CSC 207 **Assignment 10: Network Analysis** **Spring 2011**

Assigned: Tuesday 3 May 2011

Predictions Due: Friday 6 May 2011

Code/Results Due: Wednesday 11 May 2011

Objectives:

- Learn about **graphs**, their meaning, representations, and utility
- Reinforce choosing appropriate **collections** to support data structures and algorithms efficiently
- Practice writing **recursive algorithms** for data structures
- Gain experience in **predictions** and **analysis** of social networks

Collaboration: This homework assignment may be completed individually or in pairs.

Submission: Follow the instructions for submitting programs via P-Web and handing in a printed copy. Be sure to generate unit tests and a statement justifying their sufficiency.

Background

In our study of various data structures and their representations, we examined several types of trees, which feature nodes that may have any number of children (as indicated by a directed edge from parent to child) and only one parent. Two properties of trees may be relaxed. First, we can remove the stipulation that a node have only one incoming edge, meaning there is no longer a clear "parental" relationship between nodes. Second, we can eliminate the directionality of edges altogether. These relaxations give us a data structure known as an undirected graph.

The properties of graphs were studied long before social networks became popular. However, the rise of massive, digitized networks has allowed scientists to ask interesting questions. From Stanley Milgram's 1967 experiments on the small world phenomenon, to Google's PageRank algorithm, to your friends on Facebook, analyzing graphs is a fundamental computational problem. In fact, your search for a doublet chain was a graph search algorithm. With a queue, you found the shortest path between the doublets in the word graph.

In this assignment, you'll explore another important graph algorithm, determining which nodes of the graph are reachable from one another.

Graphs

More formally, a graph is a set of vertices (or nodes) V and a set of edges, E. Each edge is an unordered pair (u, v), where $u, v \in V$ are nodes of the graph. Graphs may be represented in one of two common ways. The first is an adjacency matrix. In this representation, a square matrix A is used to indicate the presence and absence of edges in the graph. In particular, $A_{u,v} = 1$ if there is an edge between nodes u and v. Because the graph is undirected, the matrix must be symmetric, meaning $A_{v,u} = 1$ also. Any zero entries mean there is no such edge.

You may notice that when a graph is very sparse (meaning very few of the possible edges exist) the adjacency matrix is mostly filled with zeros. This representation can be very inefficient for graphs with large numbers of nodes, because the matrix requires N^2 entries. The second representation—the one we will use—is an adjacency list. For each node, we store the more compact list (or set, depending on your nomenclature) of nodes it shares an edge with.

`Graph<AnyType>`

Create a parameterized class `Graph` that is `Iterable` and whose nodes are `AnyType` objects. Your class should have the following methods

`void addEdge(AnyType u, AnyType v)` adds an undirected edge between two nodes in the graph. If the nodes do not exist, they are added to the graph.

`Set<AnyType> neighbors(AnyType` *`node`*`)` returns a set of all the nodes that share an edge with the specified node. The return value should not be `null`. An `IllegalArgumentException` should be thrown if the specified node is invalid.

Processing

Now that we have a model of a graph, we can turn our attention to the task at hand: finding the nodes in the graph that are reachable from one another via a series of edges in the graph. To model this, you'll create a class called `Component` representing these clusters of interconnected nodes. Then you'll need another class that takes a graph and finds its connected components.

`Component<AnyType>`

Create a parameterized `Component<AnyType>` class that is `Comparable` (to other `Components`, based on their size) and `Iterable` over its member nodes. Include an `add` method that adds a node to the component, as well as a `size` method that returns the number of nodes in the component.

You should also override the `equals` method to determine whether two `Components` are the same (i.e., consist of the same nodes). You can easily rely on other objects' `equals` methods to correctly implement this.

`ConnectedComponents<AnyType>`

Create a `ConnectedComponents` class that is `Iterable` over the connected `Components` of a `Graph`. The iterator provided should traverse the member components in order of their size. Include a `size` method that returns the number of connected components found. The constructor should take a `Graph` and find the connected components. How might one do that?

To start, we know every node must be processed and added to some component. For every node in the graph that has not yet been processed, we create a new component, and then process the node on behalf of the component. What does it mean to process a node? If it hasn't been processed yet, add it to the component and then process each of its neighbors. In this way, you will recursively "spider" or "crawl" the graph, finding all nodes reachable from a starting node. Note that you will need to keep track of which nodes have been processed as you proceed.

`Driver`

Create a driver class with a static factory method that takes a file name and reads a list of edges from the file containing one line for each edge. Each line consists of two values (integers, strings, etc.) representing node identifiers separated by whitespace.

The main method of your driver program should take a file name from the command line, create a graph from the file (using your factory), and process the connected components. Your program should then report the total number of components and then the size of each component (in increasing order).

Note: You may need to use the `-Xmx` command line option to increase the memory available when running your program.

Unit Testing

To run unit tests, in addition to manually creating graphs within your code, I suggest you create some small graph files for which you know the answer(s) to test your factory. Please include your test files in your electronic submission and include them in the transcript (by using the `cat` program) you submit in hard copy. Note the `Tester.checkIterable` method and the `Tester.checkExpect` method that takes arbitrary object types and reflectively, recursively tests all the member fields of the object classes.

Analysis

There are several files in `~weinman/courses/CSC207/data/networks` containing actual social network graphs you can experiment with. The first is the auto-follow (a.k.a. auto-read, auto-finger) graph from GrinnellPlans[1], captured in April 2011. The graph contains an edge between u and v if member u follows the plan of member v at priority level 1.

`plans_p1.txt` A network indicating readership among GrinnellPlans users. $N = 4,996; |E| = 135,041$.

Descriptions and data for the next three graphs are taken from the Stanford Large Network Dataset Collection[2]. The first two represent scientific co-authorship patterns in two categories of the scientific e-print repository called arXiv (http://arxiv.org). The timespan is roughly the first decade of its existence, from 1993-2003. The graph contains an edge between u and v if an author u co-authored a paper with author v.

`ca-GrQc.txt` A collaboration network among authors of papers in the arXiv General Relativity and Quantum Cosmology category. $N = 5,242; |E| = 28,980$.

`ca-AstroPh.txt` A collaboration network among authors of papers in the arXiv Astro Physics category. $N = 18,772; |E| = 396,160$.

`email-Enron.txt` A communication network among Enron employees, published by the Federal Energy Regulatory Commission during their investigation of the company. Nodes of the network are email addresses. The graph contains an edge between u and v if an address u sent at least one email to address v. $N = 36,692; |E| = 367,662$.

Write-Ups

Please provide the following analyses typewritten in a word processor, double-spaced, and printed in a 12 point font with 1 inch margins.

Predictions For two of the graphs above, write up a short statement on your predictions about the connected components. Roughly how many do you think there will be? How do you expect their sizes to be distributed? Why? Explain the reasoning behind your predictions.

Results Once your code is finished (i.e., documented and tested), tabulate the actual results on the graphs above. Note where any results differ substantially from your predictions. For at least one component, do some sort of subsequent analysis of the component's topology. For example, is it strongly well connected? (Meaning, are most nodes connected to most other nodes?) Are certain parts of it well connected with only a few nodes/edges bridging these well-connected parts ? Is it generally linear?

If your component is very large, you may need to learn (and implement) some more graph analysis metrics. If it is very *very* small, you're unlikely to find anything interesting. If it is moderately sized, you may be able to draw it by hand and make some observations.

[1] http://grinnellplans.com
[2] http://snap.stanford.edu/data