Mauricio Benjamin Mossi

Project 2 Page Rank.

Preface

For this project, I developed two main APIs: one for a custom-tailored adjacency list specific to our use case, and a PageRank class which utilizes the adjacency list. It follows that the AdjacencyList class handles all graph-specific work, while the PageRank class handles all PageRank algorithm-related work. The PageRank algorithm computes the rank of website **a** by finding all websites **c** that link to website **a** and adding the result of the rank of all websites **c** times the weight of the respective website **c**, where the weight is just one over the outgoing links of **c**.

$$rank(a) = \Sigma 1/\, C_{out} * rank(c)$$

Therefore, our Page Rank algorithm requires a data structure able to store all websites pointing to any website ***a*** and able to retrieve all websites pointing to website ***a***. Therefore, our data structure needs to implement methods for adding links and for retrieving all incoming links to any website ***a***.

The data structure selected was the adjacency list because the graph representation of websites and their links is likely to be sparse. How many websites point to all websites? Or at least a several hundred? Not that many. Besides the adjacency list has the characteristics that storing the incoming edges per node resembles storing all incoming links to a website. Therefore, the retrieval of such list is quite easy; we just query for the adjacency list of website ***a***.

As mentioned above, for the Page Rank algorithm we require a data structure that implements methods for adding links and retrieving all incoming links to any website ***a***. Consequently, our AdjacencyList class will only provide those methods, it is everything needed to solve the project.

Methods and Time Complexity Analysis.

<AdjacencyList>.insertEdge [from : int][to : int]: The insertEdge method inserts an edge into the graph. The graph utilizes an unordered map as its underlying data structure to store vertices and their corresponding adjacency lists. The 'from' parameter represents a vertex and serves as one of the keys in the unordered map, while the 'to' parameter represents a value to be pushed into the vector containing the adjacent vertices of 'from'.

Since the underlying data structure is an unordered map, insertions are typically constant time on average but can degrade to O(n) in the worst case due to collisions. Inserting a value into the vector of adjacent vertices has a time complexity of O(1) on average (amortized), but can be up to O(n) in the worst case if resizing is required.

If in the worst case scenario, the number of adjacent vertices for each vertex is proportional to the total number of vertices (e = O(n)), then the worst-case time complexity of the insertEdge method becomes O(n^2), as it involves O(n) insertions into vectors, each potentially requiring O(n) time due to resizing.

<AdjacencyList>.getAdjacentVertices [source : int]: The getAdjacentVertices method retrieves all vertices adjacent to a given source vertex. The underlying data structure of the AdjacencyList is an associative container containing a vertex as its key and a vector of vertices as its value. Therefore, to retrieve the list of adjacent vertices, we find the key representing the source vertex in the map data structure.

Since searching for an element in an unordered map is typically a constant time operation on average, retrieving the key representing the source vertex is also a constant time operation on average. However, in the worst case scenario, where all vertices hash to the same bucket, retrieving the key could take O(n) time, where n is the number of vertices in the graph.

Overall, the time complexity of retrieving the list of adjacent vertices for a given source vertex using the getAdjacentVertices method is typically O(1) on average, but can degrade to O(n) in the worst case scenario.

<PageRank>.insertEdge[from : string][to : string]: The insertEdge method translates the incoming string parameters into integers so they can be inserted into the adjacency list member of the class, while keeping count of each website's outgoing links. To achieve this, the PageRank class utilizes a tree-based map to translate incoming strings to integers and a vector to hold the number of outgoing links for each website. The tree-based map ensures that websites can be printed alphabetically.

The insertEdge method performs a lookup on the tree-based map to search whether the 'from' and 'to' vertices are present. If either vertex is not present, a translation per vertex is created, and an entry is added to the vector to represent the newly created translation and its outgoing links. If both vertices are present, the number of outgoing links for the 'from' vertex is incremented. After both cases, the insertEdge method of the adjacency list member is called. It's important to note that the order of the 'from' and 'to' vertices is flipped so that the adjacency list per vertex represents the incoming vertices, i.e., the incoming links per website.

In the worst-case scenario, lookups and insertions on the tree-based map are logarithmic O(log n), where n represents the number of vertices. Similarly, insertions into the vector are linear O(n) in the worst case. Insertions into our adjacency are constant time O(1) on average but quadratic O(n^2) in the worst case due to collisions.

