Social Computing	Homework 1: Basic Network Analysis
CS5750/IS4700 Fall 2013	August 31, 2013

This project is due at 11:59:59pm on September 25, 2013 and is worth 5% of your grade. You must complete it with a partner. You may only complete it alone or in a group of three if you have the instructor's explicit permission to do so for this project.

1 Description

There are several goals for this assignment. First, you will get experience measuring and analyzing real network data. Second, you will gain experience implementing algorithms that we discussed in class. Third, you will face scalability challenges, as some of the networks are quite large.

Your assignment is to write a program that, given a network, calculates a number of statistics. Your program, called `5750netalyzer` will read input from a file given as a command-line argument, open the file and read in the network, and then output a number of statistics about the network. There are very specific requirements for the format of the input and the output; these are included below.

2 Requirements

Your program will accept a filename as a single argument, will read in the graph file in the format described below, and will output the statistics in the output described below. The instructors have provided a reference solution, and your program's output will be compared against the reference solution. The reference solution is the same code that your project will be graded against, so you should ensure that your program's output matches in order to achieve a good score.

2.1 Starter code

Very basic starter code and sample data for the assignment is available on the CCIS Linux network at `/course/cs5750f13/code/homework1`. You are free to implement this project in any language you choose (assuming the CCIS Linux machines have the necessary compiler and library support). You may also use any existing graph libraries to *store and query* the graph; you *must not* use these libraries to run the analysis (e.g., calculate the clustering coefficient). All analysis code must be written by you.

To get started, you should copy down this directory into your own local directory (i.e., `cp -r /course/cs5750f13/code/homework1 ~/cs5750`). The `Makefile` is configured to test your code on sample input (via the `test` target). You should feel free to edit the `Makefile` so that it will compile your code (or do other things). However, you should not change the `test` target. Your executable program must be named `5750netalyzer`, and must be executable by running `./5750netalyzer`.

2.2 Input format

Your `5750netalyzer` program will accept exactly one argument: a filename. The file will represent a graph, which will be a number of lines of the form:

```
node1 [tab] node2 [\n]
```

The node1 and node2 labels can be of the form (in regular expression syntax):

```
[A-Za-z0-9_-]+
```

The graph files can be arbitrarily large, and the node labels can be of any length (greater than 0 characters). The graphs will be *directed*, meaning a link from A to B does not imply a link in the reverse direction.

2.3 Features

Your program must run a number of graph analysis metrics on the graph that you read in. In particular, you should examine (in this order):

2.3.1 Total nodes and links

You should output the total number of unique nodes, number of links, and the fraction of links that are symmetric that you observed. The format should be:

```
Nodes: [space] [number of nodes] [\n]
Links: [space] [number of links] [\n]
Symmetric links: [space] [percentage to 5 decimal places] [%] [\n]
```

(i.e., the string "Nodes:", a space, the number of nodes, and a carriage return, etc). One example output might be

```
Nodes: 21
Links: 83
Symmetric links: 23.30211%
```

2.3.2 Average out-degree and in-degree

You should output the average out-degree and in-degree of nodes in the graph. Your output should be in the format:

```
Average outdegree: [space] [average outgoing links to 5 decimal places] [\n]
Average indegree: [space] [average outgoing links to 5 decimal places] [\n]
```

One example (partial) output might be

```
Average outdegree: 24.50572
```

2.3.3 Clustering coefficient

Recall that the clustering coefficient of a node i, denoted by $c(i)$, is defined as the number of directed links that exist between the node's neighbors, divided by the number of possible directed links that could exist between the node's neighbors (i.e., the nodes that this node points to). Thus, if a node i's neighbors have n directed links between them, then the clustering coefficient of i is defined as

$$c(i) = \frac{n}{d_i(d_i - 1)} \tag{1}$$

(nodes with less than 2 neighbors are disregarded) The clustering coefficient of a graph is the average clustering coefficient of all its nodes with more than 1 neighbor, and we denote it as $C(G)$, or

$$C(G) = \frac{\sum_{v \in V} c(v)}{|V|} \tag{2}$$

Thus, the clustering coefficient of a graph ranges between 0 and 1, with higher values representing a higher degree of "cliquishness" between the nodes. In particular, a graph with clustering coefficient of 0 contains no "triangles" of connected nodes, whereas a graph with clustering coefficient of 1 is a perfect clique. A graph with no nodes with more than 1 neighbor is defined to have a clustering coefficient of 0.

