Rae Chiang / Web Search Engine from Scratch Project

1. Description of how to install and run your system.

All the code is displayed in the Jupyter Notebook. The part1 notebook contains the code for the crawler function, and the crawled 5000 webpages are saved in the pickle file. The part2 notebook loads the crawled contents, provides a function to return the inverted index information and the function that given the query it will retrieve the relevant webpage based on the cosine score in ltc, ltn schema. Please refer to the comments in the notebook for detail instruction.

2. system development process

The outline of the development process is mainly following the order of the crawler development, the inverted index construction, query processing and finally, the evaluation.

A) The crawler development

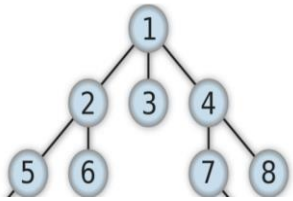


Figure (1)

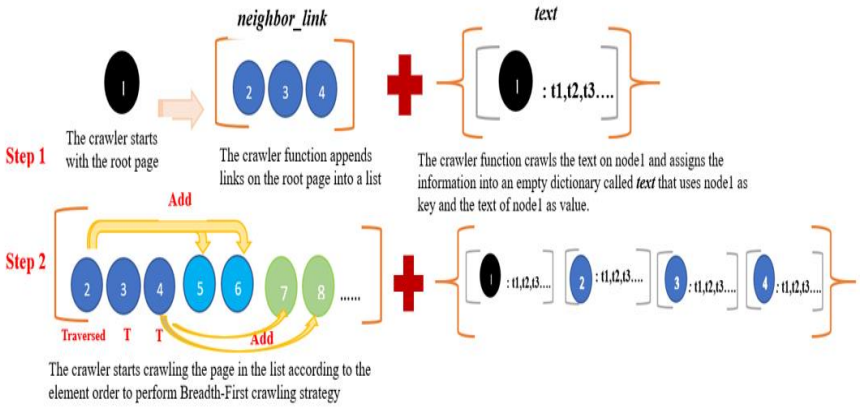


Figure (2)

The objective of the crawler is to use the Breadth-First strategy to crawl 5,000 pages. For explanation purpose, I will use figure (1) to give you a high-level explanation of the development process. To start, I first set the starting point as node1 with link "**https://www.depaul.edu**" to meet the requirement of the project.

The crawler will first collect all the links on node1, these links can be seen as node2, node3 and node 4 in figure (1) then according to figure (2), it will save them to an empty list called the ***neighbor_links***. At the same time, the crawler will also start crawling the text on node1 and assigns the information into an empty dictionary called ***text***, text={$node_i$, $text_{nodei}$}, that uses node1 as key and the text of node1 as value.

After node1 is traversed, the crawler will move to node2, using the same approach, it first collects links on node 2, that are node 5 and node 6 and appends them in ***neighbor_links*** ,the appends method adds node 5 and node 6 at the end of the list, then collect the text on node2 then assign the information into dictionary ***text***.

With this architecture in mind, after the ***neighbor_links*** and ***text*** are formed when crawler traversed the root node (node1), the crawler function can now start looping through all the element in ***neighbor_links*** to efficiently obtain links and text in the BF strategy since the element in the ***neighbor_links*** will be ordered in the correct format. The crawler function is named as *bfs_search* in my code file.

One thing to keep in mind is that before the crawler function moves on to the next node, it will particularly look into 2 aspects.

a) Has the node been traversed or not? If not, continue the process, else, move on to the next node.
b) The stopping criteria: whether the element in the ***neighbor_links*** has exceeded 5000 links or not? If yes, then the crawler will stop collecting links (stop appending nodes in the list) and just focus on crawling the text for the current node.

Before crawling the node, the crawler function will first use a *drink_soup* function to obtain all the tags in html file that used to describe the webpage, then used the *doctype* function to determine whether the node is in "html" format, if the node is in "html" format then the crawler will move on to the crawling process we mentioned above, else it will skip the node and move on to the next node in the ***neighbor_links*** list.

B) Inverted index construction and Query processing

The function *process_term* is defined for terms processing purpose, that is, providing a list of terms the function will tokenize the term, remove punctuations, lowercase the terms and apply porter stemmer to the terms. However, the function will not remove "-" if it exists between two terms, it will treat these terms and "-" as one term and return them in a string. For example, the term "second-hand" will be returned, I used this method to keep the obvious bi-gram terms.

After the collected terms are processed, we can use them to construct the Inverted index by learning the total term frequency, document frequency of the term and creating the posting list.

The 20 queries I came up also need to be processed with the *process_term* function, then the "Inverted Index Retrieval Algorithm" with the ***ltc*** (for document) ***ltn*** (for query) schema are applied for term weighting.

C) Results and Evaluation

Finally, I Computed the precision at10 based on my judgment and also compare results with Google's performance on the same queries.

