1405Z Course Project Report
Web Crawler - Search Engine

Seyedsajad Hosseini

October 2022

Introduction

This report describes all aspects of design and implementation of the web crawler and search engine excluding the calculation methods and implementation limitations illustrated in the lecture notes and the project specifications. In doing so, first, the reasoning behind the choice of file structure and different types of directory structure tested for this project is presented. Then, different steps of the algorithms and functions used in the three main modules are briefly reviewed. Discussions of runtime complexity and space complexity of the modules and their functions are followed by a review of design approaches that had significant impacts on the development of the program.

The submitted code is the final edition of several very different implementations. While all those implementations could pass the tests in a reasonable amount of time (less than 100 seconds[1] for the fruits datasets), the final version achieves a better runtime and has less amount of code. For the fruits datasets, the crawl process, which encompasses nearly all calculations, takes on average 81 seconds and the search process for the test files (e.g. *fruits2-all-test.py*) takes 0.5 seconds.

[1] Calculations are performed using MacBook Air (M1, 8GB RAM) and internet connection at my residence. Running the test file using the same laptop and the university's Wi-Fi resulted in a 75% improvement in the runtime.

File/Directory Structure

One of the main challenges of this project was the design of the file structure. Most functions in the three main modules either write data to the files or read data from the memory. Therefore, the file structure defines how, at least parts of, these functions work. The back-and-forth process of designing the file structure and writing more efficient functions resulted in several different versions of file structure and functions. Figure 1 shows two intermediate versions (top) and the final structure (bottom) for the *tinyfruits* dataset. It should be noted that all the different file structures shown below (and some other not presented in this report) result in a successful crawl and can pass all the tests.

At the beginning, a separate directory was designed for each URL (Figure 1, top). Inside the base directory, named *data,* all subdirectories were named according to the URLs of the crawled webpages. The only part of the URL left out of this hierarchy of directories was the URL's protocol. This way, if there were more than one webpage with the same title or the same path (but different domains), the *crawler* could place their data in separate directories and the functions in *searchdata* module could read the data without confusion. The difference between two file structures at the top of Figure 1 is that one uses the *words.json* file to store all the words and their relative frequencies in the corresponding webpage, while the other stores the relative frequency of each word in a different *txt* file (e.g. *banana.txt*). Moreover, the file structure at the top right uses *json* files to maintain idf and page rank values, while the one at the top left uses *txt* files, resulting in less efficient and more cumbersome functions.

Figure 1. data file structure (top: two older versions, bottom: final version)

Although these structures may work better for a much larger database, they were relatively inefficient for the fruits databases. While in the crawl phase of the program thousands of folders and files had to be created, opened, and written in, during the search, those folders and files had to be opened and read. Therefore, use of *json* files to store data (Figure 1, bottom), in addition to introducing a global dictionary in the *searchdata* module to store tf_idf data, was tested. This strategy resulted in a substantial improvement of both crawler and search efficiencies and, therefore, was chosen as the ultimate design of the file structure.

In the final design, seven *json* files store idf values, tf values, tf_idf values, incoming links, outgoing links, page ranks, and urls. While some of these data (e.g., incoming links) are not necessary for performing the final search, as per project specifications they are stored for further tests.

In general, during the design and implementation of various file structures, several aspects of the project were considered:

- Runtime efficiency of the *crawler* module

  The question of can the change in the file structure improve runtime efficiency was always a major deciding factor. Runtime measurements showed that a larger hierarchy of files causes more time wasted for creating files and folders, opening them, and writing data. The bottleneck of the crawling process is requesting and retrieving webpage content performed by *webdev* module. For different file structures, this module takes 85% of the total time spent for the whole crawling process[1]. As using this module to interact with the webpages is mandatory, the only area of improvement is reducing runtime of that 15%. Compared to the initial hierarchy of folders and files(Figure 1 top), the current file structure (Figure 1 bottom) decreases the proportion of time used by other parts of the crawler (except `webdev.read_url(url)`) from 15% to 10%.

