# COMP 1405Z
# Course Project Report

Khang Tran

School of Computer Science
Carleton University
October 2022

## Contents

# 1 Crawler Module (crawler.py):

## 1.1 Overall design:

When it comes to designing the crawler module, I just want to develop it as simply as possible. My first intention was to implement the module to get the data as stated in the course project specifications only. But later after reading the whole course requirement, I decided to precompute all the necessary information and store it file. Although it will create many files and folders, it will help my other module to run faster, because it doesn't have to compute all the data again.

Instead of writing a big function responsible for multitasking, I decided to decompose it into as specific as possible. Each function on my module only serves one or a few tasks. This will enhance the readability and maintenance of my module. During the code process, I had once named the folder by the title, but then I realized that the title of each page is not unique, thus I have to change the way to name and map the page's URL with the folder. With the decomposition, I don't have to change much, only need to change some parameters. This report will analyze the time complexity of each function on the module and evaluate the space complexity as well.

## 1.2 File structure:

### 1.2.1 File organization:

I use the folder named crawling_data to store all my data. In that directory, I will create multiple folder during my crawling process to store the information about each page. At first, I named my folder by their title, but then after realizing the title of each page not necessary be unique, I decided to name the folder by their visiting order.

In my *crawling_data* folder, along with the page's data, I also create a txt file named *crawling_url.txt* to store all page that has been gone through. In that file, every two lines will form one data set, with the first line will be the full URL and the second line being the folder containing that URL data. The *idf_data.txt* file is used to store the data about the idf value of each word, the first line of the dataset will be the word and the next line will be that word's idf value. I also create the file named *matrix_data.txt* to store the page rank, in this file I only store the value of the page on each line. How can I know which URL belongs to that value? Because I connect the URL with its visiting order, in this file, the value on line nth will also be the page rank value of page nth

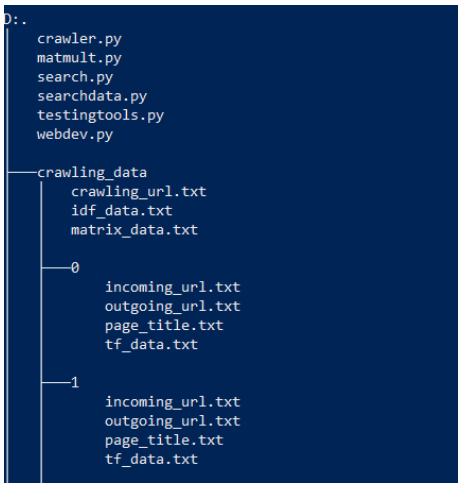


Figure 1: File organization

Each sub-folder in my crawling_data folder will contain four files, which are *incoming_url.txt, outgoing_url.txt, page_title.txt, tf_data.txt*. The data on each file is related relate to that file name. I store all the crucial data of the page into the file, so in the next module, I don't have to re-compute the data, which reduce the search run time.

There are pros and cons to this file organization. On the one hand, I don't have to create many files to store the data, only four files in each folder. On the other sides, I might have to read through

whole the file to extract the data. I fixed this problem by storing all the data into a global variable the first time the function is called. Therefore, another disadvantage of this approach is that it requires more RAM space. Besides this file organization, I also came up with the idea of storing each value in a separate file, so every time lookup for the word, I just have to open the file. But, the run time was fourth time higher compared to the current file organizations.

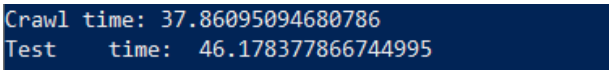


Figure 2: Another file organization

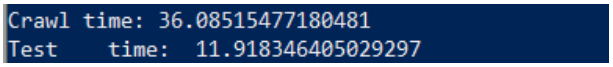


Figure 3: Current file organization

So, I have to accept this trade-off, sacrify the RAM storage for faster run time.

