Web Search Engine - UIC

Search Engine for UIC domain using TF-IDF & Page Rank

Vikram Abhishek Sah
Department of Computer Science
University of Illinois at Chicago
Chicago, Illinois, USA
vsah4@uic.edu

ABSTRACT

This report is a part of the term project for CS494: Information and Web Retrieval course at the University of Illinois at Chicago during the Fall 2021 term.

The goal of this project is to design and implement a search engine for the UIC domain that includes components for Web crawling, webpage processing, indexing, and Retrieval.

The search engine includes a multi-threading enabled web crawler, preprocessing, and indexing of web pages, a ranking methodology combining the cosine similarity and the PageRank algorithm, and an IR system implementing a vector-space model to retrieve webpages relevant to an input user query.

The implementation uses multi-threading to enable the crawling and queuing in an asynchronous way. The crawler crawls the pages and adds the relevant out links in the queue, which are then picked by the next thread in the concurrent executions. The pages are then indexed in an inverted index and a web graph of the pages is created. The resulting web graph is then used by the PageRank component to build a ranked list of the pages. Then, the TF-IDF score for all the words is calculated and added to the inverted index. The document lengths are also computed.

Once the user enters the query a combined score of the Cosine Similarity and the PageRank score is calculated, depending on which an ordered list is presented to the user.

KEYWORDS

Web Search Engine, Information Retrieval, Page Rank, Inverted Index, Crawling, TF-IDF, UIC Search Engine

ACM Reference format:

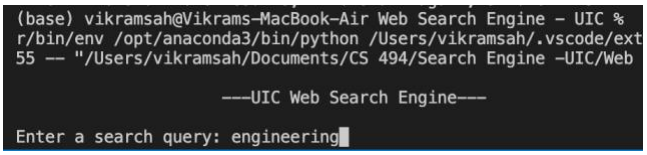
Vikram Abhishek Sah. 2021. Web Search Engine - UIC: Search Engine for UIC domain using TF-IDF and PageRank.

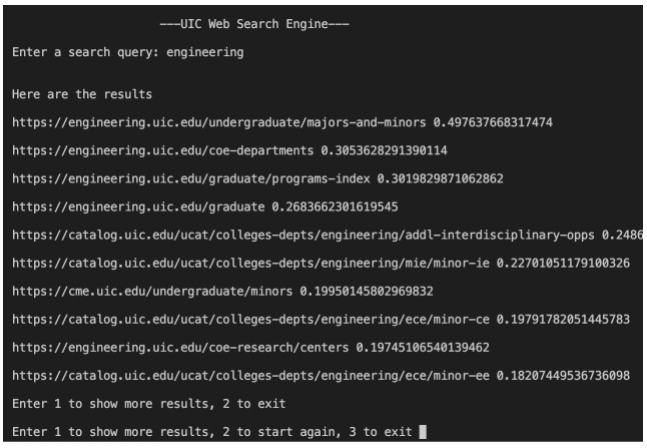
SOFTWARE DESCRIPTION

1.1 Introduction

The software code is written in Python3 adhering the OOPs design principles to allow easy testing, debugging and future extensibility. The dataset of the 4000 web pages crawled and read by the crawler are stored locally in the folder "**uic/webpages**". All these locally stored pages belong to the **uic.edu** domain. The path is configurable depending on the domain for which the crawler is to be run. Further, a text document containing the list of all the pages crawled by the crawler is maintained under "**uic**" folder with the name "**crawled_links**". This is to get a bird's eye view of all the pages crawled.

Upon opening the repository, the **search_engine.py** can be run instantly to see it in action. The queries can be entered as an input inside the terminal and the list of results is displayed on the terminal instantly.





For a more exhaustive search, the **crawling_main.py** and the **preprocessing_main.py** can be run again for a greater number of web pages. The number of web pages to consider can be easily configurable in **config.py.** These python scripts are explained in more detail in the subsequent sections.

1.2 Initial Configuration

The initial configurable parameters can be set in the **config.py** file. The parameters like the domain name, starting page, number of concurrent threads, PageRank max iteration and the number of pages to be crawled can be set in this file.

