Prof Chandra Mouli P V S S R | Data Structure & Algorithms

File System Search Engine

ADITYA JAGDEV & faizan khan 14bce0185 14bce0187



# OVERVIEW

This project deals with:

* Handling of files
  + their data
  + their metadata

which is used to describe a file.

These above mentioned parameters have been used in this ‘File System Search Engine’, to sort through a large number of files and produce fast & accurate results.

Using the python programming language, we have implemented a system that can search inside a text file in a directory or its subdirectories.

Also, files and text in files can be searched in areas with even a part of the keyword.

This project successfully uses the concepts of hashing and indexing, with the help of an inverted index data structure, for optimum performance and search results.

After indexing and hashing, all keywords’ file associations are accessible in constant time. This makes this search engine extremely fast and reliable.

# SUMMARY

#### TEXT SEARCH

We are using an inverted index to build an index. An inverted index is a data structure that has tokens to the documents they appear in. In this context, we are considering a token to be a word, so an inverted index is just something that takes in a word and returns a list of documents it appears in. Initially, we are doing some text filtering by removing all punctuations and whitespaces. We’ll do this as follows:

* For every document we want to add to our index, we’ll remove all punctuation and split it on whitespace, and create a temporary hash table that maps filenames to their list of tokens. We’ll repeatedly transform this hash table until we reach final inverted index.
* Now, for handling phrase queries we’ll need to know where in each document each word shows up, so we can check for order. So our eventual inverted index will look like this

{ WORD : { document ID [POS1, POS2, ...] …} ...}

* At first, our task is to create a mapping of words to their positions for each document and then combine these to create our completed index.
* In the function ‘index\_one\_file’, we take a list of terms in the document and add them to hash table, where the values are a list of positions of that word. We build this list up iteratively as we go through the list, until we’ve gone through all the words, leaving us with a table keyed by strings and mapping to a list of positions of those strings.
* Now, we need to combine these hash tables which is done by creating an intermediate index of the format

{ { document ID: { word : [ pos1, pos2, ...] ...} ...}

which will transform to our final index.

The function ‘make\_indices’ just takes the results of the ‘file\_to\_terms’ function and creates a new hash table, keyed by a filename with values that are the result of the previous function, making a nested hash table.

Finally, we construct our complete inverted index as follows:

* First, we make an empty hash table (python dictionary) and we use 2 nested for-loops to iterate through every word in the input hash.
* Then, we first check if that word is present as a key in the output hash table.
* If it isn’t, then we add it, setting as its value another hash table, which maps the document to the list of positions of that word.
* If it is a key, then we do another check: if the current document is in each words hash table. If it is, we extend the current positions list with this list of positions equal to the positions list for the filename and now we have our index. We can input a word and be returned a list of the documents it appears in, in the list of positions it appears in within these documents.

Now, we’ll learn how to query this index.

#### QUERYING THE INDEX

#### ONE WORD QUERY

There are 2 types of queries we want to handle:

* Standard queries
* Phrase queries

At first, we sanitize the query by removing punctuation, converting to lowercase and stemming.

Now, to implement standard queries, we split the query into words. Get a list for each word of documents they appear in and then union all of these lists.

These things are done in the functions ‘one\_word\_query’ and ‘free\_text\_query’.

#### PHRASE QUERY

Phrase query is handled by the function ‘phrase query’.

In this function, we first sanitize the input query as usual. Then we run a single word query for each word in the input and add each of these result lists to our total list.

Then we create a set called ‘setted’, which takes the intersection of the first list and all the other list which leaves us with an intermediate result set: all the docs that contain all the words in the query.

Then we have to check for ordering. So, for every list in the intermediate results, we first make a list of lists of the positions of each word of the input query. Then we use 2 nested for loops to iterate through this list of lists.

For every position in every list, we subtract a number ‘i’, which increases as we go through the list of lists. ‘i’ increments by 1 when we go through the lists of lists. Now, since python preserves order, so this list of lists contains the position lists of every word in the original query.

Then, if these words are in the proper order and we subtract an integer ‘i’, from every position in each position list and ‘i’ increments by 1 as we go to each successive position list, then if these phrases are in order, the intersection of all these modified lists of lists must have a length of at least one.

#### RANKING RESULTS

There are 5 main steps we follow in ranking

1. First, we precompute the TF and IDF scores for each term and we build the ‘N’ length vector for each document, using the TF\*IDF of each term as the entries.
2. Then, we compute the query and get a result set of matching documents.
3. After this, we compute the vector for the query which is also of length ‘N’ and uses the TF\*IDF as each of its entries.
4. Then, we calculate the similarity between the query and each document in the result set (using cosine similarity) and get a score for each document.
5. We sort the documents by this score and return them in order.

