Name: Omar Ashraf Bahgat | ID: 900211747
Analysis and Design of Algorithms Lab Project Report (Fall 2022)

Search Engine using Google's PageRank Algorithm

Brief explanation of the project:

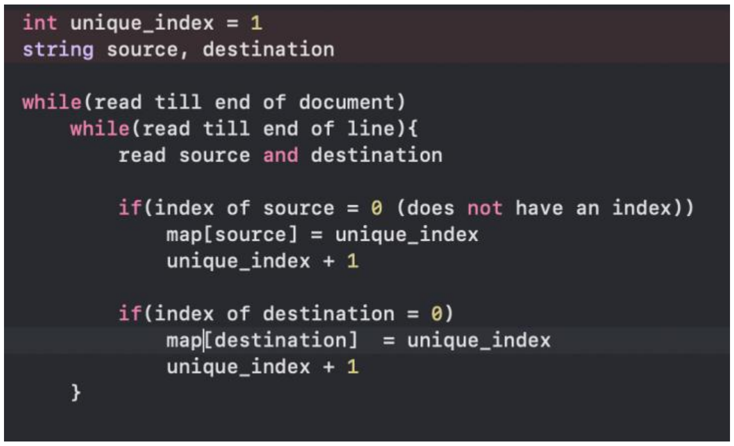
The program takes 4 files and initializes each of them to start the search engine. These files are the list of which websites point to which websites (webgraph), a list of keywords for each website, the number of impressions for each website, and the number of clicks for each website. After this initialization is done, the program would calculate and assign the page rank of each website in a vector with the help of the webgraph transpose. Then, the program receives a search query from the user which can only be a query of either two words separated by an AND, two words separated by an OR, statement or word in double quotations "", or plain text. All the websites that fit the search criteria are then added to a vector. Afterwards, each website in the vector has its score calculated and websites are then displayed in descending order of their score (based on PageRank and impressions and clicks) and the impressions of the websites that appeared in the search results are incremented by one. The user then has an option to view any of the websites, which would increase the number of clicks by one, to perform a new search, or to exit the program. When the user exits the program, the updated values for the impressions and clicks are saved onto a file and loaded when the program starts again. This way, updates won't be lost when the program ends.

1) Indexing Algorithm:

I used two unordered_maps for the process of uniquely indexing websites and mapping keywords to a vector of websites for fast and easy retrieval.

1- Mapping website URL to a unique index

Pseudocode:



```
int unique_index = 1
string source, destination

while(read till end of document)
    while(read till end of line){
        read source and destination

        if(index of source = 0 (does not have an index))
            map[source] = unique_index
            unique_index + 1

        if(index of destination = 0)
            map[destination]  = unique_index
            unique_index + 1
    }
```

Time Complexity:

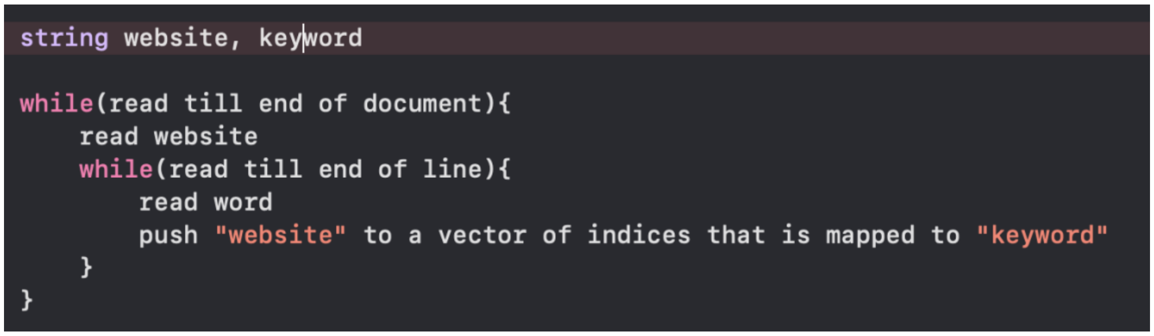
The algorithm loops over each website and assigns it a unique index. The insertion process itself takes O(1) time in an unordered_map so the overall complexity will be **O(N)** where N is the number of websites.