#### 1.2.2 Storage space complexity:

Let N represent the number of interlinked web pages.
Let M represent the number of crawled words
Because there are N URLs so there will be N directory contain the information about each URL. In each directory will contain five files to store data. The *incoming_url.txt* and *outgoing_url.txt* file will contain the information about the incoming and outgoing link to that page on each line. So the time complexity on each file in the worst case is O(N). The *page_title.txt* file only contain the title of that page so its complexity is O(1). The *tf_data.txt* will contain the tf value of the word in that page only, therefore the space complexity in the worst case will be O(M). The *idf_data.txt* will store the idf data and *crawling_url.txt* store all the interlinked paged and its data storage. The *matrix_data.txt* will contain the rank of all page.
In conclusion, the space complexity will be O(N*N*M) or O($N^2 * M$)

### 1.3 Code analysis:

#### 1.3.1 Global variables:

Because of using multiple small functions, I have used some global variable to make make it easier to work between each function. These global variables are:

- *queue*: A queue to store the URL.
- *data_dir_name*: A variable whose value is the name of the folder store all the crawling data.
- *url_checking*: A dictionary whose keys are url and the value is the corresponding folder stored all the data of that url.
- *incoming_url*: A dictionary whose keys are url and the value will be another dictionary contain all the url that link to the key url.
- *word_crawled*: A dictionary whose keys are the word and value is the total appearance of that during crawl process.
- *number_map_url*: A list whose element is a crawled url.
- *total_page*: A variable store the number of total crawled page.

Let N represents the number of interlinked pages. Let M represents the number of distinct word in the page. The space complexity of this module is O(N + M).

### 1.3.2 Queue implementation:

I use a queue to visit the page in the BFS order, which means visiting all the pages at the same depth before moving to the deeper level. I implement two function to serve for this purpose, which is *add* and *pop* function. These functions will work like their name, adding a new element to the queue to get the data. By using this implementation, it makes the process of tracing incoming and outgoing links easier. Because I can create the connection between the current node and all of its URLs while extracting the URL data without going backward.

- *add(queue, value)*: This function take queue and value as parameters, and add value into the queue. Time complexity of this function is O(1)
- *pop(queue)*: This function take queue as a parameter and returning the top element of the queue. Time complexity of this function is O(1)

### 1.3.3 File operating:

These functions below will work with the file and folder, helping my program to delete the old data, creating new file, writing new file in to the folder:

**directory_delete(dir_name, current_dir):** Using *dir_name* and *current_dir* as parameters. The purpose of this function is delete the *dir_name* directory in the *current_dir*, which is current directory. This function uses recursions because the file has a hierarchy structure recursion would be the easiest approach. Let N denote the number of file and folder in the *dir_name*. Time complexity will be O(N)

**data_reset(dir_name, current_dir), create_page_storage(folder):** These functions work like its name, the data_reset function in the worst case, which is delete the previous crawled data, is O(N) in which N is the number of file and folder, but during the crawling process it only consume O(1). Because the previous data have already deleted in the beginning to create the new folder to store the data, which is O(1).

**write_word_data(dictionary_word, total_word, folder):** Use dictionary_word, total_word, and folder as a parameter. Then using these values to calculate the term frequency value of the word write it to the *tf_data.txt* file as well as store these data to the *tf_data* variable. Each word and its tf value create a data set containing 2 lines in the file. Let N represent the number of words on that page. The time complexity of this function is O(N)

**write_url_data(url_arr, folder):** Write the outgoing link of the page, which contains in *url_arr parameter*, into a *outgoing_url.txt* file in *folder* directory. Each line of this file will contain the URL from the page. The time complexity of this function is O(N), in which N is the number of outgoing links on the page or the length of *url_arr*

**write_crawling_url_data(crawling_url_data):** Write all the crawled URLs into a file named *crwaling_url.txt* in *data_dir_name* folder, where we store the crawling data. I write the data by using the format: each data set contains a URL and the folder contain its data, the first line will be the URL, and the second will be its folder. Let N represent the crawled link, the time complexity of this function is O(N)

**write_incoming_url_data():** Iterating through each directory of the crawled link and write the data of incoming url lead to that page into the file named *incoming_url.txt*. Let N represent the number of interlinked page, M represent the biggest number of incoming link. The time complexity of this function will be O(N*M). In the worst case when every page has the link to the rest, the time complexity will be O($N^2$)

**write_idf_data():** Calculate, wrote the idf data of a word into a file named *idf_data.txt* and also stored that data into *idf_data* variable. There are multiple datasets in the *idf_data.txt* file, in which each dataset contain two lines the first one is the word and the second one is the idf value of that word. Let N represent the number of crawled words, the time complexity of this function is O(N)

**write_matrix_data(matrix):** The *matrix* parameter is a page rank vector. This function will write that data in the *matrix_data.txt* file in the *data_dir_name* folder. Each line contains the value of the page with the same index. Let N represent the number of crawled URLs, the time complexity of this function is O(N).

