Search Engine - Crawling UIC Domain and Query Search

Tejas Rajopadhye
Department of Computer Science
University of Illinois, Chicago
Chicago, IL, USA
trajop2@uic.edu

I. Abstract

This project report outlines the specification of the Search Engine made as a final project for CS 582 - Information Retrieval. The goal of this project is to design a Web Search Engine that will consist of roughly three stages - web crawler, preprocessing and indexing results based on a weighing scheme. The web crawler will start from a UIC domain link and will crawl web pages. Once the webpage is captured it will be stored. This webpage will then be preprocessed by the processing system developed during Homework 1 and the results of this will be weighted according to TF-IDF weighing scheme. This TF-IDF data along with similarity measures like Cosine Similarity will then be used to provide results to the user about the query that the user provides.

II. Software Architecture

This project makes use of Python 3.9 and couple of other libraries for crawling, text processing, etc. Python was used for this project because of its extensive support for various libraries. Also as python is interpreted, each module can be easily tested and integrated. Being dynamically/weakly typed it makes it to write description of what to do and it does it by automatically converting types and thus being fault tolerant in most cases. Apart from that, python has a large community of developer and forums which make it easy to work with python.

This project contains three stages while building a search engine. The first stage is the crawler. The crawler makes use ”requests” library from the python modules. The request module helps to fetch the desired webpage. Also while crawling the requests object doesn't contain the user-agent and thus automatically converts itself into a spider which is crawling web. The advantage with this approach is that the developer doesn't have to manually check the robots.txt file in order to determine if the webpage/domain can be accessed. If the crawler is not allowed then it is not provided the webpage. More description on the crawler is provided in the next sections.

Once the web crawling is done then and webpages are collected then the search engine has to be started. Search engine code is located in the 'searchRetrieval.py' file. From the folder where the webpages are stored, each webpage is iterated and the contents of the webpage are parsed and an index is formed. Here ”BeautifulSoup” python module is used for processing HTML documents.

Once this phase is over comes the query and retrieval phase; in this phase the user is asked to provide the string to search in the crawled webpages. This query is then passed into the text processing stage and then the cosine similarity is used to find out relevant pages.

III. Setting up the project

This project is written in Python and requires the user to have python installed on their computers.

1) First Create Folder called ”RetrievedDocs” inside the root directory
2) Install the dependencies required for this project. Please type in 'pip install requests', 'pip install BeautifulSoup', 'pip install numpy', 'pip install nltk'. If prompted while running the searchRetrivel to download the stemmer files please install and download them accordingly.
3) In order to start scraping, please run the crawler.py file by issuing command 'python crawler.py'.
4) For directly working on search queries please type 'python searchRetrival ¡path_to_RetrievedDocs Directory¿'

Note - A list of documents containing crawled pages and all the necessary mapping required to bootstrap this project to searchRetrieval file is provided with the project submission.

IV. Crawling with BFS

In order to crawl the web, the crawler makes use of BFS (Breadth First Search) strategy to crawl the web.

Before starting the crawler, the code contains some globals: the first is the BASE_URL which contains the URL from which the crawler will start crawling. The base URL is set to ”https://cs.uic.edu” in this project. In order to start and crawl the different base URL within the UIC code please change this value. Please note that the crawling in this project is always restricted to the ”uic.edu” domain irrespective of base URL. This web crawler currently tracks / crawls 3000 pages which is defined in the ”PAGES_TO_CRAWL” variable. Please modify this variable to crawl more webpages.

While Crawling webpages, some webpages point to documents like .pdf, and .docx and parsing this is a different challenge. In order to ignore these resources which are

returned while crawling webpages they are added in the ”IGNORE_LIST”. The ignore list consists of ”.pdf”, ”.doc”, ”.docx”, ”.ppt”, ”.pptx”, ”.xls”, ”.xlsx”, ”.css”, ”.js”, ”.aspx”, ”.png”, ”.jpg”, ”.jpeg”, ”.gif”, ”.svg”, ”.ico”, ”.mp4”, ”.avi”, ”.tar”, ”.gz”, ”.tgz”, ”.zip”. Please feel free to add any other file which should be ignore if you happen to use this project.All the files which are crawled are store into a folder called ”RetrievedDocs” which is located in the root folder of the project.

