Grep+ : A Word based search engine
CS512 Project

RUTGERS

Lokesh Kodavati (BK576)
Arun Kalanadhabhatla (LK586)
Shreya Rajendra Pai (SP2305)
Deevesh Chowdary Gogineni (DG1098)

December 23, 2022

Contents

1 Abstract

Utilizing advanced algorithms, tech giants such as Google and Microsoft can rapidly sift through the plethora of web pages on the internet, swiftly identifying and returning a curated selection of results pertinent to a given search query. Each website comprises a veritable lexicon of words and meanings. These companies have developed various algorithms for comprehending and indexing the content of a webpage to facilitate efficient search functionality. Similarly, there is a need within both academic and professional spheres to enable individuals to search through massive collections of documents based on a specific word or phrase. For instance, an author may desire to peruse all references to "eating" within their previous works. Still, given the vast number of books they have published, it would be onerous to review each manually. Therefore, there is a desire to create a system that can comprehend the entirety of the words contained within these books and facilitate searching through them.

2 Problem Statement

Given a huge collection of text files or word documents, it is essential upon us to devise a scalable algorithm capable of searching for a specified word. If an exact match cannot be found, we return the nearest possible match. Assuming that updates to the documents will be infrequent, and therefore a considerable volume of search queries will be made in an effort to locate the desired results.

For example, consider the following sentences contained within two separate word documents:

Word Document 1:
Following mice attacks, caring farmers were marching to Delhi for better living conditions. Delhi police on Tuesday fired water cannons and tear gas shells at protesting farmers as they tried to break barricades with their cars, automobiles, and tractors.

Word Document 2:
Sometimes to understand a word's meaning you need more than a definition; you need to see the word used in a sentence. At YourDictionary, we give you the tools to learn what a word means and how to use it correctly. With this sentence maker, simply type a word in the search bar and see a variety of sentences with that word used in different ways. Our sentence generator can provide more context and relevance, ensuring you use a word the right way.

Word Document 3:
Whether it's simple sentences for those just learning the English language or phrasing for an academic paper, this easy-to-use sentence generator will help you choose your words with confidence.

If we wish to find the documents which contain the word “sentence”; clearly, the word is present in document 2 and document 3, then the algorithm should return both the documents along with the occurrences of the word in the document.

3 Premise of the Solution

Throughout this document, we will refer to the word being searched for as the ”word of interest,” as it is the term that the user is interested in and seeks to locate the occurrences of within the complete set of documents.

The initial goal of this project is to preprocess all the documents and store the processed data in a logical format, enabling us to promptly return the instances of the word of interest within the full set of documents in response to search queries. However, we must determine what specifically should be included in the preprocessed data. To address this issue, we have conducted a series of basic analyses on the given problem, the results of which are as follows:

Analysis 1: In order to search for a word within a given list of words, it is necessary for the algorithm to at least traverse each word once, resulting in a time complexity of (N), where N is the number of words in the list, provided that the word list is unordered.

The primary takeaway from this analysis is that our algorithm must have a minimum complexity of (N).

Analysis 2: If we were able to predict the words that would be queried in the future and store the corresponding occurrences in a data structure, subsequent searches would simply necessitate iterating through said data structure.

As such, the primary conclusion drawn from these analyses is that we can store the occurrences of all the words that may potentially be queried in the future, and when one of these words is searched for, we can simply return the stored occurrences from the data structure.

Analysis 3: However, the question remains: how can we predict the words that will be queried in the future?

The central idea and takeaway of the project emerges from this query: the concept is to store the occurrences of all words. Our future search queries will always be comprised of one of these words, and it is highly likely that some words will be repeated. As a result, the length of the words in our data structure will be reduced to only the unique words.

4 Additional Analysis

- A user may be searching for the word ”sitting,” but their word of interest encompasses all words that signify ”sit.” Therefore, our search string should be

transformed into "sit," and all occurrences of the word "sit" should be returned, with the occurrences of the word "sitting" also being part of the search results.

- Another important consideration is that a user may have forgotten the spelling of the word they are seeking. For instance, if a user desires to search for the word "remember," but instead enters the spelling "remember" (perhaps due to being a primary school student), the system must be robust enough to identify the intended word and return the closest match.

5 Conceptual Depth

Consequently, as a result of the previous analyses, we have devised a system that will be able to effectively and efficiently handle the said problem in a scalable and robust manner. However, before looking at the implementation process, there are several concepts that must be thoroughly examined.