**write_page_title_data(title, folder):** Writing the *title* of a page into its corresponding *folder* by using the *page_title.txt* file. The time complexity of this function therefore is only O(1).

#### 1.3.4 Data operating:

**get_page_title(page_content):** Find the title of the page. Let N represent the number of lines in the HTML source, and M is the length of the line. In the worst case, the time complexity is O(N * M) when the title is at the end of the HTML source. But I think nobody would do that in practical, so the time complexity is O(M) which is the time complexity of the find built-in function.

**extracting_page_word(page_content, folder):** Using *page_content* and *folder* as parameters. Iterating through the *page_content*, finding the block contain the word between <p> and <\p>tag and also counting the frequency of of in that page. Moreover, marking the appearance of the word in the page through *word_crawled* dictionary. Writing the data to the file through *write_word_data* function.

Let N represent the number of line on the HTML source, M is the length of line. Time complexity is O(N*M), but because of this is a well-formatted HTML so each line will not to long, the time complexity in average is O(N).

**find_relative_url(url):** Using *url* as a parameter, which is a link. This function will check if this URL is a relative URL or absolute URL. If this is a relative URL return None, otherwise return the relative http://part of the url. Let N represents the length of the URL, time complexity is O(N).

**push_url_to_queue(url_arr):** Checking and adding all the URL in the url_arr to the queue. Then marking all the added URL to prevent multiple visiting. Let N represents the length of *url_arr*, time complexity will be O(N)

**get_matrix():** This function will calculate the rank of each page according to the algorithm in the course instruction, then call the function write_matrix_data to write the data. The first time, when implementing this function I initialized the base function as [1.0, 0.0, ...., 0.0] the page rank value is not precise compared to the test file, but then when I change the value of each element to 1.0 / N, in which N is the number of crawled link.

Let N is the number of crawled page, time complexity will be O($N^2$) which only involves the creating N*N matrix. I don't know exactly the time complexity of finding the page rank vector, but I assume that it is very small compared to $N^2$ in case N is big.

**incoming_url(url_arr, current_arr):** This function will create an connection between the *current_url* and all the outgoing url in the url_arr and store in the global variable *incoming_url_arr*.

In the beginning, I used list instead of using dictionary, thus my *in* operation will be O(N), N is the number of outgoing link on that page. But then I switch to use dictionary as a value so my *in* operation right now is O(1). Thus, making my function time complexity if O(N), instead of O($N^2$).

**extracting_page_url(page, current_url, folder):** *Page* is the HTML source of the page, *current_url* is the link of current page, *folder* is the name of the folder stored the page data. This function will extract all the URL in the page then call *incoming_url*, *write_url_data*, *push_url_to_queue* function to work and write these URL at the proper place. This function also fulfills the relative URL in the *current_url* by using the current URL. By doing this, the relative URL now becomes an absolute URL and repeats that process to its URLs. Because all the URLs are not relative, so we can not take the relative part of the first URL and assign it to the children's URL.

Let N represents the number of line on the HTML source, M is the length of the line. The time complexity will be O(N*M), but in average the time complexity is O(N), because the length of each line is not so big compared to the number of line.

**extracting_page_data(page, page_content, url, current_relative_url):** *Page* is a string of HTML source, *page_content* is the list of lines of HTML source, *url* is the link of the page and *current_relative_url* is the base URL that the current URL will base on if it is relative URL. This function is responsible for finding the title of the page, mapping the folder with the URL and vice versa. The mapping method is map the URL with it order of visiting. Moreover, it will call a smaller function to extract the URL data and word in the page.