Space Complexity:

If there are N websites, then the unordered_map that maps website URLs to a unique index uses **O(N) memory.**

2- Mapping keyword to a vector of website indices

Pseudocode:



```
string website, keyword

while(read till end of document){
    read website
    while(read till end of line){
        read word
        push "website" to a vector of indices that is mapped to "keyword"
    }
}
```

Time Complexity:

Assuming there are M keywords and N websites, the algorithm loops over each keyword and inserts index of the website into the vector which takes **constant time.** Therefore, the time complexity for this indexing process **is O(M).**

Space Complexity:

If there are M keywords and each keyword has an average of N websites, then the unordered_map that maps a keyword to a vector of websites will use **O(MN) memory.**

2) Ranking algorithm:

Brief explanation:

Calculating PageRank:

- Initialize all websites page rank to 1.0 / number of websites.
- For each website, we loop through the websites pointing to it.
- For each of those websites, we divide its page rank by the number of edges going out of this website and adding it to the website we are calculating its page rank.
- We repeat this process 100 times as it guarantees good results.

Note: After each iteration, I save the page ranks in a temporary vector and initialize the original page rank vector to zero **(how the video in the PP does it).** When we reach the last iteration, I break out of the loop before zeroing the original page rank vector.

Pseudocode:

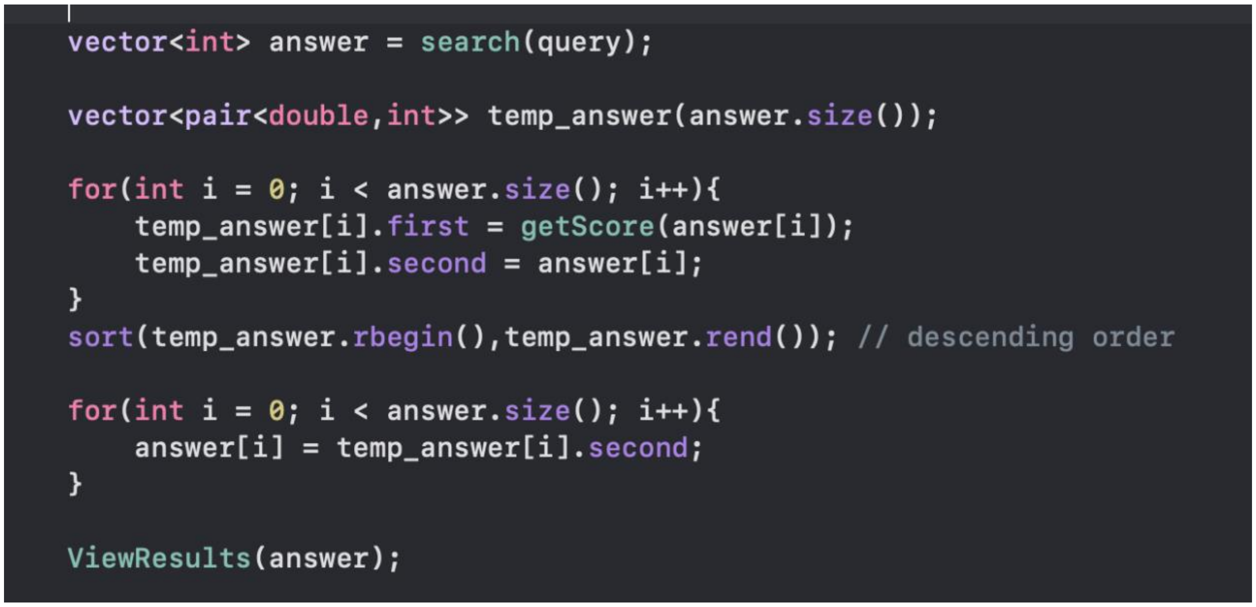
```
int N = number of websites // websites are indexed 1 to N
vector<double> PageRank  // stores page ranks of all websites

for i = 1 → 100{
    for( j = 1 → j = N){                    // j represents index of website
        for( all v : transpose){
            PageRank[j] = PageRank[j] + PageRank[v] / outgoing edges from v
        }
    }
    If (i = 100) break from loop
}
```

Time Complexity:

The function iterates 100 times. For each iteration, the function loops over each website, and for each website it goes to all its neighbors. However, since 100 is a constant, we won't include it in the complexity calculation.
Therefore, the complexity will be **O(V+E)** (vertices + edges).

