# SpidySearch: An efficent UIC search engine

**Arvind Ganesan**
Department of Computer Science
University of Illinois at Chicago
Chicago, USA
aganes25@uic.edu

## ABSTRACT

The work described in this paper is in adherence to the final project of the course, Information Retrieval (CS-582), at UIC, Chicago. A search engine is developed that crawls at least 3000 pages from the url http://www.cs.uic.edu, pre-processes the collected urls, uses vector space model to retrieve relevant documents. In addition, as part of the intelligent factor in search, PageRank and HITS algorithm are provided as an option to be used along with Cosine Similarity. Five Queries were written and were evaluated.

The code repository containing the search engine code along with top-10 results returned by queries can be viewed at the URL https://github.com/ArvindGanesan95/CS582_SearchEngineProject

## INTRODUCTION

A search engine, under its hood, searches a collection of several pages to find pages that are pertinent to the user query. In order to achieve this, the most widely used data structure is an Inverted Index. An Inverted index maintains a hash-table of sort to keep track of documents that consists of a vocabulary term. A vocabulary term is a token that is obtained after preprocessing each of the word from the text of a document. After processing each of the URLs content, using an inverted index, and using a similarity measurement technique between a document and user query, the engine returns relevant results.

## MAIN COMPONENTS

The engine is divided into 4 modules : A Web Crawler, A Preprocessor, Ranking Algorithms, and a User Query. Comments are added to cover almost all the code logic in the source code. The instructions to run the program is given in the read me file of the source code repository mentioned above.

### Web Crawling

The crawling starts from the URL http://www.cs.uic.edu. By adopting a breadth-first-search strategy, the pages are crawled and processed. Initially, the root URL is added to a queue Q. Since the number of pages to be crawled can be defined during the crawler instance creation, the crawler keeps running until the required number of pages are crawled. To speed up the creation of crawling, multi-threading concept is leveraged in Python. Python provides ThreadPoolExecutor library to create a set of fixed threads. During the instance creation, the primary function a thread must run up on and a callback function, after a thread finishes its work can be specified.

The primitive data structures in python are thread safe in itself. Hence, there is no need to handle synchronization issues for concurrent read/writes. To make sure no duplicate links are processed by the queue, a hash-table of visited URLs is maintained. Once an item is popped from the queue, it is marked as visited and submitted via thread pool to the function the thread handles. Then the outgoing links of the currently processed URL is obtained by fetching the "href" attribute of all the anchor tag in the page content of the current URL. This is done using the library called BeautifulSoup4 . For each of the outgoing link the BeautifulSoup fetched, it is checked for its validity. (See Restrictions below). All the new URLs that satisfy the restrictions are added to the Q for further processing of their outgoing links.

Each thread, after it gets an URL to process, performs the following functions : Get outgoing links of the current URL, Write the current URL's page content to the file system, and add the valid outgoing URLs back to the Q. The thread also makes sure that the URL added is not visited before.

To maintain the global shared counter of the number of pages crawled, a thread-safe unique counter library is used. In it, there exists a method with lock mechanism , that is used for incrementing the current value of counter by 1. Once a thread is ready to write a URL's page content to the file system, it gets a unique value, #uniqueCounter, from the counter library by calling increment function. Then the URL content is written with the file name : #uniqueCounter + ".json". In this way, each of the URLs get a dedicated file in the file system to maintain data consistency.

After the threshold limit for number of pages is reached, two different dictionaries are created. One is a complement of another. The structure of one of the map is : URL to #uniqueCounter, and the structure of the other map is: #uniqueCounter to URL. A outgoing links map is also maintained that keeps track of URL and its outgoing links. These two dictionaries

*CHI'20,* April 25–30, 2020, Honolulu, HI, USA

ISBN 978-1-4503-6708-0/20/04...$15.00
DOI: https://doi.org/10.1145/3313831.XXXXXXX

along with outgoing links map are written to the file system and are used later during PageRank and HITS algorithm computation.

*Restrictions*
The restrictions of the crawled URL must satisfy the following constraints: the URL must be within "uic.edu" domain, the URL must either be of http or https protocols, the URL must not match any item in the exclusion filters. The exclusion filters are (".gif"",".jpeg"",".ps"",".pdf"",".ppt"",".xml"",".docx"")

Then, every URL with its page content is written as a file to the file system.Once the limit number of pages is reached by atomic variable, the map maintaining URL and its outgoing links are written to the file system.

**Preprocessing**
The files written from the crawling step is read one by one to create an inverted index. Two model classes are created to represent an inverted index and a document. For every document that is read, it is pre-processed by applying the following sequence of operations: Tokenization, Stopword Removal, Stemming using Porter Stemmer, Removing Punctuation and Numbers. The filtered tokens are added to inverted index along with the document that contains the term. Finally, the inverted index is serialized and written to the file system using Pickle module in python. At last, the page rank scores are computed and stored by utilizing the outgoing links map created during web crawling. The preprocessor also computes document vector lengths which is independent of the user query and writes it to the file system. This is used during the computation of Cosine Similarity for a given user query.

