Local Mini Search Engine

Date: 2019-03-17

**Chapter 1: Introduction**

Every day Google processes 3.5 billion search queries[[1]](#footnote-1). To achieve this every single search engine utilizes the same procedures, downloading webpages, creating an inverted file index from the indexed pages and then process the queries and map the queries to the webpages based on their contents.

To achieve the first step of building a search engine, a WebCrawler/Spider is used to download all the contents of a given webpage. Google has developed its tool called GoogleBot, Bing has BingBot and Baidu uses Baiduspider. Once all the webpages are downloaded, then we process them for stemming and punctuations. Stemming means we get the roots of the words, the root might not be an actual word in English but it doesn’t matter since everything is stemmed using the same method; therefore the stemmed version of our query would match our inverted file index which is made from the stemmed version of the words.

For our search engine we used Porter Stemming, the algorithm is explained thoroughly [here](http://snowball.tartarus.org/algorithms/porter/stemmer.html). Once we have all the words stemmed then we need to remove any punctuations, because these characters are irrelevant, and they cause a lot of headaches for us while processing the queries. The Stemming and web crawling algorithms are discussed in the appendices as they’re not the focus of this project. The rest of algorithms used are discussed in the Chapter 2.

**Chapter 2: Data Structure/Algorithm Specification**

Mapping the Stemmed words

For mapping each word to its stemmed version, we use an unordered map, which is a hash table that is included by standard library of C++ with a very powerful hash function. To do this, while we are stemming the words using Porter Stemmer, we write each word and its stemmed version to a file, then this way we start reading the file, and then map stemmed version to its unstemmed version. To do so, we used the following algorithm:

f = open (./stemmed\_file.txt)

WordStemmer = unordered\_map(String -> String)

while (read f){

stemmed\_w = first word from f

original\_w = second word from f

WordStemmer[original\_w] = stemmed\_w

}

return WordStemmer

This function has a time complexity of O(1) and worst case scenario of O(N).

**Sorting words based on their frequencies**

Next step is mapping the root of each word to their appearances in the files. To do this again we can use an unordered map from the standard library and a vector of appearances to keep track of the words. The vector is used to sort the words based on their frequency. For sorting we just used the sorting function provided by the algorithm library.

vector(pair(string -> int)) SortedList

unordered\_map(string -> int) WordCount

files = open("./all\_stemmed\_files/\*.txt")

for f in files{

tempWord = words in the f

for w in tempWord{

if w does not exist in WordCount{

WordCount[w] = 1

}else{

WordCount[w] ++

}

}

}

for w in WordCount{

SortedList.append(pair(w, WordCount[w]))

}

sort(SortedList)

return SortedList

This algorithm helps us to identify the higher frequency words. We later use this SortedList to process our queries, and the relevancy of result is dependent on this. This algorithm has a time complexity of O(N) + O(N) + O(NlogN) on average case (assuming that the sorting method has a time complexity of O(NlogN) on average) which is O(NlogN). Next, we use the following pseudocode to generate number of appearances of each word and its location in the file.

**Index Generator and frequency recorder**

StopWords = A vector of stop words

files = open("./stemmed\_files/\*.txt")

IndexGenerator = vector(pair(String -> unordered\_map(String -> pair(int -> vector(int)))))

for f in files{

word\_loc = 1

IndexGenerator.append(pair(f, unordered\_map(String -> pair(int -> vector(int)))))

for w in words in the file{

if (w not in the StopWords){

if (w not in IndexGenerator[f].seconds){

IndexGenerator[f].second[w].first = 1

}else{

IndexGenerator[f].second[w].first++

}

IndexGenerator[f].second[w].second.append(word\_loc)

}

word\_loc++

}

}

return IndexGenerator

In this algorithm, we first open each file and add its filename as a string to the IndexGenerator. We also keep all the stop words in a vector. Then for each file we create a counter, which its job is to keep track of the index of the current word. Then we check if each word is one of the stop words, if not then we check if it already exists in the unordered map of our file, if it does we increment its counter by one and add its position to the second component, which is a vector that keeps track of the location of the word; if it doesn’t, we add it to the unordered map and add its location to the vector; if the word is just a stop word we skip this process and we just increment the word counter by one. We repeat this process for all the files, so we get a vector of all the files which has every word in the file indexed and keeps track of its location within the file, and its frequency. The time complexity of this function is dependent on the vector that stores the stop words, and the insertion of each word in the hash table. Since the vector doesn’t utilize a hash table, the search function performs a linear search or some similar method for determining whether the element exists or not; hence the time complexity of this method is O(SN), where S is the number of stop words in the vector and N is the number of words in the document. The insertion is O(N) given that the hash function is sophisticated enough to handle all the insertions. Therefore, the time complexity of this function is O(SN) given that vector performs a linear search and a powerful hash function.

