SIMPLE SEARCH ENGINE

PROJECT REPORT

Abdelaaty Rehab

Student ID: 900204245

School of Sciences and Engineering

Department of Computer Science and Computer Engineering

The American University in Cairo

November 2021

Contents

Abstract

This project is based upon the idea of a simple search engine required in the Analysis and Design of Algorithms Lab. The project is mainly based on C++. For further improvements, I decided to make a Graphical User Interface to enhance the user experience in dealing with the program instead of dealing with a black-screened console application. Consequently, the project may include some codes based on QML.

Keywords search · engine · report

1 Introduction

1.1 Overview

The project is based on the idea of a simple search engine. The project reads the input from files within the same directory of the running .exe file. Then, links this input, which represents the websites, in a graph adding edges depending on the input files. Next, it calculates the rank of each web page based on how many edges going in and the rank of those web pages that are pointing to the current web page. After this, it normalizes those ranks to obtain a kind of ration that represents the importance of each website. The main window open with a simple line edit that validates the user input to meet the search query requirements. Upon hitting search, the program searches for the inserted keyword(s) within the input data and returns a vector that satisfies the conditions in the input search query. The next step is sorting those search results in clickable labels and displaying them on a new maximized window arranged according to their ranks from greatest to smallest. Finally, upon clicking any of the web pages in the search results, the program inserts the name of that specific website in a text file whose contents (websites) are to be modified (increment clicks / impressions) upon the destruction of the main window. The program saves the new data by overwriting the current contents of the files to ensure that the next running instance of the program will not start all over again.

1.2 Requirements

To run the program correctly, it is necessary to have the following files:

websites.txt: A text file containing the names of the websites only separated by new lines.

webgraph.csv: A csv file containing the **directed** links between the websites separated by the **$** sign. Each two links are separated from each other by a new line. This link must follow the format:
*website one***$***website two*.

description.csv: A csv file containing the **description** of each website in a single line. This description must follow the format:
*website name***$***website description***$***website link* **$***website impressions***$***website clicks*.

keywords.csv: A csv file containing the **keywords** of each website in a single line. This description must follow the format:
*website name***$***keyword one* **$***keyword two* **$***keyword three* etc.

i.e. Please note that all the file names are in **lower cased** and should be added in the **same directory** of the running project.

1.3 Project Psuedocode

The project could be divided into three main phases which are:

