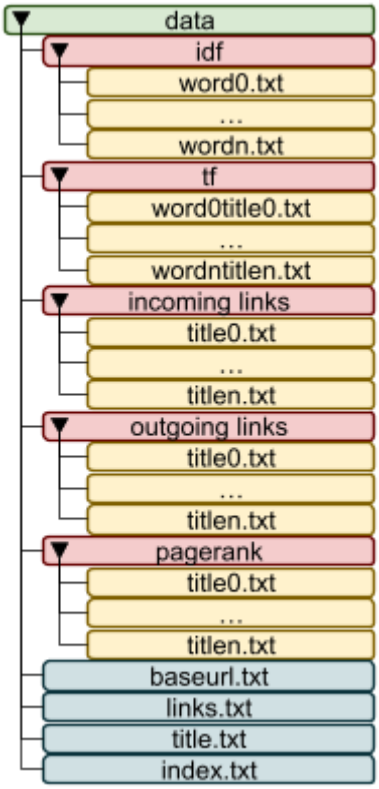
COMP 1405Z – Fall 2023
Course Project: Analysis Report
Kassem Taha & Barnabé Frimaudeau

File Structure

For our file organization, we decided to store all the data types together. So if you want to look for the tf value of something, you can look in the tf directory. This made the most sense to us. Instead of having a directory for each webpage and having 1000, you have 5. We put all the values together and search a specific directory for the value you need. We have a very simple naming convention. We can assign an index value to each url and add them to the files at the same time so that they are directly related. This means we know that the index on line 4 of index.txt is from the web page with the url on line 4 of links.txt. We use this fact to make a hashmap in searchdata.py and make searching for values instantaneous. So incominglinks, outgoinglinks, and pagerank use the index as its filename. Before this, we used to use the title as our filename but as we read the project questions FAQ we realized this solution did not work as titles are not unique so we just refactored our code. Anyways, we approach tf and idf differently as these are related to words. For idf, we only store the words as that's all we need to know for idf. For the tf value however, we need both the word and index. We simply just concatenate the word and title and make that the file name to quickly search for it. With this we made a quick, easy to read file structure that fits our needs perfectly.



crawler.py

Our Approach:

During the early development of our project, we used crawler.py as the data collector, while most of the data processing and calculations were to be handled by searchdata.py. This approach was unfortunately highly inefficient. To improve our program's performance, we restructured our approach, making crawler.py handle the processing of the data and calculations, with searchdata.py primarily responsible for data retrieval.

In the revised system, the core of our data processing takes place within the scrape_url function. Here, we extract and parse data from the requested web page, including term frequency (tf) values, outgoing links, and a list of words present on the page. Additionally, we append URLs found in the webpage to a list to

continue our crawling process. The data obtained in scrape_url is temporarily stored in lists, and the crawler then takes this data and saves it to files. This process is repeated until all interlinked web pages have been visited.

After completing the crawling process, we utilize the data to find the idf using a dictionary of which words were present in each url. We also calculate incoming links by analyzing the outgoing links from each web page. We then apply pagerank algorithms to the collected data which we can use for our calculations in the search function. This process can extract data from 1000 web pages in under two minutes.

Function Name	*Time Complexity*	*Space Complexity*	*Explanation*
`scrape_url`	O(NxMxP)	O(N)	Let N be the size of the webpage string, M be the amount of unique words in the webpage, and P be the number of interlinked web pages. Time Complexity: The two nested while loops depend on webString, which makes those two run in O(NxM) time. The if statements after those while loops use the *in* operator for a list, making them N(P). This may seem very time complex but variables like M are small, about 10-20 in size. We have another similar problems with links/<a> but it still simplifies to O(NxMxP) or $O(N^3)$ Space Complexity: The space complexity of scrape_url is not bad at all. Most space complex things are later cleared for the next scrape. Even then, they are all a subset of N, the size of the webpage string.
`crawl`	O(n(NxMxP)	O(N)	Let n be the number of interlinked web pages Let N be the size of a webpage string, M be the amount of unique words in the webpage, and P be the number of interlinked web pages. Time Complexity: The function calls other functions such as calculate_idf, which are of $O(N^2)$. The rest of the actions, excluding scrape_url(), are constant/linear. The only real time-complexity comes from the scraping process. Space Complexity: The most significant space-complex factor is the lists urls, idf and the dictionary inlinks that grow

			with the number of URLs crawled. The space complexity of urls, and inlinks can be approximated by O(N) where N is the number of URLs crawled. Idf can be approximated by O(M) where M is the amount of unique words in all webpages.
`calculate_incoming`	O(NxM)	O(N)	Let N be the number of interlinked web pages and M be the number of incoming links the webpage linked to Time Complexity: This function's time complexity is most affected by the two nested loops, which are both O(N). The time complexity is lower than O(N^2) because the number of incoming links is much smaller than the total number of URLs, hence why we gave it a new variable. Space Complexity: We have 2 basic lists and a dictionary of size N, the number of interlinked web pages.
`calculate_idf`	O(N)	O(1)	Let N be the number of unique words found in all web pages during the crawl Time Complexity: Time complexity is primarily determined by the for loop over the unique words in idf. In the worst case, where each word corresponds to all URLs, the complexity is O(N). Space Complexity: There is not much space complexity as these variables are defined previously in the file and we only define 1 variable.
`calculate_pagerank`	$O(N^2*M)$ or $O(N^3)$	O(N^2)	Let N be the number of interlinked web pages and M be the number of incoming links to the current webpage. Time Complexity: The time complexity is $O(N^3)$ for multiplying two N x N matrices. The rest of the code is negligible as it is mostly single for loops and 1 while loop. Space Complexity: The most space complex factor is 'matrix', a 2D list with size N. There are other int variables and a 1D list with size N but that is the most complex.

