CS 103 – Six Degrees of Kevin Bacon

1 Introduction

This is the second half of the previous assignment, and acts as the culmination of your C/C++ programming experience in this course. You will use certain graph algorithms to perform queries on your social network, determine relationship features, and make friend suggestions.

2 What you will learn

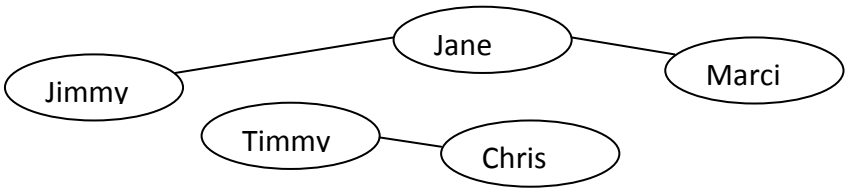
This programming assignment will have you:

1. Use breadth-first search to find a shortest path in a graph
2. Use a repeated search to find the components in a graph
3. Implement a simple friend suggestion algorithm
4. Use container classes (vector)

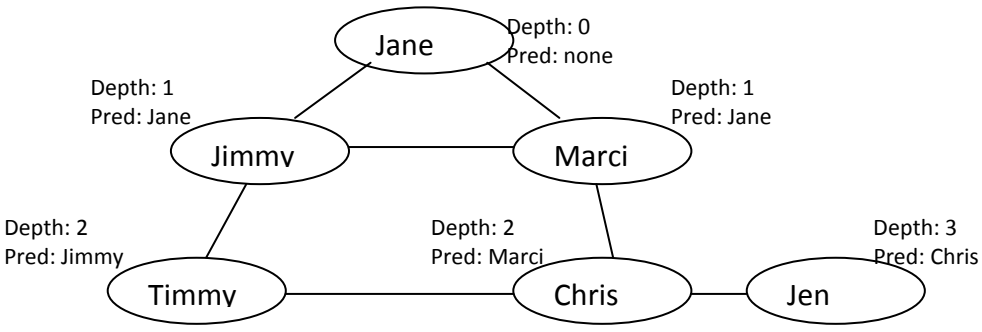
3 Background Information and Notes

BFS: Please review the Breadth-First Search slides posted on the website then continue reading. This is very similar to PA3 but in a more general setting: instead of every node having 4 neighbors, the number of neighbors varies from node to node.

Disjoint Sets: A disjoint set is a categorization of users into groups where a member of any group only has relational connections with other users in that group and none with users in any other group. In the picture below Jimmy, Jane and Marci form one disjoint set while Timmy and Chris form another.



Depth: One way to approach the "friend suggestion" part of this assignment is to use the "depth" of nodes. The shortest distance from the starting vertex to any other vertex in the network is its "depth." Here is a search starting at **Jane**, showing the depth of each node:



4 Prelab

None.

5 Procedure

1. Create a new folder, "pa5" under your "cs103" directory and then copy over your 'pa4' files to the new directory. This way you will always have your working pa4 code and can freely modify the copies to create your 'pa5' code.

```
$ cd cs103
$ mkdir pa5/
$ cd pa5
$ cp ../pa4/*  .
```

2. The main purpose of this assignment is for you to add three public functions to the Network class. See the next three sections below for descriptions of what outputs these functions should produce.

```
// see item 4 below
// a shortest path starting at user "from" and ending
// at user "to". If no path exists, returns empty vector
vector<int> Network::shortest_path(int from, int to)

// see item 5 below
// a list of the components of the network
vector<vector<int> > Network::groups()
// note! It's important you put a space between the >s

// see item 6 below
// suggested friends with highest score for this user.
// fills in "score" at the given pointer.
// if no suggestions exist, give empty vector and
//   set score to be -1
vector<int> Network::suggest_friends(int who, int& score)
```

Make all three functions public.

It is important that you exactly follow these prototypes because most of our testing will be done by calling these functions directly.

3. Add ***integer*** data members `'depth'` and `'predecessor'` to the User class. These can be used to help implement the algorithms described below. You are allowed to make them public members so you can easily set and access their values. You can create additional members like `'suggestion_score'` or `'set_member'` if you find it helpful.