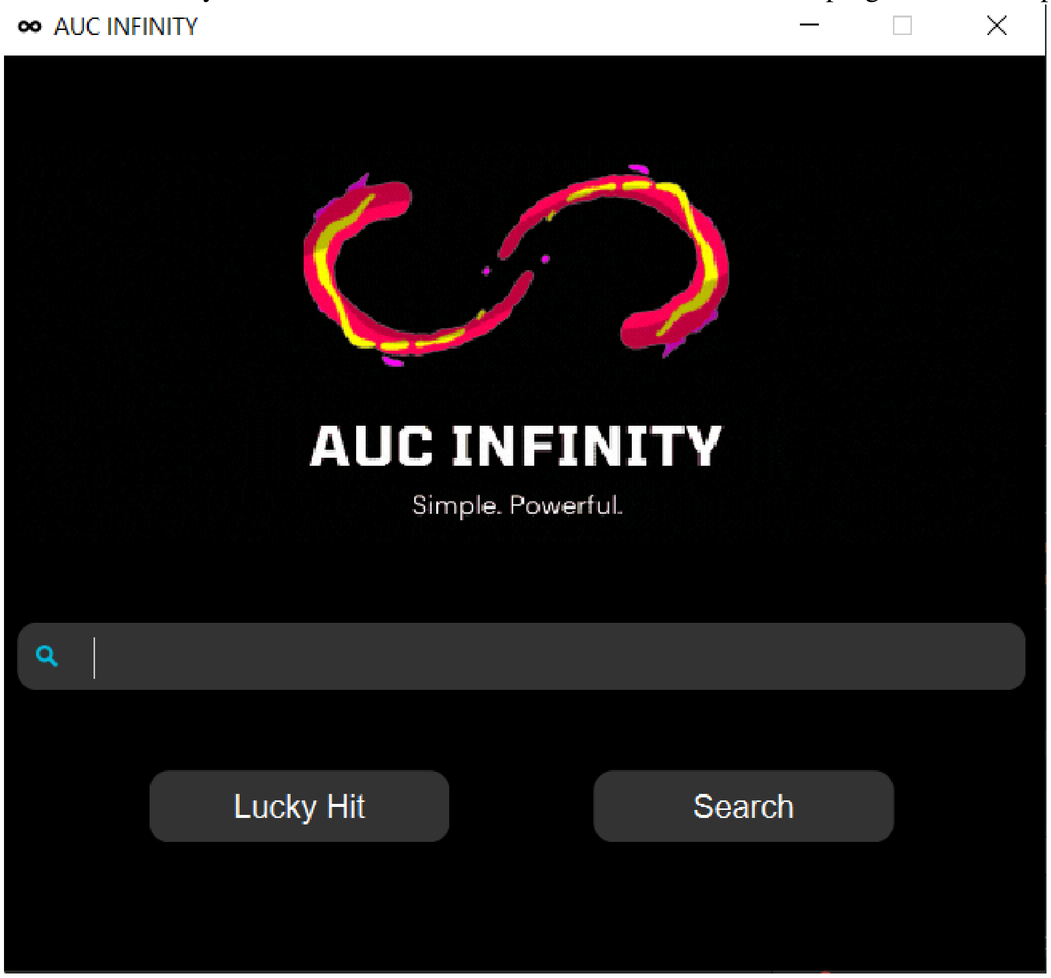
Reading phase: This phase is represented in the reading functions found in the trie and the graph. The program constructs the trie by reading and storing the websites under their relative keywords. The graph also does a similar work calculating the page rank but since it is only operating one time as long as the web graph is constant, it will be commented until we need to add more websites to our sites.

Operating phase: This phase begins once the main window appears. It is when the program finishes loading and constructing the necessary data structures to allow searching. The user inputs the search query in the line edit and the program responds as long as the process is not terminated.

Saving phase: This phase begins once the user clicks the red close button at the top of the window. The program calls the destructor which contains functions that print the information updated in the website according to the interaction of the user with the search results. The desctructor saves the information overwriting the same file so that it can be loaded again.

1.4 Features

Graphical User Interface: Instead of using dark and dull console application, I decided to make use of my basic knowledge in Qt to design and implement a user friendly GUI Below is a screenshot from the program's homepage.



Lucky hit search button: Inspired by Google, I implemented a simple lucky hit that takes a random keyword from the user's search history and does an automatic search for this keyword. Results are then displayed in a new maximized window.

Clickable Label: In Qt, a label is not a clickable object. To enable the user to click the search results, a new class is defined that has the property to be clicked. The class object emits signal *clicked()* once it encounters a *mouse event*.

Nested Search Query: To maximize efficiency as possible, the program was implemented so that it allows multiple nested search queries using the binding keywords *and, or and the quotation marks" "*. This means that the program displays the search results based on the whole input.

Input Validation: The program was implemented using trie data structure. This trie contains pointers to only 26 letters of the alphabet characters. To save extra space, the program was implemented so that it allows only the storage of those kind of websites. Moreover, the user themselves cannot enter any character other than the allowed ones in the input line edit widget. This was done using Regular expressions that allow certain symbols only to exist in the input text refusing all other symbols to be entered. This way we make sure the program will never go on a segmentation fault error.

```
QRegularExpression expr("[a-z\" ]+");
```

```
QRegularExpressionValidator *v = new QRegularExpressionValidator (expr,
    0);
ui->lineEdit->setValidator(v);
```

2 Classes

The project contains the following classes:

2.1 Node

A structure that represents the basic building unit of Trie data structure. The class is designed such that it can hold both contents(websites) and children(next nodes) where:

2.1.1 Attributes

vector<string> content A vector of strings that carries the websites that are stored in the node.

int word_end An integer that represents the number of words that are ending with the current node(letter).