- Runtime efficiency of *searchdata* and *search* modules

  The choice of file structure also affects the performance of the modules reading data from the created files. A larger hierarchy (as in top structures of Figure 1) leads to extra time needed to open and read data from several files. Flattening file structure, combined with a new global dictionary in the *searchdata* module to store tf_idf data, leads to a significant improvement in runtime efficiency. While in a larger hierarchy it takes approximately 14

[1] using MacBook Air (M1, 8GB RAM) and the university's Wi-Fi

seconds to run the test file, in the flat file structure on average 0.5 seconds is needed to perform the same task[1].

- Memory allocation (RAM)

  Considering the relatively small size of the fruits database, none of the tested approaches is constrained by the memory efficiency. The largest *json* file in the current structure is smaller than 600 KB. Therefore, considering a situation where several of the data files are open simultaneously, only a few MBs of RAM is needed to run either version of the program described above.

- Cleaner and more readable code

  Compared to the hierarchy of files and folders, the flat file structure results in simpler and more readable functions. There's no code needed to create directories and search for files in the subdirectories. In addition, some other parts of the program such as functions that delete files and folders of the previous search or read data files have become much shorter.

- Generalization of the code for larger databases

  This is one area where the top structures of Figure 1 *may* have a better performance than the flat structure of seven *json* files. If this code is executed for a dataset several order of magnitude larger than the fruits dataset and all the limitations set by the project specifications are considered, runtime efficiency of the chosen file structure is still better than the hierarchy of folders (because it uses *json* dictionaries and less time is wasted for creating directories and files). However, as the dataset grows much larger, the dumping and loading process of *json* files becomes more time consuming and may even need more memory than that of an average computer[2].

[1] There'll be more discussion on the complexity of the functions in the following sections.

[2] It should be noted that for such a mammoth database, other functions such as matrix multiplication will be the primary bottleneck of the calculations. Moreover, the problem of dumping and loading very large json files can be addressed using partial json reading algorithms or even writing and querying data using SQL libraries.

How the Modules Work

The following table gives a brief description of how the five modules are implemented. In addition to the modules required by the project specifications, two more modules are added: a *"constants"* module to store names of the files and constants used in the functions and a *"matmult"* module to perform matrix multiplication and Euclidean distance calculations.

***Crawler* module**			
This module has a main function (`crawl(seed)`) and several additional functions to perform crawling of the webpages in the following steps.			
Step	Task	Function Name	Return Variable(s)
1	Delete files containing previous crawl data	`delete_files(BASE_DIR)`	-
2	Add seed to the queue	-	-
3	Read webpage html using *webdev* module	-	-
4	Find title of the webpage	`extract_title(page_content)`	`title`
5	Find the words in the webpage	`extract_words(page_content)`	`words`
6	Find the URLs of links in the webpage	`extract_links(page_content)`	`links`
7	Add links found in step 6 to the queue	-	-
8	Calculate relative frequency (tf) of the words and idf data of the current webpage	`calculate_word_freq (words_count_dict, word_list)`	`words_count_dict, words_rel_freq`
9	Add the current URL to a list of processed URLs and delete it from the queue	-	-
10	Go to step 3 until the queue is empty	-	-
11	Save tf data (step 8, stored in `words_rel_freq_dict`) to a *json* file	`write_tf_data (words_rel_freq_dict)`	-
12	Save idf data (step 8, stored in `words_count_dict`) to a *json* file	`write_idf_data(words_count_dict, len(processed_urls))`	-
13	Calculate and write tf_idf values to a *json* file	`write_tf_idf_data(processed_urls, words_rel_freq_dict)`	-
14	Find incoming links, save outgoing and incoming links of all URLs to a *json* file	`write_inout_links(links_dict)`	-

15	Save URLs and their extracted titles to a *json* file	`write_urls(processed_urls, titles)`	-

16	Calculate and save page rank values of the crawled URLs	`calculate_page_rank(processed_urls)`
		`matrix_mult(a, b)`
		`euclidean_dist(a, b)`
			-

matmult module

This module contains `matrix_mult(a,b)` and `euclidean_dist(a, b)` for page rank calculations. The former returns a matrix, and the latter returns a number of type float.

searchdata module

Unlike the *crawler* module, *searchdata* doesn't have a main function. The functions in this module are created based on the project specifications. Some of these functions (e.g., `get_incoming_links` are not necessary for the execution of the *search* module). As tf_idf values are used repeatedly in calculations of similarity matrix in the *search* module, these values are loaded into a global dictionary in the *searchdata* module.

