Search Engine Report

Sherif H Gabr 900183120

CSCE 2203 – Spring 2020
Analysis and Design of Algorithms Lab

General Explanation of Program:

The program takes two initialization files to setup the webpage. The first file setups the graph of the webpages, which URLs are connected to which URLs. The second file setups the keywords for each URL.

Then after initializing the webpage, the program takes in a command. The command could be to search for a URL that contains specific keywords or to update URLs. The search command could be a "AND" or an "OR" depending if keywords are used with quotations or not, respectively. The AND searches for a URL the contains all keywords entered in the command. The OR searches for a URL that contains at least one keyword entered in the command. To update URLs, a update file must be entered to the command. The updates the can occur are adding a new URL, completely removing a URL, or updating certain parameters of the URL. The parameter that could be modified are the number of impressions, number of click-though, and adding or removing graph edges to a certain URL.

In the query command, the program displays URLs based on a rank that is calculated from the page rank normalized, number of impressions, and click-through rate. This equation is:

$$Rank = 0.4 * PR_{norm} + 0.6 * \left(\left(1 - \frac{0.1 * imp}{1 + 0.1 * imp} \right) * PR_{norm} + \left(\frac{0.1 * imp}{1 + 0.1 * imp} \right) * CTR \right)$$

$$where \quad PR(u) = \sum_{u \in G(v)} \frac{PR(v)}{E(v)} \quad , where\ G(v)\ is\ outgoing\ edges\ from\ node\ v\ and\ E(v)\ is\ the\ number\ of\ edges$$

$$where \quad CTR = \frac{clickthrough}{number\ of\ impressions} \quad , where\ impressions\ is\ the\ number\ of\ times\ a\ URL\ has\ been\ served$$

The URLs are sorted accordingly in descending order. They are sorted using quick sort with the partition as the first element. Then, it searches for keywords inside each URL consecutively. So to make the searching efficient, it is implemented using binary search. For it to work, the keywords of each URL must be sorted.

Pseudocodes and Complexity of Algorithms:

Let's use some symbols for calculating the complexity:

- N : the number of URLs
- M: the number of keywords
- K: the number of keywords to search for (max 10 so it can be ignored)
- G: the number of lines in graph file
- W: the number of lines in keywords file
- U: the number of lines in update file

- **Indexing (sorting keywords)**

```
Sorting:
    Int  partition (A[0…s], int p, int r){

        Pick pivot as first element A[p]
        Save the starting index (i=p) and ending index (j=r) for later

        Loop while i is less than j
            Loop while A[i] is less than the pivot and each time increment i
            Loop while A[j] is greater than the pivot and each time decrement j
            Check if i<j then swap A[i] with A[j]

        Then save the A[j] element in the pivot's location A[p]
        Then save the pivot to A[j]
        Now j has the correct element so return the location j
    }
    Void quick_sort (A[0…s], int p, int r){

        If p is less than r then continue
            Call partition with (A,p,r+1) and save the returned value in q
            Call quick_sort with the lower half of array (A,p,q-1)
            Call quick_sort with the upper half of array (A,q+1,r)
    }
```

Each URL contains a vector of keywords, where it must be sorted. So they are sorted by quick sort (only after initialization). So the complexity is N * the complexity of quick sort algorithm.
The average number of comparisons of algorithm is $T(m) = \frac{1}{m-1}\sum_{i=1}^{m-1}[T(i) + T(m-i) + m]$
Solving the recurrence, we get: $T(m) = \frac{2}{m-1}\sum_{i=1}^{m-1}[T(i) + m]$, $where\ T(1) = 0$

We get two equations: $(m-1)T(m) = 2\sum_{i=1}^{m-1} T(i) + m(m-1)$

$(m-2)T(m) = 2\sum_{i=1}^{m-2} T(i) + (m-1)(m-2)$

Subtracting both equations: $\frac{T(m)}{m} = \frac{T(m-1)}{m-1} + \frac{2}{m}$

Take $F(m) = \frac{T(m)}{m}$: $F(m) = F(m-1) + \frac{2}{m}$, $where\ a_m = 1\ \&\ b_m = \frac{2}{m}$

Linear Recurrence: $F(m) = \sum_{i=2}^{m-1}\frac{2}{i} + \frac{2}{m} = 2\ln m + \frac{2}{m}$

Return $T(m)$: $T(m) = 2m\ln m + 2$

$T(m) = \frac{2m\log m}{\log e} + 2$

So the average time complexity of the sorting algorithm is $O(m\log m)$
Then the time complexity for sorting all URLs is $O(n) * O(m\log m) = \mathbf{O(nm\log m)}$

The sorting is done in-place no extra space is needed.