Deriving from additional analysis 1, it is essential that we are able to search for a word solely based on the sense in which it is understood by the user. To achieve this, we can utilize the concept of lemmatization.

Lemmatization generally entails the proper handling of a vocabulary and the execution of morphological analysis on words, with the objective of eliminating inflectional endings and returning the base or dictionary form of a word, known as the lemma. In this project, all operations are, by default, performed on lemmatized words.

6 Implementation Steps

We await the user to upload their dataset, which is a collection of word documents or text files that contains their word of interest. Once we have obtained the dataset, the following are the implementation steps we utilized to implement the project:

- We commence by iterating through every word in the dataset and performing lemmatization, which converts each word to its root form. For instance, the words "running," "runs," and "ran" are all converted to their root word, "run."

- As we iterate through each word, we index it in a data structure. In this project, we have chosen to utilize a dictionary (Hashmap) as our data structure due to its constant access time.

- During the indexing process, we store the occurrence of the word, the document in which it occurred, and the line number of the word's occurrence.

- Once all indexing has been completed, the indexed metadata is preserved on the disk for future use. For example, if the application is turned off and later

resumed, we do not want the application to repeat the aforementioned operations.

- Finally, when the word of interest is provided to the application, it simply checks its index for a matching word and returns all occurrences of the word as a response. As we have utilized a dictionary, the access time is constant, enabling us to return the result for every query in a constant time.
- Based on additional analysis 2, if an exact match is not found in the data structure, we perform a binary search on the list of unique words to retrieve the closest matching word and return its occurrences to the user.

