Implementation of search engine for Wikipedia articles

Omar Rodrigo Muñoz Gómez
School of Engineering and Science
Tecnológico de Monterrey
Estado de México, México
A01169881@itesm.mx

Elias Alejandro Villalvazo Avila
School of Engineering and Science
Tecnológico de Monterrey
Estado de México, México
A01798097@tec.mx

Francisco J. Cantu-Ortiz
School of Engineering and Science
Tecnológico de Monterrey
Estado de México, México
fcantu@tec.mx

***Abstract*—Modern search engines are sophisticated programs that enable the user to search in files and documents using words and phrases of any human language. Search engine optimization is the act of modifying the content and structure of web pages to increase traffic. Search engine optimization is the act of modifying the content and structure of web sites to make the search engine friendly, and rank well in search engine results and improving the volume and quality of traffic to a website from a search engine [2]. It is important to understand how search engines work because searching (in general) is a hard problem that we take for granted, when in reality, there is a big engineering effort behind the search process. We implemented a full-text search engine in Python that replicates they way modern engines work. In general terms, the program that we propose uses as input a compressed file containing abstracts from several articles in Wikipedia, then the abstracts are indexed. A dictionary containing a word and the ID of the article in which they appear is created. Words are processed (splitting, lowercase and stemming) to guarantee different forms of the word map point to the same item. Finally, the search will output the article ID where the search string appears.**

***Index Terms*—search engine, full-text, Wikipedia, Python, natural language processing**

I. Introduction

A search engine is a software that searches in a database of information according to the user's query. The engine provides a list of results that best matches what the user is trying to find. The first search engine ever developed was created by Alan Emtage. He created *Archie*, a program that downloaded directory listing for all the files that were stored on anonymous FTP (File Transfer Protocol) sites in a given network of computers. After Archie, Veronica (Very Easy Rodent-Oriented Net-Wide Index to Computerized Archives) was created. One of the particularities of this program is that it is the first text-based search engine [6]. Nowadays, search engines are sophisticated programs that enable the user to search in files and documents using words and phrases of any human languages. In general terms, the user enters a search term (a word or a term), then an algorithm examines the information stored in a database and retrieves links to web pages that potentially match the term that was entered by the user. Current effort concerning web archive research and development focuses on acquiring, storing, managing and preserving data [3]. Modern web search engines perform more different types of search, such as full-text, by URL, and metadata, that complement each other. Full-text search attempts to find documents with text that match the given keywords or sentences and has become the dominant form of information access. Is is one of the preferred methods for accessing web archive data, even with the high computational resources required to provide search over a large scale of collections [4]. In the URL (Uniform Resoulce Locator) search, users can search for a web document by submitting the URL. However, this type of search is limited, as it forces users to remember URLs which may have been visited a long time ago. In metadata search, the search is performed by searching by metadata attributes such as category, language, file format, etc. For this work, we will only consider the implementation of a full-text search engine.

Search engines perform a number of activities to deliver search results [2]:

- **Crawling.** Fetching all the web pages that are linked to a website. Software responsible of doing this process is called **crawler** or **spider**. The way spiders work is by gathering keywords from web pages that are then sorted so users can locate said pages through an Internet search engine.
- **Indexing.** Creation of an index (sorting of data by creating keywords or a listing of the data) for all the fetched websites and addition of these indexes to a database where they can later be retrieved. Normally, the tasks performed when indexing are the strip of stop words (very common words that are excluded from search), record of words on the page and the frequency they occur, record links to other pages, and record information of embedded media on the page.
- **Relevance Calculation.** Search engine calculates the relevancy of the indexed pages that are related to the search string. When a search request is performed, the search engine will compare the search string with the indexed pages in the database.
- **Results Retrieval.** Retrieve the best-matching results

Search relevance is a hard problem because we take the act of searching for granted. Search applications take a user's

search queries and attempt to rank content by how likely it will satisfy. This act occurs so frequently that it's barely noticed. A "simple" search often requires extensive engineering work.

The document of this work consists of five sections including this introduction. The second section describes the method and the data that we followed in order to achieve a full-text search engine. Then, the results can be observed in section three, followed by the discussions of further work and conclusions, which are described in section four and five, respectively.

