitive*. Also, names are unique over the entire graph. The second field is 'y' if the person is a student, and 'n' if not. The third field is only present for students, and is the name of the school the student attends. The name of a school can have any character except '|', and is case *insensitive*.

     The last set of lines, following the people information, lists the friendships between people, one friendship per line. Since friendship works both ways, any friendship is only listed once, and the order in which the names of the friends is listed does not matter.

   - Result: The ajacency linked list representation, along with a data structure to be able to quickly translate from a person's name to vertex number. There should also be a quick way to translate from vertex number to the person's name.

   - Output: Graph adjacency linked lists representation in your program.
     You don't have to print the representation, we will check your code to see if you are storing the graph correctly in this representation.

3. **(12 pts) Shortest path (Intro chain)**

   - Input: Name of person who wants the intro, and the name of the other person, e.g. "sam" and "aparna" for the graph in the Background section. (Neither of these, nor any of the intermediate people are required to be students, in the same school or otherwise.)

   - Result: The shortest chain of people in the graph starting at the first and ending at the second.

   - Output: Print the chain of people in the shortest path, for example:

     ```
     sam--jane-bob--samir--aparna
     ```

     If there is no way to get from the first person to the second person, then the output should be a message to this effect.

4. **(8 pts) Cliques**

   - Input: Name of school for which cliques are to be found, e.g. "rutgers"

   - Result: The subgraphs for each of the cliques.

   - Output: Print the subgraph for each clique, in the same format as the input described in the `Graph build` section (a clique is a graph in its own right).

     For example:

     ```
     Clique 1:

     <subgraph output>

     Clique 2:

     <subgraph output>

     etc...
     ```

     Note:
     - If there is an edge `x--y` in the graph, then your output must have either `x--y` or `y--x`, but NOT both (so your output can be used as input for any other application that might use your subgraph.)
     - If there is even one student at the named school in the graph, then there must be at least one clique in the output. If the graph has no students at all at that school, then the output will be empty. (You can write out an appropriate message as you see fit.)
     - In your code, each clique must be created as a separate `Graph` object: every subgraph of some graph, is a graph in its own right.

5. **(20 pts) Connectors (Friends who keep friends together)**

   - Input: Nothing

   - Result: Names of all people who are connectors in the graph

   - Output: Print names of all people who are connectors in the graph, comma separated, in any order.

Special Implementation Notes

- Your program MUST be called `Friends` - in other words, you must have a filed named `Friends.java` with a `main` method.
- You may implement as many classes as you want, and separate them into packages as needed.
- You may import any of the classes from `java.lang`, `java.io` and `java.util`, but you may NOT import classes from any of the other packages in the standard Java API, and you may NOT import classes from any external java APIs. (Of course, if you have more than one package in yourapplication, you can cross-import classes among them.)

Grading

Since there is no pre-defined structure for the classes you will implementation, grading will be done manually. Graders will run your program, and check the printed output for correctness. For `Graph build` they will look at the code to make sure you are storing the graph correctly.

Submission

ONLY ONE PERSON per team should submit the assignment.

In the `Friends.java` file, make sure you write the names of both members of your team in comment lines at the top.

Export your entire Eclipse project as a zip archive into a file called `friends.zip`, and upload this file to Sakai.
See the Eclipse page, under Zipping up a Project, for how to do this.

If you submit an incorrect/incomplete project, you will lose credit.
So before you submit, you should check that all is well by importing your friends.zip file as a project into Eclipse (in a different workspace, so it doesn't conflict with your existing project), and running it.
The assigments you have been doing all along have been given as project files that you have imported in Eclipse, so this should be old hat by now. If you are still having trouble and cannot figure this out for some reason, ask your TA, or one of the instructors.