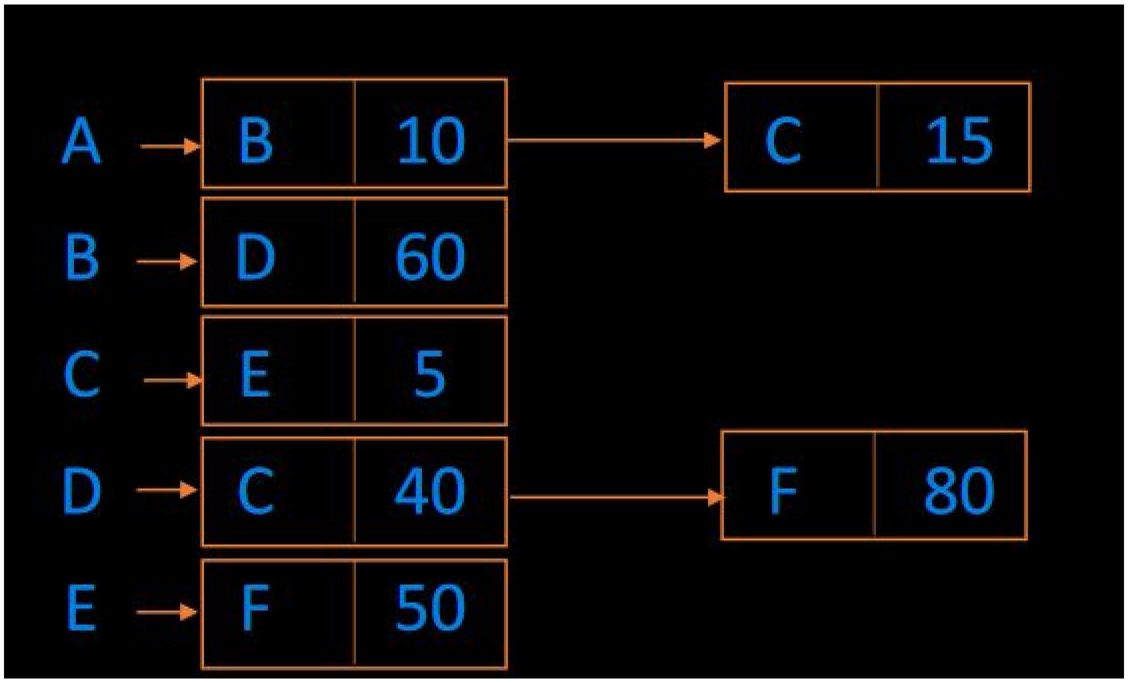
Russell Abedeen
COP 3530
November 8, 2020

<u>**Project 2: Graphs and PageRank**</u>

Implementation

I chose to implement the graph using C++ vectors. My *AdjacencyList* class consisted of a vector of *Vertex* variables, called *graph*, where *Vertex* was a data type I created to represent each graph's vertex. The *Vertex* class consisted of a *string* variable called *site* to store the url stored at the node, a *double* variable called *rank* to store the rank, and a *vector* of *strings* to store the url variables of the other objects that the *Vertex* object links to. Since this is a directed graph, only the urls of the *Vertex* objects that the object links to, and not the urls of the *Vertex* objects that link to the given object, are stored.

I chose to use this representation after looking at the depiction of adjacency lists in lectures.



(from Graphs-2.pdf, https://ufl.instructure.com/courses/406617/files/folder/Fall%202020/Powerpoints/Aman%20Presentations?preview=51799823)

Looking at this depiction gave me the idea to display the graph as a vector that stored, as a vector, all that a node pointed to. I thought that implementing each "row" of this display as a vector would be easy to work with. I ended up implementing each element of the vector as a separate *Vertex* class so I could store the values of the url and rank of the vertex as well as a vector of all urls of the objects that the vertex linked to.

<u>**Computational Complexity**</u>

void AdjacencyList::insertEdge(string from, string to)
This program inserts an edge from the *Vertex* object in *graph* whose *site* member variable matches *from* to the *Vertex* object in *graph* whose *site* member variable matches *to*. It first checks if there exists a *Vertex* object in *graph* that has a *site* member variable that matches *from* or *to*. If there are no objects in *graph* whose *site* variable matches *from*, a new *Vertex* object will be created whose *site* variable is set to *from*. After being assured that *Vertex* objects exist in *graph* that have *site* variables equal to *from* and *to*, the program adds an edge between the two objects by adding *to* to the *links* variable of *from*, which is the vector that stores the urls of *Vertex* objects that a given *Vertex* object links to.

To find the index of the *Vertex* whose *site* matches *from* and *to* in *graph* requires a traversal of *graph*. No matter what, the program must traverse *graph*, so its average, worst case, and best case complexities are all the same.

Best case: Ω(N), where N is the number of vertices in the graph
Worst case: O(N), where N is the number of vertices in the graph
Average case: Θ(N), where N is the number of vertices in the graph

void AdjacencyList::pageRank(int iteration)
This method first loops through each element in *graph* and sets its *rank* to the reciprocal of the number of elements in *graph*. This requires N operations in every case.