II. Method and Data

In this section we describe the procedures followed to create and test the search engine. To begin with, we need a database to search. We use the Wikipedia website as a starting point. The goal of the search engine will be to look for words in the abstract section of all of the articles in the English version of the website.

The Wikimedia Foundation creates data dumps of the contents of Wikipedia and makes them available to the public; this is a great resource to test the performance of a search engine. These dumps contain the contents of all the wikis in the wikitext format (a special kind of markdown used to format a page) and related metadata. These data is updated regularly. At the moment, the dumps are generated approximately twice a month. Currently, Wikipedia has more than 6 million articles. The Wikimedia foundation provides the dumps in different formats, here are some of the main formats available:

1) Pages articles: Contains all the content (text) for the current revision of all Wikipedia articles.
2) Abstract: Contains only the text inside the abstract section for the current revision of all Wikipedia articles.
3) Titles: Contains all the titles for the current revision of all Wikipedia articles.

For testing the search engine we have chosen to work with the abstracts of all the articles. By avoiding searching through all of the articles, we reduce the complexity of the problem, and by searching in the abstracts, we increase the probability that the search finds the articles relevant to the query.

For each article, the XML contains the title, the url of the article, the abstract itself, and the links to the different article sections.

Listing 1. Abstract example

```
<doc>
<title>Wikipedia: Academy Award for Best
    Production Design</title>
<url>https://en.wikipedia.org/wiki/
    Academy_Award_for_Best_Production_Design</
    url>
<abstract>The Academy Award for Best Production
    Design recognizes achievement for art
    direction in film. The categorys original
    name was Best Art Direction, but was changed
     to its current name in 2012 for the 85th
    Academy Awards.</abstract>
...
</doc>
```

The search engine divides into two main parts, first is to create the index to hold the information from all the Abstracts such that the search time is minimized (Indexing). Once this data structure is created, the second part is to create a logic that can retrieve the abstracts relevant for the user based on a query string (Results retrieval).

Indexing. To create the index, we must parse the XML file first. For this purpose, an internal representation (class) of an Abstract is created. The objects of type Abstract contain four fields: the ID of the document, the title of the article, the abstract, and the url of the article. There are three methods for each Abstract object, the *fulltext* method returns the concatenation of the title and the abstract text. This is the text that will be used to create the index. The *analyze* method uses the string created from the *fulltext* method and applies a series of filters to create a string that is easier to search. From here, we create a dictionary with the unique words in the Abstract, together with the number of occurrences; occurrences are useful for creating a logic that can rank the importance of the articles when searching. The filters in the *analyze* method include a filter to remove the uppercase letters, remove the punctuation, remove the most common words, and to preserve only the root of the different words (stem filter). The most common words in the English dictionary are removed based on the assumption that they will appear most likely in every article and thus searching for these words will yield a list with multiple results and it will be harder to filter out relevant documents. The last method for the Abstract objects is *term_frequency* which receives a term and returns the number of times it appears in the output string of the *fulltext* method.

An internal representation (class) for the Index is also necessary. The index object contains two fields: The index and the documents. The index field is a dictionary that contains all of the words in the database (after applying the filters described earlier) and for each word, a set of all the file IDs where the word is present is created. We iterate through all the article abstracts and we use each word in them as a key to the index dictionary and the values are a set that contains all the file IDs where the word occurred. This makes the process of creating this dictionary hard as all this processing is needed for each of the articles on the Wikipedia website, however, once it is created, searching for the Abstract where a given term occurred is easier and overall faster (for example, the python implementation of a dictionary has an average time complexity of O(1)). The documents field simply holds all of the Abstract objects created so that the search can return them after the user creates a query.

Results retrieval. Once the index has been created we need a way to search through it based on a query from the user. The user enters a *query* as a string of all the words she is interested in and in return she gets a list of *documents* with all the abstracts that matched the *query*. The search can be configured in two ways based on the desired *search_type*. If the user wants all the words to appear in the abstract, the search should be configured as *AND*. If the user wants at least one of the words to appear in the abstract, the search

should be configured as *OR*. Additionally, the user can activate the *ranked_search* flag which will run an algorithm to rank the search results (**relevance calculation**) based on the tf-idf metric which is a combination of the term frequency (tf) and the inverse document frequency (idf) [5].