7 Visual Representation of the Implementation Steps

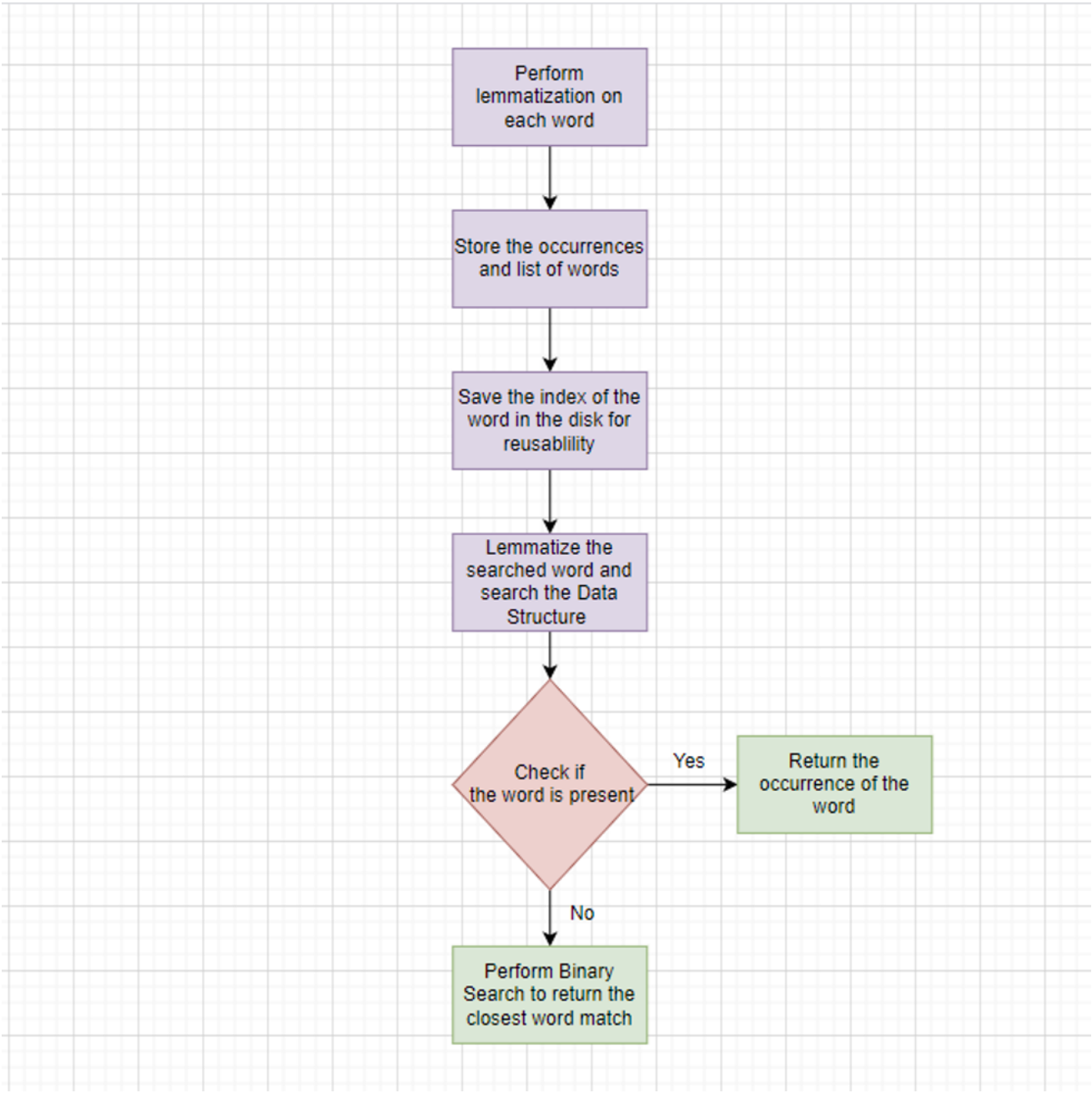

Figure 1: Flowchart

8 Technology Stack Used

The following Tech Stack has been utilized to develop the application

1. HTML
2. CSS
3. Java Script
4. Spacy Lemmatizer
5. Python/Flask

9 High-Level Description of the Project

In this project, we have used various modules to implement the project. They are as follows:

- Getting the list of all text documents in the dataset:

```
@staticmethod
def getFilesInDirectory(path):
    onlyfiles = [f for f in listdir(path) if isfile(join(path, f))]
    return onlyfiles
```

- After getting all the files, we open and iterate through every word in a file and perform lemmatization.

```
for file in files_list:


    # print(file)
    with open(os.path.join(os.path.join(os.getcwd(), GC.DATASET_FOLDER),
   file), 'r+') as f:
        lineNumber = 0
        for line in f:
            lineNumber += 1 if not line:
                continue
```

- After performing lemmatization, store every word along with its occurrence.

```
for words in lemmatized_line:
    if words.lemma_ not in indexDictionary["index"]:
        d = {}
        d[files_dictionary[file]] = {}
        d[files_dictionary[file]]["occurrence"] = [lineNumber]
        indexDictionary["index"][words.lemma_] = d
    else:
        d = indexDictionary["index"][words.lemma_]
        if files_dictionary[file] not in d:
            d[files_dictionary[file]] = {} d[files_dictionary[file]]["
   occurrence"] = [lineNumber]
        else:
            d[files_dictionary[file]]["occurrence"].append( lineNumber)
            indexDictionary["index"][words.lemma_] = d
```

- Store the occurrences in the disk for further use

```
@staticmethod
def writeToFile():
    index_file_path = os.path.join(os.path.join( os.getcwd(), GC.
   DATASET_FOLDER), GC.JSON_FILE)

    mode = 'w+'
    with open(index_file_path, mode) as idxfile:
        try:
            json.dump(GC.INDEXEDWORDS, idxfile)
        except:
            print("ERROR WRITING")


        idxfile.close()
```

- Lemmatize the word of interest and search for the data structure

```
requestWord = request.form.get('search', None)
print("REQUESTING FOR", requestWord)
lemmatizedObject = GC.SPACYLEMMATIZER(requestWord)
searchWord = lemmatizedObject[0].lemma_
print("SEARCHING FOR", searchWord)

if not searchWord:
    return render_template("search.html", response="Enter a search word")


response = AppService().searchWord(searchWord)
```

```
def searchWord(self, word, first=True):
    if first and not GC.INDEXEDWORDS and not self.isIndexed():
        self.isIndexed()
        return self.searchWord(word, False)
    if word not in GC.INDEXEDWORDS["index"]:
        # Perform Binary Search and Return the closest
        print("NOT IN THE DICT")
        closest_word = self.searchClosestWord(word)
        print("CLOSEST WORD", closest_word)
        return ResponseService().create_response(closest_word)


    if not first and not GC.INDEXEDWORDS and not self.isIndexed():
        return ResponseService().create_empty_response()

    return ResponseService().create_response(word)
```

