Report for project

The pseudo-code for your indexing and ranking algorithms

The CTR is updated every time with each edit to the increment clicks and impressions, and the set. The score is calculated in the set score function. Looped after calculating the PR, impression and click.

- input csv file “keyword” and add the url of the webpages in vector, using Split String function into vector of strings
    - checks each word in the vector of strings, if the string contains more than 1 word (using morewords function), pass, else if contains only 1 word, delete tabs and spaces since they will not be saved properly
- input csv file “impressions”, using Split String function into vector of strings
- input csv file “clicks”, using Split String function into vector of strings
- input csv file “webgraph”, find url of source and destination and add edge in graph, 2d vector with ingoing and outgoing edges
- calculate page rank using graph,
    - initialized as 1/num of pages,
    - use a vector to calculate number of outgoing edges of each webpage using the graph
    - iteration 0 for all = 1/n, iteration 1 check if in 2d page = 1, then + arr[j] / outgoing [j], same for next iterations
    - iterate diff < 0.001, where diff = difference of sum between new page rank – page rank
    - get normalized page rank, by calculating sum of pageranks, dividing each pagerank by sum
- calculate score of each page using function in webpage class, using function given in slide, score = 0.4 * pr1 + ((1 - (0.1 * i1) / (1 + 0.1 * i1)) * pr1 + (0.1 * i1) / (1 + 0.1 * i1)) * CTR1) * 0.6
- sort using insertion sort, from highest to lowest score, to be used

- Program menus, if perform new search, go into search function

    - prompt user to enter search words, and split string to vector of string
    - first checks for quotation marks, to make them belong to the same string in the vector, as well as remove the quotation marks
    - when it finds quotation marks, adds to a string until reaching ending quotation marks, then adds the new full complete string to the vector of search strings

    - removes the old words with the quotation marks by finding the first and last character of each string to see if they are “.
    - then in another for loop, go over the words to search for an AND or and OR, and specifies the operation bool.
    - in a for looping over the webpages vector, we loop over the keywords of each webpages, and loop over the user input search words vector to see when usersearch[j] == searchwords[k], if they are equal, we increment count (which is a variable counting the number of keywords in common. If the operation is OR, if the count is > 0 which means at least one keyword in common, we add the webpage, else if it’s an AND operation, we have to have count = search words, so all search words are included in the keywords, to add webpage to the vector of webpages result.
    - finally, we go over the result vector of webpages to increment impression of each and print them, and since they are sorted descending using their score, then they will be printed accordingly using the sorting and indexing algorithms
    - if result size is 0, so no webpages in common, we print the webpages, else we print the url of each webpage in result, as well as loop over the webpages vector to increment the clicks of the webpage chosen.
- After we are done with our search, we update the impressions and clicks file and repeat the loop of asking to find a new webpage or exit.

A time and space complexity analysis for your indexing and ranking algorithms

Input Keywords File

Time complexity: O(number of lines in the keywords file N * number of keywords in line M) = O(N*M)

Space complexity: O(maximum number of keywords in line M)

Input Impressions/ Clicks File

Time complexity: O(number of line in the impression/clicks file N) = O(N)

Space complexity: Θ(1)

Setting ID of each webpage

Time complexity: O(for loop of the number of webpages N) = O(N)

Space complexity: Θ(1)

PAGE RANK:

Creating webgraph

Time complexity: O(number of line in the PageRank file N) = O(N)

Space complexity: Θ(N^2) of adjacency matrix/ graph in the webgraph class

Calculating Page Rank

Time complexity: O(number of webpages N and number of iterations until convergence I) = O(N*I)

Space complexity: Θ(N) storing page rank values

Calculate Score File

Time complexity: Θ(N), with N number of webpages

Space complexity: Θ(1)

Sorting the webpages using Insertion Sort

Time complexity: Θ(N^2), with N number of webpages

Space complexity: Θ(1)

The main data structures used by your algorithm

I mostly used vector for the string of keywords, as well as a vector of the webpages class, and a 2d vector to hold the graph for the web graph, with each source and destination edge of nodes (or here webpages). I used classes, one for the Webpage with the information of each webpage, and the other is a class of web graph for each connected webpage.

- vector<Webpage>: stores instances of the Webpage class, representing webpages with their associated properties such as URL, keywords, impressions, clicks, and score.
- Webgraph: directed graph that models the relationships between webpages, adjacency matrix of the webpages
- vector<string>: used to store individual keywords extracted from user input in Search function and file data in keywords file.
- vector<vector<double>: 2D vector represents the graph structure in the Webgraph class, storing the adjacency matrix.
- vector<double>: used to calculate and store the PageRank values for each webpage
- vector<int>: vector is used to count the number of outgoing edges for each webpage in the graph, necessary for the calculation

Any design tradeoffs you made along with their justifications

- I decided to use vector and not arrays because their dynamic size, and their memory allocation, as well as their built in functions.
- I decided to use a function to increment the clicks and impressions instead of the set function, since we repeat it throughout each search iteration.
- I used Insertion Sort instead of another sorting algorithms because it works well with classes, it is simpler to implement, and has a better performance for small input sizes which is probably the case for the webpage list
- PageRank values for webpages based on a given web graph. It uses an iterative approach to update the PageRank values until convergence. I decided to repeat until convergence and not only 2 iterations, similar to the video, for the PageRank to be more specific.
- I separated the difference class files, with a header and cpp file, separate files (e.g., Webpage.cpp, Webgraph.cpp) can improve code readability, reusability, and maintainability.