The main data structure we are using in this project is the inverted index which is a form of nested hash tables.

The ranking of results uses the vector space model (the TF-IDF weights scheme) for calculating the relevancy rankings and hence displaying the results accordingly.

#### DOCUMENT SEARCH

We are using a technique for document search, similar to text search.

Initially, we create an inverted index of the given documents and files in the specified locations, which includes the subfolders and files as well. Now, we are mapping this index to a database, which is further split into four database documents

* Name.searchenginedb
* Size.searchenginedb
* Format.searchenginedb
* Path.searchenginedb

The procedure for creating an inverted index is the same as in text search. The only differences being the absence of words as tokens and the mapping of the resultant index in a database.

The only slow process is the formation of an inverted index. After the index is made the rest of the process is pretty fast. After indexing, we input the query and search in the database ‘Name.searchenginedb’ and this process happens almost instantaneously because the inverted index itself is designed according to the given query.

The data base returns the documents corresponding to the entire query or even related documents.

# CODE

|  |
| --- |
| Main.py |
| import time |
| import re |
| import math |
| import os, sys, inspect |
| cmd\_subfolder = os.path.realpath(os.path.abspath(os.path.join(os.path.split(inspect.getfile( inspect.currentframe() ))[0],"Core"))) |
| if cmd\_subfolder not in sys.path: |
| sys.path.insert(0, cmd\_subfolder) |
| import querytexts |
| from makefilesizeextpatharray import my\_list |
| from buildindex import BuildIndex |
| from querytexts import Query |
| import webbrowser as wb |
| exit=1 |
| choice=0 |
| while (exit==1): |
| os.system('cls') |
| read\_files=[] |
| print '++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++' |
| print '++++++++++++++++ FILE SYSTEM SEARCH ENGINE ++++++++++++++++' |
| print '++++ BY ADITYA JAGDEV AND FAIZAN KHAN ++++' |
| print '++++++++++++++++ 14BCE0185 14BCE0187 ++++++++++++++++' |
| print '++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++' |
| counter = 0 |
| print '\n1. Search for text in files\n2. Search for files\n3. Exit\n\nYour Choice : ', |
| choice = input() |
| if (choice == 1): #Text Search Module Begin |
| print 'Creating Index...' |
| for (dir, \_, files) in os.walk("./"): |
| for f in files: |
| path = os.path.join(dir, f) |
| if os.path.exists(path): |
| container = path |
| ext = container.rsplit( ".", 1 )[ 1 ] |
| ext=ext.lower() |
| if (ext == 'txt'): |
| read\_files.append(path) |
| query=Query(read\_files) |
| s=raw\_input('\nSearch in files for : ') |
| s=s.lower() |
| if ' ' in s: |
| test\_oneword=query.phrase\_query(s) |
| else: |
| test\_oneword=query.one\_word\_query(s) |
| if not test\_oneword: |
| print 'No Results Found' |
| else: |
| found1 = str(test\_oneword) |
| found1 = found1.rsplit("'",1)[0] |
| found1 = found1.split("/",1)[1] |
| full\_path = os.getcwd() |
| full\_path = full\_path + '\\' |
| full\_path = full\_path + found1 |
| print 'Phrase found in file \'' + (str(test\_oneword).rsplit("\\",1)[1]).rsplit("'",1)[0] + '\'' |
| choice1 = input('\nEnter 1 to open the file\nEnter 2 to open the containing folder\nEnter 3 to Continue\n\nYour Choice : ') |
| if(choice1 == 1): |
| wb.open(full\_path) |
| if(choice1 == 2): |
| wb.open(full\_path.rsplit("\\",1)[0]) #Text Search Module Ends |
| elif(choice == 2): #File Search Module Begins |
| print 'Creating Index...' |
| filenames=['Name.searchenginedb'] |
| query=Query(filenames) #printing the names path again |
| s=raw\_input('\nType filename to search for : ') #TAKING INPUT FROM USER HERE |
| s=s.lower() |
| if ' ' in s: |
| test\_phrase=query.phrase\_query(s) |
| else: |
| test\_phrase=query.one\_word\_query(s) |
| get\_fp = open('Path.searchenginedb') |
| get\_fs = open('Size.searchenginedb') |
| get\_ff = open('Format.searchenginedb') |
| get\_fn = open('Name.searchenginedb') |
| for i in range(len(my\_list)): |
| if s in my\_list[i]: |
| print '\nFile Name : ', |
| for j, line in enumerate(get\_fn): |
| if j == i: |
| print line, |
| break |
| print 'File Format : ', |
| for j, line in enumerate(get\_ff): |
| if j == i: |
| print line, |
| break |
| print 'File Size : ', |
| for j, line in enumerate(get\_fs): |
| if j == i: |
| #File Size Unit Begin |
| fsize = int(line) |
| if(fsize<=1024): |
| pwr = 0 |
| term = "Bytes" |
| elif((fsize/1024) <= 1024): |
| pwr = 1 |
| term = "KiloBytes" |
| elif(((fsize/1024)/1024) <= 1024): |
| pwr = 2 |
| term = "MegaBytes" |
| elif((((fsize/1024)/1024)/1024) <= 1024): |
| pwr = 3 |
| term = "GigaBytes" |
| #File Size Unit End |
| print str(fsize) + " " + term |
| break |
| print 'File Path : ', |
| for j, line in enumerate(get\_fp): |
| if j == i: |
| print line |
| break |
| full\_path = os.getcwd() #File Open Module Ends |
| full\_path = full\_path + '\\' |
| full\_path = full\_path + str(line).split("/",1)[1] |
| choice1 = input('\nEnter 1 to open the containing folder\nEnter 2 to Continue\n\nYour Choice : ') |
| if(choice1 == 1): |
| wb.open(full\_path.rsplit("\\",1)[0]) #File Open Module Ends |
| counter = counter + 1 |
| print '\nFound ' + str(counter) + ' result(s)' |
| get\_fp.close() |
| get\_fs.close() |
| get\_fn.close() |
| get\_ff.close() #File Search Module Ends |
| elif ( choice ==3 ): |
| exit = 0 |
| else: |
| print 'Invalid Input' |
| if (exit != 0): |
| exit=input('\nEnter 1 to Make Another Search : ') |
| print '\nHave ', |
| time.sleep(0.3) |
| print 'a ', |
| time.sleep(0.3) |
| print 'Nice ', |
| time.sleep(0.3) |
| print 'Day !!! ' |
| time.sleep(0.3) |
| print '+++++++++++++++++' |
| time.sleep(0.1) |

