Lab Project

Seifeldin Elshabshiri

November 2021

1 Introduction

Before the internet and the digitisation of files, libraries were the only place were one could get information that he or she desired. In order to look for a book in a library, a system had to be put in place were each book was given an index. Books were strategically placed in order to make finding it as easy as possible. Similarly, websites and files must be given an index and be placed in an easily accessible place where searching for them can be as easy and effective as possible. This is why the reason search engines were created: in order to make navigating through websites as easy as possible.

1.1 Page Rank

Not all websites were created equal, some are more important than others, this is why page rank was invented. Page rank is an algorithm that was invented by Larry Page and Sergey Brin that ranked each website based on their importance. The importance of a certain website was determined by the amount of websites that pointed to it and their page ranks.

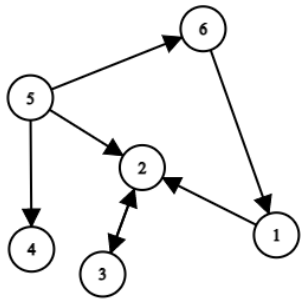

Figure 1: Directed web graph

For example in figure 1 node 2 would have the highest page rank because it has the highest number of nodes pointing to it, thus website 2 has the highest page rank.

1.1.1 Page Rank Calculation

To calculate page rank for a certain website, we first have to initialize each website with a page rank, so that they all have the same page rank. A good value to initialize them by is $\frac{1}{totalnumberofwebistes}$.
Now that we have the initial page ranks we can start to get the real value of all the websites' page ranks.

$$PR(t) = \sum_{i=0}^{n} \frac{PR(i)}{C(i)}$$

n: the number of links pointing to a certain page t
PR(i): the page rank for a website from the websites that point to page t
C(i): the number of hyperlinks that for a website i

1.1.2 Page Rank Algorithm

Algorithm 1 Page Rank

```
1:  procedure PR(pages)
2:      r ← a mod b
3:      for i ← 0 to pages.size do
4:          prI ← 1/pages.size
5:          pages_i.updateInitialPR(prI)
6:      end for
7:      for i ← 0 to pages.size do
8:          for j ← 0 to pages_i.numOfFromLinks do
9:              pr+ = PR_j / Pages_j.numOfHyperLinks
10:         end for
11:         pages_i.updateInitialPR(pr)
12:     end for
13:     for i ← 0 to pages.size do
14:         for j ← 0 to pages_i.numOfFromLinks do
15:             pr+ = PR_j / Pages_j.numOfHyperLinks
16:         end for
17:         pages_i.updatePR(pr)
18:     end for
19: end procedure
```

1.1.3 Time Complexity Of Page Rank

The page Rank of each website is calculated in two iterations to get a better estimate of the page ranks.

$$\sum_{i=0}^{n} 1 + \sum_{i=0}^{n} 1 \sum_{j=0}^{m} 2 + \sum_{i=0}^{n} 1 \sum_{j=0}^{m} 2 = n + 2 * m * n + 2 * m * n$$

The worst case will be if m = n

$$\therefore$$

$$O(n) = n^2$$

1.1.4 Space Complexity of Page Rank

For the page rank to work, two adjacency lists were used to one for the hyperlinks on each page, and one for the links that point to each page.

$$\therefore$$

The space complexity for the first adjacency list is n+m, where n is the number of pages and m is the number of hyperlinks.
The space complexity for the first adjacency list is n+p, where n is the number of pages and p is the number of links that point to that page.

$$\therefore$$

The total space complexity will be n+(m+P) because there will be n pages and each page will hold two lists.

1.2 Indexing and Search Query

1.2.1 Indexing Algorithm

```
unordered_map<string, vector<Article>> Search;
while (has not reached end of file) {

                read line


                put that line in a stringstream
                while (has not reached the end of the stringstream) {
                        getline(ss, keyword, ',');
                        for (int i = 0 to pages.size) {
                                if (one of the words == one of the urls) {
                                        ind1 = i;
                                }
                        }
                        if (keyword != page[ind1].url) {
                                pages[ind1].addKeyword(keyword);
                                Search[keyword].push_back(pages[ind1]);
                        }


                }
```

1.2.2 Input for search query function

The search query function will receive a map of vectors of pages indexed by strings, which will be the keywords

Keyword 1	page 1, page2, page 3
keyword 2	page 4, page 6, page 3
keyword 3	page 1, page 2
keyword...n	...

The table above is a sample input for the search query function

1.2.3 Search query algorithm

```
vector<Article> getResults(string entry, unordered_map<string, vector<Article>> &search) {
        string first;
        string second;
        string del1 = " AND ";
        string del2 = " OR ";
        if (Qoutations are found in entry) {
                Remove qoutations
                return search[entry];
        }
        else if(del1 is found) {
                first = first word before del1
                second = second word after del1;
                vector<Article> a;
                sort(search[second]);
                for (i = 0 to search[first].size()) {
                        if (search[first][i] is found in search[second]) {
                                a.push_back(search[first][i]);
                        }
                }
                return a;
        }
        else if (entry contains del2 ) {
                first = first word before del2;
                second = second word after del2;
                vector<Article> a;
                for (i = 0 to search[first].size()) {
                        a.push_back(search[first][i]);
                }
                sort(a);
                for (ii = 0 to search[second].size()) {
                        if (search[second][i] is not in a) ){
                                a.push_back(search[second][i]);
                        }
```

```
                }
                return a;
        }
        else {
                vector<Article> a;
                for (i = 0 to search[entry].size()) {
                        a.push_back(search[entry][i]);
                }
                sort(a);
                stringstream s(entry);
                while (s contains words separated by a space) {
                    first = each word in entry;
                        for (i = 0 to search[first].size()) {
                                if (search[first][i] is not in a) {
                                        a.push_back(search[first][i]);
                                }
                        }
                }
                return a;
        }
}
```

1.2.4 Search Query Space Complexity

The websites are stored in a map as a vector indexed by keywords.
m: number of keywords
n: number of websites in the vector

$$\therefore$$

The space complexity of the map is n*m.
The function also uses a vector of websites to store the resulting pages
t: total number of websites

$$\therefore$$

The space complexity of the resulting vector is at most equal to t.

1.2.5 Search Query Time Complexity

The time complexity varies for the type of search entered. The best case scenario is that the user enters a phrase that is in between quotations, which will result in a time complexity of $\Omega(n)$. This is because the substr (which has a time complexity of $O(n)$) function is used to remove the quotations.
The worst case scenario will be if the user enters m number of keywords in the search without "AND" or "OR". This will go to the fourth case which will: Insert all the pages that have the entire string as a keyword, which will take $O(n)$.

Then sort that vector which will take O(nlogn)
Then loop over the entire string while using binary search to check if a certain page exists in the sorted vector which will take O(nlogn*m)

$$\therefore$$

$$T(n, m) = O(n + nlogn + nlogn * m)$$

2 Data Structures Used

- unordered map: An unordered map was used. As discussed in section 1.2.1, the map was used in the search query function and made in the indexing function.
- Vector: m+2 vectors were used, where m is the number of keywords, because each keyword in the map has a vector. In addition a vector of all pages was used and a vector of pages in the search query was also used.

3 Design Trade-Offs

- Using two adjacency lists instead of one, By doing this space complexity was traded for time complexity. This is because, the adjacency list that contains the links that point to each page can be attained from the adjacency list that contains the hyperlinks that are on each page, however this would take a time complexity.
- Using an unordered map instead of searching in the file or vector for each keyword. Doing this also trades space complexity for time complexity. Because having an unordered map decreases the time complexity for accessing a vector of websites that contains that keyword to O(1).