However, after that, we must list the webpages in descending order of their score, so we use the built-in C++ sort function O(NlogN) to achieve this.

Code:



```
vector<int> answer = search(query);

vector<pair<double,int>> temp_answer(answer.size());

for(int i = 0; i < answer.size(); i++){
    temp_answer[i].first = getScore(answer[i]);
    temp_answer[i].second = answer[i];
}
sort(temp_answer.rbegin(),temp_answer.rend()); // descending order

for(int i = 0; i < answer.size(); i++){
    answer[i] = temp_answer[i].second;
}

ViewResults(answer);
```

Explanation:
After the vector containing website indices based on the search criteria is returned, we extract those website indices and put them in a vector of pairs where the first position of the pair is the score of the website and the second position is the index of the website. Score is retrieved in **O(1)** as it's a direct formula. Then, we sort the vector of pairs O(NlogN) in descending order and reassign the "answer" vector with the indices sorted in descending order of score.

Therefore, the time complexity of this part is **O(NlogN).**

Therefore, the total time complexity of the ranking process is O(NlogN + V + E).

Space Complexity:

For the webgraph, I store it in an adjacency list in a form of an unordered_map where the key of the map is the source node, and the value of the map is a vector containing all destination nodes that the source node points to. Assuming we have a fully connected graph where each graph is connected to all other nodes, the map will use **O(N^2) memory.** For the vector containing websites based on the search criteria, it will use O(N) space.

Therefore, the total space complexity for the ranking process is O(N^2)

3) Main data structures used:

All data structures listed below are heavily used in the program and all are declared globally for easy access throughout the program.

- **unordered_map<string, int> WebToIndex**
  - maps each website to a unique index

- **unordered_map<int, string> IndexToWeb**
  - retrieve website name from its index (used when displaying the websites)

- **unordered_map<int, vector<int>> adjlist**
  - stores webgraph
  - maps the source node to a vector of website indices that source points to
  - used in the PageRank algorithm to find number of outgoing links from a node

- **unordered_map<int, vector<int>> transpose**
  - stores webgraph transpose
  - maps the destination node to a vector of website indices that destination points to
  - used in the PageRank algorithm to find nodes pointing to a source node

- **unordered_map<string, vector<int>> keywords**
  - maps each keyword to a vector of website indices that contain the keyword
  - O(1) time complexity when searching for websites that contain required keyword

- **unordered_map<int, int> impressions**
  - maps each website index to its impressions
  - O(1) time complexity when retrieving impressions for a website

- **unordered_map<int, int> clicks**
  - maps each website index to its clicks
  - O(1) time complexity when retrieving clicks for a website

- **vector<double> PageRank**
  - stores page rank of each website
  - PageRank[1] stores the page rank of website with index 1 and so on.
  - O(1) time complexity when retrieving page rank for a website

4) Design Tradeoffs:

1. I used an unordered_map instead of normal map when indexing websites to numbers and websites to numbers as unordered maps in best and average cases work in **constant time** for insertion and retrieval. However, normal maps are based on self-balancing binary trees which work in **O(log(n)).**

2. I used an unordered_map to map each keyword to a vector containing website indices. I didn't do it the other way around (website to keyword) as this would require me to iterate all over the website searching for the keyword which could take O(n) time. Using my approach, I will check if the key word exists in or not in **constant time** and print the websites the keyword is included in.

3. In the search part where I search for two words, separated by an OR, I use an STL set instead of a merge function that returns all websites without duplicates. This could be easily achieved using a set and just inserting all websites in it as it **doesn't store duplicates.**