## **The pseudo-code for your indexing and ranking algorithms**

Page ranking algorithm

Rank list pagerank(graph as adjacency list) {

    Damping factor = 0.85;

    number of iterations = 100;

    inialize a list rank;

rank.size() = graph.number\_of\_nodes;

rank.initial\_values() = all rank are 1/list.size();

    initialize a list new\_rank;

new\_rank.size() = graph.number\_of\_nodes();

new\_rank.initial\_values() = all new ranks = 0;

    for (i = 0 🡺 number of iterations-1) {

        for (j = 0 🡺 graph.size() -1) {

            for (k = 0 🡺 size of the list adjacent to node j -1) {

                new\_rank[j] += rank[graph[j][k]] / graph[graph[j][k]].size();

            }

new\_rank[j] = (1 - damping\_factor) / graph.size() + damping\_factor \* new\_rank[j];

        }

        rank = new\_rank;

        new\_rank = vector<double>(graph.size(), 0);

    }

    return rank list;

}

Indexing algorithm

Psuedo code for the rest of the functions

website\_keywords\_generator(map<string, vector<string>> & website\_keywords) {

    input file websites\_keywords\_file from ("key\_word.csv");

    while (there is still line in the file)) {

header = the first word in the line

        while (there are words in line) website\_keywords[word].push\_back(header);

    }

    Close file

};

void Graph\_initializer(string\_graph, num\_graph, translator) {

    input file Graph\_file("web\_graph.csv");

  int c = 0;

    while (there is line in the file) {

header = the first word in the line

        if (header doesn’t have numerical translation) {

header’s translation = c;

c++

        }

        num\_graph.resize(c);

        while (there are words in current line) {

            if (word doesn’t have translation){//graph num string translator

                word translation = c;

                c++;

            }

            string\_graph[header].add(word);//stirng graph

            num\_graph[translator[header]].push\_back(translator[word]);//num graph

        }

    }

    Graph\_file.close();

}

void initializer(map<string, int>& mp, string name) {

    input file open (name)

    while (there is line) {

        header = first word;

cell = second word

mp[header] = cell;

    }

}

List of strings search(website\_keywords, choice) {

    string srch\_q = search\_query();

    list results;

    keywords = splitter(srch\_q);

    for (row in keywords) {

        if (row.size() > 1) {

           map freq;

            for (keyword in row) {

                for (website in website\_keywords[keyword]) {

                    freq[website]++;

                }

            }

            for (auto j : freq) {

                if (freq[j.first] == the size of the row) {

                    add j.first to the results list.

                }

            }

        }

        else {

            for (keyword in row) {

                for (website : website\_keywords[keyword]) {

                    add website to the results list

                }

            }

        }

    }

    return results;

}

vector<vector<string>> splitter(string query\_str) { //"machine AND learning OR "complexity analysis""

    ORize(query\_str); to put or between terms without operator.

    2d array keywords;

    List of strings temp;

    string temp\_str;

    bool is\_quote = false;

    for (i = 0 🡺 size of search query) {

        if (query\_str[i] == '"') {

            is\_quote = !is\_quote;

            continue;

        }

        if (is\_quote) {

            temp\_str += query\_str[i];

            continue;

        }

        if (query\_str[i] == ' ') {

            if (temp\_str.size() > 0) {

                temp.push\_back(temp\_str);

                temp\_str.clear();

            }

            continue;

        }

        if (query\_str[i] == 'A' && query\_str[i + 1] == 'N' && query\_str[i + 2] == 'D') {

            if (temp\_str.size() > 0) {

                temp.push\_back(temp\_str);

                temp\_str.clear();

            }

            i += 2;

            continue;

        }

        if ((query\_str[i] == 'O' && query\_str[i + 1] == 'R')) {

            if (temp\_str.size() > 0) {

                temp.push\_back(temp\_str);

                temp\_str.clear();

            }

            keywords.push\_back(temp);

            temp.clear();

            i++;

            continue;

        }

        temp\_str += query\_str[i];

    }

    if (temp\_str.size() > 0) {

        temp.push\_back(temp\_str);

        temp\_str.clear();

    }

    keywords.push\_back(temp);

    return keywords;

}

vector<string> search(map<string, vector<string>> website\_keywords, int& choice) {

    string srch\_q = search\_query();

    vector<string> results;

    vector<vector<string>> keywords = splitter(srch\_q);

    for (auto& row : keywords) {

        if (row.size() > 1) {

            map<string, int> freq;

            for (auto& keyword : row) {

                for (auto& website : website\_keywords[keyword]) {

                    freq[website]++;

                }

            }

            for (auto j : freq) {

                if (freq[j.first] == row.size()) {

                    results.push\_back(j.first);

                }

            }

        }

        else {

            for (auto& keyword : row) {

                for (auto& website : website\_keywords[keyword]) {

                    results.push\_back(website);

                }

            }

        }

    }

    return results;

}

## **A time complexity analysis for your indexing and ranking algorithms and the other main function**

### **Page rank O(nm)**

the first loop with counter i has 100 iterations.