Function Name	Task	Helper Function	Return Variable(s)
`read_file(url,file_name)`	Reads contents of a *json* file	-	A dictionary of corresponding data
`get_outgoing_links(url)`	Returns outgoing URLs found in the *url* webpage	`read_file (LINKS_OUT_FILE_NAME)`	A list of URLs
`get_incoming_links(url)`	Returns URLs having a connection to the *url*	`read_file (LINKS_IN_FILE_NAME)`	A list of URLs
`get_tf(url, word)`	Returns tf value of the *word* in the *url* webpage	`read_file (IDF_FILE_NAME)`	A number (of type float)
`get_idf(word)`	Returns idf value of the *word* in all crawled webpages	`read_file (IDF_FILE_NAME)`	A number (of type float)
`get_tf_idf(url, word)`	Returns tf_idf value of the *word* in the *url* webpage	`read_file (TF_IDF_FILE_NAME)`	A number (of type float)

`get_page_rank(url)`	Returns page rank of *url*	`read_file (PAGE_RANK_FILE_NAME)`	A number (of type float)

search module

This module has a main function (*search(phrase, boost)*) that searches the stored data for the best URL using the Vector Space Model. If *boost* is True, page rank from Random Surfer Model is also considered in the calculations. The following steps are performed using additional functions.

Step	Task	Function Name	Return Variable(s)
1	Read URLs and their titles from a *json* file	`read_urls_titles()`	Lists of *urls* and *titles*
2	If *boost* is True, read page ranks from a *json* file	`read_page_ranks()`	A list of *page_ranks*
	If boost is False, all pages have the same rank (page ranks = 1)	-	
3	Create tf_idf vectors for URLs and query phrase, calculate number of unique words in the query phrase	`create_vectors(urls, phrase)` (used in step 4's function)	Two vectors (*vectors, query_vector*), a number of type int (*phrase_list_size*)
4	Calculate similarities and return their top 10	`calculate_top_similarities(urls, titles, phrase, page_ranks)`	A list of top 10 URLs (each one is a dictionary of *urls, titles, scores)*

constants module

This module only contains the names of files, strings, and constant values repeatedly used in other four modules.

Discussion on Runtime Efficiency of Modules and Functions

***Crawler* module**
This table contains a discussion on complexity analysis of all functions used in the *crawler* module. There are two functions with O(N×M×P) complexity, but as two of the three variables (M and P) are usually much smaller than N, they cannot be considered as $O(N^3)$ complexity. Therefore, the function with the highest complexity, $O(N^2)$, is the one using matrix multiplication for page rank calculations. As the module is mostly affected by this function, increasing N (number of crawled URLs) will have the most unfavorable effect on performance of the *crawler* module. To evaluate the effect of this high complexity, runtime of the *crawler* module for two datasets (tinyfruits with 10 URLs and fruits5 with 1000 URLs) has been measured. In doing so, the time spent by the *webdev* module to request and retrieve data from URLs is omitted. For tinyfruits and fruits datasets the runtimes are 0.0064 and 2.5599, respectively. For the same datasets, runtimes of page rank calculation's function (that uses the matrix multiplication) are 0.0015 and 2.1637, respectively. While runtime of the whole module (minus the *webdev* module) has increased 400 times, the runtime of page rank function has grown 1470 times. As there are other factors in play, multiplying N by 100 does not result in 100^2 increase of the function's runtime, but the function's share of the total runtime of the *crawl* module has increased considerably. Figure 2 compares the effect of page rank function with $O(N^2)$ complexity on the *crawler* runtime. As size of the dataset grows, more time is allocated to the function with the highest complexity.

