Project Report

Analysis and Design of Algorithms LAb

Ahmed Jaheen
900212943

Submitted to Dr. Reem Haweel
This assignment is prepared for CSCE2203, section 2

Computer Science and Engineering Department
The American University in Cairo
22/04/2023

Brief Illustration

Firstly, in order to properly execute the project, it is imperative that you possess the following files with their corresponding names, all in lowercase:

(i) **keywords.csv:** This file saves each website with its associated keywords.

(ii) **webgraph.csv:** This file saves the web graph

There are two other files that may or may not exist at the start, but the program will initiate data for each website and store it in these files if they do not exist. However, if they are present, the program will read and update them at the end of execution:

(i) **impressions.csv:** This file stores the name of each website along with its impressions. If the impressions for a specific website are not provided, they will be set to 1 to prevent division by zero when calculating the click-through rate (CTR), where CTR = (impressions/clicks).

(ii) **clicks.csv:** This file stores the name of each website along with its clicks. If the clicks for a particular website are not provided, they will be set to 0.

Finally, there is one last file that is not initially given, called **"pagerank.csv"**. In this file, we save the pagerank of each website after the first run of the program using the information provided in **"webgraph.csv"**. Subsequently, we retrieve the pagerank of each website from this file, assuming that the web graph does not change.

Component I - Main Data Structures

The following are the primary data structures utilized in the program, all of which are global in scope and thus accessible to all functions as they are frequently utilized:

(i) **vector<Website> websites:** This is a vector that contains all of the websites provided as input. The Website class is defined to contain all the relevant information for each website.

(ii) **unordered_map<string, vector<int> > inSensitiveKeyWords:** This hash map associates keywords with a vector of indices for websites that contain these keywords, with the case of the keywords being unimportant. The indices refer to the position of the websites in the websites vector.

(iii) **unordered_map<string, vector<int> > SensitiveKeyWords:** This hash map associates keywords with a vector of indices for websites that contain these keywords in their exact case. The indices refer to the position of the websites in the websites vector.

(iv) **unordered_map<string, int> WebsitesByName:** This hash map associates a website name (URL) with its index in the websites vector.

(v) **vector<int> retrieved:** This vector stores the indices of the websites retrieved from the most recent search performed. It is initially empty.

Component II - Website Class

I have implemented a class named **"Website"** that creates instances for each website and stores their relevant information for easy access and updating. Each website object has the following variables:

(i) A string variable **"name"** that stores the name or URL of the website.

(ii) An integer variable **"impressions"** that stores the number of impressions the website has received. If not provided, it is initially set to 1 to prevent division by zero when calculating the click-through rate (CTR).

(iii) An integer variable **"clicks"** that stores the number of clicks the website has received. If not provided, it is initially set to 0.

(iv) An integer variable **"index"** that stores the index of the website in a global vector called "websites".

(v) A double variable **"PR"** that stores the page rank of the website.

(vi) A static integer variable called **"count"** that represents the count of websites created, which can be accessed and updated from outside the class.

The class has several methods, including:

(i) A void **"incrementImp()"** and a void **"incrementClick()"** that are used to increment the impressions and clicks of the website, respectively, for updates.

(ii) A double method **"getCTR()"** that calculates and returns the click-through rate (CTR) of the website.

(iii) A double method **"getScore()"** that calculates and returns the score of the website based on a given formula.

(iv) A void **"setImpressions(int impressions)"**, a void **"setClicks(int clicks)"**, and a void **"setPR(double PR)"** that are used to set the impressions, clicks, and page rank of the website, respectively.

Component III - Indexing Pseudo Code & Complexity

In order to speed up the process of retrieving websites based on keywords, I utilized three hash tables for indexing websites and assigning keywords to them. The first hash table, **WebsitesByName (<string, int>)**, maps the URL of each website to its index in the websites vector. The second hash table, **inSensitiveKeyWords (<string, vector<int> >)**, maps each keyword in all lowercase to a vector of website indices (obtained from the first hash table) that contain that keyword, with the case of the keywords being unimportant. The third hash table, **SensitiveKeyWords (<string, vector<int> >)**, is similar to the second hash table, but it takes into account the case of the keywords.
The mapping process is done when reading the keywords, and it is accomplished using the following code:

```
void InitializeKeywords(){
        ifstream  KeyInput;
        KeyInput.open("keywords.csv");

        string name, word, line;
        while (getline(KeyInput, line))
        {
            stringstream s(line);
            bool first = true;
            while (getline(s, word, ','))
            {
            if (first) {
                websites.push_back(Website(word));
                WebsitesByName[word] = websites.back().getIndex();
                first = false;
                }
                else {
                SensitiveKeyWords[word].push_back(websites.back().getIndex());
                Lower(word);
                inSensitiveKeyWords[word].push_back(websites.back().getIndex());
                }
            }
        }
        KeyInput.close();}
```

In the event that a new website is found in a separate file that was not listed in the keywords file, it is indexed using the same methodology.

The **time complexity** of this indexing algorithm is $O(N + K)$, where N represents the total number of websites and K represents the total number of keywords. The algorithm loops over each website to hash it to its corresponding index (which takes $O(1)$ time per website), and then loops over each keyword to find it in the hash tables and add its index to the corresponding vector (also taking $O(1)$ time per keyword).

