Social Graph

You have been assigned the task of building a new friend-based social network. In this network, users are able to view their own friends, friends of their friends, friends of their friends' friends, and so on. People connected to you through any number of friendship connections are considered a part of your extended social network.

The functionality behind creating users and friendships has been completed already. Your job is to implement a function that shows all the friends in a user's extended social network and chain of friendships that link them. The number of connections between one user and another are called the degrees of separation.

Your client is also interested in how the performance will scale as more users join so she has asked you to implement a feature that creates large numbers of users to the network and assigns them a random distribution of friends.

1. Generating Users and Friendships

It will be easier to build your extended social network if you have users to test it with. `populateGraph()` takes in a number of users to create and the average number of friends each user should have and creates them.

```
>>> sg = SocialGraph()
>>> sg.populateGraph(10, 2)  # Creates 10 users with an average of 2 friends each
>>> print(sg.friendships)
{1: {8, 10, 5}, 2: {10, 5, 7}, 3: {4}, 4: {9, 3}, 5: {8, 1, 2}, 6: {10}, 7: {2},
8: {1, 5}, 9: {4}, 10: {1, 2, 6}}
>>> sg = SocialGraph()
>>> sg.populateGraph(10, 2)
>>> print(sg.friendships)
{1: {8}, 2: set(), 3: {6}, 4: {9, 5, 7}, 5: {9, 10, 4, 6}, 6: {8, 3, 5}, 7: {4},
8: {1, 6}, 9: {10, 4, 5}, 10: {9, 5}}
```

Note that in the above example, the average number of friendships is exactly 2 but the actual number of friends per user ranges anywhere from 0 to 4.

- Hint 1: To create N random friendships, you could create a list with all possible friendship combinations, shuffle the list, then grab the first N elements from the list. You will need to `import random` to get shuffle.
- Hint 2: `addFriendship(1, 2)` is the same as `addFriendship(2, 1)`. You should avoid calling one after the other since it will do nothing but print a warning. You can avoid this by only creating friendships where user1 < user2.

2. Degrees of Separation

Now that you have a graph full of users and friendships, you can crawl through their social graphs. `getAllSocialPaths()` takes a userID and returns a dictionary containing every user in that user's extended network along with the shortest friendship path between each.

```
>>> sg = SocialGraph()
>>> sg.populateGraph(10, 2)
>>> print(sg.friendships)
{1: {8, 10, 5}, 2: {10, 5, 7}, 3: {4}, 4: {9, 3}, 5: {8, 1, 2}, 6: {10}, 7: {2},
8: {1, 5}, 9: {4}, 10: {1, 2, 6}}
>>> connections = sg.getAllSocialPaths(1)
>>> print(connections)
{1: [1], 8: [1, 8], 10: [1, 10], 5: [1, 5], 2: [1, 10, 2], 6: [1, 10, 6], 7: [1,
10, 2, 7]}
```

Note that in this sample, Users 3, 4 and 9 are not in User 1's extended social network.

- Hint 1: What kind of graph search guarantees you a shortest path?
- Hint 2: Instead of using a `set` to mark users as visited, you could use a `dictionary`. Similar to sets, checking if something is in a dictionary runs in O(1) time. If the visited user is the key, what would the value be?

3. Questions

1. To create 100 users with an average of 10 friends each, how many times would you need to call `addFriendship()`? Why?

2. If you create 1000 users with an average of 5 random friends each, what percentage of other users will be in a particular user's extended social network? What is the average degree of separation between a user and those in his/her extended network?

4. Stretch Goal

1. You might have found the results from question #2 above to be surprising. Would you expect results like this in real life? If not, what are some ways you could improve your friendship distribution model for more realistic results?

2. If you followed the hints for part 1, your `populateGraph()` will run in O(n^2) time. Refactor your code to run in O(n) time. Are there any tradeoffs that come with this implementation?