**Ranking Algorithms**
Two ranking algorithms are used to provide an intelligent way of providing relevant URLs to the user. One, PageRank score of every node is computed by using the link structure of the collection. The link structure is the outgoing links map that was created during web crawling step. After fetching the relevant documents using vector space model, the documents' PageRank scores are read from the file written during preprocessing step. For every document, the cosine similarity score is computed and added to PageRank score of that document by applying two heuristics. Finally, the documents are ranked in decreasing order of scores and returned to the user. Two, HITS algorithm is offered to provide a new set of relevant documents to the query. The algorithm is query-dependent and runs every time a user submits the query. Similar to PageRank, after getting relevant documents from vector space model as the root set, the set is expanded by identifying other documents from the collection that points to any page in root set as well as documents that get pointed to by any page in root set. The expanded root set is given as input to hits algorithm and a final set of hubs and authorities values are obtained. The authority values are taken as final values of the URLs and they are added with Cosine Similarity by applying two heuristics before returning the final list of ranked documents.

Using the NetworkX python library, PageRank and HITS algorithm are called. For each of the algorithm ,the input is a direction graph structure created using the outgoing links map and the documents returned from Cosine Similarity

**User Query**
A simple HTML interface is developed allowing the user to input a search query and choose either of the ranking algorithms(PageRank or HITS, if needed) and view the results. By default, if the user does not choose a ranking algorithm, only Cosine Similarity is used for returning relevant URLs. If any of the ranking algorithm are chosen, the final URLs returned are computed using the process described above. The results are the relevant URL hyperlinks pertinent to the query. At a given time, only the top 10 results are shown. The user is provided a button to view the next set of 10 results. If the user's query did not return any matching documents, a appropriate message is shown to the user. If the server encountered an error due to environmental conditions, it is shown as an alert box in JavaScript.

## MAIN CHALLENGES

- One of the main challenge was architecting the crawler flows. I had to learn multi-threading and handle the synchronization issues in Python. There are two ways in python in which worker threads could be created. One is using threading library and starting several daemon processes, and the other is creating a fixed set of threads using ThreadPoolExecutors. I chose the latter since the API syntax were easier and did the job quickly. Although the primitive data structures were thread-safe, bringing overall control over each thread in using the common data structures and creating several maps were difficult. I had to learn to create using an atomic counter and create a custom class using python locking mechanism.

- Making the parallel threads to stop at right time was very difficult. Initially, there was an infinite while loop that performed popping from the queue until the queue becomes empty. But due to asynchronous nature of threads, it is possible for the queue to momentarily become empty and then become non-empty again. To resolve this, I had to learn more about python ThreadPoolExecutor library and the mechanism behind it. I found that the library had a wait() function to wait until all pending threads complete their work and add some element to the queue. If the queue still remained empty after wait() call, I was able to break out of the infinite loop.

- Learning the Flask server in Python was a new task. I had to refer the documentation to get started. Integrating the HTML/CSS part to python took some time for learning the syntax and for adhering to the design of communication.

- To decide if I had to use Hubs or Authority values from HITS algorithm along with Cosine Similarity, I had to learn the working and application of HITS algorithm from several research literature, consult with the professor, and then take a judgement call. I decided to focus on using Authority values of URLs as they are pointed to by other pages in the collection. This would mean they are highly important and relevant to the user query.

- Identifying the tuning parameter as well as the identifying the right combination of cosine similarity with PageRank/HITS scores. Identifying the accurate parameters involves a research study with different tunings. I felt this part to be open-ended and challenging. However, I tried to apply the concept of boosting and offer an increment to PageRank/HITS value if a certain condition is met (described in Discussion section)
- The last challenge was in error handling and propagation from the backend system to the UI. I felt that it is very essential for the user to know in case of an error. Initially, I tried using individual try/catch blocks , but that was inefficient in a way that I had to write try/catch clause for each function. The code has lots of functions. Instead, I learnt the concept of Python Decorator which is defined only once, and can be added to any function using the '@' syntax. In this way, any function will get executed within the scope of decorator. The decorator takes care of exception handling and propagate it to the caller

## WEIGHTING SCHEMES

To weigh the tokens in every document, TF-IDF measure is used as weighting measure. TF stands for term frequency and IDF stands for inverse document frequency. TF refers to number of times the term t occurs in document d. IDF refers to number of documents in which t occurs.