|  |
| --- |
| Buildindex.py |
| #input = [file1, file2, ...] |
| #res = {filename: [world1, word2]} |
|  |
| import re |
| import math |
| #import Pystemmer |
|  |
| class BuildIndex: |
|  |
| def \_\_init\_\_(self, files): |
| self.tf = {} |
| self.df = {} |
| self.idf = {} |
| self.filenames = files |
| self.file\_to\_terms = self.process\_files() |
| self.regdex = self.regIndex() |
| self.totalIndex = self.execute() |
| self.vectors = self.vectorize() |
| self.mags = self.magnitudes(self.filenames) |
| self.populateScores() |
|  |
| #input:Raw file |
| #output:terms |
|  |
| def process\_files(self): |
| file\_to\_terms = {} |
| for file in self.filenames: |
| #stopwords1 = open('stopwords.txt') |
| #stopwords=stopwords1.read() |
| pattern = re.compile('[\W\_]+') |
| file\_to\_terms[file] = open(file, 'r').read().lower(); |
| file\_to\_terms[file] = pattern.sub(' ',file\_to\_terms[file]) |
| re.sub(r'[\W\_]+','', file\_to\_terms[file]) |
| file\_to\_terms[file] = file\_to\_terms[file].split() |
| #file\_to\_terms[file] = [w for w in file\_to\_terms[file] if w not in stopwords] |
| #file\_to\_terms[file] = [stemmer.stem\_word(w) for w in file\_to\_terms[file]] |
| #stopwords1.close() |
| return file\_to\_terms |
|  |
| '''input = [word1, word2, ...] |
| output = {word1: [pos1, pos2], word2: [pos2, pos434], ...}''' |
| def index\_one\_file(self, termlist): |
| fileIndex = {} |
| for index, word in enumerate(termlist): |
| if word in fileIndex.keys(): |
| fileIndex[word].append(index) |
| else: |
| fileIndex[word] = [index] |
| return fileIndex |
| #input = {filename: [word1, word2, ...], ...} |
| #res = {filename: {word: [pos1, pos2, ...]}, ...}''' |
| def make\_indices(self, termlists): |
| total = {} |
| for filename in termlists.keys(): |
| total[filename] = self.index\_one\_file(termlists[filename]) |
| return total |
|  |
| '''input = {filename: {word: [pos1, pos2, ...], ... }} |
| res = {word: {filename: [pos1, pos2]}, ...}, ...}''' |
| def fullIndex(self): |
| total\_index = {} |
| indie\_indices = self.regdex |
| for filename in indie\_indices.keys(): |
| self.tf[filename] = {} |
| for word in indie\_indices[filename].keys(): |
| self.tf[filename][word] = len(indie\_indices[filename][word]) |
| if word in self.df.keys(): |
| self.df[word] += 1 |
| else: |
| self.df[word] = 1 |
| if word in total\_index.keys(): |
| if filename in total\_index[word].keys(): |
| total\_index[word][filename].append(indie\_indices[filename][word][:]) |
| else: |
| total\_index[word][filename] = indie\_indices[filename][word] |
| else: |
| total\_index[word] = {filename: indie\_indices[filename][word]} |
| return total\_index |
|  |
| def vectorize(self): |
| vectors = {} |
| for filename in self.filenames: |
| vectors[filename] = [len(self.regdex[filename][word]) for word in self.regdex[filename].keys()] |
| return vectors |
|  |
|  |
| def document\_frequency(self, term): |
| if term in self.totalIndex.keys(): |
| return len(self.totalIndex[term].keys()) |
| else: |
| return 0 |
|  |
| def collection\_size(self): |
| return len(self.filenames) |
|  |
| def magnitudes(self, documents): |
| mags = {} |
| for document in documents: |
| mags[document] = pow(sum(map(lambda x: x\*\*2, self.vectors[document])),.5) |
| return mags |
|  |
| def term\_frequency(self, term, document): |
| return self.tf[document][term]/self.mags[document] if term in self.tf[document].keys() else 0 |
|  |
| def populateScores(self): |
| for filename in self.filenames: |
| for term in self.getUniques(): |
| self.tf[filename][term] = self.term\_frequency(term, filename) |
| if term in self.df.keys(): |
| self.idf[term] = self.idf\_func(self.collection\_size(), self.df[term]) |
| else: |
| self.idf[term] = 0 |
|  |
| return self.df, self.tf, self.idf |
|  |
| def idf\_func(self, N, N\_t): |
| if N\_t != 0: |
| return math.log(N/N\_t) |
| else: |
| return 0 |
|  |
| def generateScore(self, term, document): |
| return self.tf[document][term] \* self.idf[term] |
|  |
| def execute(self): |
| return self.fullIndex() |
|  |
| def regIndex(self): |
| return self.make\_indices(self.file\_to\_terms) |
|  |
| def getUniques(self): |
| return self.totalIndex.keys() |