Next, it calls the *genMatrix()* function, which requires N^2 operations in every case.

Finally, it calls the *powerIter()*, which requires N^2 operations in every case. It calls this function *iteration* - 1 times.

In total, this function will always perform N + N^2 + n * N^2 operations, where N is the size of *graph*, which is the number of vertices in the graph, and n is the value passed into the function as *iteration*.

Best case: Ω(n * N^2), where N is the number of vertices in the graph and n is the value passed into the function as *iteration*
Worst case: O(n * N^2), where N is the number of vertices in the graph and n is the value passed into the function as *iteration*
Average case: Θ(n * N^2), where N is the number of vertices in the graph and n is the value passed into the function as *iteration*

void AdjacencyList::genMatrix()
This method creates the matrix used to calculate ranks. It uses a vector of vector of doubles called *matrix* to represent the matrix; this vector is a member variable of *AdjacencyList*. The method then looks at each element in *graph* and checks what elements in *graph* link to it. This requires two loops, one nested in the other. The first loop creates a vector called *toAdd* and then loops through *graph* to examine each element *i*. The other is within the first loop and also

loops through *graph* to check if a given element *j* at an index specified by the second loop links to *i*. If *j* does not link to *i*, 0 is added to a vector created within the first loop (but before the second, nested loop). If *j* does link to *i*, the reciprocal of the size of the *links* vector in *j* is added to the vector. After the nested loop finishes, this vector is added to *matrix*. This method must perform as many traversals of *graph* as there are elements in *graph*, so it must perform N^2 operations, where N is the size of *graph*, which is the number of vertices in the graph. Its average, best and worst cases are all the same.

Best case: Ω(N^2), where N is the number of vertices in the graph
Worst case: O(N^2), where N is the number of vertices in the graph
Average case: Θ(N^2), where N is the number of vertices in the graph

void AdjacencyList::powerIter()
This method is what updates the ranks of the *Vertex* objects in *graph*. It loops through every *i*-th element in *matrix* and, for each element, loops through every *j*-th value, and multiplies these values by the *rank* of the *j*-th *Vertex* in *graph*. It takes the sum of these products and assigns this new value to the *rank* of the *i*-th value in *graph*. This process requires looping through every value in *matrix*, which requires N^2 operations, where N is the size of *graph*, which is the number of vertices in the graph. This is done in every case that this method is called, so its average, best, and worst cases are all the same.
Best case: Ω(N^2), where N is the number of vertices in the graph
Worst case: O(N^2), where N is the number of vertices in the graph
Average case: Θ(N^2), where N is the number of vertices in the graph

void AdjacencyList::printList()
This is a helper function used for debugging. It loops through *graph* and prints the *site* variables of each *Vertex* object in *graph* as well as the values in each *Vertex* object's *links* variable. In every case, this requires traversing every vertex and edge in the graph, and thus performs E * V operations, where E is the number of edges of the graph, and V is the number of vertices in the graph.

Best case: Ω(V * E), where V is the number of vertices in the graph, and E is the number of edges in the graph.
Worst case: O(V * E), where V is the number of vertices in the graph, and E is the number of edges in the graph.
Average case: Θ(V * E), where V is the number of vertices in the graph, and E is the number of edges in the graph.

void AdjacencyList::printRanks()
This prints the *site* of each *Vertex* object in *graph* as well as the *rank* of each *Vertex* in *graph*. Before doing so, it sorts *graph* in alphabetical order. Printing requires traversing each element in *graph*, and thus will always perform N operations. Sorting is implemented with insertion sort and, in the best case, also performs N operations, but in the worst and average cases, performs N^2 operations. N is the size of *graph*, which is the number of vertices in the graph.

Best case: Ω(N), where N is the number of vertices in the graph
Worst case: O(N^2), where N is the number of vertices in the graph
Average case: Θ(N^2), where N is the number of vertices in the graph

vector<Vertex> AdjacencyList::sort()
This method sorts *graph*. It uses insertion sort, which performs N operations in the best case, but performs N^2 operations in the worst and average cases, where N is the size of *graph*, which is the number of vertices in the graph.

Best case: Ω(N), where N is the number of vertices in the graph
Worst case: O(N^2), where N is the number of vertices in the graph
Average case: Θ(N^2), where N is the number of vertices in the graph

Reflection

This project was less frightening than it seemed at first. While graphs were a cool, interesting new data structure, they were also one that I was not familiar with, as I had never worked with one before. I also did not have as much time as I had earlier in the course to do stepik exercises to familiarize myself with graphs. However, once I looked at the depiction of adjacency lists shown in the lecture, I was quickly able to figure out a method of implementing the graph as an adjacency list. I was already familiar with matrix multiplication, so the main challenge was figuring out how to generate the matrix and update the ranks of each vertex from the graph. This assignment has therefore given me valuable experience of working with graphs and visualizing graphs, which are exciting structures that I look forward to working with in the future. The only thing I would do differently would be to change which algorithm I used for sorting. Insertion sort does not work well for smaller data sets, so implementing an algorithm that used insertion sort for small graph sizes and another algorithm such as quicksort for larger sizes would be ideal.