Let N represents the total line of the page, M represents the length of the line. Time complexity will be O(N*M), but M <<N, so the time complexity will be O(N).

#### 1.3.5 Crawl process:

**crawl(seed):** At the beginning of my function I call the function *data_reset* to reset all the previous data of previous crawl, then add the *seed* to the queue for crawl process. The idea of crawl process is using the queue to visit the other links in the BFS order. While the queue is not empty, which

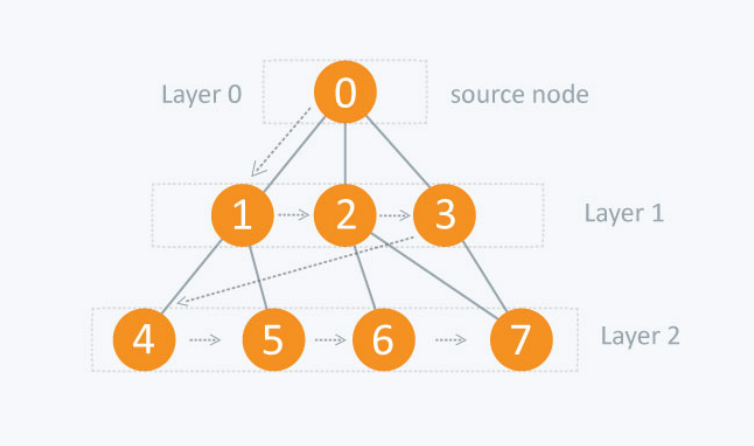


Figure 4: BFS visualization

means there are still other links that had not been visited, the program will go to that link, extract the HTML source, and get data from that source. During this process, it also check if the URL is an absolute URL or not and return the base URL. After visiting all the interlinked URL and getting enough information to process the page rank data, I will called the function *get_matrix()* to calculate the page rank. Furthermore, writing all of these data to the proper directory serving for later process.

Let N represents the number of interlinked URL, M represents the number of line on the page. The queue searching process take O(N) and during the search process, extracting the data from the URL source take O(M). Therefore, the time complexity of this function will be O(N * M).

# 2 Data required for Search (searchdata.py):

## 2.1 Overall design:

Because of having precomputed all the necessary data used for searching process. This module will mainly reading the data and then return the required data.

## 2.2 Code analysis:

### 2.2.1 Global variable:

I use several global variables to store all the data that has been precomputed so I don't have to go through each file over and over again every time call a function. These variables are:

- *data_dir_name:* A variable stores the name of the data directory.
- *url_map_folder:* A dictionary contains key as an URL and value data directory.

- *page_rank:* A list store the rank of each page.
- *idf_data*: A dictionary contains key as a word, and value as a idf value of that word.
- *tf_data:* A dictionary contains key as a directory of the URL and value as another dictionary store all the tf data of that page.

Let N represents the number of interlinked page. Let M represents the number of distinct word. The space complexity is O(N + M)

#### 2.2.2 Load data:

**get_mapping_data():** This function will open the file *crawling_url.txt* in the main directory to take and store the data of URL and its corresponding folder to the *url_map_folder* dictionary.
Let N represents the number of interlinked URL. The time complexity of this function is O(N).

**get_matrix_data():** This function will open the file *matrix_data.txt* in the main directory where store the rank of each page, then store it to the *page_rank* list.
Let N represents the number of crawled URL. The time complexity will be O(N).

**get_idf_data():** This function will open the file *idf_data.txt* file in the main directory to extract data and then store it to the *idf_data* dictionary.
Let N represents the number of word. Time complexity will be O(N).

**get_tf_data():** This function will iterating through all the folder and extract the tf data from the *tf_data.txt* file and store it to the *tf_data* dictionary.
Let N represents the number of folder, M is the number of word. The time complexity will be O(N*M).

**get_data():** This function will check whether all the data has been loaded to the dictionary for the searching process or not. If not, this function will call all above function to get the data for the searching process. Hence why, this function has to be called in the beginning of the every other search data function in order to get all the necessary information.
Let N represents the number of crawled word, M represents the number of crawled word. Time complexity will be O(N*M) in the worst case which is the time complexity of the function *get_tf_data()*. But after the first call, the time complexity will be O(1).