When the crawling starts the BFS Queue is initialized with the BASE_URL which was declared in the global variables. The function which crawls is called ”crawlBFS” and is located in the crawler.py file. With this BFSQueue, during each iteration a webpage URL is popped out and that webpage is retrieved. This is made possible with the ”requests” module. To check if the webpage was retrieved or not the status code is checked. If the status code is between 200 to 300 range (excluding 300) then the webpage is successfully retrieved. This webpage is then retrieved and passed to the BeautifulSoup module which parses the HTML tags and fetches all the text excluding the SGML/XML tags associated with them. The reason that HTML tags are removed is that they don’t provide any relevant information about the content of the document. Once the text is extracted it is stored into the ”RetrievedDocs” folder with the unique integer. This unique is then mapped with the URL of the page. This is made available in the ”docToURL” dictionary. All this information about crawled webpages and docToURL mapping is all stored in the root directory of the webpage. This information is used while text processing and searching for text inside the documents gathered from the URLs. More on this in the next section.

While crawling, URL links which are having secure connection are crawled ”https”. Also, if the URL has relative URL then it is converted to the absolute URL with a domain attached to it. Apart from that each URL is converted to lower case to make the URL text uniform. At the end only those URL are added which are not visited and are not present in the BFSQueue frontier.

In order to start crawling, please type ”python crawler.py”

V. Text Processing and TF-IDF

Once the HTML documents are parsed and collected in the ”RetrievedDocs” folder they are passed through the text processing stage. The text processing stage consists of removing excessive white space, splitting the document and forming tokens. These tokens are then used to find the similarities. Also, punctuations and digits are removed from the documents as these text values do not provide any meaningful data. Also, words that are connected by a ”-” are also not separated as they provide a special meaning. Then the stop words are removed . Here nltk’s stop word collection is used to remove the words which are present in the list. Once the stop words are removed they are passed through the stemmer which uses a porter stemmer from the nltk module to stem the words. Thus now tokens are into their base format. After stemming words containing 1 or 2 characters are removed and then the final data set is loaded onto a dictionary. This dictionary is then used for further processing like TF-IDF.

In order to calculate the TF-IDF, the term frequency has to be calculated. This is the next stage in the processing. Once the TF is calculated and an inverted index is constructed, this is used to calculate the TF-IDF score. The values are TF-IDF are then stored.

In order to start the search engine and start searching type ”python searchRetrieval.py”

VI. Query Processing and Similarity

In order to search and get URLs that are relevant to the search text, initially the user is asked to enter the search string. This search string is then passed through the text processing function in-which just like the previous stages it tokenizes and then performs string operations to clear out unimportant items like stop words, etc. After this, the tokenized text is passed through the TF-IDF calculation unit which then measures the weight of each token term. This is required to calculate the similarity between the docs and the query. The next step is the one that finds the documents that are relevant to the query string. This happens by calculating the Cosine-Similarity between the documents. In order to calculate this each query is compared with the words in the query and the words in the document measured by the TF-IDF given by the cosine similarity formula.

Once this process is done, an index containing the query and all the documents fetched is retrieved. These retrieved docs and their corresponding similarity score are then sorted in descending sort. Initially, the user is shown the top 10 queries from the search results based on the query results. Next, the user is prompted if more results are to be displayed OR a new query is to be searched.

VII. Main Challenges Encountered

- While crawling, it was very difficult to detect the robots.txt file and only crawl the webpages which are allowed. In order to solve the problems requests module provides a feature that doesn’t specify the User-agent which is supplied by the browser to identify that the user is an actual user and not a crawler. By not attaching this field a lot of code and parsing checks are avoided as if the crawler is not allowed it is given a 4xx response for such websites.
- When crawling a webpage, it can happen that often a link points to a resource which contains .pdf, .docx, .mp4. When I started the crawling process some of the pdfs were getting loaded and the size of these pdfs was very high. Apart from that some files like .mp4 were not opening with the HTML parsers which were creating a lot of exceptions. Also, no important information was getting collected in this approach. Thus, I decided to include these in my ignore list. One major issue was creating a holistic list that contains all the extension formats so that they can be ignored. For this I made use of web search and some websites which had this list readily available.

