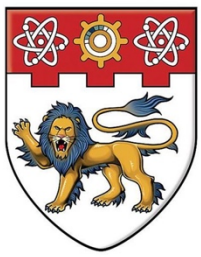


NANYANG TECHNOLOGICAL UNIVERSITY

Wee Kim Wee School of Communication and Information

18S2-SCI-CI6226-INFORMATION RETRIEVAL & ANALYSIS

Group Project Report

A Basic Information Retrieval System Using Inverted Index

Wang Sixin (G1801764B)
Zhang Yunyun (G1801061H)
Xia Xiaolong (G1801412A)
Zhao Hengrui (G1801739D)
Wang Xiangchang(G1801101L)

Date: 29 March 2019
Nanyang Technological University

First of all, as the advance preparation, we create two components: getFilePath and getFileContent. By using getFilePath, we can locate and list file paths from local directory and read all the resource file path in the list. After that, we can use getFileContent to get the text content of each file though a specific path. These two sections bring together all of our textual data and help to simplify future going through documents at indexing stage.

```
# output: list of paths. e.g. [path1, path2]
def getFilePath(self):
    textPath = os.getcwd() + '/HillaryEmails'
    if not os.path.exists(textPath):
        raise print('Please put HillaryEmails folder in work path')
    pathList = []
    for (dirpath, dirnames, filenames) in os.walk(textPath):
        for filename in filenames:
            pathList.append(os.path.join(dirpath, filename))
    pathList = sorted(pathList, key=lambda k: int(re.findall(r'(\d+)\.txt', k)[0]))
    return pathList

# output: string content of one file. e.g. 'this is file content'
def getFileContent(self, filePath):
    f = open(filePath, 'r')
    fileContent = f.read()
    f.close()
    return fileContent
```

Since text can come in a variety of forms from a list of individual words, to sentences to multiple paragraphs with special characters, before designing retrieval system, one essential part is source content preprocessing. This process involves several crucial steps: tokenization (to split text into words token) and text normalization(to remove all punctuation symbols, to lowercase all tokens and to stem the variant forms of a word to a common form).

1. Tokenization

The first step of preprocessing is tokenization. As the input type is string, we split all text data to a list by whitespace. By using split(), we get a list of all the words in the string with str as the separator (splits on all whitespace if left unspecified. After removing empty tokens, we combine tokens and the matching path to create token-path pairs. The output of this step is a list of token-path dictionaries.

```
# output: list of token pair. e.g. [{token1: path1}, {token2: path1}]
def tokenization(self, fileContent, filePath):
    tokenList = []
    tokens = fileContent.split()
    for token in tokens:
        if token != "":
            tokenPair = {token: filePath}
            tokenList.append(tokenPair)
    return tokenList
```

2. Linguistic Modules

Next, we should create linguistic module to handle more complicated text preprocessing. Since the punctuation marks are so frequently used in text but carry so less meanings(comparing to words), we first remove all punctuation symbols (!@#$%^&*()-_=+’`~”:;/.,?[]{}<>) so that we can focus on the important words. Then we turn remaining tokens into lowercase, which is applicable to most Natural Language Processing problems and can help in cases where the dataset is not very large and significantly helps with consistency of expected output. After removing all punctuation symbols and lowercasing tokens, we implement Porter Stemmer for stemming. As the most commonly used word reduction algorithm in English word processing, Porter algorithm has been proved effective repeatedly by practices.

The output is still a list of token-path dictionaries, with the tokens have been linguistically processed.

```
    # output: list of stemmed token pair
    def stemmer(self, tokenList):
        stemmer = PorterStemmer()
        newTokenList = []
        for tokenPair in tokenList:
            token = list(tokenPair.keys())[0]
            newToken = stemmer.stem(re.sub(r'[^\w]', '', token.lower()))
            if newToken != "":
                tokenPair[newToken] = tokenPair.pop(token)
                newTokenList.append(tokenPair)
        return newTokenList
```

3. Sorting the Tokens

For the basic postings list, main way to reduce the total times of merging index is to lessen the changings. In order to quickly generate an inverted index and merge posting, we need to sort the token-path pairs. Unlike mass ordering, sorting tokens could save much of merging time and extra effort of preparation for the inverse index in the following steps.

In this part, we start with concatenating all processed token list and then sorting the token list by tokens and paths. The sorting follows two steps: first we sort them by tokens in alphabetical order, and then further sort by document paths in alphabetical order as well.

In sortTokens, we select sorting key to order tokens by alphabetical order and then

document paths respectively. “Key” is any content that can be compared, and lambda is a function which could change the input string into a tuple and then the function of sortTokens would do the sort by tuple rather than by string, thus to decide the placement of each string.