1.3 Multi-Threaded Crawling

The web crawling can be started with the script **crawling_main.py**. Designing the crawler was challenging, in terms of how the queuing and crawling to be performed in sync. A total of 20 worker threads accepts jobs from a queue implemented using the python's Queue module. The first thread starts crawling from the CS department homepage *https://www.cs.uic.edu/*, which is also configurable from the config file. The crawler implements a breadth first search strategy. Every page is dequeued, downloaded and parsed using the HTMLParser of the urllib.request library. The "**content type**" in the page header is checked explicitly to be of the type 'text/html'. The pages are saved and all the out links from the current page is extracted and added back to the queue.

After the first crawl of over 10000 pages a block list was created which prevented the crawling of the pages ending with the below format types.

(".doc", ".docx", ".avi", ".mp4", ".jpg", ".jpeg", ".png", ".gif", ".pdf",".gz", ".rar", ".tar", ".tgz", ".zip", ".exe", ".js", ".css", ".ppt", ".pptx", ".mov")

Before adding the out links back into the queue, the urls are modified. The urls starting with *http* was converted to *https*, the query strings are deleted, the ending slashes, hash symbols and in-page links are removed. Moreover, further manual scans were done to identify the instances where redundancy in crawling was identified and eliminated.

During the crawling, two dictionaries, **url_from_file** and **file_from_url** was created and saved on the disk as a pickle file, to be used later at the time of web graph creation and the retrieval of relevant links.

It took less than 7 mins to crawl and save 4000 pages, due to the multi-threading approach.

1.4 Preprocessing

The preprocessing can be started from **preprocessing_main.py** Script. Parameters like **page_rank_max_iter,** and **page_count** can be modified from the config file.

The preprocessing is carried out by going through each saved page one by one and extracting the text by parsing it using the HTMLParser of the **selectolax** library built using the Modest and Lexbor engines, known for its speed of execution. The parser takes all the plain text except for the text within the JavaScript tags <*scripts*> and css tags <*style*>. The text is then tokenized and stemmed using the porter stemmer by nltk. The basic preprocessing techniques like removal of special characters, punctuations, digits and stop words removal is carried out. Further only words having length greater than 2 are considered.

The tokenized words are then stored in an inverted index. Using the TF-IDF weighing scheme the TF-IDF values were calculated and added to the inverted index. The document length was also calculated for each document to be used later for similarity calculation. The data objects like page ranks, inverted index, document lengths, and tokens were saved as pickle files to be used later. The precomputation of these data dictionaries enabled the retrieval system to be so efficient as none of these expensive calculations are required to be done at the time of query result retrieval.

From the extracted out-links within the pages a directed web graph was created and was fed to the Page Rank implementation.

For preprocessing 4000 pages a total of 104 seconds was taken out of which approx. 2 seconds was the time required for Page Rank convergence.

1.5 Search Module

The search engine can be fired by running the "**main.py**" script. By default, the search engine fetches 50 documents and 10 results per page. These parameters can be configured by altering the global declaration of **RESULTS_PER_PAGE** and **MAX_RESULTS_TO_CONSIDER.**

The tokenizer and TF-IDF ranker is declared globally to process the query. The pickle files of the data objects **url_from_file, file_from_url, inverted index, document length and page rank** are loaded. Once the query is fired a ranked list of most relevant documents is fetched which considers a combination of the scores of the cosine similarity and the page rank ranking.

The weightage of the PageRank rankings can be altered my changing the value of the global variable PAGE_RANK_MULTIPLIER. By default, it is set to 10.
The script upon running asks the user for input in the command line and presents users with the list of documents and multiple options to interact with the search engine.

MAIN CHALLENGES

The most challenging part was designing the crawler as I was new to the asynchronous coding practices and multi-threading. Implementing the interaction between the worker threads and the queue was meticulous.

Further the crawler script had to be run several times to identify the repetition and redundancies in the selection of out links. Each of these runs was very time consuming. And I had to manually scan the links which were processed to identify the techniques of improvement.

I struggled to select the ranking methodology to go with. The cosine similarity could have sufficed but I wanted to experiment with page rank and web graph. Due to this I feel that a lot can be done in terms of tweaking the weightage of both these ranking schemes.

Implementing the page rank and designing the component to handshake properly with the tf-idf ranker required a lot of code changes and foresight.

MEASURES AND WEIGHTING SCHEMES

3.1 Weighting Scheme

A TF-IDF weighting scheme was used to its popularity and it also taking into account the frequency of words in the overall collection. The other weighing schemes like simple term frequency, logarithmic term frequency and Boolean term frequency can be used. But given the nature and density of the retrieval system the TF-IDF gives the correct weightage to the words in the collection. And the results seemed to justify the same.

