I will explain several aspects of my project and analyze the complexity of a some procedures in this report.

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. pageRank.csv: save the page rank calculations
3. pages.csv: save the web graph
4. impressions.csv: keep track of how frequently each website appears in the search results.
5. cicks.csv: record the number of times each website was visited..

**Main Data Structures Used:**

The primary data structures are listed below.

1. I implemented a graph class for the web graph.
2. I used 2d vectors of strings to read the keywords, number of impressions, and number of clicks from the files.
3. I used a vector of strings to split the sentence the user will type into keywords.
4. I used vectors of strings to process the queries that the user will type.

**Indexing Pseudo Code & Complexity:**

**Main functions and their time and space complexity**

**Split: it takes an sentence and return a vector of words.**

Function split( s, delimiter)

vector<string> words

word = “”

flag = true

Loop from i=0 to I =s.size()

if (s(i) == '\"')

i++;

while (s(i) != '\"')

word += s[i];

i++;

words.push\_back(word)

i++

word = ""

else

if (s[i] == delimiter)

words.push\_back(word)

word = "";

else

word += s[i];

if(word!="")

words.push\_back(word)

return words;

Time Compexity O(n) AND Space Complexity = O(n)

**Get\_Keywords: it returns the 2d vector of keywords.**

Function Get\_Keywords(fileName)

arr // 2D vector

index = 0

ifstream fin

line = “”;

open file whose name is fileName

while (getline(fin, line, '#'))

arr.push\_back(split(line, ','))

return return the table of keywords arr

Time Compexity O(n) AND Space Complexity = O(n)

**Removespace: Remove spaces from string**

Function removespace(s)

Loop I from 0 to s.length()

if (s[i] == ' ')

s.erase(s.begin() + i)

i--

return the word s without spaces

Time Compexity O(n) AND Space Complexity = O(1)

**read\_excel\_file: return it reads the graph and return the matrix of the graph.**

Function read\_excel\_file(file\_name)

2D-vector result;

ifstream file(file\_name)

line = “”

while (getline(file, line))

stringstream ss(line)

vector<int> temp

string word

while (getline(ss, word, ','))

word = removespace(word)

if (word.length() == 0)

continue

temp.push\_back(stoi(word))

result.push\_back(temp);

return result;

Time Compexity O(V+E) AND Space Complexity = O(V+E)

**read\_excel\_file\_2: the same as the previous function but has some modifications to work on the PageRank file.**

**calculatePageRank: it calculates the page rank for the web graphs.**

The website raking differs depending on whether we are using the programme for the first time or not. We will make a graph because this will be our first time. If not, we will simply calculate the page rank using the saved data before continuing the sorting.

Let’s assume we are at the first iteration, so we will calculate the page rank for the graph:

Function Calculate PageRank

// for designing the graph

Graph graph(number of websites = n)

ifstream in // to read form file

vector<string> pages

vector<vector<int>> arr = read\_excel\_file("Pages.csv") // read from file and return the graph matrix.

// for initializing the graph

Loop from 0 to number of web pages-1

Loop from 0 to the number of web pages that this page[i] is pointing to -1

graph.Add edge (arr[i][0], arr[i][j])

// start calculate the page ranks

vector<vector<double>> PageRank of size n // 2D vector that save the previous and current iteration

Loop from 0 to number of web pages-1

//Initilizaiton

PageRank(i).push\_back(1.0 / n)

PageRank(i).push\_back(0)

MIN = 1e9;//I will use it for normalization

MAX = 0;

//Update the page rank according to this video //<https://www.youtube.com/watch?v=P8Kt6Abq_rM>

Loop from 1 to 100

achieved = true

Loop from 0 to number of web pages-1

d = 0.85;//damping factor

cur = 0;

loop throughout the pages that web[i] points to

//all websites poinitng at webs

cur += (PageRank[j][(i - 1) % 2] / graph.get\_adj\_list(j).size())

cur \*= d

cur += (1 - d) / n

PageRank[webs][i % 2] = cur

MIN = min(cur, MIN)