The first step of the search algorithm is to transform the *query* applying the same filters used to create the index, that is, removing the upper case letters, punctuation, common words, applying a stemming filter to keep only the root of the words in the *query*, and creating tokens for each word in the *query*. This corresponds to line 2 in Algorithm 1.

Using this transformed query, we search for the words (tokens) in the index. The index is a dictionary where the keys are all the words in the database and the values are the indexes of the files (Abstracts) where the word appears. This initial search will return a list with the indexes of the files for each word. See lines 4 through 7 of Algorithm 1.

Depending on the search type we can ask for the intersection of the indexes of all the words (AND-based search), the union (OR-based search) or apply the inverse document frequency algorithm (rank-based search). Apart from the dictionary described earlier, the Index also contains a list of all the documents (Abstracts). The indexes from the initial search are used to retrieve the documents that match the user query. See lines 8 through 21 of Algorithm 1. A diagram of the workflow followed by our code is described in Figure 1.

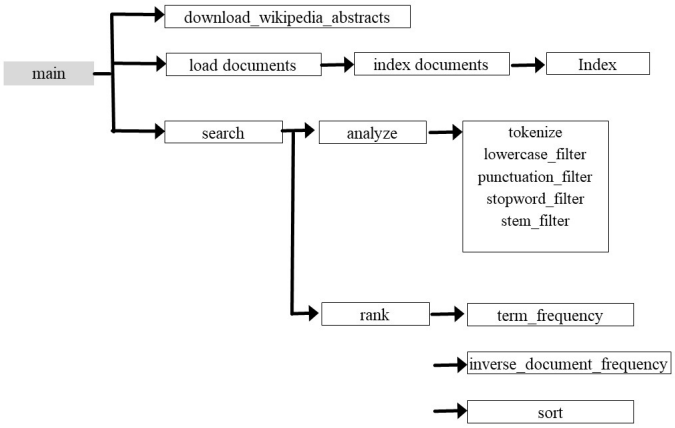

Fig. 1. Graphical hierarchy of the functions used in the search engine.

Algorithm 1 Search user query in the Index

Input: User query $query$, Index $index$, Search type $search\_type$, Rank search flag $ranked\_search$
Output: List $documents$ with all Abstracts matching the query

1: **function** $search$(query, index, search_type, ranked_search)
2: $analyzed\_query \leftarrow analyze(query)$
3:
4: $results \leftarrow \{\}$
5: **for** each token $t \in analyzed\_query$ **do**
6: $file\_ids \leftarrow index.get(t)$
7: $results.append(file\_ids)$
8: $documents \leftarrow \{\}$
9: **if** $search\_type\ is$ **AND then**
10: $results \leftarrow intersetcion(results)$
11: **for** each file_id $fid \in results$ **do**
12: $file \leftarrow index.documents(fid)$
13: $documents.append(file)$
14: **if** $search\_type\ is$ **OR then**
15: $results \leftarrow union(results)$
16: **for** each file_id $fid \in results$ **do**
17: $file \leftarrow index.documents(fid)$
18: $documents.append(file)$
19: **if** $ranked\_search\ is$ **True then**
20: $documents \leftarrow rank(index.documents)$
21: **return** $documents$

III. Results

In this section we present the results of the two phases described in the methodology section: indexing and results retrieval.

A. Indexing

All the abstracts of the Wikipedia website up to 02-Nov-2021 were downloaded (6.4 million) and an index was created based on the steps described in section II. The creation of the index is the most time consuming process as it needs to parse all the words in the abstracts and apply all the word filters described in the previous section. The creation of the index took around 16 minutes on a laptop with an AMD Ryzen 7 5800H 3.2 GHz processor and 16 GB of RAM. After the parsing and the filtering of words is done, the resulting index contains roughly 2.8 million individual terms together with the id of the article where the term appears. Figure 2 shows the wordcloud of the terms in the index. The larger the size, the larger the number of repetitions in the index. Notice that in this context, term refer to the words added to the index after all the filters described earlier were applied, thus the stop words, the uppercase, and punctuation are omitted, and stem filter is applied. The least common terms are terms that appear only once, however, there exists 1,916,286 terms that appear only once. A few examples of terms that appear only once are the terms *"actrius"*, *"isbn9789221115175"*, *"ansix3"*, and *"alkīmiyā"*. Table I shows the articles where the abstract contains the highest number of unique terms. One could think that having the largest number of unique terms

in an article may increase the probability of the article to be found, however, this is not necessarily true. Consider the case of the article *Phan Van Quy*, the article refers to a member of Vietnam's XIII National Assembly. This article shows 2nd, based on the highest number of unique terms, because the references used for the article come from websites written in Vietnamese language, which possesses special characters that are not frequently repeated throughout the articles of the English Wikipedia.

