(Fall 2021)

CSCE2203 PROJECT:

Creating a Search Engine Algorithm Using PageRank

Sara Mohamed

900203032

**File Input:**

In file\_input.h and file\_input.cpp, the algorithms for inputting all information from csv files can be found. For inputting the files, the algorithm takes all files, and iterates along them until it cannot find any more values, and in each iteration, it splits the line up into variables that represent each column. Important to note is that the way webpages are identified between files is through the creation of an unordered\_map<string, int>, static to the webpage class. While inputting values from the first keywords file, the program starts making webpage objects, and inputting them into a vector, and saving their name as well as their index in the vector in the unordered\_map. In this way, it is able to identify the same webpage in the vector from one file to another, and use the indices to access the exact webpage in the vector and edit its values.

For the file with keywords and names:

vector <webpage> pages;

int i = 0;

while (getline(file, name, ‘,’))

{

string kws;

getline(file, kws, ‘\n’)

Vector <string> keywords;

split up the string kws into separate keywords based on commas

and then insert them one by one into the vector “keywords”

//create webpage with name and keywords, and push it into the existing vector //of keywords “pages”

webpage pg(name, keywords);

pages.push\_back(pg);

//in “webpage::unordered\_map keys”, save the name of the webpage along with its //index in the vector of webpages;

webpage::keys.insert({name, i});

i++;

}

For the impression (and clicks) file:

while (getline(file, name, ‘,’))

{

string imp, clicks, impressions;

getline(file, imp, ‘\n’)

//check if imp contains a comma, if it does, it contains both impressions and

//clicks. If it does not, it contains only impressions.

if (imp.find(“,”))

split into two and save first in impressions and second in clicks;

else

impressions = imp;

//access the unordered\_map<string, int> using “name”, to find which index of //the vector the webpage is saved in, and save that index in int key.

int key = webpage::keys[name];

pages[key].setimpressions(impressions);

pages[key].setclicks(clicks); // only if clicks exist in file for this webpage

}

For the hyperlinks file:

there exists a static array “pointingto” of the webpage class, which saves the number of pages each page points to (or has hyperlinks to), and is initialized to zero.

create matrix int\*\* links; // this is our adjacency matrix

while (getline(file, name, ‘,’))

{

string name2;

getline(file, name2, ‘\n’)

//access the unordered\_map<string, int> using “name” and “name2” to find the //indices of both webpages based on their names;

int key1 = webpage::keys[name]; //index of webpage with “name” in vector

int key2 = webpage::keys[name2]; //index of webpage with “name2” in vector

links[key1][key2] = 1;

webpage::pointingto[key1]++;

}

For the file inputting as a whole, the time complexity is necessarily O(n). We must iterate through all the n columns of the keywords and impressions files, to get all the information on the webpages. However, through the creation of the unordered\_map, we decreased comparisons that would otherwise be required to determine which name belonged to which webpage in our vector. If we had done that instead, our complexity would have reached O(n²) or even O(n³) even for the hyperlinks file, because we would have gotten a name from a file, and then had to iterate through the vector to determine which object corresponds to the name. Thus, the creation of the unordered\_map increased the space complexity by 2n spaces, n for the strings and n for the ints, but greatly decreased the time complexity. Also, the creation of the poinitngto array increases space complexity by n, but helps with later time complexity in the pagerank function, so no iteration would be required to find how many pages one webpage points to. The adjacency matrix also raises the space complexity by n², but is useful in the upcoming pagerank function to lower its time complexity. The file input function, when done, calls on a static function, pagerank, which takes the adjacency matrix created, and the vector of webpages, to calculate the pagerank of each one.

**PageRank:**

void webpage::pagerank(int\*\* links, vector <webpage> pages)

{

double d = 0.85;

double N = (double)count;

double\* previousrank;

double\* newrank;

previousrank = new double[count];

newrank = new double[count];

//initializing damping factor d, number of elements N, and two arrays //previousrank and newrank to be of size N, to be able to use them in the //main pagerank algorithm

for (int i = 0; i < count; i++)

{

newrank[i] = (1.0 / count);

previousrank[i] = 0.0;

}

//initializing newrank of all elements to be 1/N, and previousrank of all //elements to be 0

for (int k = 0; k < 10; k++)

{

for (int i = 0; i < count; i++)

previousrank[i] = newrank[i];

for (int i = 0; i < count; i++)

newrank[i] = 0.0;

//every iteration, we set previousrank to be equal to newrank of the //previous iteration, and we zero the entire newrank array again.

double rank;

for (int i = 0; i < count; i++)

{

rank = 0;

for (int j = 0; j < count; j++)

{

if (links[j][i] == 1)

rank += (previousrank[j] / pointingto[j]);

}

//we hold onto each index i, and check through all indices j to //see if a j points to i. If it does, we divide the previous rank //of j by the number of webpages j points to, and add the result //to the rank of i.

newrank[i] = ((1 - d)/N) + (d \* rank);

// we then add the damping factor and set the new rank of i to be //equal to the previous equation.