MAX = max(cur, MAX)

achieved &= (abs(cur - PageRank[webs][(i - 1) % 2]) < 0.01);

if (achieved)

if (i % 2 == 1) //the answer in the second column, so we put it in the first column as well

Loop from 0 to number of web pages-1

if (MIN != MAX) //that would be prbably always the case as we used damping factor

PageRank[i][0] = (PageRank[i][1] - MIN) / (MAX - MIN)//normalizing usin Min-Max Normalizaition

ofstream out // open file

out.open("pageRank.csv");

Loop from 0 to number of PageRank -1

Print the page ranks in this file

for (int i = 0; i < n && MIN != MAX; i++)

{

PageRank[i][0] = PageRank[i][1] - MIN / MAX - MIN;//normalizing

}

The construction of the graph takes **O(V+E)**.

Then we initialize the page rank in O(V) where v is the number of pages in the graph.

Then it makes at most 100 iterations which is constant to update the page rank.

Print the page ranks in the file which takes O(V) where v is the number of pages in the graph.

**Find\_Keyword: returns all websites that have this keyword.**

Function Find\_Keyword(Keyword)

2D-vector web\_Keywords = Get\_Keywords("Keywords.csv")

vector<string> websites

Loop from 0 to web\_Keywords.size()-1

Loop from 2 to web\_Keywords[i].size()-1

if (web\_Keywords[i][j] == Keyword)

websites.push\_back(web\_Keywords[i][1])

return websites

Time Compexity O(n\*n) AND Space Complexity = O(n\*n)

**Find\_Similarity: return the keywords that are in both two vectors of strings.**

Function Find\_Similarity(vector<string> web1 , vector<string> web2)

vector<string> web3

Loop from 0 to web1.size()-1

Loop from 0 to web2.size()-1

if (web1[i] == web2[j])

web3.push\_back(web1[i])

return web3

Time Compexity O(n) AND Space Complexity = O(n)

**Is\_Exist: to find whether the key exists in a particular website.**

Function Is\_Exist(vector<string> web, string key)

Loop from 0 to web.size()-1

if (web[i] == key)

return 1

return 0

Time Compexity O(n) AND Space Complexity = O(1)

**Merge: merges the keywords that exist in two vectors of strings into 1 vector.**

Function Merge(vector<string> web1, vector<string> web2)

vector<string> web3;

Loop from 0 to web1.size()-1

web3.push\_back(web1[i])

Loop from 0 to web2.size()-1

if(!Is\_Exist(web3, web2[j]))

web3.push\_back(web2[j])

return web3;

Time Compexity O(n) AND Space Complexity = O(n)

**AddOR: it takes the sentence that the user type and if there is no link such as (AND & OR), it will add OR between them.**

Function AddOR(vector<string> keys)

k = 0;

Loop from 0 to keys.size()-2

if ((keys[i] != "AND" && keys[i] != "OR") && (keys[i+1] != "AND" && keys[i+1] != "OR"))

keys.insert(keys.begin() + i+1, "OR")

return keys

Time Compexity O(n) AND Space Complexity = O(n)

**UpdateClicks: it updates the number of clicks for the visited websites.**

Function UpdateClicks(string page)

{

vector<vector<int>> arr = read\_excel\_file("clicks.csv") cost n\*n

vector<vector<string>> pages = Get\_Keywords("Keywords.csv") // cost n\*n

pageNumber =0

Loop from 0 to pages.size()-1

if (pages[i][1] == page)

pageNumber = i

break

arr[pageNumber][1] = arr[pageNumber][1] + 1

ofstream out // open file

out.open("clicks.csv")

Loop from 0 to arr.size()-1

out << i << ',' << arr[i][1] <<'\n'

Time Compexity O(n\*n) AND Space Complexity = O(n)

**UpdateImpressions: it updates the number of appearances for the searched websites.**

Function UpdateImpressions(vector<string> pgs)

vector<vector<int>> arr = read\_excel\_file("impressions.csv") // it costs n\*n

vector<vector<string>> pages = Get\_Keywords("Keywords.csv") // it costs n\*n

pageNumber=0

Loop from 0 to pgs.size()-1

Loop from 0 to pages.size()-1

if (pages[i][1] == pgs[k])

pageNumber = i;

break;

arr[pageNumber][1] = arr[pageNumber][1] + 1

ofstream out // open file

out.open("impressions.csv")

