0 Rationale

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

I used three things to make the graph. The main item that makes up the majority of the graph is an adjacency list implemented via a `std::unordered_map` whose first component is an `int`, which I called the index. The map's second component was a `std::vector` of `std::pair`s whose first component was an integer and whose second component was a `double`. The first component in the pair would represent the index j of the vertex which the parent vertex would be adjacent to. The second component is used to contain the weight, which would be $1/d_j$, the reciprocal of the outdegree of the vertex with index j. I used this implementation because it seemed to make the most sense for storing both the weight and the adjacency data, and I also saw how Prof. Kapoor showed us in the lectures, so I thought this made the most sense.

These outdegrees d_j were stored in another `std::unordered_map` with again the first component being `int`s and the second components being `double`s. I computed the outdegrees while placing all of the items in the adjacency list, and then stored them in this map so that I could then adjust the weights of the items in the adjacency list after all of the "links"/edges were added. This was done for simplicity, I did not want to store these outdegrees in the same structure as the one the graph itself was being encoded into since I was worried about confusing myself.

One more item was used in order to create a bijection between indices and website urls: I used a `std::map` whose first component was a `std::string` and whose second component was of course an `int`. This map is ordered, so this made printing out the final ranks in alphabetical order easier as well.

1 Documentation

What is the computational complexity of each method in your implementation? Reflect for each scenario: Best, Worst and Average.

Let ℓ be the number of lines passed in as input, then let $|V|$ be the number of vertices in the graph made after passing in all the lines into `main`.

1. `AddSite` In the worst case, we have that adding another vertex to the map of websites-indices is going to take $O(|V| \log |V|)$ complexity, same as the average case. This is when we successfully add a site to the map. If the site is already added to the map, then it will not add it to the map, so that only requires $O(\log |V|)$ complexity, which is the best case.

2. `AddLink` In all cases we have to find the index of the "to" url, which takes $O(\log |V|)$ complexity (sorted map). Then we push back on the vector a pair containing the from index and a default weight, which I will treat as taking constant time complexity. Hence the time complexity in all cases is $O(\log |V|)$.

3. `PlaceWeight` This method seeks to replace all the default weights in the adjacency list with the correct weights, computed via taking the reciprocals of all of the outdegrees (which are computed as links/edges are

added to the list). In the best and average cases we should find that the graph is a sparse graph, so that the number of lines ℓ is approximated by $|V|$. This means that in going through the whole list (a traversal, essentially) to replace the weights with $1/d_j$, we should expect the traversal to take $O(|V|)$ complexity. In the worst case this traversal should take $O(|V|^2)$ complexity since we no longer have a sparse graph (so $\ell \propto |V|^2$), and so the adjacency list resembles a dense adjacency matrix.

4. `RankInit` This method initializes the rank vector, which is equivalent to setting the number of power iterations equal to 1. This will take $O(|V|)$ in all cases since this function just initializes an array of size $|V|$ on the heap with all values set to $1/|V|$.

5. `IterateOnce` This method is simulating sparse matrix multiplication with a vector without doing any multiplications with zero. In the best case the adjacency list is very sparse, so that the vector associated to the each index is of a very small size compared to the number of vertices - treat this as constant relative to the number of vertices. So in taking the repeated "dot products" we are only doing a constant number of operations for every component of the rank vector, and we do them for every such element in the rank vector, which is size $|V|$. So in the very best case, we have a complexity of $O(|V|)$. In the average and worse case, we cannot assume anymore that the adjacency list is sparse, though for this average case we may expect it to be partially sparse (this estimation is an overestimate). In the case where there are many edges, so that the number of edges, or the size of the vector associated to each index in the adjacency list, is proportional to the number of vertices, then the complexity is increased. Because we are no longer doing a constant number of operations per index/element of the rank vector in taking the "dot product", we instead take around $O(|V|)$ operations. And we do this $|V|$ times, so the complexity in the average or worst case (leaning more towards the worse case, in the average case it would never be this bad) is $O(|V|^2)$.

6. `PageRank` There are no differences in how this function works in any cases since all this does is print the rank for every url in alphabetical order. The function traverses via an iterator all of the urls in the original bijection map from the string urls to access the items at every corresponding index of the rank vector after the "sparse matrix multiplication" is done. Because of the way we are iterating throught the map, the complexity will always be $O(|V|)$ since we have to traverse all of the vertices/urls. (accessing the index numbers and hence the rank is constant since we are using an iterator)

What is the computational complexity of your main method in your implementation? Reflect for each scenario: Best, Worst and Average.

Let p be the number of power iterations passed into the main method.

`main` The main method is going to use all of the above functions in a very direct manner. Constructing the `PageRanker` object as well as initializing some variables would take constant time complexity. Then we call `AddSite` and `AddLink` 2ℓ and ℓ times, respectively. We can consider what happens in the best case as having the number of lines be proportional to the number of unique urls (so then $\ell \propto |V|$). We also must note that by constraints we will not have repeated lines (no parallel edges!). Then we can say that the complexity of this

process will be bounded above by $O(|V|^2 \log|V| + |V| \log|V|) = O(|V|^2 \log|V|)$. In the average and worst cases let us say that $\ell \propto |V|^2$, so that we get complexity bounded above by $O(|V|^3 \log|V| + |V|^2 \log|V|) = O(|V|^3 \log|V|)$. Then `PlaceWeight`'s contribution, $O(|V|)$ in the best/average case, and $O(|V|^2)$ in the very worst case, are both subsumed by the previous complexity. Same goes for `RankInit` which is $O(|V|)$ in all cases. Then observe that the complexity for the next loop where we iteratively do the "sparse matrix multiplication" with `IterateOnce` is either $O(p|V|)$ in the best case or $O(p|V|^2)$ in the average/worst case, so then we add that on as well. (Optionally we may choose to drop this since we know that the values of p are constrained to be far smaller than the number of urls/lines.) Finally we consider `PageRank` which is $O(|V|)$ complexity, and can be dropped. So the initial loading of the graph dominates the time complexity of `main`, and so the final complexities are $O(|V|^2 \log|V| + P|V|)$ in the better/best cases and $O(|V|^3 \log|V| + p|V|^2)$ in the more dense/average/worst cases.

2 Reflection

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

I learned that I should not be scared of trying to compute these things by hands, at least for sake of understanding the algorithm. I initially went off of the instructions that Prof. Kapoor gave us, and thought that it would be easy enough to simply implement it directly using my head, but I was being very arrogant. I eventually got confused and had to try computing his example by hand and comparing that with what my code was doing to correct some mistakes. If I had to redo this project, I would spend more time just trying to understand the algorithms on paper first before trying to program anything.

Citations for the project:

Kapoor, A. (2021). Graphs-2. Gainesville, FL; University of Florida.