Query1	data science program	Query11	professor noriko tomuro research area
Query2	lincoln park campus	Query12	fun and free event in spring quarter
Query3	depaul safety	Query13	campus libraries
Query4	student future career path	Query14	most popular student organization
Query5	how many international students in depaul	Query15	sports for spring
Query6	student discount	Query16	most popular program
Query7	student insurance plan	Query17	cdm ranking
Query8	restaurant recommendation near campus	Query18	depaul mascot blue demons
Query9	depaul ranking in the world	Query19	depaul famous alumni
Query10	salary for cdm student	Query20	who is depaul university founder

Precision	Query1	Query2	Query3	Query4	Query5	Query6	Query7	Query8	Query9	Query10
My system	0.1	0.4	1	0.3	0.2	0.9	0.2	0.1	0.6	0
Google	0.8	0.8	1	0.8	0	1	1	0.7	0.9	0
Precision	Query11	Query12	Query13	Query14	Query15	Query16	Query17	Query18	Query19	Query20
My system	0	0.7	0.6	0	0.7	0	0	0	0.4	0.1
Google	0.8	0.7	1	0.8	1	0.7	0.2	0.9	0.3	0.5

According to the experiment, we can learn that the retrieved links from Google are more relevant compared to the retrieved links returned by my system. That's mainly because my system is a naïve system that only focusing on term matching. When the key concept term of the query happens to have a high frequency in the web page ("safety" in query 3), the result will tend to be great, therefore, the vector space model seems to work well when the query is short and concise.

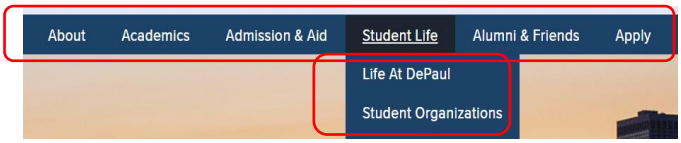
On the other hand, when the query is asking for some information that is not prominent on the internet, such as query 5 and 10, the results are bad in both Google and my system.

3. Your comments and reflection on the project overall

One of the challenges I ran into is the run time, when scraping the web page I first tried to use the "html.parser" in the beautifulsoup python module, after reading some threads on the forum[1][2], I decided to try the "lxml" parser. From what I read, the "lxml" parser will be faster, other than that there is no difference between the parser if the given document is perfectly-form in HTML format[3]. Applying different parsers with the same function I defined above, the bfs_search, when conducting web scraping for 10 web pages to collect the links and corresponding text on these pages, the "lxml" is able to be faster for more than 5 minutes. In future work, I will also read more about the multiprocessing [4] and apply it to optimize the scraping process.

The other interesting part of the development process is the hyperlink normalization. The crawled links might be the partial links, for example, the crawled link might look like this *questions/120951/how-can-i-normalize-a-url-in-python* therefore, in the *obtain_links* function I defined, it will join the partial link with the link it crawled it from, that is, https://stackoverflow.com/ + questions/120951/how-can-i-normalize-a-url-in-python. Also, some links might be formalized differently but they are actually leading to the same page, for example, *https://www.depaul.edu:443/Pages/default.aspx*, *http://www.depaul.edu/Pages/default.aspx*, *and https://depaul.edu/* all point to the same page, therefore, I defined a function called the *redirect_url* to find the final destination of every URL before I start crawling to make sure I don't craw duplicate pages[5]. On the other hand, when start crawling the pages, if the page cannot be accessed, I also have to determine whether that is because the link is no longer exist or because of the protection, then try to access the page if it is protected [6].

Additionally, since we are applying the vector space model for this project, basically the documents that have more mapping terms to the query will be retrieved, therefore I tried to get rid of some unnecessary contents that are common in all the pages, and mainly keeping the text in titles and body paragraph. For example, the menu text of each page and their corresponding contents will not be collected during scraping process. The code can be referred to the *obtain_HTML_text* function in my code file.



Future work of this project, I would like to do more research on how to return more accurate link when given a query. I will continue to do more experiments on adjusting term weights to put more emphasis on title text, and read some materials about page rank to apply it into my work. Overall, I enjoy doing this project, because I learn more about web crawling, which I have never encounter before.

[1] python: difference between 'lxml' and “html.parser” and “html5lib” with beautiful soup?https://stackoverflow.com/questions/45494505/python-difference-between-lxml-and-html-parser-and-html5lib-with-beautifu
[2] Speeding up beautifulsoup https://stackoverflow.com/questions/25539330/speeding-up-beautifulsoup
[3] Beautiful Soup Documentation
https://www.crummy.com/software/BeautifulSoup/bs4/doc/#specifying-the-parser-to-use
[4] How to speed up your python web scraper by using multiprocessing
https://medium.com/python-pandemonium/how-to-speed-up-your-python-web-scraper-by-using-multiprocessing-f2f4ef838686
[5] how to get redirect url using python requests [duplicate]
https://stackoverflow.com/questions/36070821/how-to-get-redirect-url-using-python-requests
[6] Unable to scrap a Website using beautifulSoup
https://stackoverflow.com/questions/50501203/unable-to-scrap-a-website-using-beautifulsoup