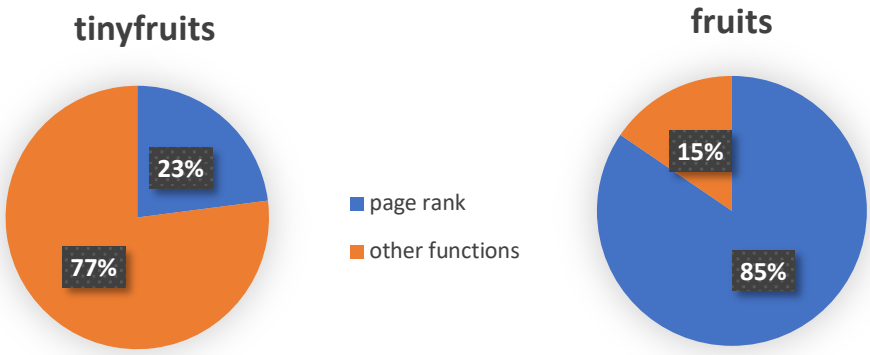

Figure 2. share of page rank function vs. other functions of the *crawler* runtime for a dataset of size 10 (tinyfruits) and a dataset of size 1000 (fruits)

Function/Task	Complexity	N
delete_files(BASE_DIR)	O(1)	-
This function finds the list of files in the base directory (named *data*) and deletes those files. In the previous versions of the file structure, where hierarchy of folders was used, a recursive function was responsible for searching the folders and deleting files and subfolders. In that version, as the number of URLs increased, more files and folders were created to save data. Therefore, their deletion was of O(N) complexity. In the final code only seven data files are created for all sizes of databases and, therefore, deletion of the base directory and its files is independent of number of URLs.		
Appending URLs to the queue	O(1)	-
Deleting URLs from the queue	O(N)	Number of URLs
The processed URLs are deleted from the start of the queue (which is a list), therefore all the items at positions 1 to N-1 should be transferred to one position lower.		
extract_title(page_content)	O(N)	Number of characters in the crawled text
Using the built-in *find()* function to find index of each of <*title*> and </*title*> requires at most N operations. However, as titles are at headers of HTMLs, in practice, this isn't a time-consuming operation. When indices of <*title*> and </*title*> are found, slice operation of complexity of O(M<N) is used. M (the length of the page title) is considerably smaller than N.		
extract_words(page_content)	O(N×M×P)	Described below
HTML content of the webpage is first split using the built-in *split(separator)* function where the separator is <*p*>. This operation is of O(N) complexity. Then if each string in		

the resulting list contains * * it will be split into more string parts using the same function and *"\n"* as the separator (if the words are separated by space instead of new line, the split will be performed once more, this time using space as separator). Finally, the split words in each new string part are added to a list. Therefore, this function is of O(N×M×P) complexity (N: number of the characters in the webpage HTML, M: size of the list of string parts resulting from split by , P: size of string parts that have * *). In the fruits database, M is always 1, while N and P are proportional to the number and length of words. Hence, for this database the complexity can be considered as $O(N^2)$.		
extract_links(base_url, page_content)	O(N×M×P)	Described below
HTML content of the webpage is first split using the built-in *split(separator)* function where the separator is *"href="*. This operation is of O(N) complexity. Then for each string in the resulting list that contains **, the index of ** is found using the built-in *index()* function (an operation of O(M) complexity). Finally, the absolute or relative URL is found using slice operation (an operation of O(P) complexity). Therefore, this function is of O(N×M×P) complexity (N: number of the characters in the webpage HTML, M: size of the list of string parts resulting from split by *"href="*, P: size of string parts that have **). In the fruits database, while M and P are small, their multiplication is not negligible.		
calculate_word_freq (words_count_dict, word_list)	O(N)	Number of words in the webpage
The function once loops through the list of words in the webpage (complexity of O(N)) and adds data to a dictionary and then retrieves the dictionary data to perform some calculations (complexity of O(1)).		
write_tf_data (words_rel_freq_dict)	O(N)	Number of characters in the input dictionary
This function writes tf dictionary of every URL to a *json* file. If writing every character in the dictionary is considered as one operation, N operations are necessary to write tf values of the database to the file. If a hierarchy structure was implemented, N would be smaller, because only tf values would be written to tf files and words would be used as the names of the files for each URL. However, as the number of files increases the operation of opening and closing files should also be considered.		
write_idf_data(words_count_dict, len(urls))	O(N)	Number of characters in the input dictionary
Like the previous function, this function writes idf values to a *json* file. If there are N characters in the dictionary, it will be of O(N) complexity.		