- **Indexing (searching for keywords)**

```
Searching:
        bool  binarySearch (A[0…s], key, int start, int end){

                if start is greater than end then key is not found

                calculate mid of array (start+end)/2

                check if key is less than A[mid] then call binarysearch with lower half of array
                check if key is greater than A[mid] then call binary search with higher half or array
                check if key is equal to A[mid] then key is found
        }
```

After sorting, the keyword is searched through the vector by a binary search implementation. So the complexity is N* the complexity of binary search algorithm. The worst number of comparisons of algorithm is $T(m) = T\left(\frac{m}{2}\right) + 2$

Recurrence with $c = 2, a = 1, f(n) = 2, l = \log m, T(1) = 1$: $T(l) = T(l-1) + 2$

Solving linear recurrence, we get: $T(l) = 1 + \sum_{i=2}^{l-1} 2 + 2$

$T(l) = 1 + 2l - 2 + 2$

Solving it back for T(m), we get: $T(m) = 1 + 2\log m$

So the complexity for searching the vector is $O(\log m)$

Then the complexity for searching through all URLs is $O(n) * O(\log m) = \mathbf{O(n \log m)}$

Binary search does not require any additional space.

- **Ranking (calculating PageRank)**

```
computePR (URL [0…n], graph[0…n][0…n]){

        create new array (npr) of size n and initialize it to 0
        set a max variable to 0

        for j=0->n-1
                for i=0->n-1
                        if graph[i][j] == 1
                                npr[j] += PR of URL[i] / Number of edges of URL[i]
                if npr[j]>max
                        set max to npr[j]

        for i=0->n-1
                if npr[i]!=0
                        PR of URL[i] = npr[i]
                        PRN of URL[i] = npr[i]/max;
                Else
                        PRN of URL[i] = npr[i]/max;
}
```

Each URL has a certain page rank that must be calculated as soon as the graph is modified. The array accesses is the significant factor in this algorithm. So the number of array accesses operations is $T(n) = \sum_{i=0}^{n-1} 2 \sum_{j=0}^{n-1} 4 + \sum_{i=0}^{n-1} 5 = 8n^2 + 5n$
So the algorithm has a complexity of $\mathbf{O}(\boldsymbol{n^2})$
The algorithm also requires additional space of size n so the space complexity is $\mathbf{O}(\boldsymbol{n})$

- **Ranking (sorting URLs)**

Each URL contains a specific rank that differs from URL to another and the greater the rank, the better the URL is (will appear higher in searching for keywords). So, the URLs must be sorted based on their rank in descending order. It is also implemented using quick sort.

```
Sorting based on Rank (descending order):
    Int  partition (A[0…s], int p, int r){

        Pick pivot as first element A[p]
        Save the starting index (i=p) and ending index (j=r) for later

        Loop while i is less than j
            Loop while A[i] is greater than the pivot and each time increment i
            Loop while A[j] is less than the pivot and each time decrement j
            Check if i<j then swap A[i] with A[j]

        Then save the A[j] element in the pivot's location A[p]
        Then save the pivot to A[j]
        Now j has the correct element so return the location j
    }
    Void quick_sort (A[0…s], int p, int r){

        If p is less than r then continue
            Call partition with (A,p,r+1) and save the returned value in q
            Call quick_sort with the lower half of array (A,p,q-1)
            Call quick_sort with the upper half of array (A,q+1,r)
    }
```

*To rank URLs, an operator overloading must happen to be able to compare objects.

The same analysis holds of the quick sort holds. But this time we are sorting URLs. The average number of comparisons of algorithm is $T(n) = \frac{1}{n-1}\sum_{i=1}^{n-1}[T(i) + T(n-i) + n]$ which when we solve, we get $T(n) = \frac{2n \log n}{\log e} + 2$
So the complexity of quick sort is $\mathbf{O}(\boldsymbol{n \log n})$

The algorithm does not require any additional space.

Main Data Structure:

The main data structures used are vectors. Since they use very efficient memory allocation methods, dynamically sized, and has many built-in functions that utilizes it greatly.

A directional graph is used to represent the edges between URLs. It is implemented using a 2d vector (matrix) with 1 representing an edge and 0 representing not an edge.

The main class (webpage) has a vector of URLs, vector of URLs' names, and 2d binary vector representing a graph.

Design Tradeoffs:

The graph and vector of URLs are hard to traverse. So, a third vector is introduced which is a vector of URLs' names. It is used to find the location of a certain name inside of it then with that location, we access it from the URL vector and graph vector.

So, the tradeoff is extra space along with the dependability of the URL vector and graph vector on the URLs' names vector. Therefore, any change in the URLs' names vector must be accounted for in the URL vector and graph vector and vice versa. So, the three vectors depends greatly on each other and if one changes, then all must change.