|  |
| --- |
| makefilesizeextpatharray.py |
| import os |
| fn = open("Name.searchenginedb", "w") |
| fs = open("Size.searchenginedb", "w") |
| ff = open("Format.searchenginedb", "w") |
| fp = open("Path.searchenginedb", "w") |
| container = 'p' |
| my\_list = [] |
| for (dir, \_, files) in os.walk("./"): |
| for f in files: |
| path = os.path.join(dir, f) |
| if os.path.exists(path): |
| container = path |
| if(container.rsplit(".",1)[1] != "searchenginedb"): |
| fp.write(path) |
| fp.write('\n') |
| fsize = os.path.getsize(path) |
| fname = container.rsplit( ".", 1 )[ 0 ] |
| fs.write(str(fsize)) |
| fs.write('\n') |
| fname = container.rsplit( ".", 1 )[ 0 ] |
| fname = fname.rsplit( "/", 1 )[ 1 ] |
| if "\\" in fname: |
| fname = fname.rsplit( "\\", 1 )[ 1 ] |
| fname = fname.lower() |
| fn.write(fname) |
| fn.write('\n') |
| my\_list.append(fname) |
| ext = container.rsplit( ".", 1 )[ 1 ] |
| ext = ext.lower() |
| ff.write(ext) |
| ff.write('\n') |
| fp.close() |
| fs.close() |
| fn.close() |
| ff.close() |