Node* children [ ] An array containing 26 (no. of English letters) pointers to nodes that represent the children of the current node (next letters in the keyword).

2.1.2 Methods

int isWordEnd() Returns the number of words ending with a specific node (letter).

Node* getChild(T) Returns a child based on the parameter passed. If it is a character, it returns the pointer at position *T-'a'* in the array. On the other hand, if it is an integer, it returns the position at array position *T* directly.

2.2 Trie

This class serves as a data structure that stores the websites and their keywords related to them to facilitate the process of searching within the given input. The class contains attributes and methods that are efficiently customised to achieve that purpose where:

2.2.1 Attributes

Node* root A pointer to a node representing the head pointer that points to the first distinct letters to be inserted in the trie.

int size An integer to keep track of the size of the trie.

2.2.2 Methods

int isWordEnd() Returns the number of words ending with a specific node (letter).

vector<string> search (string) A method that searches a certain keyword in the trie and returns the vector content of the last node if there are words that end with that character. Otherwise, it returns an empty vector.

void insert (string,string) A method that inserts a certain value of string in a passed keyword. It is used in constructing the trie at the beginning of the program.

2.3 Website

This class serves as an object oriented version of each website. Each website is represented in the program by an object instantiated by the values read from the input files:

2.3.1 Attributes

string name A string that stores the name of the website.

string description A string that stores the description of the website.

string link A string that stores the link of the website.

int impressions An integer that stores the number of times this website appeared in search results.

int clicks An integer that stores the number of times the website was clicked when it appeared in a search result.

double CTR A double value that represents the *click-through ratio* of the website. Where:

$$CTR = clicks/impressions \tag{1}$$

double score Stores the score of each website based on its normalized rank calculated by the graph.

2.3.2 Methods

The methods do not differ from any class declared in any OOP program. They are mainly setters and getters for the attributes in the class to ensure encapsulation of the data inside them. They also include functions that increment clicks or impressions based on a certain action

3 Data Structures

3.1 Vectors

In all of the coming scenarios, vector is chosen as a data structure to store them because vectors offer simple slots of storing objects inside and in addition, they offer the possibility of pushing back objects instantly without needing to access the contents of the last index. They are also dynamically sizeable which makes vectors perfect for storing.

vector<string> contents A vector that contains the websites that have a certain keyword in common.

vector<string> results A vector that contains the results of a certain search using a particular search query so that it can be displayed in the search results page.

vector<string> adj A vector that contains the edges coming out/into a certain website (used to represent the directed web graph).

vector<double> P0 A vector that contains the rank of each website calculated by the ranking algorithm.

3.2 Ordered maps

In all of the upcoming scenarios, ordered maps are chosen to store because they offer two advantages: first, they offer the capability of storing pairs which makes them perfect for hashing websites to indices. Second, they are automatically sorted according to the key value. The map sorts the elements based on the key value. This makes the hash maps perfect for displaying the search results based on their calculated scores.

map<string,int> hashIndex A hash table that contains the website followed by its index number.

map<string,website> websites A hash table that contains the website index followed by its corresponding object.

map<int,Pair<int,int» impressions A hash table that contains the website index followed by its impressions and clicks grouped in a pair. It is used once in the graph to calculate the rank of each page.

map<string,int> hashIndex A hash table that contains the website followed by its index number.

3.3 Arrays

In all of these scenarios, arrays are used because no further operations are need to be done with the stored data. The only operation to be done is matrix multiplication which does not need special requirements. A simple array will do the job for this.

vector<double> adj[ ] An array of vector used to hold the adjacency matrix of the whole graph representation.

double power_mat [ ] A 2-D array used to store the power matrix of the graph. This power matrix is essential to calculate the rank of each web page.