**Query Preprocessing**

Now that we have finished the preprocessing of the indexed files, it’s time to move on to the queries. To process the queries, we need to first make sure that the words from queries match our list of words. To do that, we need to perform stemming on the queries. Since we have all the words recorded with their stemmed version, we can just search through them and get the root of the word.

StopWords = A vector of stop words

NewQuery = vector(String)

Query = input()

f = open('stemmed\_file.txt')

OriginalWord = first word of f in each line

StemmedWord = second word of f in each line

Stemmer = pair(OriginalWord, StemmedWord)

for word in Query{

if word not in StopWords {

if word in Stemmer{

NewQuery.append(Stemmer[word])

}

}

}

return NewQuery

**Query Processing**

Once we have the queries preprocessed, we store them in vector, then for each word we go through every file in the IndexGenerator, and check if the word exists in the file, if it does we record its first appearance location. We repeat the process for each word in the query and if they do exist in file, we increment our counter for the file by one, and append the location of them to our WordLocation variable. The file that has the most word is chosen as the result of our query.

Query = QueryProcessor(input())

if Query.length == 0{

throw runtime\_error

}

MaxFreq = -1

pair(String -> vector(int)) Result

for file in IndexGenerator{

TempFreq = 0

vector(int) Word\_Locations

for word in Query{

if word in file.second{

TempFreq += file.second[word].first

Word\_Locations.append(file.second[word].second[0]) // appending the first appearance of the word in the file

}

}

if (TempFreq > MaxFreq){

MaxFreq = TempFreq

Result.first = file.first // file name

Result.second = Word\_Locations // location of the words

}

}

return Result

**Main File**

After preprocessing the files and the query, we can process the query and print the corresponding file name, its location in the file.

Query = query\_processor(input)

if there are any results{

f = open(Query.first)

print(Query matched this file: Query.first)

for location in Query.second{

print(The words appeared in:)

print(f.loc[location-5:location+5])

}

}else{

print(Could not find any results)

}

**Chapter 3: Testing Results**

1. Common case with punctuation and stop words:

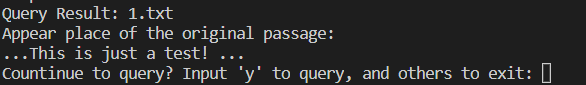


After the word stemming program, the result is:



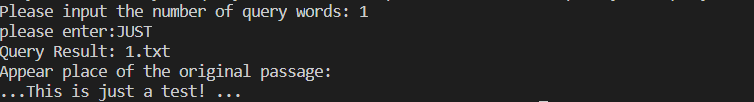
The result shows that the word stemming program successfully stems the words. The uppercase letters are changed to lowercase, and stemming worked as intended, i.e. this is changed to thi; further, the punctuation is dropped.

1. The results from searching the query “just”:



As expected, the search engine returned the right sentence.

1. Here is the results from the query “JUST”:



This shows that the program can deal with both uppercase and lowercase letters.

1. We try to search the stop word ‘a’, and the result is:



As you can see, there is no returned value. The reason is that the program omits all the stop words when it processes queries.

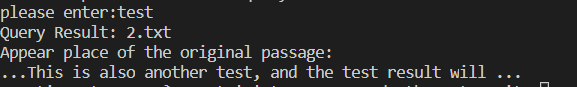
1. Many files with different words’ frequency

We generate two .txt files, and they are showed below:





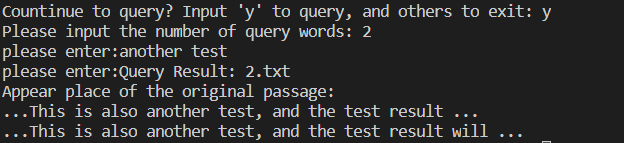
It is obvious that 1.txt has one “test”, 2.txt has two “test”. We try to search the word “test” and the result are shown below:



It shows that our program correctly identifies which file has higher frequency of “test”.

