In this report, I am going to illustrate different parts of my project along with the complexity analysis of some processes.

First of all, to correctly run the project you should have the following files with the exact names, all letters are small:

1. keywords.csv: save each website with its keywords
2. webgraph.csv: save the web graph

There are 2 other files but they do not need to exist at the beginning as the program itself will initiate the data for each website and put it in these files. If they do exist at the beginning, then the program will read them and update them at the end:

1. impressions.csv: saves each website name along with its impression. If the impressions for a certain website is not given, it will be set to 1 in order to avoid dividing by zero when calculating CTR (CTR=impressions/clicks)
2. clicks.csv: saves each website name along with its clicks. If the cliks for a certain website is not given, it will be set to 0.

There is a last file that is not given which is called “pagerank.csv”. In that file, we save the pagerank of each website after the first run of the program using the info given in “webgraph.csv”. After that, we just retrieve pagerank of each website from the file as we **assume the webgraph does not change.**

**Main Data Structures Used:**

The upcoming are the main data structures used, and all of them are global so they are seen by all functions as they are frequently used.

* vector<Website> websites:

This is a vector of all websites given from the input. (Website is a class defined to have all the information of a website)

* unordered\_map<string, vector<int>> inSensitiveKeyWords:

This hash map maps key words to vector of indices of the websites that contain these key words **and the case of the keywords are not important.** The indices here are the indices of the websites in the vector websites

* unordered\_map<string, vector<int>> SensitiveKeyWords:

This hash map maps key words to vector of indices of the websites that contain these key words **in the exact case.** The indices here are the indices of the websites in the vector websites

* unordered\_map<string, int> WebsitesByName:

This hash map maps a website name (URL) to its index in the vector websites.

* vector<int> retrieved:

This vector saves the indices of the websites retrieved from the last done search. Initially, it is empty.

Now, I will discuss some of the components of my project:

1. **Website Class:**

I have a class called Website, and I create instance for each website where the information of it is saved there and can be accessed and updated easily.

Each website has variables:

* string name;
* int impressions;//Initially set to 1 if not given
* int clicks;//initially set to 0 if not given
* int index: index of the website in a global vector websites
* double PR; page rank

And here are some but not all methods:

* void incrementImp(); void incrementClick();\\for updates
* double getCTR();
* double getScore();//calculates the score based on the given formula
* void setImpressions(int impressions);
* void setClicks(int clicks);
* void setPR(double PR);

And the class has static int variable called count which represents the count of websites created.

1. **Indexing Pseudo Code & Complexity:**

I used 3 hash tables for the process of indexing websites, and assigning keywords to websites in order for the retrieval to be fast. The first one is WebsitesByName (<string, int> )which maps the URL of the website to the index of it in websites vector.

The second one inSensitiveKeyWords(<string, vector<int>> ) maps lower the keywords (all lower case) and maps each one to a vector of websites indices (we get it from the first map) that contain these words.

The third one is SensitiveKeyWords (<string, vector<int>> ) is similar to the second one except the case of the key words matters.

The mapping happens when reading the key words using this code



Also, if we found a new website in another file that was not in the keywords file(that may happen because of a human mistake while writing the input files), we index it using the same way

**Complexity:**

Assume we have N websites, and a total of K keywords

* The algorithm only loops over the websites and add insert website to the hash map (insertion is O(1)), so hashing websites to its indices takes O(N) time.
* The algorithm loops over all K keywords and for each one, it finds it in the map, O(1), then it inserts an index of the associating website in the vector, O(1), so this indexing uses O(K) time.
* So, the total time is O(N+K).

For the space complexity, the WebsiteByName hash map have N websites, so it uses O(N) memory

For the other two maps, assume every keyword has an average of x website, so each map would use O(K x) memory as we have K words and x websites for each one

1. **Websites Ranking Algorithm and Pseudocode:**

The website ranking differs depending on whether we are running the program for the first time or not. As for the first time, we will build a graph. Otherwise, we will just use the data saved to get the page rank and then continue sorting.

Let’s assume we are at the first iteration, so I make an instance of a class called webgraph which has the following variables:

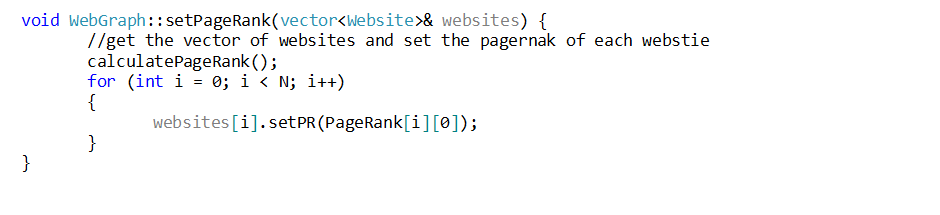
* vector<vector<int>> adjlist;
* vector<vector<int>> adjlistTranspose;//adjlist of the transpose of the graph to help us calculate the page rank
* int N;//num of vertices (Websites
* vector<vector<double>> PageRank:

2D matrix of size N\*2 that saves the previous and current iteration of calculating Page Rank.

To calculate the page rank, we first add all the edges using the following function in the main:



The construction of the graph takes **O(E)** as we just loop over the edges and the method addEdge takes O(1)

Then we call the method SetPageRak and give it the websites vector

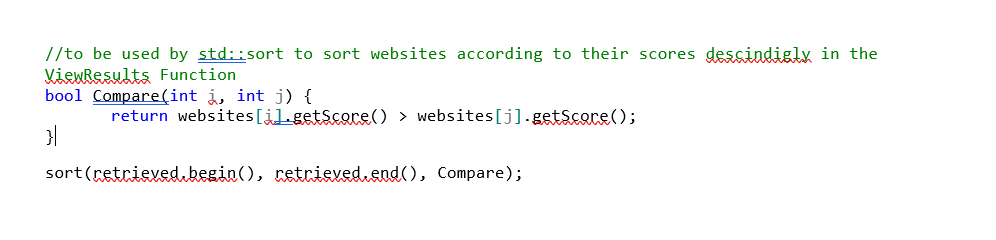
It calls CalculatePageRank() which calculates the rank, and then it sets the rank in O(N) where N is the number of websites.

Now, we see CalculatePageRank()



The Function only loops Over each website, and for each website it goes to all its neighbors, so it takes O(N+E) time, and then it makes at most 100 iterations which is constant.

After that, we use the std::sort of c++ to sort the websites when we requested to view them after any search query. We direct the function to sort them descending according to their scores as follows



As getScore() is O(1), it only uses a formula. Then this part’s complexity is O(N\*logN)

So, the total time complexity in the worst case, when the graph is constructed is O(N\*log N+N+E+E)=O(N\*log N+E)

In case the graph is already constructed, so we wil have the last sort only which takes O(N\*log N)

As for the space complexity, the graph itself uses adjlist which takes O(N+E) memory. And then, we use PageRank 2D vector which takes O(2N)=O(N) space, and the sorting if done out of place it would take O(N), so we would have a total of O(N+E) space Complexity in the case of the graph.

In the case of not constructing the graph, we will only need O(N) memory if the sorting is done outside of the graph.

**Design Tradeoffs:**

1. I used a hash map instead of a normal map in the indexing because hash maps have constant time in insertion and retrieval. However, maps are balanced binary search trees with insertion in O(log(n)) and retrieval O(log(n)).
2. I decided use a hash map to map the keywords to a vector of website indices and not the other way around (I don’t map the website to its key words). This is mainly to avoid iterating over all the websites to check whether a keyword present or not. Using my approach, I will just check if the key word exist in O(1) file, and then print its associated websites.