Fig. 2. Wordcloud of the most common terms in the created index for the Wikipedia website.

TABLE I
WIKIPEDIA ARTICLES WITH THE HIGHEST NUMBER OF UNIQUE TERMS.

Article	# terms
List of English-language poets	132
Phan Van Quy	117
Characters of Ackley Bridge	116
Ikhshidid dynasty	108
Pierre Minet	107
1996–97 New Jersey Nets season	105
Alfonso de Borbón y Borbón	105
Camera Camera (Nazia and Zoheb Hassan album)	103
Lee Byeong-cheon	100
Nahum Eitingon	100

B. Results retrieval

Once the index has been created, performing a search for a keyword and retrieving the articles containing the words is an efficient process thanks to the design of the index which is implemented as a Python dictionary. In this section, we show the different types of search that the engine supports together with the search time as computed by the *timing* method of the *time* module.

AND search is the default search type and for each word in the query it retrieves a set of articles and the intersection of the sets is returned to the user. For example, the query: *"mexico city"* with the AND search type returns a total of 4,656 articles. The search took on average 0.0026 seconds (as measured by averaging the search time of 100 searches of the *"mexico city"* query). The search can be either ranked or not ranked based on the tf-idf score. For the non-ranked search type, the three articles that showed up first were: *"Thirteenth Federal Electoral District of the Federal District"*, *"Gerardo Alcántara"*, and *"Diego Rivera"*. However, the articles are not presented in any particular order and are just articles that contain both words. When using a ranked search, the three articles that showed up first were: *"Circuito Exterior Mexiquense"* with a score of 19.49, *"Mexico City Grand Prix (badminton)"* with a score of 18.87, and *"Voit Mexico City Open"* with a score of 18.87.

OR search type will retrieve the union of all the articles that match the individual keywords of the user query. For example, the query: *"mexico city"* with the OR search type returns a total of 136,109 articles which is almost 30 times more results than the equivalent AND search. The search took on average 0.06 seconds (as measured by averaging the search time of 100 searches of the *"mexico city"* query). For the non-ranked search type, the three articles that showed up first were: *"Springdale, Brampton"*, *"Tlatilco"*, and *"César Rengifo"*. The articles are shown without any particular order. When using a ranked search, the three articles that showed up first were: *Kansas City, Missouri City Council"* with a score of 21.00, *"Circuito Exterior Mexiquense"* with a score of 19.49, and *"Mexico City Grand Prix (badminton)"* with a score of 18.87.

IV. DISCUSSION

The scope of this project is to illustrate concepts of search only, thus, it still requires lots of effort to convert it into a production-quality software.

The search runs entirely on a local computer, and it requires every time that a search needs to be computed, that data is downloaded and an index is created for all of the words found. Since indexes are stored in computer's RAM, then the length of the search space may be limited by the capacity of the local computer. Moreover, the current implementation relies on the usage of Python dictionaries, which, in general terms, consist of buckets of at least 12 bytes on 32-bit machines, and 24 bytes on 64-bit machines. These buckets contain the hash code of the object currently stored, a pointer to the key object, and a pointer to the value object. Initially a dictionary starts with 8 empty buckets, and then this is resized by double the number of entries when the capacity of the container is reached. As per the abstracts of the Wikipedia website up to 02-Nov-2021 used in this work, for the memory analysis we performed another experiment, the computer used for this test had a AMD Ryzen 7 4800H 2.8 GHz processor, and 8 GB RAM Memory. Around 6.4 million documents were indexed in 3529 seconds (1 hour approximately) and the memory used to abstract information was of 822,256,080 bytes (784.16 MB). The complexity of a search in a dictionary is O(1). For the AND search of "mexico city" the search took 0.001 milliseconds. However, indexing task consumes a lot of resources of time and space. We can conclude that in future works, a more efficient data structure and disk optimization are needed. One alternative would be to distribute data in different nodes (or servers), just as Elasticsearch [1] does by grouping data by shards (or self-contained index), which grants them high scallability and high availability. It is also necessary to design indices well, by

defining what goes and what does not go into the indices. The data that does not go into the indices can be stored elsewhere and redundant data can be removed. However, the trade-off is that the more storage is optimised, the less flexibility there is when in querying data.