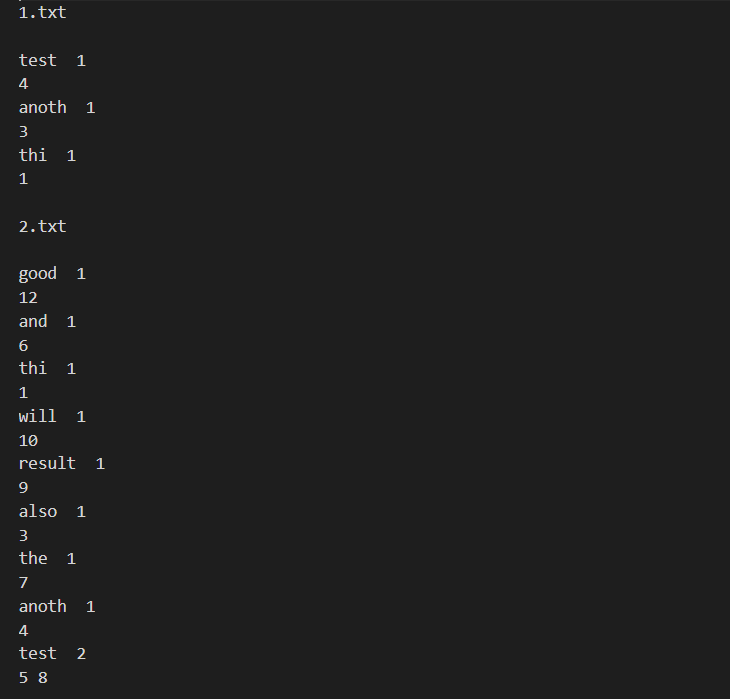
1. Multi-words query

We try to search for both “another” and “test” and the results are:



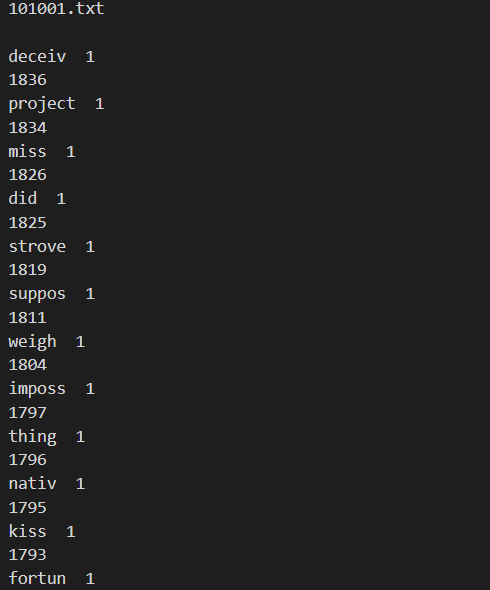
The results are correct. In our algorithm, we want the text which has higher sum of frequency of query words to display on the terminal.

To specify the answer, the inverted index is shown below:

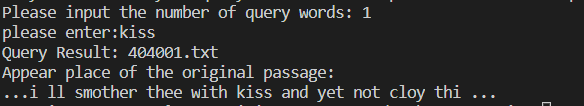


1. Shakespeare

The inverted index file looks like:



If we try to search for some word, like “kiss”, the results look like this:



**Chapter 4: Analysis and Comments**

**Time complexity**

We use N to denote the total number of words in the text, F for total number of files and Q for the length of query.

**Index Builder**

get\_word\_count()

In this part, we read in every word and sort them by their frequency. In worst case, the time complexity is O(NlogN). But in common cases, there are lots of words are repeated, so the average time complexity is O(N).

word\_stemmer()

The time complexity is O(N).

get\_index\_generation()

The time complexity is O(N), since it scans every word in text.

So the total time complexity is O(N) in average case. O(NlogN) in worst case.

**Search**

We use hash table to store the inverted index and scans every file and every query word to get the answer. So the time complexity is O(FQ).

**Space Complexity**

Because we use hash table to store the inverted index, in worst case scenario the space complexity is O(N).

**Further possible improvements**

1. The way we compute stop words is naïve. We define that the words appear more than 2000 times and length shorter than 2 are stop words, which is definitely incorrect. It only work for Shakespeare data set. One simple improvement is downloading the stop words online and apply to the “StopWordList” data structure.
2. The space complexity is huge. If we have tremendous amount of data, we can’t load it into the memory. As an improvement, we can use B+ tree data structure to store our inverted index.
3. The page rank method is naïve. We simply compute the frequency and select the text which has the highest frequency word. To make an improvement, we can improve our ranking algorithm, use more advanced data structures and so on.
4. We didn’t sort the query terms by their frequency in ascending order and search according to only some percentage of the original query terms. So it’s impossible to test the threshold of the query.
5. We can try to make some compression of our inverted index. For example, in posting list, we can store the difference of position instead of the position itself.
6. We didn’t implement the Boolean queries.
7. By implementing Word2Vec (pre-trained models are available) not only we can use the same words for the queries, but we can also use the neighbouring words for the queries. This can both improve the relevancy of the results and also the speed of the program. The reason it can improve the performance of the algorithm is that we no longer need to search for every word, we can combine few words together, get their vector sum; then find a word that can resemble those words with similar vectors. Further, we can use Word2Vec to create a vector for each document, and then measure the cosine distance of vector representation of the query with the document’s vector, whichever has the smallest difference is the

