**Page Rank Documentation**

By Aarithi Rajendren

* I used an adjacency list to implement the graph because the graph is usually a large, sparse graph since most web pages may not have direct links to other web pages. So, an adjacency list only stores the incoming and outgoing edges for each node but in two different unordered maps. I stored the edges in unordered maps since it allows for only the existing edges to be stored, which limits any excess space being used unnecessarily. Thus, using an adjacency list and unordered maps allowed me to use space efficiently and make doing operation such as insertion, search, and deletion more efficient as well.
* Computational complexity of each method:
  + addEdge: this function performs several operations that all contribute to the overall worst case time complexity of the function. For all these time complexities, n represents the number of elements/nodes in the unordered map. The push\_back function for vectors has a worst-case time complexity of O(n) since in the case of a collision all keys will be placed in the same bucket so the function might have to traverse the entire list of n elements to find the correct position to be inserted. The find function would have a time complexity of O(n) in the worst case since in the case of a collision, all elements will be placed in a single bucket so the map must traverse the entire list to find the key. Also, incrementing the key will have a worst-case time complexity of O(n) since to update the key, the key must be found first, which has already been established to have a worst case time complexity of O(n). Thus, the overall worst case time complexity of the addEdge function is O(n) where n represents the number of elements in the unordered map.
  + setUpInitialRank: this function has a worst case time complexity of O(V) where v represents the number of vertices (pages) in the graph. This is because this function simply loops through all the vertices and initializes their rank. The initialization of the ranks has a time complexity of O(1) so this does not affect the function’s overall time complexity.
  + computeRankUpdates: this function has a worst-case time complexity of O(V\*E) where v represents the number of vertices (pages) and e represents the number of edges, specifically incoming edges, in the graph. This is because the outer for loop, loops through all the vertices and the inner for loop, loops through each incoming edge associated with that particular vertex. So, for each vertex, all its incoming edges must be looped through, which is why the time complexity is V\*E. The actual updating of the ranks has a worst-case time complexity of O(1) so it does not affect the function’s overall time complexity.
  + displayRanks: this function has a worst-case time complexity of O(n logn) where n represents the number of elements in the page\_ranks map. This is because the sort function has a worst-case time complexity of O(n log n) due to the underlying algorithm used like quicksort or merge sort. Finally, iterating through every element in the map to display what is in the final map would have a worst-case time complexity of O(n) since it would have go through each page in the final map to be able to display the name of the page and its associated rank. Thus, the overall worst-case time complexity of the function would be O(n log n).
  + executePageRank: this function calls some of the other functions that have already been analyzed. setUpInitialRanks has a worst-case time complexity of O(V), displayRanks has a worst-case time complexity of O(n logn), and computeRankUpdates has a worst-case time complexity of O(V\*E), however, computeRankUpdates will be called k number of times where k represents the number of iterations that need to be performed. So, this will result in the for loop having a worst-case time complexity of O(k\*V\*E). So, when the worst-case time complexity of all the functions are added together, the final worst case time complexity of the overall function is O(n logn) + O(k\*V\*E). n represents the number of elements in the map, v represents the number of vertices, and e represents the number of edges.
  + generateRankString: this function has a worst-case time complexity of O(n logn) where n represents the final entries in the map, which will be displayed. This is because the function first copies the elements in a map and puts them in vector, so if there are n elements in the map then the operation takes O(n) time. The sort function as mentioned earlier has a worst-case time complexity of O(n logn). Finally, the function iterates through the sorted vector so if there are n elements in the vector the operation takes O(n) time. So, the overall function’s worst-case time complexity is O(n logn).
  + buildFromInput: the for loop of this function has a worst-case time complexity of O(n\*E) where n represents the elements in the map and E represents the number of edges in the map. This is because as established earlier the addEdge function has a worst-case time complexity of O(n) and this function is called E number of times, resulting in a worst-case time complexity of O(n\*E). The executePageRank function has already been established to have a worst-case time complexity of O(n logn) + O(k\*V\*E). So, the function’s overall worst-case time complexity is O(n\*E) + O(n logn) + O(k\*V\*E).
* The main function has a worst-case time complexity of O(n2) + O(n logn) + O(k\*V\*E). n represents the number of pages/elements being added to the map, k represents the number of iterations of page rank, v represents the number of vertices in the graph, and e represents the number of edges in the graph. This is because the for loop has a worst-case time complexity of O(n2) since it has already been established the addEdge function has a worst-case time complexity of O(n) and this function is called n number of times, resulting in a time complexity of O(n2). Similarly, executePageRank has already been established to have a worst-case time complexity of O(n logn) + O(k\*V\*E). This results in the main function having a worst-case time complexity of O(n2) + O(n logn) + O(k\*V\*E).
* The main thing I learned from this assignment is that depending on the problem given there are more appropriate data structures that can be used. This assignment could have been done with an adjacency matrix, however, it is important to realize that an adjacency list is a much more time and space efficient data structure to use for this problem. This helped me realize the importance of data structures and knowing when to implement each data structure. The one thing I would do differently if I had to start over was make sure I understood what page rank is doing in each iteration on paper before trying to implement it in code. When I started this project, I was not very confident with what this assignment was asking for so trying to code it ended up being frustrating in the beginning. However, once I watched some videos and read the paper on page rank I got a better understanding of what I am trying to do and then coding it was simpler.