From another perspective, stemming improves recall (number of correct words; true positives) by automatically managing the ending of words by reducing the words to their roots. In our case, stemming was done using the built-in script Stemming, which uses a snowball stemmer. This type of stemming, reduces the word to its word stem that removes suffixes and prefixes (e.g cared and caring convert to care). One of the outcomes of stemming, is that the number of retrieved documents can increase. However, stemming has to main errors: over-stemming and under-stemming. Over-stemming is when two words with different stems are stemmed to the same root (false positive). On the contrary, under-stemming is when two words that should be stemmed to the same root are not (false negative). The lighter the stemming, the less prone to over-stemming errors, however, under-stemming errors increase. On the other hand, applying a hard stemming reduces under-stemming errors, but it increases over-stemming. Moreover, stemming may sometimes decrease the total size of the index and lose some valuable information that directly impacts relevance (e.g *university* and *universe* stem into *univers*). One possible solution would be to use lemmatization instead to resolve words to their dictionary form. However, this requires more time and implementation effort.

Another critical element of search engines is the order in which the results are presented. In this work, we used the tf-idf metric, which assigns a score depending on the frequency of the terms in the query. This kind of score is relevant for searching Wikipedia articles since most of the articles themselves are static and describe facts. However, when searching other databases, it might be good to introduce a different ordering scheme. For instance, when searching through news articles, one might be more interested in the latest results according to their publication date. In other types of problems, ranking by popularity may be more relevant. The usefulness of the tf-idf metric can be seen by comparing the results of the AND search and the OR search performed in section III. The query in both cases is *"mexico city",* and as one would expect, when performing an OR search, the articles that match the search will not necessarily refer to Mexico as the search looks for either "mexico" or "city." However, when the ranking feature is enabled, the articles that show up in the first positions are relevant for both keywords in the query independent of the search type. This behavior may or may not be desired but it is up to the users to customize the search to suit their needs.

V. Conclusion

Modern search engines allow us to search through large databases efficiently using words and phrases of any human language. In this work, one of the critical enablers of efficient search is the creation of an index that acts as a lookup table. On average, searching for any query in the Wikipedia abstracts took less than 0.06 seconds once the index was created. The creation of the index is the bottleneck of the engine since it requires a lot of processing and computational power. Depending on the hardware available, this can take minutes or hours for this particular database. However, once that indexing is performed, the search process is fast. Search types allow us to customize our queries. In this work, we looked at the AND and OR search types which yielded different results, making the search more flexible. Other search engines provide additional features such as ordering by date, searching for images, document type, among others. This search engine may be extended and customized further to meet the user's needs. One of the limitations of our search engine is that the database used is not regularly updated. The search results are limited to a specific snapshot of the website. That is, the website is expanding daily, and new articles will not be considered. In future work, the implementation of a more efficient data structure, a non-local indexing scheme, the implementation of additional search features such as search by date and by document type, the implementation of a periodic update of the database, as well as improving the interaction with the user through a friendly GUI would be an improvement area.

References

[1] What is elasticsearch?. recovered from https://www.elastic.co/es/what-is/elasticsearch.

[2] Prashant Ankalkoti. Survey on search engine optimization tools & techniques. *"Imperial Journal of Interdisciplinary Research (IJIR)*, Vol-3:40–43, 01 2017.

[3] Miguel Costa. Full-text and URL search over web archives. pages 71–84. Springer International Publishing, 2021.

[4] Miguel Costa and Mário J. Silva. Characterizing search behavior in web archives. In *In Proc. of the 1st International Temporal Web Analytics Workshop*, 2011.

[5] Standford Natural Language Processing Group. Inverse document frequency, 2008. [Online; accessed 24-October-2021].

[6] J.L. Ledford. *Search Engine Optimization Bible*. Bible. Wiley, 2015.