Benjamin Simonson

Describe the data structure you used to implement the graph and why?

I used a map data structure that consisted of string keys and a list of strings as values. I implemented the list of strings using a node struct within an AdjacencyList class where I had the string url and a next pointer (singly linked list). I used this because I am very familiar and comfortable with working with linked lists as I am building my own RTOS in one of my other classes. Thus, having my values of my map be contained within a linked list for my adjacency list implementation was very easy. I used a map (ordered) as it has a low space requirement mainly. I was initially thinking of using a vector of paired strings and linked lists but operating within this vector container would be a bit difficult to do and would require more work than just using a map and iterating through the map using a for each loop.

What is the computational complexity of each method in your implementation in the worst case in terms of Big O notation?

I will go top down in accordance with my code. My first method is the insertNode method. This implementation depends on the number nodes already within the list IE the number of edges a specific node is already pointing to. This is because I insert the new node being pointed at at the end of the existing list. All other operations within this method are O(1). Thus, the computational complexity of this method is O(|E|) where |E| is the number of edges a node has.

My next method is the getNodes method. This method just creates a vector of all of the nodes within a specific keys value (the keys adjacency list). Thus, similar to the first method, the computational complexity of this method is dependent on the number of edges that a key has, so, the computational complexity is O(|E|) where |E| is the number of edges of a key.

My next method is the AddEdge method. The first complexities I will analyze are building the set of all of the nodes. Inserting into a set is O(Log(N)) where N is the number of elements in the set / size of the set. Next, this is where I am building my adjacency list which calls the insertNode method I explained previously. Thus, the time complexity of this would be O(|E|) , where |E| is the number of edges in the value (linked list). Thus, the overall time complexity of this method would be O(|E| + Log(N)).

My final method is the PageRank method. First, I have nested for loops where the outer loop iterates through all of the keys in the adjListMap (map containing the strings (keys) and AdjacencyList (values)), then the nested for loop iterates through the respective keys values (it iterates through the linked list of the key). Within the nested for loop there is an if condition that tries to find if the specific node is a key of the adjacency list. The find method is an O(Log(N)) operation where N is |V| or the number of vertices (keys) in the map. The inner for each loop will iterate for the number of nodes in the specific keys linked list, thus this time complexity is O(N) where N is the number of nodes / vertices a specific key has. Finally, the outer for loop will

iterate again for O(M) times where M is the number of keys in the map. The overall time complexity of this first nested for loop is O(M*N*Log(|V|))

Next, I have another for each loop that will iterate through the adjListMap I have and set each of the adjListMap keys and map them to values of the initialPage rank. Since this is dependent on the number of unique keys in the adjListMap, the time complexity of this operation is O(N) where N is the number of vertices or unique keys.

Similarly, my next for each loop does something similar except rather than populating an oldRank map, I populate a newRank map and initialize the adjListMap keys to 0.0. This whole operation is O(N) where N is the number of unique vertices or keys.

Again, somewhat similarly, I have a new map for the outDegree values of each node that will be used for a page rank calculation. I have a for each loop for the entire adjListMap values. Within that loop, I access each key's value (or the singly linked list) and call the function to get the list as a vector (which I explained previously as O(N) where N is the number of nodes in the linked list), and then I call the size function on the vector which is O(1). Then I populate the outDegree map by setting the adjListMap key as the key in the outDegree map and then setting its value to be the size of the returned vector. Thus the time complexity of this is O(N * M) where N is the number of nodes in the linked list for the key, and then M is the number of keys in the adjListMap.

Next, I have a nested for loop that creates a reversedAdjList where the keys contain a linked list of nodes that point to it rather than a list of nodes that are being pointed to by the key. The first outer for loop iterates for the number of keys in the adjListMap, the inner for loop then iterates through each of the keys linked list. While doing this, it sets the key of the reversedAdjList to the first string value of the linked list, and then starts building a vector of the keys that point to it which is O(1). Ignoring the getNodes function complexity, the overall complexity of this is O(N * M) where M is the number of keys in the adjListMap and M is the number of vertices or nodes in the keys linked list.

Next, I have a for loop that will iterate through all of the keys in the adjListMap, it then checks if the key is found within the oldRank. This is because dangling nodes have not been initialized in the oldRank map. The find method is O(Log N) where N is the number of keys within the oldRank map, and the for each loop complexity is O(M) where M is the number of keys in the adjListMap, thus the complexity is O(M Log N).

Finally, I have my final page rank for loop that takes all of the previous maps I explained above and uses them to calculate the pageRank values. This outer for loop will iterate for O(P - 1) times or just O(P) where P is the number of power iterations passed in. In the first part of the for loop, I iterate through the adjListMap keys and set the corresponding keys values to 0 in the newRank. This is to reset the newRank values, this operation is O(M) where M is the number of keys in the adjListMap. Next, I have a for loop that will again iterate through the number of keys in the adjListMap O(M), it then calls the find function on the reversedAdjList which is O(Log N) where N is the number of keys in the reversedAdjList. If the key is found, another for each loop is entered that will iterate through the vector of the key in the reversedAdjList and calculate the

sum of the oldRank value and the outDegree value for the key. This operation is O(F) where F is the number of elements in the vector. Thus the final time complexity for this section is O(P * (M + F)). After all of this I have a separate for each loop that goes through all of the oldRank page rank values and prints them which is O(M) for the number of keys in the oldRank map. Thus, for PageRank, the (simplified) overall time complexity of this function is O(N Log M + P * N) where N is the total number of edges in the graph |N|, where M is the total number of unique keys or vertices in the graph |V|, and P is the number of power iterations performed.

What is the computational complexity of your main method in your implementation in the worst case in terms of Big O notation

Since the main method depends on pretty much all of the aforementioned methods, along with the number of inputs I will use my previously mentioned time complexity analysis for all of the functions, O(N Log M + P * N) , and combine that with input complexity. There is a for loop that will iterate for the number of lines mentioned, which contains the from and to values, and it builds the adjacency list based on this. Thus, the time complexity of this for loop is O(N Log M + N) where N is the number of edges |E| in the graph and M is the number of vertices or unique keys |V| in the graph. The time complexity of the PageRank function is, as mentioned, O(N Log M + P * N) where N is the total number of edges in the graph |N|, where M is the total number of unique keys or vertices in the graph |V|, and P is the number of power iterations performed. Thus the time complexity for the main method is O((N Log M + N) + (N Log M + P * N)).

What did you learn from this assignment and what would you do differently if you had to start over?

I learned how to draw maps efficiently on paper through my efforts of debugging. I also learned how to use for each loops efficiently to iterate through maps. However, I think I could have made the program a little more efficient. One thing I could have done differently if I started over would be to put more thought into the data structure I was using to represent the graph. I was too far in to change the data structure from my ordered map of strings and linked lists. I would have changed it to an ordered map of strings and a vector of pairs of strings and integers. This would have made my page rank calculations a bit easier. Thus, I would sit down and seriously consider the drawbacks and benefits of using a particular data structure for an assignment and see how that might affect the different functions I would have to implement later on in the code.