4 Indexing Psuedocode

Indexing websites in the program is done using hash tables in a linear manner. Where the program reads the name of the websites from the file, hashes the website to the key corresponding to the number of line this website represents in its file. **e.g.** A website in line 3 will be indexed to value 2 in the hash table.
The double hashing method enables the program to link the website name with its key and relate this integer key with the object representing the website. This way enables us to access the object website easily using different means depending on the way the accessing is done.

$\boxed{\text{Website name}} \rightarrow \boxed{\text{Website index}} \rightarrow \boxed{\text{Website object}}$.

This is how strings are hashed to integers:

```
int i=0;
QTextStream in(&myfile);

while (!in.atEnd())
{
    QString line = in.readLine();

    hashIndex[line]=i;
    i++;
}
myfile.close();
```

This is how key integers are hashed to website objects:

```
QTextStream in1(&myfile2);

while (!in1.atEnd())
{
    QString line1 = in1.readLine();

    QStringList list1 = line1.split(QLatin1Char('$'));
    Website* wb;
    wb = new Website(list1[3],list1[4],list1[0],list1[1],list1[2]);

    websites[hashIndex[list1[0]]]=*wb;
    i++;
}
myfile.close();
```

5 Complexity

5.1 Time Complexity

The highest time complexity in the program is O(kn) where k represents the number of iterations the page ranking algorithm makes to make the ranks of the pages stabilize to a certain range of values.

5.1.1 Input processing complexity

Maps hashing

The program hashes the input values to maps. If we have **n** websites and the insertion in a map takes **O(1)** time
The total hashing time of one complete map is **O(n)**.
i.e. This process is repeated for every input file the program opens. The input time will be a bit larger but of the same complexity **O(n)**.

Trie Constructing

If we considered we have *W* keywords to be inserted in the trie and the average length of those keywords is *L*, the average time for inserting all of those keywords will be **O(W*L)** because the program will have to make *L* searches for each keyword to find them then insert the content in the last node of the keyword in the trie.

5.1.2 Ranking complexity

The program uses the page ranking algorithm using the power matrix method to calculate the rankings of each web page. The time complexity of ranking depends on repeating an algorithm that consumes **O(x*n + y*m)** a certain number of times till we reach that *error* < *tolerance value*. This condition is met after at most 100 iterations. This value is chosen arbitrarily according to some sources to ensure a reasonable amount of precision.

5.1.3 Searching complexity

As mentioned above, the program uses *Trie* data structure to store websites and keywords in it. This allows the search to be of nearly a constant time. This is because the search iterations will only depend on the length of the keyword which is of constant size even if it gets bigger from one runtime to another. Below is the code of the search function in the trie:

```
vector<string> Trie::search(string key)
{
    Node* curr = root;

    for (int i = 0; i < key.length(); i++)
    {
        int index = key[i].toLatin1() - 97;
        if (!curr->children[index])
            return vector<string>();

        curr = curr->children[index];
    }

    return (curr->content);
}
```

As clearly appearing in the code, the search function iterates from the first to the last character of the keyword. It does not depend on the contents of the trie at all or even at on the number of websites. If we considered we have *W* keywords to be searched and the average length of those keywords is *L*, the average time for searching for all of those keywords will be **O(W*L)** because the program will have to make *L* searches for each keyword.

5.2 Space Complexity

Most of the program's space complexity lies in the Graph class where it uses matrices, arrays, maps to hash the websites and calculate their ranks and store them in those maps. But, this class is to be implemented once so it will not run except for the first run time only. That's why the space complexity will depend on the space complexity of the trie and the pages classes only.

Trie Space Complexity

As for the trie, in the worst case all the keywords are distinct without any common letters between them. If we have *W* keywords and their average length is *L*. For each ending node, there is a vector containing websites linked to that specific keyword. If the size of this vector is *R*, then the space complexity in the trie **O(R*L)** where R represents the number of websites that are linked with a certain keyword.