searchdata.py

Our Approach:

Within searchdata.py, the functions are straightforward and efficient. The most significant computation takes place in the init function. This arises from the necessity to quickly determine the file name of a webpage based on its URL. Our system stores files based on each web page's index. Since we store links and titles at the same time, a webpage's index corresponds to the same line of a webpage's URL in index.txt and links.txt (where we store links and titles). Therefore we utilize a hashmap to rapidly locate the index of a URL. This optimization allows most functions to operate at O(1) complexity, with only a few operations working at O(N) complexity since they return lists.

Function Name	*Time Complexity*	*Space Complexity*	*Explanation*
`init`	O(N)	O(N)	Where N is the number of linked web pages. Time Complexity: It is O(N) because there are 2 read_files which depend on the number of interlinked web pages, N, and the for loop to construct the hashmap is size N. Space Complexity: Because we have 2 lists of size N and a dictionary also of size N.
`get_tag`	O(1)*	N/A	The *in* operator is being used for a dictionary, not a list, therefore it can't be O(N). We do not store any variables either so it is not space complex at all.
`get_outgoing_links`	O(N)	O(N)	Time Complexity: It is O(N) where N is the number of outgoing links in the file that were read. Space Complexity: The list we return is size N.
`get_incoming_links`	O(N)	O(N)	Time Complexity It is O(N) where N is the number of incoming links in the file that were read. Space Complexity: The list we return is size N.

`get_idf`	O(1)	O(1)	Time Complexity: There exists only 1 line in the file that is being read, therefore not O(N). There is also a single if-else statement. Space Complexity: We assigned the result to 1 variable.
`get_tf`	O(1)*	O(1)	Time Complexity: There exists only 1 line in the file that is being read, therefore not O(N). There is also a single if-else statement. Space Complexity: We assigned the result to 1 variable.
`get_tf_idf`	O(1)*	N/A	There is only a return statement, and the functions being called in this statement are technically O(1).
`get_page_rank`	O(1)*	O(1)	Time Complexity: There exists only 1 line in the file that is being read, therefore not O(N). Space Complexity: We assigned the result to 1 variable.
*** `init()` just runs once, so this is O(1) unless init was never run again then it is O(N).**			

search.py

Our Approach:

When we initially developed the search function, our primary goal was to make it work. We began this by following the steps in the 'Example Calculations' document from the Project Lectures. The initial approach involved creating a vector space model for the documents. We converted the search phrase into a list by using the split function and mapped the idf_tf values of all the documents to the words. However, this approach had a major flow as it did not account for duplicate words. To address this, we reworked the way we calculated the query vector and vector space model, using a dictionary-based solution.

With this data, we calculated the cosine similarity of all the document vectors with the query vector. This step was relatively straightforward, involving a few for loops and a list to store the results.

At this point, we were nearly done, but there was one more thing we needed to implement: applying the pagerank boost if requested by the user. Given the way in which we structured the vector space model, we knew that each index of the cosine similarity result list corresponded to the index of the URLs list.

Therefore, we iterated through the cosine similarity list and multiplied it to the calculated PageRank value of the same index in the URLs list.

Finally, we proceeded to sort the search results. As specified in the assignment instructions, we only needed the top 10 results. To minimize unnecessary time complexity, we developed our own sorting algorithm. This algorithm involved a basic linear search that iterated through the entire list to find the highest value, added it to a list of already selected results, and then located the second-highest value, and so on. After 10 iterations, the search algorithm was complete, yielding the top 10 search results. Adding the URL, title, and cosine score to the results was straightforward, as the cosine index directly corresponded to both the URL and title indices.

Function Name	*Time Complexity*	*Space Complexity*	*Explanation*
`search`	$O(N^2)$	$O(N^2)$	Time complexity: Where N is the amount of interlinked web pages. When we construct the matrix we have two for loops and in the end it may seem like O(N^3) but it is more similar to O(N) because we first only loop 10 times and the list, already_picked, can have a max size of 10. Space complexity: We have a 2D list with a width and height of N and a 1D list with size N which simplifies to $O(N^2)$

List of functionalities:

Functionality	*Complete / Incomplete*
Scrape through all interlinked web pages	**COMPLETE**
Store necessary data in directories and files	**COMPLETE**
Read all the necessary data in a reasonable amount of time	**COMPLETE**
Correctly calculate the search results asked by the user and sort them in a reasonable amount of time	**COMPLETE**

Other qualities of the project:

We believe to have a good balance of time and space complexity in this project, instead of trying to make one very performant and neglect the other.