EECS-214 Assignment 5
A social network graph

Out: Friday, May 15
Due: Friday, May 22, noon

Overview

In this assignment, you'll work with an undirected graph representing relationships between people. You task will be to implement the data structures to represent the graph, and to implement a graph walk algorithm to compute the distance between two people in the network, defined as the number of edges along the shortest path connecting the two people. Thus the distance between a person and themselves is 0, between two people with a direct connection is 1, and so on.

This assignment **differs from previous assignments** in three important ways:

- We are **not specifying how** you should represent the graph. You may use implement a node class to represent the nodes of the graph and place a list of edges inside the nodes, if you like. Or you may use an array of adjacency lists, if you prefer. Or you may even use a matrix representation. It's up to you.

- You are **may use the built-in classes** for Queues, Lists, LinkedLists, and Hashtables, if you like (but you are not required to):

    - Queue<T> is a queue whose elements are of type T. For example, Queue<int> is a queue of integers, while Queue<Node> is a queue of Nodes, assuming you've defined some class named Node.

    - List<T> is a dynamic array whose elements are of type T. Again, List<int> would be a dynamic array of integers, while List<string> would be a dynamics array of strings.

    - LinkedList<T> is a linked list of elements of type T.

    - Dictionary<TKey, TValue> is a hashtable whose keys are of type TKey and whose values are of type TValue. So Dictionary<string, int> would be a hash table that mapped strings to integers, since Dictionary<string, string> would make strings to other strings.

Click on the links above to see the methods that are available in these data types. You are not required to use them, but feel free to do so if you like.

- Finally, we are giving you an incomplete set of test cases. You are **expected to add additional tests** to verify that your code is working correctly. We will test them with a different test graph and a different set of test cases than the one included in the assignment, so you should think carefully about what kinds of things we didn't test for and add your own tests. You will be graded based on whether your code passes the tests with our tests and graph.

The graph

You should fill in the implementation of the PersonGraph class in the file PersonGraph.cs. PersonGraph should be an undirected graph of representing people (the nodes) and their connections (the edges). Again, you may implement the graph any way you like, but your must support the three methods we have included in the PersonGraph.cs file:

- void **AddPerson**(string personName)
  Adds a new node to the graph representing a person with the specified name.
  - You do not have to worry about the case of someone trying to add the same person twice.
- void **AddConnection**(string person1, string person2)
  Adds a new undirected edge to the graph connecting the nodes representing person1 and person2.
  - If person1 or person2 have not been added to the graph already, AddConnection should add them.
  - You do not have to worry about the case of someone trying to add the same edge twice.
- int **Distance**(string person1, string person2)
  Returns the distance in the graph between person1 and person2, that is, the number of edges in the shortest path connecting the nodes for person1 and person2.
  - If person1 and person2 are unconnected (i.e. there not only is no edge connecting them, there isn't even a path), then Distance should return -1.

Note that this means that you will need to have some data structure such as a hashtable for keeping track of what node objects or node numbers (if using an array of adjacency lists) correspond to what person names.

Important: while the file PersonGraph.cs only contains a single class, you should feel free to add additional classes if you like. You do not have to create additional classes, but if you do, you should include them in PersonGraph.cs rather than making extra files, since our testing code won't use the extra files.

The tests

We have included a set of tests as well as a test graph in the file PersonGraphTests.cs in the UnitTests project. You can find the test graph in the method MakeTestGraph inside of PersonGraphTests.cs.

As mentioned above, we have included a deliberately incomplete set of tests. You should add additional tests for your own use, although you will not be turning those tests in. The basic format for a test is:

```
[TestMethod]
public void TestName()
{
   Assert.AreEqual(7,
                   Graph.Distance("person1 name",
                                  "person2 name"));
}
```

Here the `[TestMethod]` annotation tells the testing system that the method below contains tests that should be run when testing. The Assert.AreEqual call tells the system that its two arguments should be equal, and that if they aren't, the test has failed.

To make a new test, just copy one of the existing tests in PersonGraphTests.cs and change its method name (which will show up as the name of the test), the names of the people the system is computing the distance between, and the correct distance (7 in the example above) that Assert.AreEqual should compare to. That's all you have to do.

Turning it in

To turn in your assignment, upload your PersonGraph.cs file to Canvas. Again, since you can only submit the PersonGraph.cs file, all your classes (if you make more than one) need to be in this file. Do not upload your tests file.