At last, output of this step is a dictionary, tokens are keys of the dictionary and list of paths are values of the dictionary.

```
# output: list of all sorted stemmed token pair. e.g. [{token1: path1}, {token1: path2}, {token2: path1}]
def sortTokens(self):
    allTokenList = []
    pathList = self.getFilePath()
    for filePath in pathList:
        fileContent = self.getFileContent(filePath)
        tokenList = self.tokenization(fileContent, filePath)
        newTokenList = self.stemmer(tokenList)
        allTokenList += newTokenList
    sortedToken = sorted(allTokenList, key=lambda k: (list(k.keys())[0], list(k.values())[0]))
    return sortedToken
```

4. Transformation into Postings

Generally speaking string matching can be carried out in two forms: sequential string matching and indexed string matching. But indexed string-matching permits finding all the occurrences of a certain pattern without traversing the whole text, which is more efficient and less costly (there is a trade-off in memory cost and time save). So we move down to build an index.

After a prescribed ordering of the list of tokens, we could choose binary tree to determine if every key term exists in the vocabulary that is in order to assure the pointer to the corresponding lists.

If we use forward index, which is finding out value by key, all the documents should be searched for the key words, scoring them according score models and ranking them to return output. This kind of index structure cannot meet the requirements of returning results. By contrast, with inverse index, we could get a quick path list of documents that contain the token based on the token itself. That is, the mapping of document paths to tokens is transformed into the mapping of tokens to document paths.

Data structure to this point has been changed: each token corresponds to a series of files, and all these files contain this specific token, such as {token1: [path1, path2], token2: [path1]}

```
# output: dictionary of inverted index. e.g. {token1: [path1, path2], token2: [path1]}
def posting(self, sortedToken):
    invertedIndex = {}
    for tokenPair in sortedToken:
        token = list(tokenPair.keys())[0]
        if token not in invertedIndex:
            invertedIndex[token] = [(tokenPair[token])]
        elif (token in invertedIndex) & (tokenPair[token] not in set(invertedIndex[token])):
            invertedIndex[token].append(tokenPair[token])
    return invertedIndex
```

Next, we use pickle to dumpInvertedIndex function to serialize the inverted index and store it into a file. By doing this, system only need to create inverted index at the first searching. In future search, system would directly load in the next time by the function of loadInvertedIndex.

```
# once dump invertedIndex, we can directly load it next time
def dumpInvertedIndex(self, invertedIndex):
    outputFile = open("invertedIndex.txt", "wb")
    pickle.dump(invertedIndex, outputFile)
    outputFile.close()

# load invertedIndex
def loadInvertedIndex(self):
    inputFile = open("invertedIndex.txt", "rb")
    invertedIndex = pickle.load(inputFile)
    inputFile.close()
    return invertedIndex
```

Besides, search requests submitted by the users are preprocessed by the information retrieval systems. The query processing is done by implementing the same tokenization and lexical transformation on query content like what we do with the document texts.

```
# output: list of stemmed query tokens. e.g. [queryToken1, queryToken2]
def queryProcess(self, query):
    queries = []
    tokens = query.split()
    stemmer = PorterStemmer()
    for token in tokens:
        newToken = stemmer.stem(token.lower())
        if newToken != "":
            queries.append(newToken)
    return queries
```

5. Posting List Merge Component

After indexing the documents, information search is started. One of the most important modules is postings list merge. In this component, we first use postingList function to get the list of path lists by matching the tokens of inverted index and query. Then, we use postingMerge function to get common paths appear in all query tokens.

```
    # output: list of matched pathList. e.g. if queryToken1=token1 & queryToken2=token2 then [[path1, path2],[path1]]
    def postingList(self, queries, invertedIndex):
        matchedPostings = []
        for query in queries:
            if query not in invertedIndex:
                matchedPostings = []
                break
            matchedPostings.append(invertedIndex[query])
        return matchedPostings

    # output: list of common matched path. e.g. [path1], which is the common path of two pathList
    def postingMerge(self, matchedPostings):
        if len(matchedPostings) == 0:
            return []
        result = matchedPostings[0]
        if len(matchedPostings) == 1:
            return result
        for posting in matchedPostings[1:]:
            resultTmp = []
            index1, index2 = 0, 0
            len1, len2 = len(result), len(posting)
            while ((index1 < len1) & (index2 < len2)):
                if result[index1] == posting[index2]:
                    resultTmp.append(result[index1])
                    index1 += 1
                    index2 += 1
                elif result[index1] < posting[index2]:
                    index1 += 1
                elif result[index1] > posting[index2]:
                    index2 += 1
            result = resultTmp
        #    result = sorted(list(set(result)), key=lambda k: int(re.findall(r'(\d+)\.txt', k)[0]))
        return result
```