- Creating the BFS Frontier, it was so happening that for the relative URL the same URL was getting appended by the relative URL (e.g. uic.edu/about was getting appended as uic.edu/about/about) and this was creating problems in which this was not getting detected as an old visited page and was considered as a new URL. In order to make things right, the BFS updating strategy had to be changed. The new logic changed the way relative URL where appended to the Domain URL and while creating the new URL it was checked if was visited or was in the BFS Frontier.
- During the process of parsing and tokenizing web pages, as the documents were around 3000 it was taking time to process those pages and parsing the HTML contents and fetching the text took a long time. In order to solve this problem an efficient parsing scheme "lxml" was used.
- Some webpages were encoded in different format and it was creating exceptions when trying to read and write it to the file. In order to solve this problems I tried removing characters which are causing the problem. But this didn't work and scale well as there were a lot of combinations of characters I was missing. Thus, I made encoded by file writer to utf-8 which takes care of making sure the file is in one format and can be later read by my parser in the search retrieval part.
- One of the major issues with crawling multiple webpages and inspecting the code was that during runtime when the code failed and knowing where it failed was a big problem. In order to solve the problem although using efficient use of logging is the basic strategy, I made use of the information state that was captured before the crash happened. In order to solve this, I made use of logging each and every state variable i.e. number of pages visited, visited pages and all dictionaries. For dictionaries I converted into JSON data and stored them so that they can be retrieved. Directly storing didn't work as python is not able to load a dictionary that is in the text format easily.

VIII. Weighting scheme and Similarity measure

A. *Weighting Scheme*

For this project, I used TF-IDF weighing scheme. The TF-IDF score computes scores in which the frequently but rarely over terms are given more weight. This gives a good measure of the weight of the term considering its frequency but also considers the occurrence of that term in other documents. Also, TF-IDF is simple to work with. Apart from the TF-IDF has been found to work well experimentally and it gave good results while working in Homework 2. Apart from that with the help of inverted index it can be computed in polynomial time and can be a very precise and concise measure of the token in the entire document.

B. *Similarity Measure*

Cosine Similarity measure will be used in this project. Cosine Similarity provides measure of similarity between associated terms which are weighted using some scheme like TF-IDF. Cosine similarity works well as the query or the document is considered vectors in an n-dimensional space and Cosine similarity provides the similarity measure of projection of one vector on the other in an n-dimensional space. The n-dimensional vector is formed by using some weighting scheme like TF-IDF in this project. For this project, the n-dimensional vectors will be calculated by parsing HTML documents and processing them (to remove stop-words, punctuations, using stemmer porter, etc.) and then weighted with the TF-IDF scheme (di). Also, the queries are similarly processed and then the resulting vector is produced (dj). The similarity score is then calculated with cosine similarity formula.

$$\text{Cosine Similarity} = \frac{\sum d_i * d_j}{\sqrt{\sum d_i^2 * \sum d_j^2}}$$

C. *Other Weighting and Similarity measures*

For the weighing scheme, as the process is unsupervised some other weighting schemes which could be used are probabilistic idf, which considers the probability of df measure in the documents, weighted inverse document Frequency (widf) which uses inverse of term frequency measure for the idf component. These idf based components can be clubbed with tf of the term to give another resulting TF-IDF. For the similarity measure, Inner product, Dice coefficient, Jaccard coefficient can be used to give results instead of Cosine Similarity. Inner product is naïve similarity measure and other measure can be computationally intensive as compared to Cosine Similarity.

IX. Manual Query Evaluation

The following section goes over evaluating the results obtained by the search engine. To get the results, the web pages were crawled around 18th November 2022.

1. Searching for Query String - Cornelia Caragea

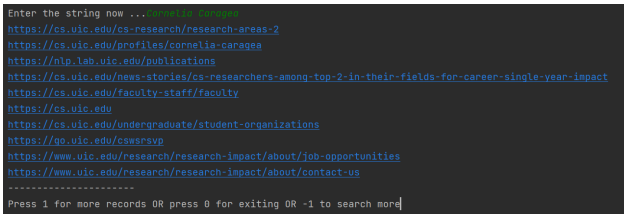

Fig. 1. Query - Cornelia Caragea

In this query, the search engine provides results which have similarities to the search text. For this query, documents with timestamp and containing search text are provided in that order. Here, the top 5 documents signify the results that were expected. The precision for this query can be estimated to be 0.5

2. Searching for Query String - Natural Language Processing

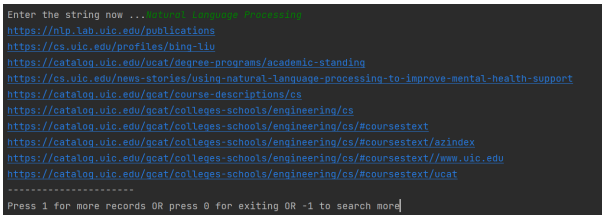

Fig. 2. Query - Natural Language Processing

For the second query, the results provided are news articles and profiles, but this time the searching doesn't actually provide the required documents. This can be improved if more documents are crawled. I would give this a precision of 0.3.