write_tf_idf_data(urls, words_rel_freq_dict)	O(N×M)	Number of characters in the input dictionary
In this function, first tf-idf values are calculated. This task is performed for N URLs and M unique words in the dictionary. Therefore, the complexity of this part is O(N×M). Then the result is written to a *json* file, which results in P operations, where P is the number of characters in the data written to file. Hence, if M is significantly smaller than N, this function is of linear complexity.		
write_inout_links(links_dict)	$O(N^2)$	Number of URLs
There is a nested loop in this function that finds incoming links of each URL from outgoing links of other URLs. Therefore, as there are N URLs in the dataset, the complexity of this function is of type $O(N^2)$.		
write_urls(urls, titles)	O(N)	Number of characters in the input lists
calculate_page_rank(urls)	$O(N^2)$	Number of URLs
There is a nested loop in this function that assigns 0 or 1 to a list of lists. In addition, it uses *matrix_mult(a,b)*, which is of $O(N^2)$ complexity. Therefore, as the number of URLs (N) increases, complexity of the function grows quadratically.		
***matmult* module**		
matrix_mult(a,b)	$O(N^2)$	Number of URLs
This function calculates matrix multiplication using a triple nested loop. As a result, in general, it is an $O(N^3)$ complex function. However, for the current application one of matrices is a vector, which reduces the complexity to $O(N^2)$.		
euclidean_dist(a, b)	O(N)	Number of URLs
Calculation of Euclidean distance between two vectors of size N is of complexity of O(N).		

***searchdata* module**
All the functions in this module only read the saved data calculated in the *crawler* module. Therefore, if reading each character is considered as one operation, every function in this module is of O(N) complexity, where N is the number of characters in the file.

<table>
<tr><th colspan="3">search module</th></tr>
<tr><td colspan="3">As the following discussion on the runtime complexity of functions in the module shows, the search module has a complexity of O(N×M), where N is the number of URLs and M the number of unique words in the search query. In most cases, M is significantly smaller than N, making this module of linear complexity.</td></tr>
<tr><th>Function</th><th>Complexity</th><th>N</th></tr>
<tr><td>read_urls_titles()</td><td>O(N)</td><td>Number of characters in the file</td></tr>
<tr><td colspan="3">The name of the function reveals its sole purpose. Increasing the size of the data file written in the crawler module will linearly increase runtime of this function.</td></tr>
<tr><td>read_page_ranks()</td><td>O(N)</td><td>Number of characters in the file</td></tr>
<tr><td>create_vectors(urls, phrase)</td><td>O(N×M)</td><td>Described below</td></tr>
<tr><td colspan="3">This function creates one vector for each URL. The vector’s size is dependent on the number of unique words in the input phrase. To create the vector, the function must open the files that store values of tf and idf for that specific URL-word combination. However, to improve the efficiency of the code, a global dictionary has been designed in the searchdata module to reduce the number of operations related to reading data from the files.
Therefore, for N URLs and M unique words in the search phrase, the complexity of this function is O(N×M).</td></tr>
<tr><td>calculate_top_similarities(urls, titles, phrase, page_ranks)</td><td>O(N×M)</td><td>Described below</td></tr>
<tr><td colspan="3">Like the previous function, this function performs cosine similarity calculations on all elements of N vectors. It there are M unique words in the search query, the complexity of this part of function is O(N×M).
To find and return URLs with the highest similarity to the search query, the function searches the scores of all URLs (N operations). Number of searches in this project is 10 (P). Therefore, the complexity of this part of the function is also O(N×P).
In the fruits database M and P are much smaller than N and the complexity of the function is O(N).</td></tr>
</table>

Discussion on Space Efficiency of Modules and Functions

Space complexity of various functions can be reduced at the expense of the runtime complexity and vice versa. Therefore, it must be decided which one has the priority and what are the red lines for each one. Due to the relatively small size of the fruits database, memory usage of the variables in the program is small. As discussed in the file structure section, the largest dictionary has a size of less than 600 KB, when stored as a *json* file. However, as this program can be used for far larger databases, a discussion on the space complexity of the variables is necessary.