6. Results Output

In getResult function, we combine all the components together. We also add step to determine whether the inverted index file has been generated. If the file exists, system will load it in directly. For those search queries which contain more than one word, system basically will consider it to be a Boolean AND query. What's more, we add OR rule to make the system more flexible (users are required to include "OR" in query so as to trigger the OR rule).

```
    # output: string of combined path. e.g. 'path1 \n path2 \n'
    def getResult(self, query):
        if os.path.isfile('invertedIndex.txt'):
            invertedIndex = self.loadInvertedIndex()
        else:
            sortedToken = self.sortTokens()
            invertedIndex = self.posting(sortedToken)
            self.dumpInvertedIndex(invertedIndex)

        queryList = query.split(' OR ')
        result = []
        for query in queryList:
            queries = self.queryProcess(query)
            matchedPostings = self.postingList(queries, invertedIndex)
            result += self.postingMerge(matchedPostings)
        result = sorted(list(set(result)), key=lambda k: int(re.findall(r'(\d+)\.txt', k)[0]))
        finalResult = '<ol>'
        for path in result:
            finalResult = finalResult + "<li>Path:<a href='file://" + path + "'>" + path + '</a></li>'
        finalResult = finalResult + '</ol>'
        return finalResult

    # output: string to show paths
    def noResult(self, query):
        start = time.clock()
        result = self.getResult(query)
        elapsed = "Searching time: " + '%.4f' % (time.clock() - start) + " s"
        if result == "<ol></ol>":
            return 'No result',elapsed
        return result,elapsed
```

Lastly, we use PyQt5 package to design our UI. There are three buttons including

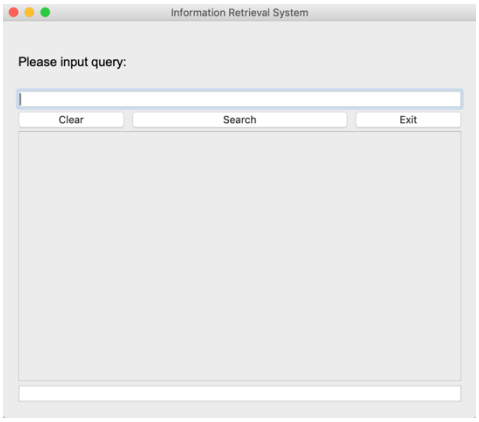
‘Search’, ‘Clear’ and ‘Exit’. The text box display sorted path names and we can also open the file by clicking the path. The small text box in the bottom shows the searching time.

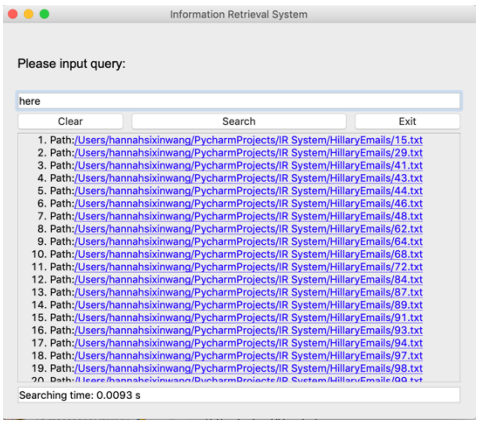
To enhance our user experience and enrich our functionality, we used a string containing HTML tags to present the final result.

- Use ordered list tag to give the answer sort id.
- Use the <a> tag to make the hyperlink to each result item to make it clickable.