3. Searching for Query String - Internships

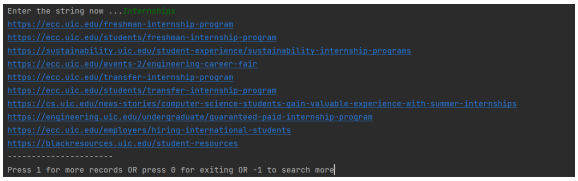

Fig. 3. Query - Internships

During this search, all the documents seem to have been correctly fetched. The query string and the documents fetched are very relevant. It seems that for one word the similarity scores and documents fetched are in unison. I would give it a precision of 1.

4. Searching for Query String - Housing

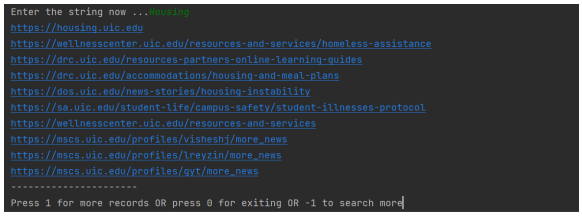

Fig. 4. Query - Housing

Even for this query the documents (and thus the URLs) fetched by the search engine are relevant and it exactly gives the results that are expected (although some results are unrelated). I would give it a precision of 0.7.

5. Searching for Query String - Final Exam Fall 2022

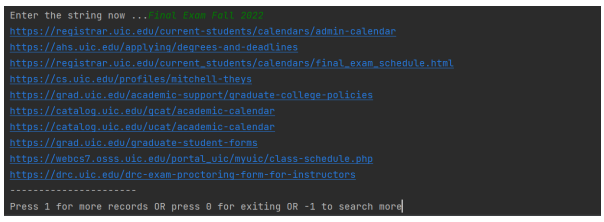

Fig. 5. Query - Final Exam Fall 2022

Although in this search engine and query parsing each token is considered independent of all it performs very well and fetches the retrieves the relevant pages. I would give it a precision of 0.7 as well.

The overall precision of the search engine comes out to be 0.64. Although this is subjective precision it signifies that the search engine works very well.

The accuracy of the search engine can be improved by crawling more pages and using other ranking algorithms and similarity measures like page rank, etc.

X. Results And Analysis

- The results obtained by the search engine for various queries yield good results considering the pages crawled
- The queries containing single word fetch good results as compared to multiple words in a query string. This can be attributed to the fact that a single word results in a very narrow search space when doing Cosine Similarity.
- The top 5 results that are fetched by the search engine are very relevant and gives good context into the search query. If top 5 queries are considered then the precision for this search will be around 0.9.
- Considering the top 10 results, it seems that the precision drop to around 0.7. This precision can be improved by crawling more pages and using more accurate similarity measure. In this search engine though the results provided by cosine similarity are rich and are very relevant to the document.
- The use of crawling with "requests" module without adding the user-agent works perfectly in that the user doesn't have to manually check the robots.txt files.

XI. Related Work

There are multiple application text-based and NLP-based applications where the use of cosine similarity is used to find the similarity measure. For searching some documents for coarse searching and sometimes for scalable search cosine similarity has been used. I found the use of cosine similarity and its implementations in multiple data science projects and also it is used for search and retrieval of documents for internal searching within an organization. Although, there are multiple applications that use BFS. Using BFS and traversing the web is not a novel idea. Indeed though, the use of BFS and then parsing href links from the webpage is a tedious job and for this, there are multiple blogs that are available that do a good job. BFS Searching is not only used in web searching but also used in unsupervised A.I. algorithms to scrap the landscape.

XII. Future Work

Although the search engine performs well and achieves a precision of 0.9 considering top 5 results. There are multiple enhancements that could be made in order to improve this as well as to speed up the application crawling process

- In terms of the information crawled, currently the crawler crawls only 3000 pages. To improve the precision and provide more diverse data more than 3000 pages need to be crawled. Say around 20k to 30k.

- The speed of the crawler is not restricted to only one main thread. Multi-threading can be used which will parallelly crawl multiple pages and update the frontier. In doing this though, great care has to be taken that different crawlers do not crawl the same frontier multiple times.
- Text ranking algorithms like page rank can be used which uses the information about the in-links and out-links. This naturally can help define authoritative pages in the given search space and provide more accurate results.
- As described in the previous sections other similarity and weighting measure can be used. Models like Word2Vec can be incorporated and the results of this can be seen.
- Currently the words which appear on the queries are thought of as independent and the relation between them is not measured by the cosine similarity. A relation factor can be used between these words which will further improve the results.