Mina Ashraf Gamil

900182973

Analysis and Design of Algorithms Lab Project report

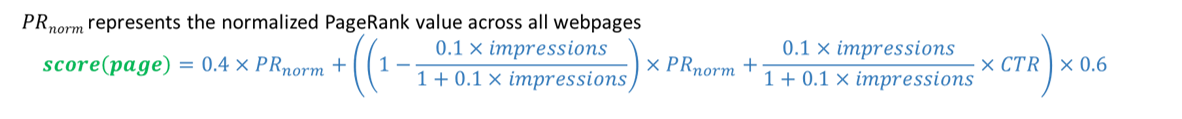
**Search engine using PageRank algorithm**

**General explanation of the project:**

The program takes multiple flies to start the search engine initialization. These files are the list of which websites point to which websites, a list of keyword for each website, and a number of impressions and clicks for each website.

After this initialization is done, the program would compute the page rank and the score of each website which would later be used to return the results of the search where websites of higher ranks are shown at the top of the list. Then, the program receives a search query from the user which can only be a simple query of either ANDs or ORs. All the websites that fit the search criteria are then added to a vector, sorted by their score, and displayed to the user. The user then has an option to view each website, which would increase its CTR, or to perform a new search, or to get back the search results.

The score formula used is:



**pseudo code:**

The indexing part of the project is assumed to include everything from reading from the files in the beginning till the searching for the websites that include our search keywords.

1. Search engine initialization:

* Open the web graph file and read the two websites from it (Source and destination)
* For each of them, check if it was not mapped before, and then map its link (string) to a unique ID which we will use later. Then map each ID to a website pointer. If the website was mapped before, increment the number of its outgoing links if it was the source. If it was the destination, add the ID of the website that was pointing to it to a vector of integers.
* Now we load the keywords of each website by opening the file and then reading it line by line and accessing the map and pushing back the word to a vector of strings.
* We do the same for clicks and impressions.

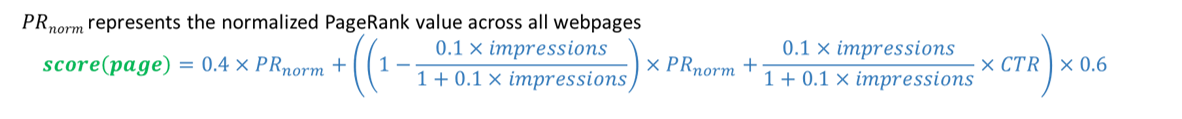
1. Calculating page rank:

* Initialize all websites page ranks to 1 / number of websites
* loop over all websites and for each of them we loop over all the websites that were pointing to it (which we already added to a vector in the previous step).

For each of them, we calculate the result of dividing the page rank by the number of outgoing links of that website and adding it to the page rank of our website that we are trying to calculate its page rank.

* We repeat this process for x times, where x is calculated by a factor epsilon that check the difference between the last iteration page rank and the current one.

1. Calculate the score from the following formula



1. Finally, we use set union and set intersection operations to perform the searching operation.

Page rank Pseudo code

For I =0 🡪 x {

For (W in Websites) {

Rank = 0;

For (u in W.websites\_pointing\_to\_W){

Rank += u.page\_rank() / u.number\_of\_outgoing\_links

}

w.set\_page\_rank(Rank);

}

For (W in Websites)

W.prev\_page\_rank = w.current\_page\_rank

w.current\_page\_rank = 0

}

Complexity of above page rank:

as this is code is repeated x times and each time we loop over the adjacency list of all websites, which will be equal to the number of nodes in a fully connected graph, hence the W2. Thus, the complexity is

Space complexity is constant as we don’t create any new data structures except for a single variable. Thus, **O(1)**

For the indexing (searching for websites by keywords)

After parsing the search keywords, we append them to a vector of keywords and send them to the respective search handler (Or/And).

Before this step we sort all keywords ids in O(kn log n) where K is the number of keywords and n is the number of ids per keyword

Or\_Handler(){

Vector search result

For (keyword : Keywords){

If(Keyword is mapped to a vector of websites){

Search\_result = set\_union(Keyword.vector\_of\_websites, search\_result);

}

Else skip this keyword

}

}

And\_Handler(){

Vector search result

For (keyword : Keywords){

If(Keyword is mapped to a vector of websites){

Search\_result = set\_intersection(Keyword.vector\_of\_websites, search\_result);

}

Else skip this keyword

}

}

Complexity of above search:

the sorting in (K N log N) happens only 1 time and is not repeated any more in the project. But every search call, we perform K (number of keywords) set intersections or set unions. Since, we sorted the ids that are going to unionid or intersected, the operation of set\_union and set\_intersection is Linear in time. Thus, the overall complexity of searching is **O(KN)** where K is the number of keywords in the search query and N is the max number of IDs per keyword.

After this, we sort the returned results by their score in descending order. Which happens in another

(N log N) where N is the number of websites in the returned vector of search results.

Making the overall search complexity

Space complexity is O(KN) since we merge N ids per Keyword

**Data structures used:**

I tried to follow an object-oriented design for clarity as well as maintainability of the project. The main data structure that was used is the C++ Map from the STL. This is used for direct access to save speed, but it affects the space to achieve this.

Vectors of integers for IDs, strings for keywords, website object pointers were also used to store the relevant data in an array-like structure that we can iterate over. The choice of the vectors was based on their efficient memory allocation and dynamic resizing that makes them superior to normal arrays.

Custom-designed classes:

1. Search engine class which contains
   1. File paths and file streams to the 4 main files for the project
   2. Maps strings (website links) to integers (unique IDs)
   3. Maps each unique ID (int) to a pointer to a Website object
   4. Contains a map for each keyword (string) to a pointer to a vector of integers that are the IDs of each website.
2. Website class which contains
   1. Link
   2. Score
   3. PageRank
   4. Keywords
   5. Number of outgoing links
   6. Many other elementary data types.

Design tradeoffs:

Usage of Maps makes data access faster at the expense of space. Thus, instead of having an O(N) access time to find the website we are looking for, it is reduced to 2 \* Log N for the usage of normal maps.

I could have used unordered maps to handle this, but it has a worst-case complexity of O(N) so I sticked with the normal map for this task as the difference gets huge for larger data. This was a tradeoff as I could have used the unordered maps also for O(1) direct access but this is not guaranteed.

To do the search in O(KN) after the initial sorting, we had to sacrifice storing a list of websites for each separate keyword which uses a lot of memory. To solve this, I replaced each website with an 4-byte integer that points to it instead and then I can use that integer as a unique ID to get the website from the original map. Thus, we save tremendous amount of space by only increasing 1 map and the cost of an additional lookup. Thus, 2 log N instead of Log N alone, but that was the cost of saving space and making the program memory friendly and space efficient.