COURSE PROJECT ANALYSIS REPORT

COMP 1405Z – FALL 2022

Aria Zhang

Crawler Module

Overall Design

The majority of data are processed in the crawl function and the outputs are stored in files to be read when necessary.

Regarding the set-up of my files, I decided to create small files in different directories to reduce the difficulties in reading files, as well as save my memory space. For example, the pagerank.txt, idf.txt, tf.txt, and tf_idf.txt, all have only one line in the files.

Global Variables

I created 3 global variables during the crawling process:

1) **mapping_dic & mapping_list:**
These two variables are to give every URL a unique id. The mapping_list is to record the sequence of URLs since the dictionary may not have a precise sequence of how URLs are added. And in the **mapping_dic**, the key is the URL, and the value is the sequence of the URL in the mapping_list:

```
Mapping_dic
{
url-0: 0
url-1: 1
…
}
```

Based on this, I created two functions: **get_num_from_url(url)** and **get_url_from_num(num)** so that when I need the URL when given a number or I need the number of a URL, I can easily get it from the **mapping_dic** and these two functions in O(1) time complexity. Besides, I use the unique id number of each URL as the name of its directory to store data, which will be discussed later.

I also store the **mapping_dic** in a file named **mapping.txt** to be used in the search module. I can read this file and then get the URL and number information in the crawler module, but it's more convenient to get information from a dictionary and a list, and it can save me the time of reading a file, so I decided to keep it.

2) unique_word

In the crawler module, I need to read the body content of each URL page and then save the content into each URL's directory. To generate the tf, idf, and tf_idf of all words that appeared in pages, I save the unrepeated unique words into a global **unique_word** list.

Crawl(seed) function design

Here are the steps in the crawl(seed) function:

1) **Reset existing data:**
   the first step is to delete all the data created before during the crawling process. In this step, I use a function named **delete_directory(dir_name)** which will recursively delete all the files and directories. This will have a time complexity of O(n), n represents the files and directory of the files in the dir_name.

2) **Add seed URL to the queue:**
   I append the seed into the **mapping_dic** and **mapping_list** that I mentioned above and append the seed to a list named **url_queue** which will be used to get the first URL in the list, parse it then remove it from the queue until the queue is empty. While **url_queue** is not empty, I need to continue parsing the first URL inside it using the **wevdev module.**

   The adding process has O(1) time complexity since it only appends 1 URL to the list, but overall it will have O(n) time complexity, n represents the number of URL found during the whole crawling process.

   The **mapping_dic** and **mapping_list** have O(n) space complexity. The space complexity of **url_queue** in the worst case is O(n).

3) **Find the URL:**
   Then I split each line of this page and then use the **find_url(page_line)** function to find if there are any URLs within lines with "a herf=" inside the page, then turned relative URLs into absolute ones and add them into the **outgoing_links.txt** of this url_queue[0] and also add this url_queue[0] into the **incoming_links.txt** of the URLs that are found in this page. I'll continue adding URLs that are not in the **mapping_dic** into the **url_queue** as well as **mapping_dic** and **mapping_list.**

   For every URL in the mapping_dic, I need to go through every line of its page content to find if there's a URL in it, so this step has a time complexity of O(mn) where m represents the lines in an HTML page content and n represents the number of URLs found during the crawling process.

4) **Find the body content:**
I use the unique number of each URL stored in the **mapping_dic** to create the directory of this URL. And then use the **find_body(page_content)** function to get the words inside this page and then save the body in the file within the directory.

This step has O(mn) time complexity where m represents the length of the 'page_content' HTML string and n represents the number of URLs found during the process. And it has O(n) space complexity by using the mapping_dic variable.

After finishing this while loop, I have already got the necessary information to do the calculations that are required in the searchdata module. So, the next step is to calculate.

5) **Get the page rank:**
I started calculating the page rank using function **get_page_rank().** Based on the above explanation, we now have **mapping_list**, **mapping_dic**, and the directory that is named after the number stored in the **mapping_dic** for each URL. The sequence is ordered perfectly, and this is helpful for me to create the adjacency matrix.

The first part of this function involves a nested for loop to create an adjacent matrix. I use the **outgoing.txt** that was created in the **crawler module** to initialize the adjacency matrix. In this part, we strictly follow the sequence in the mapping_dic to create each row and column of the matrix. For example, the first column should be the outgoing links values of the URL with the number 0 in the mapping_dic. the first row should be the incoming links values of the URL with the number 0 in the mapping_dic. This part has O(n^2) time complexity because of the nested for loop and O(n^2) space complexity because of the 2D adjacent matrix.

