CS 3100 – Data Structures and Algorithms

Project #4 – Graph Statistics

Learning Objectives

- Use an existing graph data structure (or implement a new one) to represent graphical data
- Apply the existing graph implementation's functionality to output established statistics
- Expand the existing graph implementation to provide additional statistics about the data

Overview

Your task for this assignment is to use the graph data structure we have been implementing in lecture to model graphical data contained in a file and output various statistics about it. In addition, you will need to expand the current implementation to allow for new functionality, as described below.

Ground Rules

For this project you are welcome to use any and all graph-related code I have posted to Pilot as a starting point. You may alternatively implement your own graph data structure from scratch if you desire. You may use any classes from the C++ Standard Template Library (STL), included vector, queue, set, unordered_map, etc., in your implementation. You may NOT use significant portions of any code or libraries from any other source (i.e. other than mine and the C++ STL). If you have any questions about what is allowable, ASK ME rather than risk an academic integrity violation.

Requirements

1. You will need to write a main method (preferably in a file called project4.cpp) that prompts the user for the name of a data file, reads in that file, and stores its contents in a graph data structure.

Data files will be organized like this (there is a sample on Pilot called `mine.graph`):

```
Vertices
v1_label
v2_label
…
Edges
from1_label to1_label
from2_label to2_label
…
```

Vertex labels should be considered ***strings*** (this is different than what I said in lecture – I'm changing it to make it easier), and the edges should be considered directed.

Example prompt and input:

```
Graph filename? mine.graph
```

Note: The vertices in the sample file have names like 1, 2, etc. but you should still store them as strings.

2. After reading in the data file, your code should print the following message:

Ingested graph from <input_filename>: V vertices and E edges, where V is the number of vertices in the graph and E is the number of edges. For mine.graph, this message will be:

```
Ingested graph from mine.graph: 11 vertices and 19 edges
```

3. Print out the following messages related to node degree:

v1_label communicates with the most others (v1_out)

v2_label is who others communicate with most (v2_in)

where v1_label is the label of the vertex with the greatest number of outgoing edges, and v1_out is the number of outgoing edges that vertex has, and v2_label is the label of the vertex with the greatest number of incoming edges, and v2_in is the number of incoming edges that vertex has. For mine.graph, this will be:

```
1 communicates with the most others (4)

5 is who others communicate with most (4)
```

4. Prompt the user for a starting and ending vertex label and print out the number of edges on the shortest path from the start to the end. For example:

```
Starting vertex? 1

Ending vertex? 9

1 is 3 hops away from 9
```

If there is no path from the starting to the ending vertex, state that as follows:

```
Starting vertex? 1

Ending vertex? 8

There is no path from 1 to 8
```

Note: You only need to prompt for one starting vertex and one ending vertex. I will run your program multiple times to check its answer on different inputs.

5. Prompt the user for an integer and display a list of all nodes that are within that number of hops (edges) from the starting vertex you read in during the previous step:

```
Neighborhood size? 2

1’s 2-hop neighbors are: 1 2 5 7 10 3
```

Notes: The order in which you list the neighbors is not important. Be sure to include the starting node itself as part of the neighborhood. Do not list the same vertex label multiple times. Again, you only need to prompt for the neighborhood size once. I will run your program multiple times to check its answer on different inputs.

6. Display statistics about the number of connected components and the size of the largest one:

```
The largest connected component contains 7 vertices

There are 3 connected components
```

Note: When determining the number of connected components in a directed graph, do not count components that are “subsumed” by larger components. For example, in mine.graph, if we start at vertex 1 and look at what vertices are connected to it, we have 1 2 3 5 7 9 10. If we start at vertex 2, since the edges are directed, we have 2 3 9. Because 2 3 9 is a subset of 1 2 3 5 7 9 10, we would not count 2 3 9 as a separate connected component. Using this definition, the three connected components in mine.graph are:

```
1 2 3 5 7 9 10

4 6 8

11
```

(Your program does not need to display these – only count them. I’m just providing this for clarification.)

Turn in and Grading

Turn in any source code files needed for your program, including your main.cpp file. If you used one or more of my files as-is, upload them as part of your submission. The goal is for me to have everything I need to compile and run your program in one place. You may upload the files individually or in a zip file.

Projects that don’t compile will receive a zero.

Each of these things is worth 10 points, for a total of 100 possible points:

- Reads in a data file with the specified format and stores it in a graph
- Correctly displays the number of vertices and edges
- Correctly determines the node with the highest outdegree and the number of outgoing edges
- Correctly determines the node with the highest indegree and the number of incoming edges
- Correctly states the minimum number of hops between two vertices when a path exists
- Correctly determines when a path between two vertices does not exist
- Correctly finds the 2-hop neighborhood of a node
- Correctly finds the n-hop neighborhood of a node for any integer n
- Correctly finds the size of the largest connected component
- Correctly finds the number of connected components