//the second loop has graph.size() iterations. the third loop has graph[j].size() iterations. then multiply all of these together to get the total number of iterations of the nested for loops.

// let n = graph.size() and m = graph[j].size() then the total number of iterations of the nested for loops is 100 \* n \* m.

//**the time complexity** of the above function is O(100 \* n \* m) which is O(n^2) because the 100 is a constant and m is atmost = n because a node can have edges to all other nodes in the graph.

//**the space complexity** of the above function is O(n) because the vector rank is of size n and the vector new\_rank is of size n.

### **Website keywords generator function O(nm). n: number of lines in file. m; number of cells in line**

At the beginning, the program uses the “keywords.csv” file to generate a list of the keywords and their associated websites in

map<string, vector<string>> website\_keywords

void website\_keywords\_generator(map<string, vector<string>> & website\_keywords) {

    ifstream websites\_keywords\_file;

    websites\_keywords\_file.open("key\_word.csv");

    string line;

    while (getline(websites\_keywords\_file, line)) {

        stringstream row(line);

        string cell;

        getline(row, cell, ',');

        string header = cell;

        while (getline(row, cell, ',')) website\_keywords[cell].push\_back(header);

    }

    websites\_keywords\_file.close();

};

### **Then graph initializer function initializes the graph in O(nm). n: number of lines in file. m; number of cells in line**

void Graph\_initializer(unordered\_map<string, vector<string>>& string\_graph\_websites, vector<vector<int>>& num\_graph\_websites, map<string, int>& string\_num\_websites\_translator) {

    ifstream Graph\_file("web\_graph.csv");

    string line = "";

    int c = 0;

    while (getline(Graph\_file, line)) {

        stringstream row(line);

        string cell;

        getline(row, cell, ',');

        string header = cell;

        if (string\_num\_websites\_translator.find(header) == string\_num\_websites\_translator.end()) {//graph num string translator

            string\_num\_websites\_translator[header] = c;

            c++;

        }

        num\_graph\_websites.resize(c);

        while (getline(row, cell, ',')) {

            if (string\_num\_websites\_translator.find(cell) == string\_num\_websites\_translator.end()) {//graph num string translator

                string\_num\_websites\_translator[cell] = c;

                c++;

            }

            string\_graph\_websites[header].push\_back(cell);//stirng graph

            num\_graph\_websites[string\_num\_websites\_translator[header]].push\_back(string\_num\_websites\_translator[cell]);//num graph

        }

    }

    Graph\_file.close();

}

The function initializes the following variables

unordered\_map<string, vector<string>> string\_graph\_websites; (string nodes)

vector<vector<int>> num\_graph\_websites; (numerical nodes)

map<string, int> string\_num\_websites\_translator; (to translate from string nodes to numerical and vice vers)

### **Impressions and clicks initializer function O(n). n: number of lines in file**

void initializer(map<string, int>& mp, string name) {

    ifstream file;

    file.open(name);

    string line = "";

    while (getline(file, line)) {

        stringstream row(line);

        string header;

        string num;

        getline(row, header, ',');

        getline(row, num, ',');

        mp[header] = stoi(num);

    }

    file.close();

}

### **ORize function O(n). n: length of search string.**

takes a search string and put OR between the words that do not have any operator, for example, “data structures” 🡺 “data OR structures”. N is the length of the search string

void ORize(string& s) {

    vector<string> vec;

    string word = "";

    for (int i = 0; i < s.length(); i++) {

        if (s[i] == '"') {//quotation handling

            word += s[i];

            i++;

            while (s[i] != '"') {

                word += s[i];

                i++;

            }

            word += s[i];

            if (i == s.length() - 1) vec.push\_back(word);

        }

        else if (s[i] != ' ') {//words

            word += s[i];

            if (i == s.length() - 1) vec.push\_back(word);