**Declaration:**

**We hereby declare that all the work done in this project titled "Local Mini Search Engine" is of our independent effort as a group.**

**Appendix A: Web Crawling**

To process the webpages and write them to the disk we used Python’s built-in library request and BeautifulSoup which eases the process of accessing the text of body within a webpage. Here is our code for indexing the webpages:

## all the libraries needed for preprocessing

import requests ## needed to do web requests

from bs4 import BeautifulSoup ## needed for crawling

import os

website = requests.get("http://shakespeare.mit.edu") # performs a get request

bs\_obj = BeautifulSoup(website.text) # creates a new instance of beautifulsoup object based on the url

tables = bs\_obj.find\_all("table") # finds all the tables in the page

links = []

for a in tables[1].find\_all("a"): # adds all the url in the table[1] to the links array

links.append("http://shakespeare.mit.edu/"+str(a['href']))

# the crawling process is a bit different for peotry and non-poetry works

non\_poetry = [] # array used to keep track of non-poetry works

poetry = [] # array used to keep track of poetry works

for a in links: # classification of urls

if "Poetry" in a:

poetry.append(a)

else:

non\_poetry.append(a[:a.find("index.html")]+"full.html") # adds the full text of the play to the list

for link in non\_poetry: # processing non\_poetry links

web\_req = requests.get(link)

bs\_obj = BeautifulSoup(web\_req.text)

with open(bs\_obj.title.text[:bs\_obj.title.text.find("\n")]+".txt", "a")\

as file:

file.write(bs\_obj.text) # writting the files on the disk

sonnets = [] # an array to keep track of sonnets

for links in poetry: # processing all the poetry works

if "sonnets" in links:

web\_req = requests.get(links)

bs\_obj = BeautifulSoup(web\_req.text)

dts = bs\_obj.find\_all("dt")

for dt in dts:

sonnets.append("http://shakespeare.mit.edu/Poetry/"+\

dt.find("a")["href"])

else: # writting the poetry works that are not sonnets

web\_req = requests.get(links)

bs\_obj = BeautifulSoup(web\_req.text)

with open(bs\_obj.title.text+".txt", "a") as file:

file.write(bs\_obj.text)

for sonnet in sonnets: # processing all the sonnets and writting them to the disk

web\_req = requests.get(sonnet)

bs\_obj = BeautifulSoup(web\_req.text)

with open(bs\_obj.title.text+".txt", "a") as file:

file.write(bs\_obj.text)

**Appendix B: Stemming**

To process the files and perform stemming on the words we used the NLTK (Natural Language Tool Kit) library for python. This library provides various stemming algorithms; the one we used for this project is called Porter Stemming. Here is the code for our stemming part:

import os

import string

from nltk.stem.porter import PorterStemmer # word stemming, need to install package "nltk"

porter\_stemmer = PorterStemmer() # initialize PorterStemmer object

path = "./shakespeare" # original files' path

new\_path = "./shakespeare\_dealt" # new files' path

def remove\_symbols(sentence): # eliminate the punctuation function

del\_estr = string.punctuation + string.digits

replace = " "\*len(del\_estr)

tran\_tab = str.maketrans(del\_estr, replace)

sentence = sentence.translate(tran\_tab)

return sentence

word\_stem\_file = open("./stem.txt", 'w')

for dirpath, dirnames, filenames in os.walk(path): # eliminate the punctuation function, remaining space and do the word stemming

for file in filenames: # loop among every file

fp = open(path + "/" + file, "r")

temp\_str = fp.read()

temp\_str = remove\_symbols(temp\_str)

temp\_str = temp\_str.lower()

temp\_list = temp\_str.split()

for i, item in enumerate(temp\_list): # store the original word and the result after word stemming

temp\_stem\_word = porter\_stemmer.stem(temp\_list[i])

word\_stem\_file.write(temp\_list[i] + " " + temp\_stem\_word + " ")

temp\_list[i] = temp\_stem\_word

temp\_str=' '.join(temp\_list)

new\_file = open(new\_path + "/" + file, 'w')

new\_file.write(temp\_str)

1. (internetlivestats, n.d.) [↑](#footnote-ref-1)