Your program must output the clustering coefficient of the graph as

```
Clustering coefficient: [space] [coefficient to 5 decimal places] [\n]
```

One example output might be

```
Clustering coefficient: 0.20347
```

2.3.4 Assortativity

The assortativity of a graph captures the linking behavior of nodes. Assortativity is measured in the range $[-1, 1]$, where 1 means that nodes always link to other nodes of equal degree, and -1 indicates that nodes link to other nodes of different degree. A graph with assortativity >0 is called assortative, while a graph with assortativity <0 is called disassortative. For directed graphs, you can actually calculate four different assortativities: $r(in, in)$, $r(in, out)$, $r(out, in)$, and $r(out, out)$. Here, in and out refer to which types of degrees you are comparing, i.e. in-degree vs. in-degree, in-degree vs. out-degree, etc.

Taken form Foster et al.'s PNAS paper, let $\alpha, \beta \in \{in, out\}$ be the degree type, and j_i^α and k_i^β be the α- and β-degree of the source and target node for edge i. We can then define the assortativity as

$$r(\alpha, \beta) = \frac{E^{-1} \sum_i [(j_i^\alpha - \overline{j^\alpha})(k_i^\beta - \overline{k^\beta})]}{\sqrt{E^{-1} \sum_i (j_i^\alpha - \overline{j^\alpha})^2} \sqrt{E^{-1} \sum_i (k_i^\beta - \overline{k^\beta})^2}} \tag{3}$$

Where E is the number of edges in the graph, $\overline{j^\alpha} = E^{-1} \sum_i j_i^\alpha$, and $\overline{k^\beta}$ is similarly defined. In cases where the denominator is 0, the metric is defined to be 0 (in such cases, the numerator must also be 0).

Your program must output $r(in, in)$ and $r(out, out)$ as

```
Assortativity (in/in): [space] [in-degree assortativity to 5 decimal places] [\n]
Assortativity (out/out): [space] [out-degree assortativity to 5 decimal places] [\n]
```

One example output might be

```
Assortativity (in/in): 0.59270
Assortativity (out/out): 0.01373
```

2.3.5 Radius and diameter

Recall that the radius and diameter of a graph represent how far away nodes are from each other in the network. Formally, the *eccentricity* of a node v is the maximal directed shortest path distance between v and any other node (for the purposes of this project, pairs of nodes that have no path between them should be ignored and nodes with no outgoing links have an undefined eccentricity). The *radius* of a graph is then the minimum eccentricity across all vertices, and the *diameter* is the maximum eccentricity across all vertices. Thus, the radius represents the maximal distance from the most "central" node in the graph to all other nodes, and the diameter represents the maximal distance from the least "central" node in the graph to all other nodes.

Your program must output the radius and diameter of the graph as

```
Radius: [space] [radius] [\n]
Diameter: [space] [diameter] [\n]
```

2.3.6 Average path length

The *shortest path length* between two nodes is defined as the minimum number of directed hops that must be traversed to get from one node to another. Note that (a) due to the directed nature of graphs, the shortest path length is not necessarily commutative, and (b) certain pairs of nodes may have no defined shortest path length. The *average path length* is defined as the average value of the shortest path length, measured across all pairs of nodes that have a defined shortest path length.

Your program must output the average path length of the graph as

```
Average path length: [space] [average path length to 5 decimal places] [\n]
```

2.3.7 Connected components

Recall that for an undirected graph, a *connected component* is a subset of the nodes where that there is a path in the network between all pairs of nodes in the set. For a directed graph, we distinguish between a *strongly connected component* and a *weakly connected component*. A strongly connected component (SCC) is defined as a set of nodes such that there is a path in the network between all pairs of nodes in the set. In contrast, a weakly connected component (WCC) is defined as a set of nodes such that there is a path in the network between all pairs of nodes in set if the all links in the network were viewed as undirected.