|  |
| --- |
| querytexts.py |
| import buildindex |
| import re |
|  |
| #input = [file1, file2, ...] |
| #res = {word: {filename: {pos1, pos2}, ...}, ...} |
| class Query: |
|  |
| def \_\_init\_\_(self, filenames): |
| self.filenames = filenames |
| self.index = buildindex.BuildIndex(self.filenames) |
| self.invertedIndex = self.index.totalIndex |
| self.regularIndex = self.index.regdex |
|  |
|  |
| def one\_word\_query(self, word): |
| pattern = re.compile('[\W\_]+') |
| word = pattern.sub(' ',word) |
| if word in self.invertedIndex.keys(): |
| return self.rankResults([filename for filename in self.invertedIndex[word].keys()], word) |
| else: |
| return [] |
|  |
| def free\_text\_query(self, string): |
| pattern = re.compile('[\W\_]+') |
| string = pattern.sub(' ',string) |
| result = [] |
| for word in string.split(): |
| result += self.one\_word\_query(word) |
| return self.rankResults(list(set(result)), string) |
|  |
| #inputs = 'query string', {word: {filename: [pos1, pos2, ...], ...}, ...} |
| #inter = {filename: [pos1, pos2]} |
| def phrase\_query(self, string): |
| pattern = re.compile('[\W\_]+') |
| string = pattern.sub(' ',string) |
| listOfLists, result = [],[] |
| for word in string.split(): |
| listOfLists.append(self.one\_word\_query(word)) |
| setted = set(listOfLists[0]).intersection(\*listOfLists) |
| for filename in setted: |
| temp = [] |
| for word in string.split(): |
| temp.append(self.invertedIndex[word][filename][:]) |
| for i in range(len(temp)): |
| for ind in range(len(temp[i])): |
| temp[i][ind] -= i |
| if set(temp[0]).intersection(\*temp): |
| result.append(filename) |
| return self.rankResults(result, string) |
|  |
| def make\_vectors(self, documents): |
| vecs = {} |
| for doc in documents: |
| docVec = [0]\*len(self.index.getUniques()) |
| for ind, term in enumerate(self.index.getUniques()): |
| docVec[ind] = self.index.generateScore(term, doc) |
| vecs[doc] = docVec |
| return vecs |
|  |
|  |
|  |
| def query\_vec(self, query): |
| pattern = re.compile('[\W\_]+') |
| query = pattern.sub(' ',query) |
| queryls = query.split() |
| queryVec = [0]\*len(queryls) |
| index = 0 |
| for ind, word in enumerate(queryls): |
| queryVec[index] = self.queryFreq(word, query) |
| index += 1 |
| queryidf = [self.index.idf[word] for word in self.index.getUniques()] |
| magnitude = pow(sum(map(lambda x: x\*\*2, queryVec)),.5) |
| freq = self.termfreq(self.index.getUniques(), query) |
| tf = [x/magnitude for x in freq] |
| final = [tf[i]\*queryidf[i] for i in range(len(self.index.getUniques()))] |
| return final |
|  |
| def queryFreq(self, term, query): |
| count = 0 |
| for word in query.split(): |
| if word == term: |
| count += 1 |
| return count |
|  |
| def termfreq(self, terms, query): |
| temp = [0]\*len(terms) |
| for i,term in enumerate(terms): |
| temp[i] = self.queryFreq(term, query) |
| return temp |
|  |
| def dotProduct(self, doc1, doc2): |
| if len(doc1) != len(doc2): |
| return 0 |
| return sum([x\*y for x,y in zip(doc1, doc2)]) |
|  |
| def rankResults(self, resultDocs, query): |
| #print 'Start rankResults' |
| vectors = self.make\_vectors(resultDocs) |
| #print(vectors) |
| queryVec = self.query\_vec(query) |
| #print(queryVec) |
| results = [[self.dotProduct(vectors[result], queryVec), result] for result in resultDocs] |
| #print(results) |
| results.sort(key=lambda x: x[0]) |
| #print(results) |
| results = [x[1] for x in results] |
|  |
| #print 'End rankResults' |
| return results |
|  |
|  |
| """Do this: |
| Calculate a tf-idf score for every unique term in the collection, for each document. As in, find all unique terms, and for each document, got through |
| each unique term and calculate a tf-idf score for it in the doc. You can do this already with the generateScore function. Doc becomes array of scores. |
| Calculate a tf-idf score for every unique term in the collection for the query. |
| Find the cosine distance between each document and the query, and put the results in descending order. |
| """ |
|  |
| #q = Query(['pg135.txt', 'pg76.txt', 'pg5200.txt']) |

# EXPERIENCE

Through this project, we learnt the basic components of a search engine:

* Indexing
* Querying
* Implementing the above using python

For indexing, we learnt how to implement an inverted index data structure in two different ways:

* Document mapping
* Word mapping

For text search, we used the famous bag of words model and hence learnt how to implement that.

For ranking and querying, we learnt about the vector space model and how to calculate relevancy and yield the results accordingly.

Overall, the project was a very interesting one and we learnt a lot and thoroughly enjoyed working on it.