When the user clicks on the result item, the .txt file of the corresponding path will be automatically opened.

```
def initUI(self):
    searchLabel = QLabel('Please input query:')
    searchLabel.setFont(QFont("Arial", 16))
    searchItem = QLineEdit()
    searchTime = QLineEdit()
    searchTime.setReadOnly(True)
    btn1 = QPushButton('Search', self)
    btn2 = QPushButton('Clear', self)
    btn3 = QPushButton('Exit', self)
    searchResult = QLabel()
    searchResult.setOpenExternalLinks(True)
    searchResult.setAlignment(Qt.AlignTop)
    resultScroll = QScrollArea()
    resultScroll.setWidget(searchResult)
    resultScroll.setWidgetResizable(True)
    grid = QGridLayout()
    grid.setColumnStretch(0, 1)
    grid.setColumnStretch(1, 0.5)
    grid.setColumnStretch(2, 1)
    grid.setColumnStretch(3, 1)
    grid.setColumnStretch(4, 0.5)
    grid.setColumnStretch(5, 1)
    grid.setSpacing(5)
    grid.addWidget(searchLabel, 1, 0, 1, 6)
    grid.addWidget(searchItem, 2, 0, 1, 6)
    grid.addWidget(btn1, 3, 2, 1, 2)
    grid.addWidget(btn2, 3, 0, 1, 1)
    grid.addWidget(btn3, 3, 5, 1, 1)
    grid.addWidget(resultScroll, 4, 0, 5, 6)
    grid.addWidget(searchTime, 9, 0, 1, 6)
    self.setLayout(grid)

    def searching():
        result, elapsed = self.noResult(searchItem.text())
        searchResult.setText(result)
        searchTime.setText(elapsed)
        self.repaint()

    btn1.clicked.connect(searching)

    def clear():
        searchResult.setText('')
        searchItem.setText('')
        searchTime.setText('')
        self.repaint()

    btn2.clicked.connect(clear)
    btn3.clicked.connect(QCoreApplication.instance().quit)
    # btn3.clicked.connect(self.close)
    self.setGeometry(400, 150, 600, 500)
    self.setWindowTitle('Information Retrieval System')
```





7. Test and Evaluation

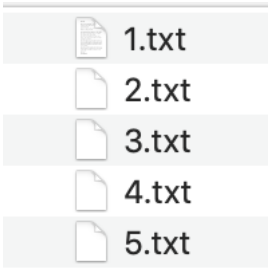
In this section, we start with a small document set of 5 files to do a set of demo functionality test, after which we will do black-box test to the whole document to check the accuracy of the query results which includes multiple test cases, such as single word query, multi-word query and OR mixed query. After that, we will test the time and size requirement of Information Retrieval System before and after optimization to evaluate our optimization.

7.1. Testing Environment

- Device: Macbook Pro (13-inch, 2017, Four Thunderbolt 3 Ports)
- OS: macOS Mojave 10.14.3
- IDE: PyCharm
- RAM: 8GB
- Time test tool: Profile
- Memory test tool: Memory_profiler

7.2. Demo-test (Small document set of 5 files)

We have retained 5 documents for demo testing.



Test case 1: Query = “father”. Results as:

Expected Answer: “2.txt, 5.txt”

Result: “2.txt, 5.txt”

Test case 2: Query = “mother”.

Expected Answer: “No Result”.

Result: “No Result”

Test case 3: Query = “wednesday”. Results as:

1. Path:/Users/zhaohengrui/Desktop/IR/demo-test/HillaryEmails/1.txt
2. Path:/Users/zhaohengrui/Desktop/IR/demo-test/HillaryEmails/3.txt
3. Path:/Users/zhaohengrui/Desktop/IR/demo-test/HillaryEmails/4.txt

Test case 4: Query = “father wednesday”

Expected Answer: “no result”

No Result

Test case 5: Query = “father OR wednesday”

Expected Answer: “1.txt, 2.txt, 3.txt, 4.txt, 5.txt”

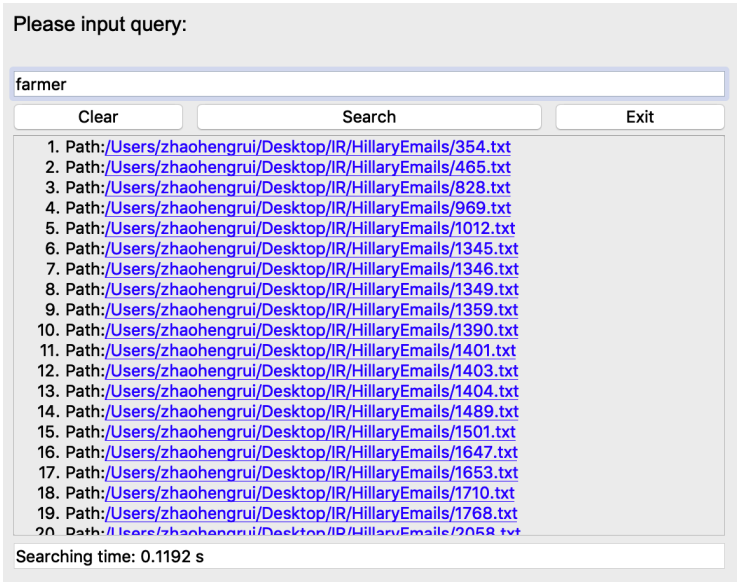
Result: “1.txt, 2.txt, 3.txt, 4.txt, 5.txt”

