Jeffrey Lansford

9-25-2019

Lab 4

Introduction

In this Lab, we completed the data structure to represent graphs. We completed the scanning and saving graphs to a tab-separated text file. Then we used R to visualize the graphs and to test our program with small and very large graphs.

Methods

We are provided a template for the Graph class, so we had to finish the template with scan, save, and addEdge. The class has two vectors, m_nodes and m_adjList. M_nodes holds all the different nodes within the graph and m_adjList holds a list of all the adjacent nodes for a given node. For adding edges, we used a list to hold the adjacent nodes to make it faster when adding new nodes to it. What we do is pull the list from where it is located according to the node's id then push the adjacent node to the list and store the new list back where we pulled from. For scanning, we must open the file and parse each line to get the two nodes individual strings. Then we use a hash table to map our nodes to keep track of the unique nodes. We go through the entire file, adding the unique nodes to the hash table. Then we add the nodes mapped from the hash table to the vector. We must reopen the file to go back through the list of edges to now get the edges store into the vector. Now we fid the two nodes read in and use the addEdge function to add the edges. Saving is simple where we read the m_nodes and the m_adjList vectors and go though the list to write each edge. The pseudo code below shows the Graph Class described earlier:

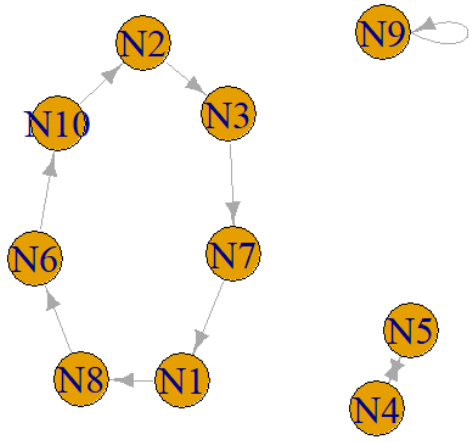
```
Graph Class
Vector<Node> m_nodes
Vector<list<Node>> m_adjList


addEdge (Node a, Node b) {
        l = m_adjList[a.id]
        add node to l
        m_adjList[a.id] = l
}
Scan () {
        Hash table nodeMap
        While Loop for reading in lines of edges, nodes stored in c1 and c2 as strings {
                Find node c1 in nodeMap
                If c1 is not in node map
                        Add c1 to nodeMap
                Find node c2 in nodeMap
                If c2 is not in node map
                        Add c2 to nodeMap
        }
        Resize vectors to number of nodes
        Reopen file
        For loop of going through nodeMap {
                Add to vector m_nodes
        }
        While Loop for reading in lines of edges, nodes stored in c1 and c2 as strings {
                Get c1 and c2 from nodeMap
                Get nodes from m_nodes using nodeMap, node1 and node2
                AddEdge(node1 and node2)
        }
```

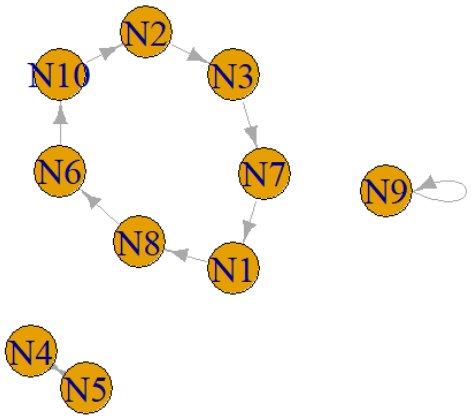
For visualizing and testing the graph, we will R to compare the two text files to see if they have the same edges and graph them if they are small enough from the files produced.

Results

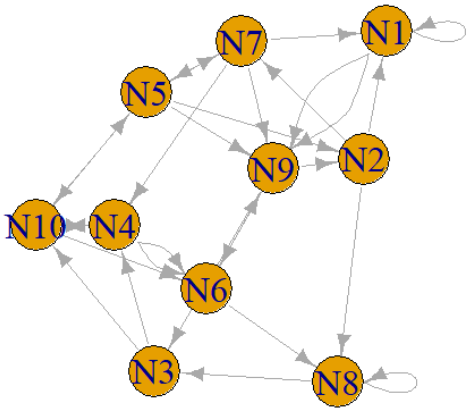
I graph two small graphs, test_small and test_mid. Test_small had graph this first from the scan file:



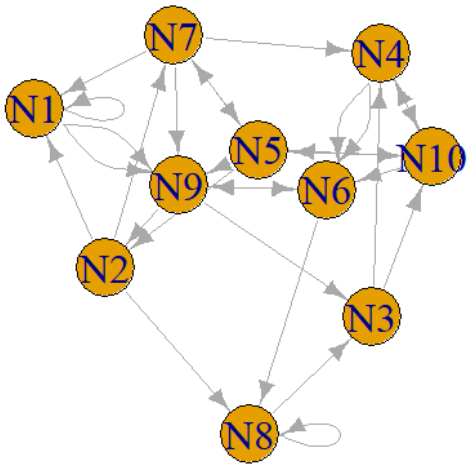
And this from save file:



Test_mid had these to show for scan:



And this for save:



We can see that they are the same graphs, just arranged differently due to how the vector is organized.

For larger graphs, we do not want to draw the graph as it will take forever, instead we left it in its text format. We tested the runtime for the larger graphs to see if our algorithm is running at an efficient rate. For test_midder which has 50000 nodes and 500000 edges, the runtime is 1.49 seconds. For test_big which has 100000 nodes and 1000000 edges, the runtime is 3.33 seconds. For test_really_big which has 1000000 nodes and 2000000 edges, the runtime is 6.56 seconds. Seeing the runtime doubles for when the number of edges double means that our algorithm is running at O(n).

Discussions

At first, I was developing the code I was using vectors to try and hold the nodes before inserting them into m_nodes. But this strategy would pollute our runtime as pushing into the vector would make it resize the vector for each new node. Thus, making the runtime even longer than it must be. Using the hash table was the best strategy to keep the runtime low and to make the code easier to read. I am glad the Dr. Song gave us an extra week and explained how hash tables work in C++ to help me better understand the Graph database and make it more efficient.

Conclusions

This lab gave us the foundation to create Graphs in way for the computer to interpret. We created a class that can read in a text file, break it down into data structures, and rewrite

into an output file for the data structures. Knowing how to read in the text files efficiently helps us keep future programs from running very long. This provides the groundwork to begin working the DFS and BFS algorithms that we are learning in class.