Therefore, the worst-case time complexity of the insertEdge method can be represented as O(log n + n + n^2), which simplifies to O(n^2), where n represents the number of vertices in the graph. All data structures—the translation map, the vector, and the adjacency list—will always contain the same number of items corresponding to the number of vertices in the graph.

<PageRank>.setDefaultRank[]: The setDefaultRank method initializes a rank vector containing the initial ranks of the websites. To achieve this, the method iterates over the number of vertices and pushes a float value per vertex with the initial value 1/vertices.

If the vector needs to resize during the insertion of elements, it involves reallocating memory, copying the existing elements, and deallocating the old memory block, resulting in a linear time complexity operation. Since we perform n such insertions, where n is the number of vertices, the worst-case time complexity for initializing the rank vector becomes O(n^2).

Therefore, the worst-case time complexity of the setDefaultRank method is O(n^2), where n represents the number of vertices in the graph.

<PageRank>.powerIterate[n : int]: The powerIterate method performs the specified n number of power iterations to compute the rank of each website.

During each iteration, the method copies the ranks of the websites to an array, which is linear in the number of vertices. Then, it performs a matrix-vector multiplication operation, where the ranks represent a vector and the weights per vertex represent the matrix.

For each vertex in the graph, the method retrieves its adjacent vertices from the adjacency list. In the worst case, fetching the adjacent vertices for a given vertex is an O(v) operation, where v is the number of vertices. Then, for each adjacent vertex, the method calculates a dot product between the rank of the adjacent vertex and its weight. This operation is repeated for all adjacent vertices of the given vertex.

Therefore, the time complexity for each power iteration is O(v + v*e), where v is the number of vertices and e represents the average number of adjacent vertices per vertex.

Since we perform n power iterations, the overall time complexity of the powerIterate method is O(n * (v + v*e)), where n is the number of iterations, v is the number of vertices, and e is the average number of adjacent vertices per vertex, if we take the adjacent vertices per vertex to be proportional to the number of vertices, the time complexity simplifies to O(n*v^2).

<PageRank>.getRankString[]: The getRankString method computes a string containing information about the rank of each website. Developed primarily for testability purposes, this method initializes a stringstream and iterates over the translation ordered map, which contains mappings from website names to integers. This iteration is linear in the number of websites.

For each item in the translation map, a string is inserted into the buffer. The time complexity of inserting strings into the buffer is proportional to the length of the longest website name. Although the inserted string length is constant except for the website name, we'll assume that the insertion operation could potentially involve resizing of the buffer in the worst case.

Therefore, in the worst case, the time complexity of this method is O(n*w), where n is the number of websites and w is the length of the longest website name.

<PageRank>.printRank[]: The printRank method prints the string computed by getRankString, thus its time complexity is directly dependent on the time complexity of getRankString, which is O(n*w), where n is the number of websites and w is the length of the longest website name. Therefore, the printRank method also has a time complexity of O(n*w).

<main>[]: The main method uses several of the methods above. It computes and displays the page rank for a given input. Where the input is the number of edges in the graph, the number of power iterations to be performed, and the edges themselves. The methods waits for std::cin for the input mentioned before, the input of importance is the number of edges, since <PageRank>.insertEdge is called per edge in the graph. After all inputs have been entered <PageRank>.setDefaultRank is called to initialize the rank of the websites, the power iterations specified in the input are performed by calling <PageRank>.powerIterate, and finally <PageRank>.printRank is called to print the rank string. The time complexities of the methods of PageRank are specified above, where the inputs for the methods are n the power iterations and v the number of edges. In the worst case the number of edges is proportional to the number of vertices. Therefore the time complexity of the main method is O(v*v^2 + v^2 + n * v^2 + v*w), which simplifies to O(v^3+n*v^2 + w*v), where v is the number of vertices, n the number of power iterations and w the length of the longest website.

Learning.

While doing this assignment I learned the importance of designing well-structured programs. This program took me significantly less than Project 1 as I designed the program before starting its implementation, therefore I had a good idea of what to implement and how to do so. For example, through this process I realized the AdjacencyList should be agnostic, not concerned with PageRank and its details like rank, website ordering. This allowed me to use an unordered map for the AdjacencyLIst implementation and an ordered map as a translation between website strings in PageRank and the integer representation of a graph in AdjacencyList. If I had to start over, I would implement the unit tests per each small feature as in the long run I spent a lot of time manually debugging and testing code. One feature that allowed me to test my code was the getRankString method which creates a string representing the output, with this method I was able to significantly speed up development.