1. Path:/Users/zhaohengrui/Desktop/IR/demo-test/HillaryEmails/1.txt
2. Path:/Users/zhaohengrui/Desktop/IR/demo-test/HillaryEmails/3.txt
3. Path:/Users/zhaohengrui/Desktop/IR/demo-test/HillaryEmails/4.txt
4. Path:/Users/zhaohengrui/Desktop/IR/demo-test/HillaryEmails/2.txt
5. Path:/Users/zhaohengrui/Desktop/IR/demo-test/HillaryEmails/5.txt

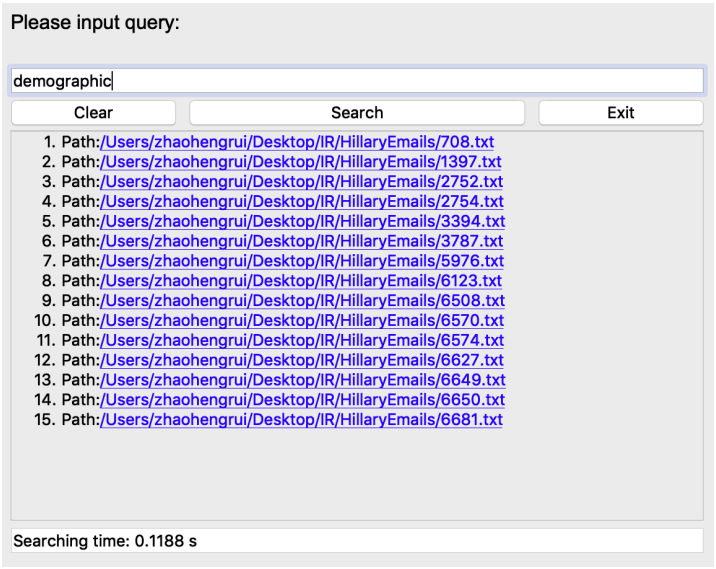
7.3. Black-box testing for whole document set

7.3.1. One-word query

Test case 1: Query = “farmer”. The result has 62 items. After check, the result is accurate. (Searching time: 0.1192s)

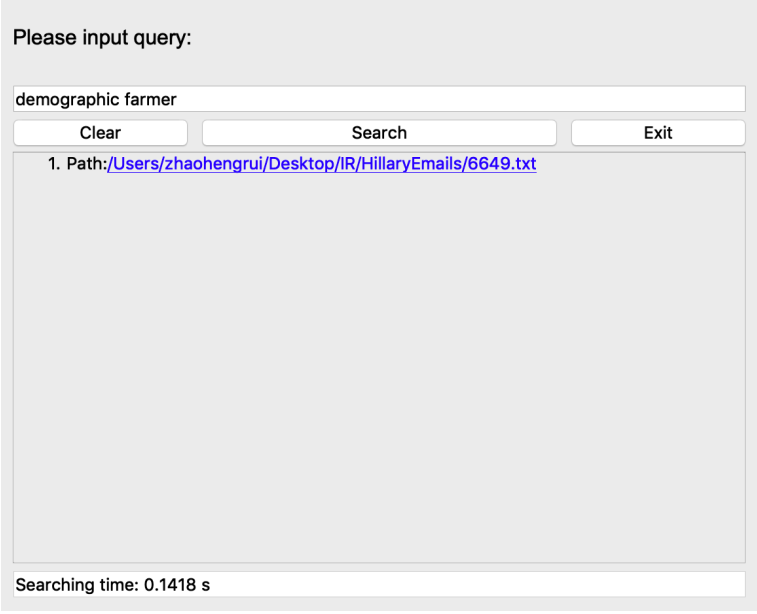


Test case 2: Query = “demographic”. The result has 15 items. After check, the result is accurate. (Searching time: 0.1188s)

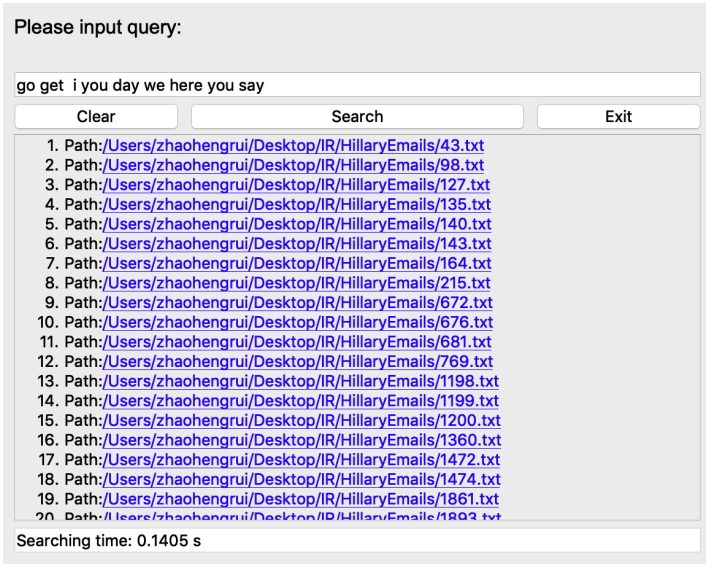


7.3.2. Multi-words query

Test case 1(Shorter Query): Query = “farmer demographic”. The result has 1 item. After check, the result is accurate. (Searching time: 0.1418s)

