Friends – 100 course points

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

Refer to our Programming Assignments FAQ for instructions on how to install VSCode, how to use the command line and how to submit your assignments.

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. In other words, this algorithm is not about students, let alone students at a particular school. So, for instance, you may need to find the shortest path (intro chain) from `nick` to `samir`, which will be:

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

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

2. **Cliques: Student cliques at a school**

   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 cliques of students at rutgers:

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

   Note that in the full graph these are not islands since `samir` connects them. However, since `samir` is not a student at rutgers, it results in two cliques of rutgers students that don't know each other through another rutgers student.

   At penn state, there is a single clique of students:

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

   Also, a single clique of students at ucla:

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

   And a single clique of students at cornell:

   ```
   (michele,cornell)
   ```

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.

   Definition: 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
     ```

     Neighbors for a vertex are stored in adjacency linked lists like this:

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

     The DFS starts at A.

     ```
     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 trace below 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 adjacency linked lists as in Example 1.

     ```
     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
     ```

     Neighbors stored in adjacency linked lists like this:

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

     DFS starts at A.

     ```
     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 graph as in Example 3, but neighbors are stored in adjacency linked lists in a different sequence:

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

     DFS starts at D, Connectors are still correctly identified as B and 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

At the bottom of the Autolab assignment page, under **Attachments**, you will see a `friends.zip` file. Download and unzip in your computer. You will see there are:

- 2 directories/folders, here are the contents of src:
  - A class, `friends.Friends`. This is where you will fill in your code, details follow.
  - A class, `Graph`, that holds the graph on which the the friends algorithms operate.
    - The file `Graph.java` defines supporting classes `Friend` and `Person` that are used to store a graph in adjacency linked lists format.
    - The file `Graph.java` also defines a class called `Edge` that you are free to use in your implementation in the `Friends` class.

    You will NOT change ANY of the contents of `Graph.java`.
  - Classes `structures.Queue` and `structures.Stack` that you may use in your implementation in the `Friends` class. You will NOT change ANY of the contents of `Stack.java` and `Queue.java`.
  - `A class FriendsApp.java` (initial test client, update this file for testing).
  - **bin**: contains the folders friends and structures with the class files after the assignment is compiled
- 2 files
  - `Makefile`: used to automate building and executing the target program.
  - sampleGraph.txt (sample graph below as a starting point for you)

Every graph that on which you might want to run your algorithms will have the following input format – the sample graph input here is for the friendship graph shown in the `Background` section above. (The `Graph` class constructor should be passed a `Scanner` with the input file as its target.)

```
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.
- 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.

Names of people and schools (disregarding case) are unique.

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.

You will complete the following static methods in the `Friends` class, to implement the three algorithms in the previous section. (All of these methods take a `Graph` instance as a parameter, aside from other possible inputs detailed below.)

Methods

1. **(35 pts) shortestChain**

   - Input: Name of person who wants the intro, and the name of the other person. For instance, inputs could be "sam" and "aparna" for the graph in the `Background` section. (Neither of these, nor any of the intermediate people in the chain, are required to be students, in the same school or otherwise.)
   - Result: The shortest chain (list) of people in the graph starting at the first and ending at the second, returned in an array list.

     For example, if the inputs are `sam` and `aparna` (`sam` wants an intro to `aparna`), then the shortest chain from `sam` to `aparna` is `[sam,jane,bob,samir,aparna]`

     (This represents the path `sam--jane--bob--samir--aparna`)

     If there is more than one shortest path, ANY of them is acceptable.

     If there is no way to get from the first person to the second person, then the returned list is empty (null or zero-sized array list).

2. **(25 pts) cliques**

   - Input: Name of school for which cliques are to be found, e.g. "rutgers"
   - Result: The names of people in each of the cliques, in any order, returned in an array list of array lists. Moreover, the cliques themselves could be in any order in the top level array list.

     For the example cited in the `Cliques` part of the `Algorithms` section above, with input `rutgers`, the result is:

     ```
     [[sam,jane,bob,kaitlin],[sergei,aparna]]
     ```

     In other words, an array list that has two cliques, each of which is an array list.

     The names in the clique array list can be in any order. So, the same cliques could have been returned as:

     ```
     [[jane,sam,kaitlin,bob],[aparna,sergei]]
     ```

     and it would be correct.

     The cliques themselves can be in any order within the top level array lists, so the following variation (for example) is also acceptable:

     ```
     [[sergei,aparna],[sam,jane,bob,kaitlin]]
     ```

     However, names must not be repeated in a clique.

     If there are no students in the input school, the result is empty (null or zero-sized array list).

3. **(40 pts) connectors**

   - Input: None
   - Result: Names of all connectors, in any order, returned in an array list. If there are no connectors, the result is empty (null or zero-sized array list).

     In the sample friendship graph of the `Background` section, the connectors list is `[jane,aparna,nick,tom,michele]`. Any other perumtation of the names in the list is fine, since the order does not matter.

     Names in the list must not be repeated.

Implementation Rules

Do NOT change ANY of the contents of `Graph.java`, `Queue.java`, and `Stack.java`.

In `Friends.java`, you may NOT MAKE ANY CHANGES EXCEPT to (a) fill in the body of the required methods, or (b) add private helper methods.

Note:

- You may use the `Stack` or `Queue` classes that are imported in `Friends.java`, as needed. But you are not required to use either.
- Also, you may use any of the code in the Graph class in Resources, or solutions to problems in the problem sets.

Compiling and Executing

Once inside the polynomial directory, type:

- **`javac -d bin src/friends/*.java src/structures/*.java`** to compile
- **`java -cp bin friends.FriendsApp graph_filename`** to execute

If you have a unix operating system (Linux or Mac OS) you CAN use the Makefile file provided. Once inside the polynomial directory type:

- **make** to compile
- **make run** to execute
- **make clean** remove all the .class files

Before submission

4. *Collaboration policy*. Read our collaboration policy here.
5. *Submitting the assignment*. Submit *Friends.java* via the web submission system called Autolab. To do this, click the *Assignments* link from the course website; click the *Submit* link for that assignment.

Getting help

If anything is unclear, don't hesitate to drop by office hours or post a question on Piazza. Find instructors office hours by clicking the *Staff* link from the course website. In addition to office hours we have the CAVE (Collaborative Academic Versatile Environment), a community space staffed with lab assistants which are undergraduate students further along the CS major to answer questions.