#### 2.2.3 Get search data:

**get_outgoing_incoming_data(directory, method):** Because of my data organization, all information about the incoming and outgoing URL is stored in the page's folder, thus the difference between the process of taking the URL is only based on the file. So I use this general function to parse through the file to take the URL data. This function will use *directory*, which is the page's storage, and *method* which is what information we are required right now.
Let N represent the number of urls. The time complexity will be O(N).

**get_outgoing_links(url) and get_incoming_links(url):** These two functions will return the list of incoming or outgoing links if the link appears during the crawled process. Otherwise, return *None*. These two functions will call the *get_outgoing_incoming_data()* function and pass the correspondence argument to the method. Thus, the time complexity of these two functions are the time complexity of the *get_outgoing_incoming_data()* function.

**get_page_rank(url), get_idf(word), get_tf(url, word) and get_tf_idf(url, word)** : Because of precomputing and loading all the necessary data to the dictionary, these functions will check if the URL is existed and return the correct value from the proper dictionary. The time complexity of these functions is O(1). Only in the first call of these functions, the time complexity would be O(N).

# 3 Search (search.py):

## 3.1 Overall design:

The main function of this module is *search(phrase, boost)* function. By using the cosine similarities between the phrase and the other document, we can suggest the top 10 most relevant URLs to the phrase. This function will use other functions from the *searchdata* module to get the required data.

## 3.2 Code Analysis:

### 3.2.1 Global variables:

Using two global dictionary named *url_map_title*, in which store the URL as a key and its corresponding title as a value, and *folder_map_url* to store the folder as a key and its corresponding URL as a value. Let N represents the number of interlinked page. The space complexity would be O(N).

### 3.2.2 Function

**phrase_tf_data(words):** This function will calculate the tf value of each word in a phrase by counting the frequency of each word and return the dictionary which contains the word as a key and that word tf data as a value.
Let N represents the number of the word in that phrase. Time complexity is O(N).

**phrase_idf_data(words):** This function will return the dictionary contains the word in the phrase and its idf data get by calling the *get_idf()* function from the *searchdata* module.
Let N represents the length of *words*. Time complexity is O(N).

**get_cosin_similarity(document_vector, phrase_vector):** This function will return the cosin similarity between the *document_vector* and *phrase_vector* based the formula:

$$\cos(\mathbf{t}, \mathbf{e}) = \frac{\mathbf{te}}{\|\mathbf{t}\|\|\mathbf{e}\|} = \frac{\sum_{i=1}^{n} \mathbf{t}_i \mathbf{e}_i}{\sqrt{\sum_{i=1}^{n} (\mathbf{t}_i)^2} \sqrt{\sum_{i=1}^{n} (\mathbf{e}_i)^2}} \tag{1}$$

The function will iterate through the vector, add the value, and compute the cosine similarity. Let N represents the length of the vector. Time complexity will be O(N).

**get_url_data():** This function will iterate through the *crawling_url.txt* to get all the interlinked URLs and their corresponding folders and store them in the *folder_map_url* dictionary. During that process, the function will open each folder to get the title of the URL from the *page_title.txt* file and store it at the *url_map_title* dictionary.
Let N represents the number of interlinked pages. The time complexity of this function will be O(N).

**search(phrase, boost):** This is the main function of this module. This function uses *phrase* which is the search phrase, and *boost*, which is the argument to determine whether to use the page rank or not. Firstly, the function will split the phrase into separate word, then get the *phrase_vector*, which contain the tf_idf value of each word in the phrase by using the above functions. Secondly, the function will iterate through all the interlinked URLs and get the vector containing the tf_idf value of the phrase word on that page, then get the cosine similarity between that and the phrase vector. Finally, multiply the URL rank if the *boost* is *True* then append the cosine similarity into the list for sorting. About the sorting process, I will repeat the process of finding the max then remove that from the *cosin_list*, instead of using the sort function. Because the time complexity of the sort function is O(N*log(N)) but only doing this will cost O(10*N), with N = 1000, the difference is not significant because log(1000) is approximately 10.

Let N represent the number of interlinked URL, M represents the number of distinct word in the phrase. Time complexity will be O(N*M)