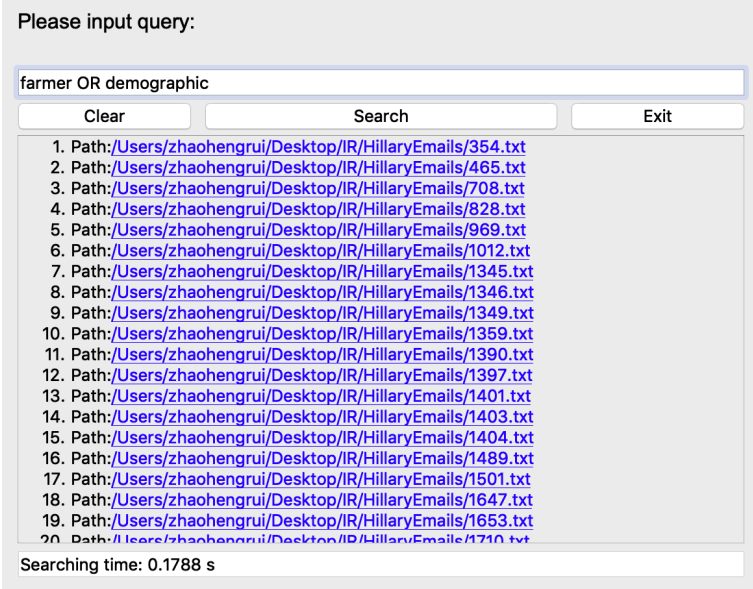


Test case 2(Longer query): Query = “go get I you day we here you say”. The result has 91 items. After check, the result is accurate. (Searching time: 0.1405s)

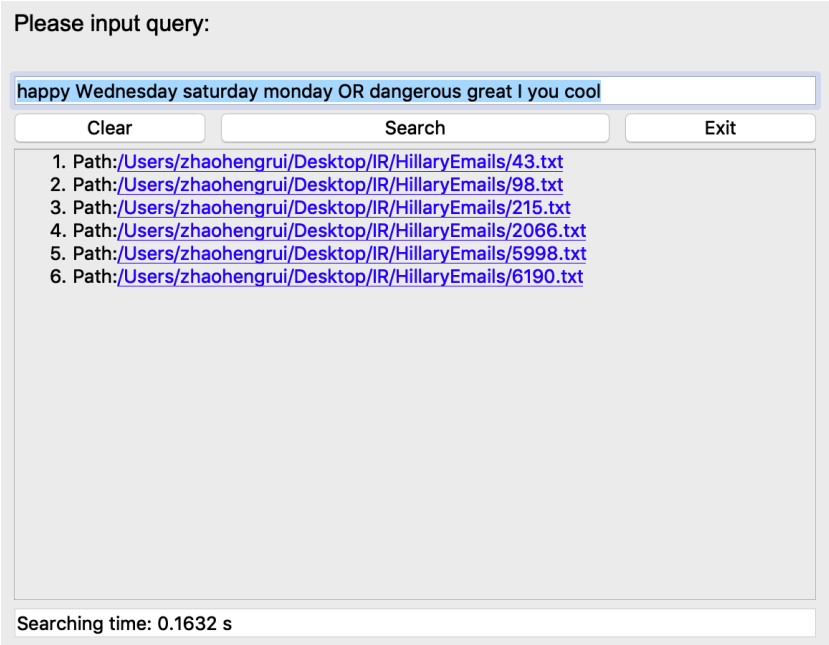


7.3.3. ‘OR’ Mixed query

Test case 1: Query = “farmer OR demographic”. The result has 76 items. After check, the result is accurate. (Searching time: 0.1788s)



Test case 2: Query = “happy Wednesday saturday monday OR dangerous great I you cool”. The result has 2 items. After check, the result is accurate. (Searching time: 0.1632s)



8. Optimization

We previously store the full document paths in the posting list. Each document path is stored multiple times, which we believe would affects the memory and time requirements for creating and querying indexes. Thus, in the optimization process, we try to store all the file path index in the posting list to improve the efficiency. Below are our optimization steps:

8.1. In sortTokens function, after optimization, we get the corresponding file path by traversing the index of the pathList instead of the pathlist itself. When we call getFileContent, we pass in self.__pathList(index) (which refers to file path). Then for the tokenization part, we just pass in the index instead of whole path.