<table>
<tr><th colspan="3">Crawler module</th></tr>
<tr><td colspan="3">Although this module doesn't have any global variables, because of its main function (crawl(seed)) it stores several variables from the start of the crawl until it finishes. The data that these variables store can be stored in the files and retrieved later. This results in smaller memory usage but also a slower process. Therefore, it seems (and proves) to be more runtime efficient if the variables save more data and write all of them to files at the end of the crawl process.</td></tr>
<tr><th>Function/Task</th><th>Complexity</th><th>N</th></tr>
<tr><td>crawl(seed)</td><td>O(N×M)</td><td>Described below</td></tr>
<tr><td colspan="3">The function creates several lists and dictionaries, but the largest of them is a dictionary with URLs as keys and dictionaries of unique words in the webpages as values. If there are N URLs and, on average, M unique words in every webpage, the function is of O(N×M) space complexity.</td></tr>
<tr><td>delete_files(BASE_DIR)</td><td>O(1)</td><td>-</td></tr>
<tr><td colspan="3">Path of files and folders to be deleted are stored in a list in the function. As this number is constant, this function can be considered of O(1) complexity.</td></tr>
<tr><td>Appending URLs to the queue</td><td>O(N)</td><td>Number of URLs</td></tr>
<tr><td colspan="3">As the number of crawled URLs increases, the space requirement of the queue increases linearly.</td></tr>
<tr><td>extract_title(page_content)</td><td>O(1)</td><td>-</td></tr>
</table>

<table>
<tr><td colspan="3">The function returns title of each webpage one every call. Given there's not a big difference between size of webpage titles, this function can be considered of O(1) space complexity.</td></tr>
<tr><td>extract_words(page_content)</td><td>O(N)</td><td>Number of words in the webpage</td></tr>
<tr><td colspan="3">While looping through the webpage content to find and extract words, HTML content of the webpage is split and stored in a list. The extracted words are also stored in a list. Increasing number of paragraphs in the webpage and words in the whole page, results in a linear increase of space complexity of the function.</td></tr>
<tr><td>extract_links(base_url, page_content)</td><td>O(N)</td><td>Number of URLs in the webpage</td></tr>
<tr><td colspan="3">Similar to the previous function, increasing number of URLs in the webpage results in a linear increase of the memory usage of the function.</td></tr>
<tr><td>calculate_word_freq (words_count_dict, word_list)</td><td>O(N)</td><td>Number of unique words in the webpage</td></tr>
<tr><td colspan="3">The function creates two dictionaries whose sizes are dependent on the number of unique words in the crawled webpage and number of unique words in the whole dataset. For the fruits dataset, these numbers are small and in practice, the memory usage of the function is negligible.</td></tr>
<tr><td>write_idf_data(words_count_dict, len(urls))</td><td>O(N)</td><td>Number of unique words in the dataset</td></tr>
<tr><td colspan="3">Like the previous function, the space complexity of this function is dependent on the number of unique words in the dataset.</td></tr>
<tr><td>write_tf_idf_data(urls, words_rel_freq_dict)</td><td>O(N×M)</td><td>Described below</td></tr>
<tr><td colspan="3">Here, a dictionary whose values are themselves of type dictionary is created. The key in the parent and child dictionaries are URLs and unique words of the dataset, respectively. Therefore, the space complexity of this function is affected by number of URLs (N) and number of unique words (M).</td></tr>
<tr><td>write_inout_links(links_dict)</td><td>O(N)</td><td>Number of URLs</td></tr>
<tr><td colspan="3">This function creates a dictionary to save incoming links to each URL in the dataset. Therefore, it is of O(N) space complexity.</td></tr>
<tr><td>write_urls(urls, titles)</td><td>O(N)</td><td>Number of URLs</td></tr>
<tr><td>calculate_page_rank(urls)</td><td>$O(N^2)$</td><td>Number of URLs</td></tr>
<tr><td colspan="3">The adjacency matrix used for calculation of page rank is a list of lists. The size of rows and columns of this matrix is equal to the number of URLs.</td></tr>
</table>