}

}

double minpr = 0.0, maxpr = 0.0;

for (int i = 0; i < count; i++)

{

if (newrank[i] > maxpr)

maxpr = newrank[i];

if (newrank[i] < minpr)

minpr = newrank[i];

}

//we then determine the minimum and the maximum pageranks in newrank

for (int i = 0; i < count; i++)

{

double p\_rank;

p\_rank = ((newrank[i] - minpr) / (maxpr - minpr));

wp[i].pr = p\_rank;

wp[i].updateCTR();

}

//we use those minimum and maximum values to normalize all the pageranks, so //the minimum value is 0, and the maximum is one. Then we set the pagerank of //each page to be the normalized value, and then call the function updateCTR //for each webpage, so it can calculate the CTR for the first time. updateCTR() //calls updateScore() every time, so the score is updated whenever the CTR //itself is updated.

}

The time complexity of pagerank is n². It iterates over two nested for loops 100 times, and does comparisons and arithmetic within those for loops. Please note the number of loops was chosen to be 100 times after testing to see when the numbers would stop changing significantly, and 100 was found to be a safe spot for that. Outside the two large for loops, the program carries out a loop of complexity O(n) to find the maximum and minimum pageranks, and then another O(n) loop to normalize each pagerank and update each webpage. Thus, in total, the algorithm is of complexity n². The two auxiliary arrays newrank and previousrank take up space complexities of order O(n), but are necessary in order to calculate the pagerank accurately. The adjacency matrix, of size n², is also a necessary part of the algorithm, to determine which nodes have edges with which other nodes.

**CTR and Score Calculations**

void webpage::updateCTR()

{

CTR = (clicks / impressions);

updatescore();

}

void webpage::updatescore()

{

double temp1, temp2;

temp1 = (1 - ((0.1 \* impressions) / (1 + 0.1 \* impressions)));

temp1 \*= pr;

temp2 = ((0.1 \* impressions) / (1 + 0.1 \* impressions));

temp2 \*= CTR;

score = (0.4 \* pr) + ((temp1 + temp2) \* 0.6);

}

Both the CTR and score calculations are of order O(1), as they are simple arithmetic calculations that do not depend on size of data. Note that updateCTR() automatically calls on updatescore(), so the score is updated every time the CTR is updated. The updatescore function contains two auxiliary double variables, of total auxiliary space complexity O(1).

**Program Flow and Searching:**

For the program flow, the program offers choices to the user, one of them being search. When a user searches for keyword(s), the program sends the keyword(s) into one of four functions, depending on the nature of the search (if it includes “AND”, if it includes “OR”, if it includes quotations, if it includes a space, or if it's simply one word). Each one of those search functions has a complexity of O(n), as the program loops through all webpages, and in every webpage it loops through a vector of keywords. The keywords however, are usually between 5 to 10, and so, can be considered to be considerably less than O(n), making the search function be equivalent to a constant multiplied by n, which is O(n). Note that the search functions use very minimal auxiliary space equal to O(1) in the form of strings or bools to manage the function flow. The results found from the search are pushed into a vector, which is an auxiliary space, with maximum capacity n, and thus, space complexity O(n). It displays the results, and increments the impressions of each webpage by one, as well as updates the CTR and score. Updating the score and CTR of each webpage, as previously mentioned, is of order O(1). If the user chooses to click on a webpage, the clicks for that webpage are incremented, and the CTR and score of the webpage are also updated. Every time the user chooses to make a new search, the program uses std::sort to sort the vector of webpages based on score. The std::score algorithm is of complexity O(nlog(n)), and, since it is only carried out the amount of times the user makes a new search, can be considered to be in total a constant multiplied by nlog(n), which is still O(nlog(n)). At the very end of the program, when the user decides to exit, the program creates a new csv file, where it outputs the name of each webpage, as well as the impressions and clicks (if available). It then deletes the old file for impressions and clicks, and renames the new file to be the same name as the old one. This process takes O(n) time to print each webpage in the vector, and constant time to rename the files. Thus, it takes O(n) time in total.

{

ask user for choice;

if (choice = newsearch)

{

sort vector based on score;

Determine which kind of search to execute;

Execute search and store results in vector results;

display results and increment impressions of each result by one;

if (choose to click on webpage)

{

increment clicks of webpage by one;

ask again for choice;

}

}

If (choice = exit)

{

Create new csv file and store names, impressions and clicks in it;

Delete old impressions csv file;

Rename new csv file the same name as old one;

exit();

}

}

In total, the time complexity of the entire program is O(n²), which is due to the pagerank algorithm taking O(n²) time, and being the most time consuming process in the entire program. The space complexity of the total program is also O(n²), which is due mainly to the existence of the matrix of size n². The program mainly uses basic data structures, such as matrices of ints, as well as vectors. The other data structure used is an unordered\_map, which, as we have seen, helps decrease the time complexity by a large order.

**Tradeoffs:**

The tradeoffs for this program are not many. They consist mostly of the usage of an unordered\_map in the file input stage, which raised the space complexity to decrease the time complexity by allowing data accesses in O(1) time. This one unordered\_map allowed webpages to be represented by integers, which helped simplify the graph, as well as simplify time complexity in pagerank for webpage accesses. The algorithm for searching keywords, is, in truth, perhaps not the most efficient, seeing as the keywords could have been placed in a tree of some kind to minimize time complexity. However, as seen, seeing as keywords vary between only 5 to 10 keywords, the complexity is only a constant multiplied by n, which is relatively still quite efficient. In this particular project, with n being a relatively small number, this efficiency suits our purposes perfectly. However, note that with larger n, such as for real-life practical-use search engines, such a search mechanism would definitely not be suitable in terms of efficiency.