### Bag-of-words (BoW) vs TF-IDF

In Bag-Of-Words model , the frequency of term occurence is the primary factor and it does not give a sense about the overall collection of documents for a given term. It is possible for a vocabulary term to be present many times in only one document than other terms. BoW assigns higher weight to it, whereas TF-IDF may not give higher weight to it because with respect to the collection, it is not distributed across several documents. TF-IDF gives an idea of relevance of a term to a collection

## SIMILARITY MEASURES

Cosine Similarity is used as a similarity measure since it is widely established is proven to work well with vector space model. Here, a possible alternatives are mentioned along with a comparison to Cosine Similarity.

### Euclidean Distance vs Cosine Similarity

For a two n-dimensional vector, the euclidean distance calculates the distance between the vectors. However, there exists a problem in this measurement. If the distance between the points are the same, this method still does not tell which of the vectors are closer in content. In case of Cosine Similarity, orientation of vectors is considered which tells the closeness between two vectors, which is required for finding closely matching documents for a user query.

### Jaccard Similarity Measure vs Cosine Similarity

Jaccard similarity considers unique set of words for a given document and does not consider the frequency the term occurs in the document. A term maybe frequent due to its importance. Since, Jaccard Similarity measure does not take into account the term frequency, a vital piece of information in returning relevant ranked results to the user's query is missing. On the other hand, Cosine Similarity used with TF or TF-IDF considers term frequency of every vocabulary term.
Other alternatives would be to use Dice Coefficient, Minkowski distance and Manhattan distance and compare the accuracy with Cosine Similarity.

## EVALUATION OF QUERIES

The following queries were run and their precision were computed. Only Top-10 precision were calculated. Relevance were not able to be computed due to the large volume of collection and it requires human judgements. Three different modes of ranking were applied : Only Cosine Similarity, Cosine Similarity + Page-Rank , Cosine Similarity + HITS.

To provide a benchmark-like comparison, the queries were evaluated on UIC Website to do a visual comparison of top 10 URLs returned. Two precision values for each of the query are given. One is by comparing results from CS-UIC search engine and the other is by manual analysis by clicking on each page and analyzing the contents.

One important point is the way the UIC website search engine is implemented. It is unknown and may involve various search optimization parameters. The values are just provided for a side comparison. The UIC search engine maybe optimized by Google searches. The top-10 results for 5 queries are given below. Here, Cosine Similarity is referred shortly as "CS" for presentation purposes.

### Query 1 : UIC Computer Science Courses

*Based on Computer Science-UIC website search results* :
CS Score = 2/10
CS + PageRank (damping factor: 0.85): 1/10
$CS + HITS = 0/10$.

The results for HITS were from UIC catalog pages which did not list computer science courses

*Manually Analyzed and human-judged results* :
CS Score : 2/10
$CS + PageRank(dampingfactor: 0.85): 2/10$.
$CS + HITS: 0/10$

### Query 2 : Software Model Checking

*Based on Computer Science-UIC website search results* :
CS Score = 1/10
CS + PageRank (damping factor : 0.85): 1/10.
$CS + HITS: = 0/10$

The cosine similarity and PageRank returned same URLs

*Manually Analyzed and human-judged results* :
CS Score = 2 / 10
CS + PageRank (damping factor : 0.85): 2/10
$CS + HITS: 0/10$.

### Query 3 : Open Positions

*Based on Computer Science-UIC website search results* :
CS Score = 1 /10
CS + PageRank (damping factor : 0.85) = 1/10
$CS + HITS = 1/10$

*Manually Analyzed and human-judged CS Score* :
CS Score = 3 / 10
CS + PageRank (damping factor: 0.85) = 3/10
CS + HITS = 3 / 10

**Query 4 : Alumni Association**
*Based on Computer Science-UIC website search results* :
CS Score = 2/10
CS + PageRank $(\text{damping factor} : 0.85) = 2/10$
$CS + HITS = 2/10$

*Manually Analyzed and human-judged results* :
CS Score = 7/10
CS + PageRank $(\text{damping factor} : 0.85) = 6/10$
$CS + HITS = 7/10$

**Query 5 : Academic Calendar**
*Based on Computer Science-UIC website search results* :
CS Score = 0/10
CS + PageRank $(\text{damping factor} : 0.85) = 1/10$
$CS + HITS = 1/10$

*Manually Analyzed and human-judged results* :
CS Score = 2/10
CS + PageRank $(\text{damping factor} : 0.85) = 3/10$
$CS + HITS = 2/10$

## DISCUSSION AND ERROR ANALYSIS

Initially a linear addition of Cosine Similarity values and PageRank/HITS scores were applied and the results were checked. However, due to the dominating value of cosine similarity scores, there was very negligible change in the ranked documents that were returned.