The second step is to do the math calculation to generate the probability matrix. This part also has O(n^2) time complexity because of a nested for loop and O(n^2) space complexity because of the 2D adjacent matrix.

The third step is to use the **mult_matrix** function and **Euclidean_dist** function imported from tutorial 4 to generate the page rank list. While the result of Euclidean_dist is less than 0.0001, the calculation ends. The result of the loop is a list that stores each page rank of the page.

[pagerank-0, pragerank-1, ...]

PageRank list

This step has O(1) time complexity because I think the calculation will repeat rather smaller times compared with the number of URLs. And the space complexity is O(n^2) because we use a 2D matrix to do the calculation.

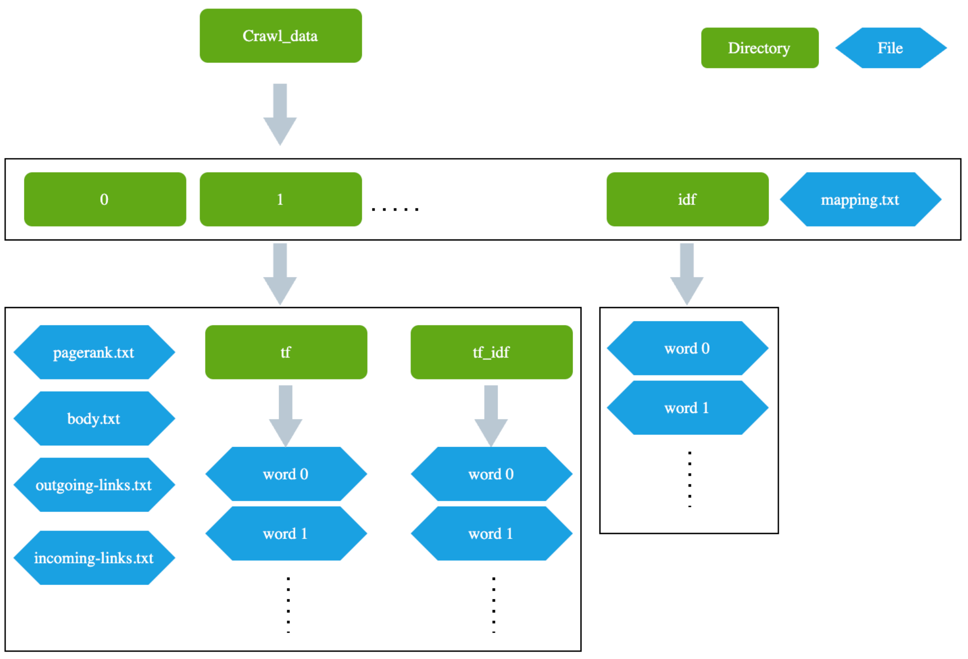
The final step is to find the page rank of each URL in the PageRank list and save it into separate files. The sequence of this list is aligned with our **mapping_dic**, **mapping_list,** and the directories' name. So, I stored each page rank of the specific URL in the **pagerank.txt** within its directory, the page Rank value will be stored on the first line. This step has O(n) time complexity for the length of the PageRank list being n and O(n) space complexity for the same reason.

6) **Calculate tf, idf, tf_idf:**
term_frequency() function is to calculate the term frequency of all URLs and words. For each URL in the **mapping_dic**, we can use the **get_num_from_url(url)** function to get its number and open its directory, then get the body content. For each unique word in our global variable **unique_word**, and for each **body.txt**, we can use the **binarycount()** function that was created in tutorial 5 to get the word's frequency and then store a single term frequency result into the file in the same directory with the **body.txt.** The time complexity of this function is O(mn) where m is the number of words in unique_word and n is the number of URLs. By using the mapping_list and unique_word, the space complexity is O(n).

calculate_idf(): similar to the term_frequency() function, for each unique word in our global variable **unique_word**, and for each **body.txt** where I stored the body contents, we see if the unique words are in the **body.txt** and calculate its idf. Then we store each idf for each word in a single file and store it in the idf directory. The time complexity and space complexity of this function is the same as term_frequency().

calculate_tf_idf(): this function is to open and read the term frequency file and idf file then calculate the tfidf of each URL and word then save a single result into the file named by the word in each URL's directory. The time complexity and space complexity of this function is the same as term_frequency().