Loop from 0 to arr.size()-1

out << i << ',' << arr[i][1] << '\n';

Time Compexity O(n\*n) AND Space Complexity = O(n)

**getScore: it returns the score of each website.**

**Function getScore()**

{

impressions = read\_excel\_file\_2("impressions.csv") // costs n\*n

pageRank = read\_excel\_file\_2("pageRank.csv") // costs n\*n

= read\_excel\_file("clicks.csv")

vector<double> scores

Loop from 0 to pageRank.size()-1

double frac = (0.1 \* impressions[i][1]) / (1.0 + 0.1 \* impressions[i][1])

scores.push\_back(0.4 \* pageRank[i][2] + ((1 - frac) \* pageRank[i][2] + frac \* clicks[i][1]) \* 0.6)

return scores

Time Compexity O(n\*n) AND Space Complexity = O(n)

**sort\_by\_value: it takes a vector of indexes that represents the websites and the score of each of them and sorts them descending.**

Function sort\_by\_value(vector<int> index, vector<double> values) {

vector<pair<double, int>> temp

Loop from 0 to index.size()-1

temp.push\_back({ values[index[i]], index[i] });

sort(temp.begin(), temp.end(), std::greater<pair<double, int>>())

vector<int> result

Loop from 0 to temp.size()-1

result.push\_back(temp[i].second)

return result

Time Compexity O(n\*n) AND Space Complexity = O(n)

**Sort: it takes the pages that have the same keywords I am searching for and then call the function sort\_by\_value to sort the websites.**

Function sort(vector<string> pages)

scores = getScore()

Loop from 0 to scores.size()-1

cout << scores[i] << " "

vector<int> numberPages

all = Get\_Keywords("Keywords.csv")

pageNumber=0

Loop from 0 to all.size()-1

Loop from 0 to pages.size()-1

if (all[i][1] == pages[j])

numberPages.push\_back(i);

result = sort\_by\_value(numberPages, scores)

vector<string> finalResults

Loop from 0 to result.size()-1

finalResults.push\_back(all[result[j]][1]);

return finalResults;

Time Compexity O(n\*n) AND Space Complexity = O(n)

**search: the main function that I am responsible for calls all other functions.**

Function search(fileName, words)

{

keywords = split(words,' ');

if(keywords.size() > 2)

keywords = AddOR(keywords)

webs1;

webs2;

webs3;

a = 0;

Loop from 0 to keywords.size()-1

if (keywords[i] == "AND")

if (a == 0)

webs1 = Find\_Keyword(keywords[i - 1])

webs2 = Find\_Keyword(keywords[i + 1])

webs3 = Merge(webs3, Find\_Similarity(webs1, webs2))

else if (a == 1)

webs1 = Find\_Keyword(keywords[i + 1])

webs3 = Find\_Similarity(webs3, webs1) // it costs n\*n

a = 1

else if (keywords[i] == "OR")

a = 0

Loop from 0 to keywords.size()-1

if (keywords[i] == "AND")

keywords.erase(keywords.begin() + i - 1, keywords.begin() + i+2)

i = 0;

else if (keywords[i] == "OR")

keywords.erase(keywords.begin() + i)

i = 0;

Loop from 0 to keywords.size()-1

webs3 = Merge(webs3, Find\_Keyword(keywords[i]));

return webs3;

Time Compexity O(n\*n\*n) AND Space Complexity = O(n)

**Search: it is responsible for the UI and waiting for the user to input his command and call the search function to perform this command.**

**Design Tradeoffs:**

1. I created a powerful algorithm that can search for all the keywords in any query, regardless of length, using the logical priority and will process first, then OR. For Example, it can search for a query like this (data structure AND algorithms OR java AND "learning" OR complexity); however, it costs n power 3 instead of n power 2 if I design the algorithm to work on only two keywords.