### Ranking Heuristics

I could think of two heuristics to experiment with the ranked results returned to the user. Heuristic 1 is of the following formula :
$\text{ranked\_score}_{\text{url}}$ = $\text{maximum\_document\_score}_{\text{query q}}$ + cosine_similarity_score[url] + (page_rank_result[url] * 2)

The reason for multiplying the page ranks score by 2 is because on an average, the page rank score was at least two times smaller than a cosine similarity. This was found by an offline analysis for 500 documents that were downloaded. To minimize the risk of multiplication, the score is multiplied by 2 so that there is not a drastic boost to page rank score and return incorrect values.

Maximum document score is the maximum value obtained for a query q in the cosine similarity scores. Even though the cosine similarity takes into consideration of IDF and TF for a query term t, I thought that having a booster to include the maximum document score would bring more accurate results. This heuristic did better than just addition of cosine similarity and page rank score.

Heuristic 2 is of the following formula : If the cosine similarity is above 75 percent of cosine similarity, to avoid the dominating effect, a booster score is given to Page-Rank algorithm . For the booster score, 25 percent of current Page-Rank score is added to the current score and then only 75 percent of Cosine Similarity score is taken.

ranked_score = 0.75 * cosine_similarity_score[page] + boosted_page_rank_score[page]

where boosted_page_rank_score = page_rank_score[page] + page_rank_score[page] * 0.25

Finally , The set of documents whose total heuristic score is maximum is returned to the user.

In most of the cases, HITS did not result any relevant results to the user query. The issue of Topic Drifting during HITS could also be seen. The results bought some significant irrelevant results for the user query. Even for query with one term, the algorithm took an average of 200 seconds to return the results. This means that HITS would not be optimal to run every time.

It was observed that there was not significant drift in bring relevant documents when PageRank/HITS are compared with Cosine Similarity. There could be several factors that was causing almost similar performance of Page-Rank and Cosine Similarity.

But, by experimenting with these new heuristics I could think of, I understood the importance of parameter tuning and adding more factors to optimize a search engine.

### Mean Average Precision (MAP)

For the obtained precision values of manual judgements, Mean Average Precision (MAP) formula is applied. Since only precision at top 10 is taken, MAP can be calculated as follows.
MAP(Top 10 ) = Sum of precision value of each query / (number of queries)

$\text{MAP}_{\text{Top 10 / Cosine Similarity}}$ = ( 2/10 + 2/10 + 3/10 + 2/10 + 2/10 ) / 5 = 0.22
$\text{MAP}_{\text{Top 10 / Cosine Similarity + PageRank}}$ = ( 2/10 + 2/10 + 3/10 + 6/10 + 3/10 ) / 5 = 0.32
$\text{MAP}_{\text{Top 10 / Cosine Similarity + HITS}}$ = ( 0/10 + 0/10 + 3/10 + 7/10 + 2/10 ) / 5 = 0.24

## RELATED WORK

Cosine Similarity in a vector space model has been widely discussed and used for rank a set of documents relevant to the user query. The method gives its best performance when used with TF-IDF. Other methods that can boost the similarity is by using pseudo-relevant feedback mechanism. There are two ways providing feedback: Local Query expansion and Automatic Thesaurus generation. After fetching the query tokens from pre-processor, words that are synonyms of each of the query term are added to the query set. Then, Cosine Similarity is applied to retrieve relevant documents. However, this method is disadvantageous and is too expensive. HITS algorithm is query dependent and hence, needs to execute at run-time as soon as the user enters the query. This method is also expensive, however it offers a good set of authority and hubs which is a good indicator of the importance of the web pages. This method requires a root set to start with. PageRank algorithm runs on the entire collections link structure and can be precomputed. The disadvantage is that it is query independent and does not focus on the content of the page. Recent developments include using LDA algorithm with Cosine Similarity, PageRank algorithm using document clustering, SetRank, Context-Adaptive ranking model

## FUTURE WORK

- A potential improvement would be to come to a consensus when using HITS algorithm in choosing Hubs or Authority values. During the analysis, HITS algorithm returned pages which had more frequent occurrence of a single token in a query sentence rather than returning pages that had entirety of the query. This could be due to Topic Drifting problem where the neighbor nodes have more hub and authority values. A pruning mechanism to disregard the nodes that are irrelevant to the query could be explored.

- Page-Rank and HITS scores was used as a normal linear addition with Cosine Similarity values. However, there was not much difference in the returned ranked results. It was closer to Cosine Similarity. Instead some form of extensive tuning study is required for combining Cosine Similarity and PageRank/HITS scores

- In the user side, the search results time, the total number of pages actually returned could be shown to give the user an idea of the search engine efficiency

- Some of the URLs crawled have some redirects being performed. For example if the url is `https://abc.com` , once the user clicks or makes a request, the url request is forwarded to another url, say `https://cde.com`. This issue could be solved by identifying the resolved URL ahead of time and update our outgoing links map.