Pages Space Complexity

As for the web pages stored in the class, the class *Pages* stores the websites in two hash tables. If we have *n* websites, then the total space used in memory will be of **O(2*n)** which is equivalent to **O(n)**.

6 Page Rank Algorithm

6.1 Idea

A recursive approach is used when implementing the page rank algorithm where:

Edges Representations: Edges are represented using power matrix of the graph. The power matrix is an indication of the initial number of links from and to a website. Before the ranking algorithm is called, **constructPowerMatrix()** function is called. The function iterates over the adjacency matrix of the graph and updates each node power to be 1/n where n is the size of the adjacency list of a certain node. The function also validates the power matrix as it goes on where it does not allow the sum of any column to be zero. If in case this happens, each item in this column will be initialized to be 1/total number of websites in the graph. At the end, the function updates the array of ranks of each website to start with equal initial values 1/total number of websites.

Matrix Multiplication Each recursive iteration of the function, the power matrix is multiplied with the ranks matrix and the ranks are updated with the latest results. The program introduces two multiplication algorithms which are:

Enhanced normal multiplication: The normal matrix multiplication is built upon three nested for loops each of them iterates one of the two matrices from one dimension. This will cost us $O(n^3)$ time. But, if we know that the ranks vector is always one dimensional, we can actually dispose the third nested loop because it would iterate over the same positions over and over again wasting more time. The enhanced algorithm multiplies element i from the page ranks vectors with the sum of elements of row i from the power matrix array. The result is stored in the same place to be updates. Therefore, The algorithm is of complexity $O(n^2)$.

Sparse matrix multiplication: The sparse matrix multiplication is implemented in another function because it is a little faster than the ordinary one. It depends on storing the matrix in form of triplets $(row, column, value)$ in a 2-D array (array representation of sparse matrices). This helps us avoid the zero elements in the matrix to avoid wasting time in their calculations. The functions transposes the second matrix and traverses the result with the first one till it finds an element that has a common column, then it multiplies it and stores it in the result. The result

is traversed once more to merge any elements with the same rows and columns indices.
As obvious, the time complexity of this algorithm is $O(x * n + y * m)$, where (x, m) is number of columns and terms in the second matrix; and (y, n) is number of rows and terms in the first matrix. In the worst case scenario, it is $O(n^2)$but this worst case will not always happen.

Stopping criteria: The program continues to iterate over the matrices again and again till the values of the page ranks stabilize. This is achieved when the average difference in the page rank from one iteration to another is less than 0.01. Two significant figures were chosen to gather between precision and also speed where it is unnecessary to loop for more precision because the values of the page ranks will not significantly change the think which cost a lot of wasted time.

Below is the code used for ranking:

```
void Graph::Rank()
{
    if (error < tol || steps>=100)
        return ;

    steps++;
    P0 = multiply();
    Rank();
}
```

6.2 Time Complexity

The ranking algorithm depends on multiplying the power matrix of the web page ranks recursively till we reach an error value that is less than 0.01 or we reach 100 recursive iterations. This ensures that the program at worst case will make 100 repetitions of the matrix multiplications. The ordinary matrix multiplication was implemented in an algorithm that takes $O(n^2)$. This ensures that the program will at most make $100 * O(n^2) = O(n^2)$. Another thing we should take into consideration is that the program uses the *Sparse matrix multiplication* which takes $O(x * n + y * m)$ complexity. This means that the program repeats that number of iterations for at most 100 times. Consequently, the complexity will be less than $O(n^2)$. Eventually, in either cases we can conclude that the complexity is around $O(n^2)$. A bit greater in case of normal matrix multiplication and a bit smaller if we chose to use sparse matrix multiplication.

6.3 Space Complexity

As clear, the algorithm depends on the power matrix of the web pages which is a 2-D arrays of size nxn. Thus, the space complexity is of $O(n^2)$.