- If the word is not present in the data structure, perform a binary search and search for the closest word

```
def searchClosestWord(self, word, first=True):
    if not GC.SORTEDWORDLIST:
        self.computeSortedWordList()
        return self.searchClosestWord(word, False)


    if not first and not GC.SORTEDWORDLIST:
        return ResponseService().create_empty_response()
    low = 0
    high = len(GC.SORTEDWORDLIST) - 1


    closest_word = self.binarySearch(low, high, word)
    return closest_word

@staticmethod
def binarySearch(low, high, word):
    mid = -1
    while(low < high):
        mid = (low + high)//2
        midpoint = GC.SORTEDWORDLIST[mid]
        if midpoint > word:
            high = mid - 1
        elif midpoint < word:
            low = mid + 1
        else:
            break
    return GC.SORTEDWORDLIST[mid]
```

10 Overall Time and Space Complexity

There are various parts to the proposed algorithm. Average time complexity cannot be determined. However, the time complexity for every step and every case is listed below:

10.1 Time Complexity

- Preprocessing Step: The preprocessing step takes (N) where N is the total number of words present in the dataset. time to perform and store the index of all the words in a data structure.
- For every query or word of interest, the search takes a constant, (1) time to retrieve all the occurrences of the given word.
- If the word of interest is not present in the whole dataset, we perform a binary search which takes (M) time where M is the number of unique words in the dataset.

10.2 Space Complexity

In all the cases, we perform the index and store the occurrence of all the words. Therefore, the space complexity is (M) where M is the number of unique words in the dataset.

11 Results

Our algorithm performed exceptionally well when the word of interest was present in the dataset. It retrieved results in constant time outperforming some of the in-built search algorithms used in modern systems. However, when the exact match is not present, it still performed very well retrieving the result in logarithmic time.

12 Limitations and Improvements:

We are tracking the addition of new files in the dataset. However, tracking the edits in the dataset is currently out of scope for this project. If a dataset is updated after the preprocessing steps, then we perform the preprocessing again and this step is performed explicitly by the user.

Building on the TA's comments, we tried implementing a function for adding new files in dataset. However, the scope of functions should be reducing human effort. We docked this idea realising it is easier to change the dataset manually.