3.2 Similarity Measure

The similarity measure used was cosine similarity. Along with which inner product similarity was also used. The inner product was later divided with the document lengths to produce the cosine similarity which yielded better results. Considering the document and the query lengths certainly provides better results and eliminates the bias caused due to certain words repeating way more than often.

Integrating page rank with the similarity measure was intuitive, where initially a simple sum of cosine similarity and the page ranks was tried. For a simple sum the effect of the page rank was nullified. And a for a very high page rank weightage gave unnecessary weightage to the homepages and the authority pages. So, tweaking with the values and analyzing the results was important.

EVALUATION

An evaluation was done on top 10 results for 5 different queries.
The query set was kept diverse to gauge the performance of the system on different types of entities.

- Query – “Computer Science”

The first link was *https://cs.uic.edu/undergraduate/majors* and the subsequent links in the result also belonged to the cs department page and the cs page in the engineering department website. All the results seemed relevant. So, resulting precision is **p =1**

- Query – “Cornelia Caragea”

The first link result was *https://cs.uic.edu/profiles/cornelia-caragea* and the remaining links was pointing to faculty and instructor pages somehow related to the search query. The precision would be p= 1

- Query – “internships and jobs”

The first link retrieved was *https://careerservices.uic.edu/students/internships*. Of the remaining pages 6 were relevant and remaining 4 were not. As a result. The precision would be 0.6.

- Query – “research assistantship”

The first link retrieved was *https://cs.uic.edu/graduate/admissions/financial-aid-and-funding*. Of the remaining links 7 links were relevant on a stingy examination. The precision would be 0.7

- Query – “Admissions Department Address”

Of the links retrieved initially the first link was *https://disabilityresources.uic.edu* and only two of the links retrieved had the address. But upon decreasing the page rank weightage to 3 from 10, there were 5 links which had the address and almost all the remaining pages had at least one outlink pointed to the admissions department website which had the address.

DISCUSSION

For a number of queries which had little to no relevant documents, upon manual analysis, it was found that the said website had not been indexed by the crawler. As a result the overall performance of the system can significantly be improved by crawling more

pages. A lot of other improvements can be done on the lines of accepting words like “StudentCenterEast” or misspelt words like “career fare”. A pseudo relevance feedback or query expansion type of synthesis to recompute the scores, can allow us to identify relevant pages which contain the words similar to what is in the query expression but not exactly the same.
The Page Rank clearly improved the overall search quality and enabled us to attach a bias to our normal cosine similarity-based search and alter the tradeoff between preferring authoritative pages and syntactically similar pages.

RELATED WORK

Page rank and Web IR Models have been researched extensively in the past. Researchers have always tried to improve page rank by trying to eliminate the effect due to highly authoritative but less relevant pages using techniques like query optimization[1] (Roul et al., 2021). Some research is also on lines of continual learning in IR where the systems makes a query log[2] (Sharma et al., 2020) and the stores the user clicks against the retrieved queries. This log is used to refine query searches.

Some research on IR systems exploiting other ways of referencing pages. Like the citation network in research publications being used to improve the retrieval scores[3](Caragea et al., 2014).

FUTURE WORK

Some machine learning techniques can be used along with word embeddings in IR systems. Having some kind of query relevance feedback to the final Page Rank list could also be very helpful.

A few ideas novel ideas which come to my mind includes *taking the web page topology into account such that the ranking methodology indexes the subpages within a homepage*. Also, creating *a topic taxonomy of the entire web* can help create new areas of research possibilities in IR.

REFERENCES

[1] Rajendra Kumar Roul, Jajati Keshari Sahoo:
A novel approach for ranking web documents based on query-optimized personalized pagerank. Int. J. Data Sci. Anal. 11(1): 37-55 (2021)

[2] Prem Sagar Sharma, Divakar Yadav*:*
Incremental Refinement of Page Ranking of Web Pages. Int. J. Inf. Retr. Res. 10(3)*: 57-73 (2020)*

[3] Cornelia Caragea, Florin Bulgarov, Andreea Godea, and Sujatha Das Gollapalli. "*Citation-Enhanced Keyphrase Extraction from Research Papers: A Supervised Approach*." In: Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP 2014), Doha, Qatar, 2014.