As for **space complexity**, the WebsitesByName hash table uses $O(N)$ memory for the N websites. The other two hash tables, inSensitiveKeyWords and SensitiveKeyWords, would use $O(xK)$ memory each, assuming that each keyword has an average of x websites associated with it.

Component IV - Websites Ranking Algorithm & Pseudocode

In order to rank websites, the approach taken depends on whether the program is being run for the first time or not. When the program is run for the first time, a graph is built. On subsequent runs, the existing data is used to calculate the page rank and continue sorting. Assuming that we are on the first iteration, an instance of the "webgraph" class is created with the following variables:

(i) **"adjlist"**, a vector of vectors of integers, representing the adjacency list of the graph.

(ii) **"adjlistTranspose"**, another vector of vectors of integers, representing the adjacency list of the transpose of the graph. This is used to help calculate the page rank.

(iii) **"N"**, an integer representing the number of vertices (i.e., websites) in the graph.

(iv) **"PageRank"**, a 2D matrix of size $N \times N$ that saves the previous and current iteration of calculating the page rank.

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

```
WebGraph G(Website::count); //this loop will actually construct the graph

        while (getline(GraphInput, line))
        {
            stringstream s(line);
            getline(s, web1, ',');
            getline(s, web2, ',');
            G.addEdge(WebsitesByName[web1], WebsitesByName[web2]);//add the edge
        }

        G.setPageRank(websites);//assign PR to each website
```

The construction of the graph takes $O(E)$ **time complexity** as we simply loop over the edges and the addEdge method has a constant **time complexity** of $O(1)$.
After constructing the graph, we call the SetPageRank method and pass in the websites vector.

```
void WebGraph::setPageRank(vector<Website>& websites){
        calculatePageRank();
        for (int i = 0; i < N; i++){
            websites[i].setPR(PageRank[i][0]);}}
```

After calculating the page rank using the method CalculatePageRank(), the SetPageRank method sets the calculated rank for each website in the PageRank matrix, which takes $O(N)$ time, where N is the number of websites.
Now, we construct the CalculatePageRank()

```
void WebGraph::calculatePageRank() {
        Set the number of websites as N
        Use 2D array PageRank of size N*2. Initialize PageRank[i][0]=1/N for all i
        double MIN := Infinity //Will be used for normalization
        double MAX := 0
        bool achieved:= false
        Do the following from i=0 to i=100 times or until abs(PageRank[w][1]
        -PageRank[w][0]<0.01) for all w<N:{
            for (int webs = 0; webs < N; webs++) {
                double d := 0.85;//Damping Factor
                double cur := 0;
                for (int j : adjlistTranspose[webs]) { //neighbors of webs
                cur += (PageRank[j][(i - 1) % 2] / adjlist[j].size());
                // PageRank of j in the previous iteration / its outDegree
                }
                cur *= d;
                cur += (1 - d) / N;
                PageRank[webs][i % 2] = cur;//set PageRank
                MIN = min(cur, MIN);
                MAX = max(cur, MAX);
                achieved &= (abs(cur - PageRank[webs][(i - 1) % 2]) < 0.01);

            if (achieved=true) {
            //page rank have stabilized, so we don't need more iterations
            //save the pageranks in the first Column in PageRank
            //but normalized through the following formula
PageRank[i][0] = (PageRank[i][1]-MIN) / (MAX-MIN)
return;
                }
}
        If we finished 100 iterations, then normalize here
}
```

The function loops over each website, and for each website, it traverses all its neighboring nodes, which takes $O(N+E)$ time, where N is the number of websites and E is the number of edges in the graph. The function then performs at most 100 iterations, which is constant. Therefore, the **time complexity** of the function is $O(N+E)$ since it dominates the number of iterations.
After the page rank calculation is done, we use the built-in sorting function in C++, **std::sort**, to sort the websites in descending order based on their scores. We pass a custom comparison function to **std::sort** which compares two websites based on their scores. The implementation of this function ensures that websites with higher scores appear first in the sorted list.

```
bool Compare(int i, int j){
    return websites[i].getScore() > websites[j].getScore();
}
sort(retrieved.begin(), retrieved.end(), Compare);
```

The **time complexity** of the sorting function is $O(NlogN)$ because getScore() is $O(1)$, and it only uses a formula. Therefore, the total time complexity in the worst case, when constructing the graph, is $O(NlogN + E)$. If the graph is already constructed, we only have the last sort, which takes $O(NlogN)$ time.

Regarding **space complexity**, the graph itself uses adjlist, which takes $O(N + E)$ memory. Additionally, PageRank 2D vector takes $O(2N) = O(N)$ space. If the sorting is done out of place, it would take $O(N)$ space, resulting in a total of $O(N + E)$ space complexity in the case of the graph. If we are not constructing the graph, we only need $O(N)$ memory for sorting if done outside the graph.

Component V - Design Tradeoffs

(i) I decided to use a hash map to map keywords to a vector of website indices instead of mapping websites to their keywords. This was mainly done to avoid having to iterate over all the websites to check if a keyword was present or not. With my approach, I can check if a keyword exists in $O(1)$ time complexity, and then print its associated websites, which can significantly improve the program's performance.

(ii) I chose to use a hash map instead of a normal map for the indexing process because hash maps provide constant time complexity for both insertion and retrieval. On the other hand, maps are balanced binary search trees, with insertion and retrieval taking $O(log(n))$ time complexity. By using a hash map, I was able to optimize the indexing process for faster retrieval.