13 Working Snapshots of the Project

- Main Screen

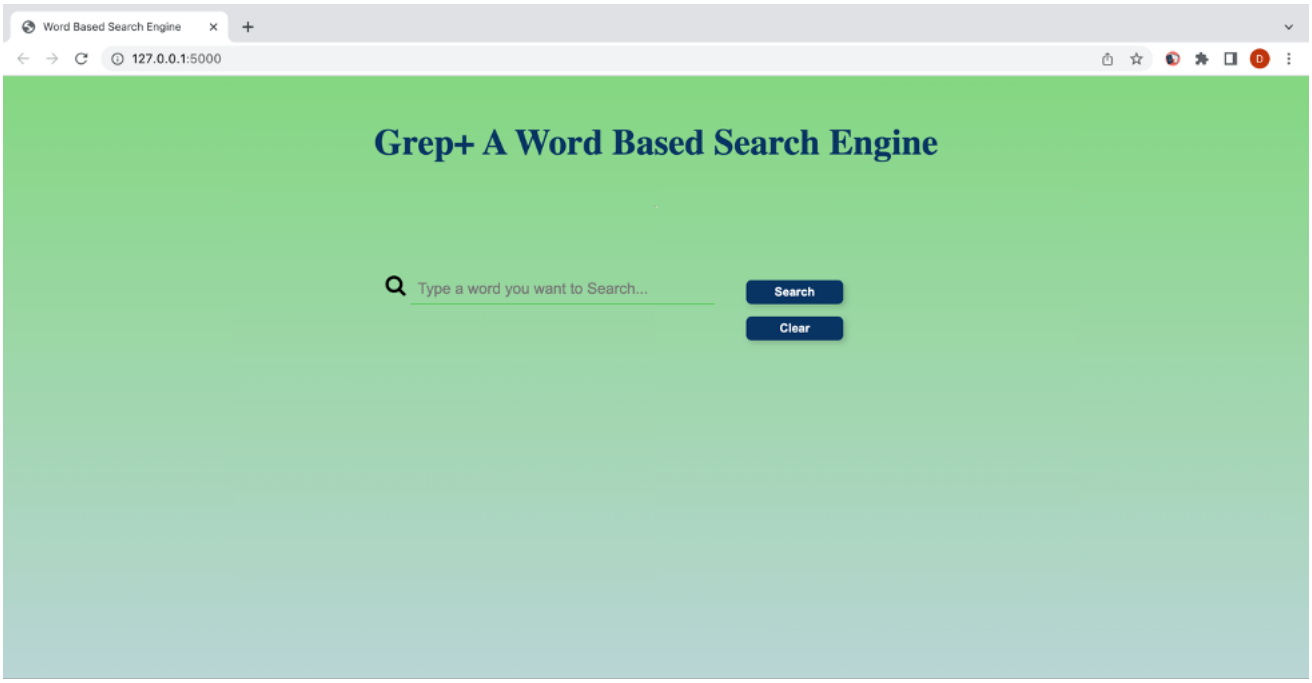

Figure 2: Main Screen

- Search Results

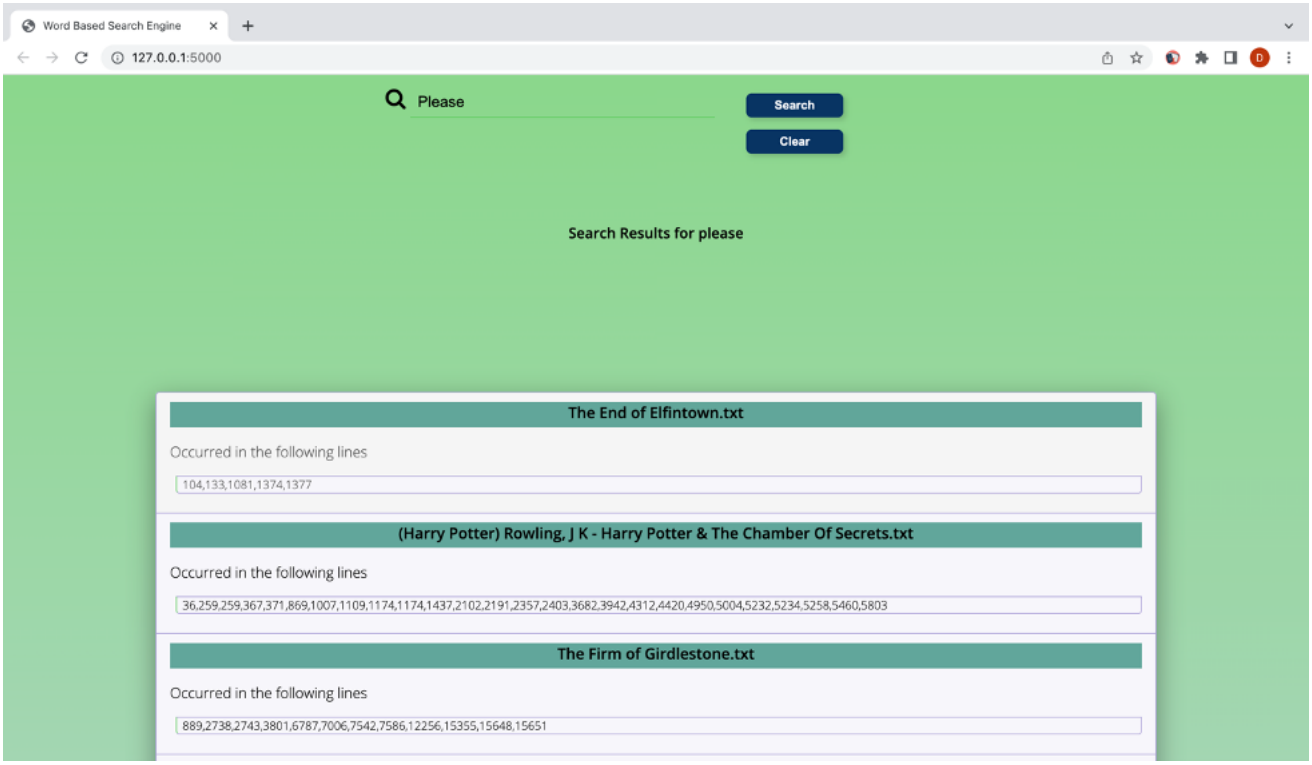

Figure 3: Search Results Screen

- If the exact word doesn't exist, then:

Figure 4: Exact word does not exist Screen