Before Optimization:

```
# output: list of all sorted stemmed token pair. e.g. [{token1: path1}, {token1: path2}, {token2: path1}]
def sortTokens(self):
    allTokenList = []
    pathList = self.getFilePath()
    for filePath in pathList:
        fileContent = self.getFileContent(filePath)
        tokenList = self.tokenization(fileContent, filePath)
        newTokenList = self.stemmer(tokenList)
        allTokenList += newTokenList
    sortedToken = sorted(allTokenList, key=lambda k: (list(k.keys())[0], list(k.values())[0]))
    return sortedToken
```

Optimized:

```
def sortTokens(self):
    allTokenList = []
    for index in range(len(self.__pathList)):
        fileContent = self.getFileContent(self.__pathList[index])
        tokenList = self.tokenization(fileContent, index)
        newTokenList = self.stemmer(tokenList)
        allTokenList += newTokenList
    sortedToken = sorted(allTokenList, key=lambda k: (list(k.keys())[0], list(k.values())[0]))
    return sortedToken
```

8.2. In getResult(function, we use__filelist[index] to get the final document.

Before Optimization:

```
# output: string of combined path. e.g. 'path1 \n path2 \n'
def getResult(self, query):
    if os.path.isfile('invertedIndex.txt'):
        invertedIndex = self.loadInvertedIndex()
    else:
        sortedToken = self.sortTokens()
        invertedIndex = self.posting(sortedToken)
        self.dumpInvertedIndex(invertedIndex)

    queryList = query.split(' OR ')
    result = []
    for query in queryList:
        queries = self.queryProcess(query)
        matchedPostings = self.postingList(queries, invertedIndex)
        result += self.postingMerge(matchedPostings)
    result = sorted(list(set(result)), key=lambda k: int(re.findall(r'(\d+)\.txt', k)[0]))
    finalResult = '<ol>'
    for path in result:
        finalResult = finalResult + "<li>Path:<a href='file://" + path + "'>" + path + '</a></li>'
    finalResult = finalResult + '</ol>'
    return finalResult
```

Optimized:

```
def getResult(self, query):
    if os.path.isfile('opti_invertedIndex.txt'):
        invertedIndex = self.loadInvertedIndex()
    else:
        invertedIndex = self.posting(self.sortTokens())
        self.dumpInvertedIndex(invertedIndex)
    queryList = query.split(' OR ')
    result = []
    for query in queryList:
        queries = self.queryProcess(query)
        matchedPostings = self.postingList(queries, invertedIndex)
        result += self.postingMerge(matchedPostings)
    result = sorted(list(set(result)))
    finalResult = '<ol>'
    for index in result:
        finalResult = finalResult + "<li>Path:<a href='file://" + self.__pathList[index] + "'>" + self.__pathList[index] + '</a></li>'
    finalResult = finalResult + '</ol>'
    return finalResult
```

8.3. Time & Size requirement Testing

In this test session, we try to ensure that the environment is consistent and close other unrelated processes.

Our first test scenario is the first time to run the program and search "architecture" keyword, click on the search to get the search results, then close the GUI interface to complete the test.

Another test scenario is to search “architecture farmer” in both before and after

optimization system after creating indexes and calculate the average of 5 search times.

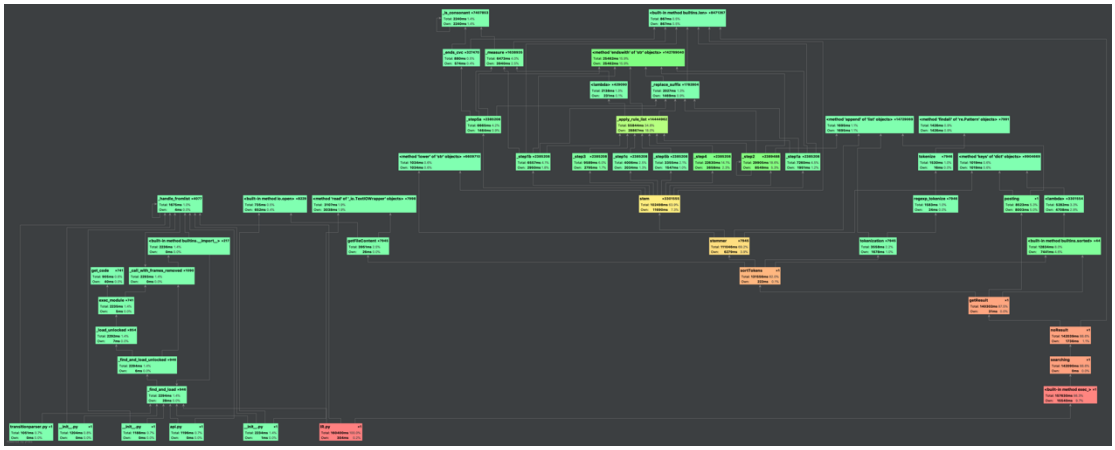
8.3.1. Before Optimization

Time profile result: It takes 160400ms for the whole process.