Your program must calculate the *weakly connected components* in the graph, and output their sizes (i.e., the number of nodes in each component). The format should be

```
Weakly connected components: [space] [size_1] [space] [size_2] ... [size_n]  [\n]
```

Your program *must* print these out in decreasing order of size. For example, one output might be

```
Weakly connected components: 45 39 9 3 2 1 1
```

2.4 Exit message

Finally, your program should output

```
So long and thanks for all the fish! [\n]
```

before it exits.

2.5 Error handling

If your program is unable to open the file, or if the file does not exist, you should output

```
Error: Unable to open graph file.
```

and exit. If you encounter a graph file which does not meet the specification listed above, you should output

```
Error: Malformed graph file on line LINE.
```

(where LINE is the [0-indexed] offending line number) and exit. If you encounter a duplicate link, you should output

```
Error: Duplicate link on line LINE.
```

and if you encounter a self-loop, you should output

```
Error: Self-loop on line LINE.
```

If you encounter any other error while running the program, you should output

```
Error: [meaningful description of the error]
```

3 Implementation hints

You should develop your client program on the CCIS Linux machines, as these have the necessary compiler and library support. You are welcome to use your own Linux/OS X machines, but you are responsible for getting your code working, and your code *must* work when graded on the CCIS Linux machines. If you do not have a CCIS account, you should get one ASAP in order to complete the project. If you are using C, your code must be `-Wall` clean on `gcc`. Do not ask the TA for help on (or post to the forum) code that is not `-Wall` clean unless getting rid of the warning is what the problem is in the first place.

You will want to make sure that you break your program up into modules, such that each module represents a sensible type abstraction. If you have questions on how to do this, please come see the instructors or the TA.

4 Testing

First test that your program can run on very simple graph files. Your program should not crash, no matter how weird the input; in all cases, your program must either complete successfully or print an error.

Additionally, we have included a basic test script to check the output of your code against our reference solution and check your code's compatibility with the grading script. If your code fails in the test script we provide, you can be assured that it will fare poorly when run under the grading script. To run the test script, simply type

```
bash$ make test
```

This will compile your code and then test your shell on a number of inputs, comparing the results against the reference solution. If any errors are detected, the test will print out the side-by-side diff-ed output (the left side is the reference solution, the right side is yours). For example, you might see something like

```
bash$ make test
Milestone tests
  Trying with graph 'graph1'                                    [FAIL]
    Diff in expected output:
Nodes: 10                                                     <  Nodes: 11
Links: 20                                                     <  Links: 19
So long and thanks for all the fish!                             So long and thanks for all the
```

This indicates that the test with input `./5750netalyzer` was expected to print out the lines with a `<` in the left column, but instead returned the right column; both output the "So long..." text. We include a few sample tests, but these are by no means exhaustive. We expect that you will create additional test to ensure that your programs behave as expected.

5 Submitting your project

5.1 Registering your team

You and your partner should first register as a team by running the `/course/cs5750f13/bin/register` script. You should pick out a team name (no spaces or non-alphanumeric characters, please) and run

```
/course/cs5750f13/bin/register homework1 <teamname>
```

This will either report back success or will give you an error message. If you have trouble registering, please contact the course staff.

You must register your team by 11:59:59pm on September 9, 2013.

5.2 Final submission

For the final submission, you should submit you (thoroughly documented) code along with a plain-text (no Word or PDF) README file. In this file, you should describe your high-level approach, the challenges you faced, a list of properties/features of your design that you think is good, and an overview of how you tested your code. In your README, you should also point out any extra features of your project that you have implemented.

You should submit your project by running the `/course/cs5750f13/bin/turnin` script. Specifically, you should create a `homework1` directory, and place all of your code and README files in it. Then, run

```
/course/cs5750f13/bin/turnin homework1 <dir>
```

Where `<dir>` is the name of the directory with your submission. Again, the script will print out every file that you are submitting, make sure that it prints out all of the files you wish to submit!

You must submit your project by 11:59:59pm on September 25, 2013.

6 Grading

The grading in this project will consist of

75% Program functionality

15% Correct error handling

10% Style and documentation

7 Advice

A few pointers that you may find useful while working on this project:

- Check the Piazza forum for question and clarifications. You should post project-specific questions there first, before emailing the course staff.
- Finally, get started early!