File Structure



Summary

Function Name	Function Purpose	Runtime Complexity	Space Complexity
delete_directory(dir_name)	Reset existing data recursively	O(n) n: the number of files and directory in the dir_name directory	O(1)
create_file(dir_name,file_name,contents)	to avoid duplicate codes to create files	O(1)	O(1)
find_body(page_content)	To find the body when a page content is read and then save the body in the file within the directory of this URL	The function itself have O(m) time complexity but the whole step has O(mn) m: the length of 'page_content' HTML string n: the number of urls	O(n)
find_url(page_line)	To find the URL link when a page content is read and split into lines	The function itself have O(m) time complexity but the whole step has O(mn) m: the lines in a html page content n: the number of urls	O(n)
get_num_from_url(url)	Using the global variable **mapping_dic** to get the unique number of a URL	O(1)	O(n)
get_url_from_num(num)	Using the global variable **mapping_dic** and **mapping_list** to get the URL when I have the number of it	O(1)	O(n)
get_mapping()	To save the global variable **mapping_dic** to a file named **mapping.txt**	O(1)	O(n)
get_page_rank()	To calculate and store the page rank	$O(n^2)$	$O(n^2)$
term_frequency()	To calculate the term frequency of all URLs and words and then save a single term frequency result into the file named by the word in the url directory.	O(mn) m is the number of words in unique_word and n is the number of urls	O(n)

calculate_idf()	To calculate the idf of every unique word and save it into the file with name of each word in idf directory	O(mn) m is the number of words in unique_word and n is the number of urls	O(n)
calculate_tf_idf()	To open and read the term frequency file and idf file then calculate the tfidf of each url and word then save a single result into the file named by the word in each URL's directory	O(mn) m is the number of words in unique_word and n is the number of urls	O(n)
crawler(seed)	See above explanation	$O(n^2)$	$O(n^2)$

Searchdata Module

Overall Design

After the crawler module that finishes all the calculations and store data into files, the searchdata module is quite easy to proceed. All we need to do is open the corresponding directory and read the file and then get the number. Since all the file I stored in the crawler module is a small file with only one line in it, the open and read file and get the data process will have O(1) time complexity as well as O(1) space complexity.

Runtime Complexity & Space Complexity

Function Name	Runtime Complexity	Space Complexity
get_out_going_links(URL)	O(1) This function calls **get_num_from_url(url)** to get the unique number of the url and then open its directory to read the file. Since the time complexity of **get_num_from_url(url)** function is O(1), it will not have an impact on the time complexity of this function.	O(1)
get_incoming_links(URL)	O(1) Calls **get_num_from_url(url)** which is O(1)	O(1)
get_page_rank(URL)	O(1) Calls **get_num_from_url(url)** which is O(1)	O(1)
get_idf(word)	O(1)	O(1)
get_tf(URL, word)	O(1) Calls **get_num_from_url(url)** which is O(1)	O(1)
get_tf_idf(URL, word)	O(1) Calls **get_num_from_url(url)** which is O(1)	O(1)

Search Module

Overall Design

1) **Get the vector of tf_idf**:
First I turn the phrase into a list that each word as a single value and then turn the words into lower case, which has O(1) time complexity since the phrase will not be too long when people are searching on the internet.

For each word in the phrase, and for each url in the **mapping.txt**, we can get a tf_idf value by calling **get_tf_idf(URL, word)** function. So, I create a nested dictionary to store all the information. The dictionary will be like:

```
{
url-0: {word-0: tf_idf-0, wor-1: tf_idf-1, ...}
url-1: {word-0: tf_idf-0, wor-1: tf_idf-1, ...}
1...
}
```

Vector dictionary

This will take O(n) of time complexity as well as O(n) space complexity, n represents the number of the urls and m is the number of unique words in all the url pages. I also regard the number of words in the phrase as constant the same reason above.

2) **Get the tf_idf of the words in phrase**:
Then I calculate the tf of each word in the phrase using a simple math calculation which has O(1) time complexity also because the phrase will not be too long.

Next I get the idf of each word using **get_idf(word)** function which has O(1) time complexity based on the explanation in Searchdata Module.

Based on above, I can get a dictionary that store all the information:

```
{
Word0: tf_idf-0
Word1: tf_idf-1
...
}
```

query_vector dictionary

The space complexity is O(n) where n is the number of words in the dictionary .

3) **Get the cosine similarity**:
Then what we need to do is to get the information from the two dictionaries above and then do the math calculation. And if the boost is true, we use the **get_page_rank(url)** function which has O(1) time complexity to get the page rank and multiply it into the cosine similarity.

```
{
url-0: cosine_similarity-0
url-1: cosine_similarity-1
```

…
}
cosine similarity dictionary

This step has O(n) time complexity and O(n) space complexity where n represents the number of the URLs.

4) **Generate the top 10 results:**
Then we use python's built-in sort function to sort the cosine similarity dictionary which has O(n) time complexity. And then we revise the cosine similarity dictionary into the required result list and output the first 10 results which has O(n) time complexity and O(1) space complexity.

Overall, the search module has O(n) time complexity and O(n) space complexity.