        }

        else {//if space is encountered

            vec.push\_back(word);

            word = "";

            if (s[i + 1] == 'O' && s[i + 2] == 'R') { i += 3; vec.push\_back(" OR "); }

            else if (s[i + 1] == 'A' && s[i + 2] == 'N') { i += 4; vec.push\_back(" AND "); }

            else { vec.push\_back(" "); }

        }

    }

    s = "";

    for (auto& x : vec) {

        if (x == " ") x = " OR ";

        s += x;

    }

}

### **Splitter function O(n). n: length of search string, however because ORize is called inside Splitter the total complexity is O(n^2)**

(takes the search query and extract the keywords in 2d array to be used by search function. If one row of the array contains more than keyword it means the operator AND was between them. If keywords are in different rows, then operator OR was between them.

vector<string> search(map<string, vector<string>> website\_keywords, int& choice) {

    string srch\_q = search\_query();

    vector<string> results;

    vector<vector<string>> keywords = splitter(srch\_q);

    for (auto& row : keywords) {

        if (row.size() > 1) {

            map<string, int> freq;

            for (auto& keyword : row) {

                for (auto& website : website\_keywords[keyword]) {

                    freq[website]++;

                }

            }

            for (auto j : freq) {

                if (freq[j.first] == row.size()) {

                    results.push\_back(j.first);

                }

            }

        }

        else {

            for (auto& keyword : row) {

                for (auto& website : website\_keywords[keyword]) {

                    results.push\_back(website);

                }

            }

        }

    }

    return results;

}

vector<vector<string>> splitter(string query\_str) { //"machine AND learning OR "complexity analysis""

    ORize(query\_str);

    vector<vector<string>> keywords;

    vector<string> temp;

    string temp\_str;

    bool is\_quote = false;

    for (int i = 0; i < query\_str.size(); i++) {

        //cout << query\_str[i] << '\n';

        if (query\_str[i] == '"') {

            is\_quote = !is\_quote;

            continue;

        }

        if (is\_quote) {

            temp\_str += query\_str[i];

            continue;

        }

        if (query\_str[i] == ' ') {

            if (temp\_str.size() > 0) {

                temp.push\_back(temp\_str);

                temp\_str.clear();

            }

            continue;

        }

        if (query\_str[i] == 'A' && query\_str[i + 1] == 'N' && query\_str[i + 2] == 'D') {

            if (temp\_str.size() > 0) {

                temp.push\_back(temp\_str);

                temp\_str.clear();

            }

            i += 2;

            continue;

        }

        if ((query\_str[i] == 'O' && query\_str[i + 1] == 'R')) {

            if (temp\_str.size() > 0) {

                temp.push\_back(temp\_str);

                temp\_str.clear();

            }

            keywords.push\_back(temp);

            temp.clear();

            i++;

            continue;

        }

        temp\_str += query\_str[i];

    }

    if (temp\_str.size() > 0) {

        temp.push\_back(temp\_str);

        temp\_str.clear();

    }

    keywords.push\_back(temp);

    return keywords;

}

### **Search function O(u v m) u: number of rows in the keywords data structure we got from splitter. v: number of keywords in each row. m: number of websites associated with this keyword**

vector<string> search(map<string, vector<string>> website\_keywords, int& choice) {

    string srch\_q = search\_query();

    vector<string> results;

    vector<vector<string>> keywords = splitter(srch\_q);

    for (auto& row : keywords) {

        if (row.size() > 1) {

            map<string, int> freq;

            for (auto& keyword : row) {

                for (auto& website : website\_keywords[keyword]) {

                    freq[website]++;

                }

            }

            for (auto j : freq) {

                if (freq[j.first] == row.size()) {

                    results.push\_back(j.first);

                }

            }

        }

        else {

            for (auto& keyword : row) {

                for (auto& website : website\_keywords[keyword]) {

                    results.push\_back(website);

                }

            }

        }

    }

    return results;

}

## **The main data structures used by your algorithm and space complexity**

map<T1, vector<T2>> O(n m) n:num. of keys in the map, m num.of elements in vector

map<T1, T2> O(n)

vector<T> O(n)

vector<vector<T>> O(n^2)

vector<pair<T1, T2>> O(n)

so the wors space complexity I quadratic.

## **Any design tradeoffs you made along with their justifications**

1. Adjacency list representation of the graph was used for better storage efficiency

## **Extra Features**

Multiple “AND”s, “OR”s and quotations can be used in the sear query, for example, [“data structures” AND programming OR java OR python] is a valid search query which prioritizes the operator “AND”. So the final results for the above website will be the websites that contain both “data structures” and programming, and the websites that contain solely “java” and the websites that contain solely “python”.