7 Design Trade-offs

7.1 Choosing Trie

After searching for data structures and thinking of many of them that could serve as a container to the keywords and the related websites, the trie was the perfect option to choose for some reasons:

Space complexity: The space complexity of the trie is not bad at all. It does not only store the websites vector once at the end of the word, but also it saves us the space of storing other keywords containing similar letters. It is simply achieved through the *word_end* integer that defines how many words end at the current character.

Time complexity: The time complexity of constructing a trie is not very big. It costs us **O(W*L)** if we had *W* keywords of average length *L*.It will not grow very big with the input since the websites will indeed have common keywords between them.

Searching: Searching in a trie is efficient because it depends only on the length of the keyword given. It does not depend on the amount of input or their length. It mainly depends on the length of the input keyword. In the worst case scenario, the search will go for finding all the keywords which will cost us **O(W*L)** too.

7.2 Sparse Multiplication of matrix

The sparse matrix multiplication is chosen to provide a faster method for multiplying matrices without having to iterate through all of their elements. It depends on storing the matrix in form of triplets **(row,column,value)** in a 2-D array (array representation of sparse matrices). This helps us avoid the zero elements in the matrix to avoid wasting time in their calculations. The functions transposes the second matrix and traverses the result with the first one till it finds an element that has a common column, then it multiplies it and stores it in the result. The result is traversed once more to merge any elements with the same rows and columns indices. Below is the code of sparse matrix multiplication:

```
void multiply(sparse_matrix &b)
{
    if (col != b.row)
    {

        // Invalid multiplication
        qDebug() << "Can't multiply, Invalid dimensions";
        return;
    }

    // transpose b to compare row
    // and col values and to add them at the end
    b = b.transpose();
    int apos, bpos;

    // result matrix of dimension row X b.col
    // however b has been transposed,
    // hence row X b.row
    sparse_matrix result(row, b.row);

    // iterate over all elements of A
    for (apos = 0; apos < len;)
    {

        // current row of result matrix
        int r = data[apos][0];

        // iterate over all elements of B
        for (bpos = 0; bpos < b.len;)
        {

            // current column of result matrix
            // data[,0] used as b is transposed
            int c = b.data[bpos][0];

            // temporary pointers created to add all
            // multiplied values to obtain current
            // element of result matrix
            int tempa = apos;
            int tempb = bpos;

            double sum = 0;
```

```
                // iterate over all elements with
                // same row and col value
                // to calculate result[r]
                while (tempa < len && data[tempa][0] == r &&
                    tempb < b.len && b.data[tempb][0] == c)
                {
                    if (data[tempa][1] < b.data[tempb][1])

                        // skip a
                        tempa++;

                    else if (data[tempa][1] > b.data[tempb][1])

                        // skip b
                        tempb++;
                    else

                        // same col, so multiply and increment
                        sum += data[tempa++][2] *
                            b.data[tempb++][2];
                }

                // insert sum obtained in result[r]
                // if its not equal to 0
                if (sum != 0)
                    result.insert(r, c, sum);

                while (bpos < b.len &&
                    b.data[bpos][0] == c)

                    // jump to next column
                    bpos++;
            }
            while (apos < len && data[apos][0] == r)

                // jump to next row
                apos++;
        }
        b =result;
        //result.print();
    }
```

7.3 Hashing method

It has been decided to store the websites under the keywords and not the other way round because this will be faster in searching. The user will input a certain keyword, the program will be required to find the websites related to it. If we stored the keywords of each website under it, we would have to iterate over each and every website to see if it has the required keyword or not. This will cost us linear time which will increase in case the user inputs multiple keywords in the search query.

7.4 Using Ordered maps instead of unordered ones

The advantage that is given by the ordered map is that it performs the heap sort of the data automatically based on the key. It sorts the keys lexicographic-ally or numerically depending on its data type. This advantage is useful in displaying the search results where the maps insert each element in **O(logn)** time.