4. **Shortest path.** Add a menu option (option 7), to compute the shortest relational distance between two users (minimum number of edges between two users/vertices). The output of this option should print the relational distance to the screen and then list the shortest path of relational connections that make up the given distance.

   Make this menu option 7. Follow the format of this example:

   User input:
   ```
   7 Mark Redekopp Max Nikias
   ```

   Printed output:
   ```
   Distance: 2
   Mark Redekopp -> Tommy Trojan -> Max Nikias
   ```

   In this example, Mark and Max have a path through a common friend Tommy Trojan, but no shorter path (they are not direct friends).

   Print None if there is no path present between the two specified users.

   Computing the relational distance can be accomplished by performing a Breadth-First Search starting from the source user. Set the 'depth' data member and the 'predecessor' member of each user encountered in the BFS. Once the destination user is found, their depth can be displayed and the reverse path back to the source can be found by following the 'predecessor' ID members. However, realize we want to print the path in the forward direction (from source user to destination). The predecessor of User x should be set to the ID of the user whose edge first allowed discovery of User x. By placing an invalid ID (such as -1) in the original source User's 'predecessor' value, we can traverse from the destination user back the source user by iterating over each until the 'predecessor' field contains -1. Starting from the destination user, we can adding theses predecessor ID's to a temporary vector/deque and then traverse it in the appropriate order to print out the relational path.

5. **Disjoint Sets**. Compute how many disjoint sets of users are present in the network. A disjoint set is a group of users who are isolated from all other users (i.e. have no edges to any users not in that set). Think about how BFS (or really multiple BFS searches) can be used to categorize each User into a different set. The output should be a list of sets and the Users in each set.

   For example, if the network only had four users and Mark and Tommy were each other's only friends while Max and Andrew were friends, the result of selecting this feature should be:

```
Set 1 => Mark Redekopp, Tommy Trojan
Set 2 => Max Nikias, Andrew Viterbi
```

To include this, make this menu option 8. The user shall simply input the command number (i.e. 8) to execute this feature.

6. **Friend Suggestions**. Compute a list of users who are most likely suggestions for the specified user to add as a friend. These users can obviously not be current friends of the specified user but should have a 'strong' connection to the ***friends*** of the specified user. Let us limit our search for 'strong' candidates to those who are at a relational distance of 2 from the specified user. The **score** of a potential new friend is the number of **common friends they would share with the specified user**.

   Make this menu option 9. Follow the format of this example:

   User input:
   ```
   9 Shaquille O'Neal
   ```

   The output from the command should be:

   ```
   The suggested friend(s) is/are:
      TA Tommy          Score: 3
      Kobe Bryant       Score: 3
   ```

   So in this example, Shaquille is not yet friends with Tommy or Kobe, but has 3 friends in common with both Tommy and Kobe, and this is the maximum.

   Display all candidates that have the highest score (print the score as well). There can be more than one user if more than one person is tied for the highest score. In this case print out all the users with the highest score! If there are no possible candidates, print None instead.

7. Compile your program (the Makefile from the part1 should work just fine for this assignment).

8. Download a new user database and one command script that can be used for these kinds of queries. Also, when you are just starting to debug your code you may want to make up your own database.

   ```
   $ wget http://ee.usc.edu/~redekopp/cs103/users_large.txt
   $ wget http://ee.usc.edu/~redekopp/cs103/users_large_disjoint.txt
   $ wget http://ee.usc.edu/~redekopp/cs103/grade_input1
   ```

9. Run your program: `$ ./social_network users_large.txt` or
   `$ ./social_network users_large_disjoint.txt`

10. Test your program with our canned inputs in `grade_input1` (view them in a text editor to understand what commands will be executed). Verify the output is as you would expect.

```
$ ./social_network users_large.txt < grade_input1
```

11. Obtain the readme from:

```
http://cs103.usc.edu/files/pa5/readme.txt
```

6 Rubric

8 points: `Network::suggest_friends` correctly computes friends and score
11 points: `Network::shortest_path` correctly uses BFS to find shortest path
11 points: `Network::groups` correctly finds connected components
5 points: Menu operations implemented correctly
5 points: Style, documentation, and readme