***matmult* module**		
matrix_mult(a,b)	$O(N^2)$	Number of URLs
Like *calculate_page_rank(urls)*, this function uses a matrix of N^2 size, where N is the number of URLs in the dataset.		
euclidean_dist(a, b)	O(N)	Number of URLs
Calculation of Euclidean distance between two vectors of size N is of space complexity of O(N).		

***searchdata* module**
All the functions in this module first read the saved data calculated in the *crawler* module, then find the required value based on an input key, and finally pass the value to the *search* module or the user. Therefore, the space complexity of these functions is dependent on the data saved during the crawl process. To improve runtime efficiency of the search, a dictionary with URLs as keys and dictionaries of tf-idf as values is implemented as a global variable. This dictionary is the largest variable used in the module. Therefore, the *searchdata* module is of O(N×M) space complexity.

***search* module**		
Function	Complexity	N
search(phrase, boost)	O(N)	Number of URLs
The search function implements the following functions. The largest variable in its scope is a list of N items, where N is the number of URLs.		
read_urls_titles()	O(N)	Number of URLs
The function reads a dictionary with key size of N URLs and then creates two lists of size N. Therefore, it is of O(N) complexity.		
read_page_ranks()	O(N)	Number of URLs
Same as the previous function, this function reads a dictionary with key size of N URLs and then creates a list of size N.		
create_vectors(urls, phrase)	O(N×M)	Described below
Here, a matrix of size N×M is created. N is the number of URLs and M is the number of unique words in the search query.		
calculate_top_similarities(urls, titles, phrase, page_ranks)	O(N×M)	Described below

The matrix created in the previous function is used here and has the largest impact on the space complexity. Therefore, given N is the number of URLs and M is the number of unique words in the search query, the function is of O(N×M) complexity.

Major Design Approaches

- A global dictionary in the *searchdata* module to save tf-idf values

  The *search* module repeatedly calls *searchdata.get_tf_idf(url, word)* to calculate elements of similarity vectors. As the tf-idf values are stored in a *json* file, cumulative time spent for reading a relatively large file for a large number of URLs will be considerable. By saving the contents of the file on the memory, retrieval of the data is performed considerably faster. While the same approach can be implemented for the page rank datafile too, because of smaller size of this file and the fact that if boost is False, page rank data will not be used in the calculations, it hasn't been stored as a global variable.
- Saving data files in *json* format (instead of *txt*)

  If dictionaries were to be written to a *txt* file, a rather cumbersome scheme for writing and reading data was needed. Using *json* results in a much simpler code.
- Flat structure vs hierarchy of directories

  File structure design affects all aspects of the program. Discussions in the file structure section show the reasoning behind the chosen flat file structure.
- Addition of a module containing shared constants and file names

  To reduce typing errors and improve code reusability and modification, all file names and constants are moved to a separate module.
- Performing all the calculations in the *crawler* and *matmult* modules

  At the beginning stages of the project, most tf, idf, and page rank calculations were performed in the *searchdata* module. However, as the raw data required for these calculations are in the *crawler*, it became clear that moving the

calculations to this module results in less operations (i.e., file opening and reading).

- Writing helper functions

  To follow the DRY concept, moving lines of code that perform repeated tasks to a helper function was necessary. In doing so, functions for writing and reading files were implemented in the *crawler* and *searchdata* modules, respectively. Moreover, to improve readability and maintenance of the code, every function that performed several tasks was split into several smaller functions with specific tasks.