Name	Call Count	Time (ms)		Own Time (ms)	
IR.py	1	160400	100.0%	304	0.2%
<built-in method exec_>	1	157630	98.3%	15540	9.7%
searching	1	142090	88.6%	0	0.0%
noResult	1	142039	88.6%	1736	1.1%
getResult	1	140302	87.5%	31	0.0%
sortTokens	1	131556	82.0%	222	0.1%
stemmer	7945	111046	69.2%	6279	3.9%
stem	3301555	102498	63.9%	11690	7.3%
_apply_rule_list	14444962	55844	34.8%	28867	18.0%
_step2	2389488	29905	18.6%	8549	5.3%
<method 'endswith' of 'str' objects>	142789040	25462	15.9%	25462	15.9%
_step4	2385208	22630	14.1%	3658	2.3%
<built-in method builtins.sorted>	44	12834	8.0%	7441	4.6%
_step3	2385208	9589	6.0%	2795	1.7%
posting	1	8523	5.3%	8003	5.0%
_step1a	2385208	7260	4.5%	1951	1.2%
_step5a	2385208	6665	4.2%	1464	0.9%

As shown in the picture, <built-in method exec_>(Refer to the index establishing time) takes 157630 ms.

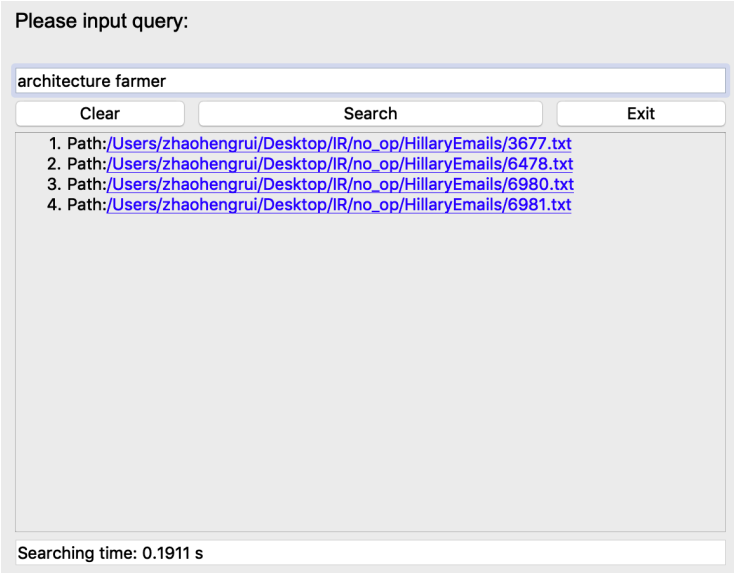
Call graph:



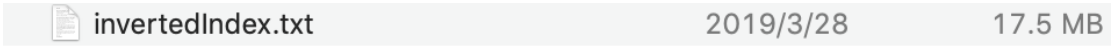
Size Consumption Result:

```
Henrys-MBP:no_op zhaohengrui$ python -m memory_profiler IR.py
Filename: IR.py

Line #    Mem usage    Increment   Line Contents
================================================
    62    138.4 MiB    138.4 MiB       @profile
    63                                 def sortTokens(self):
    64    138.4 MiB      0.0 MiB           allTokenList = []
    65    139.3 MiB      0.9 MiB           pathList = self.getFilePath()
    66    695.7 MiB      0.0 MiB           for filePath in pathList:
    67    695.7 MiB      0.2 MiB               fileContent = self.getFileContent(filePath)
    68    696.5 MiB      4.7 MiB               tokenList = self.tokenization(fileContent, filePath)
    69    695.7 MiB      0.3 MiB               newTokenList = self.stemmer(tokenList)
    70    695.7 MiB      0.1 MiB               allTokenList += newTokenList
    71   1187.7 MiB     82.8 MiB           sortedToken = sorted(allTokenList, key=lambda k: (list(k.keys())[0], list(k.values())[0]))
    72   1187.7 MiB      0.0 MiB           return sortedToken
```

Searching time（query case=”farmer architecture”）: Average searching time is 0.1913s (in 5 times)



Index Size:



The unoptimized invertedIndex size is 17.5 MB

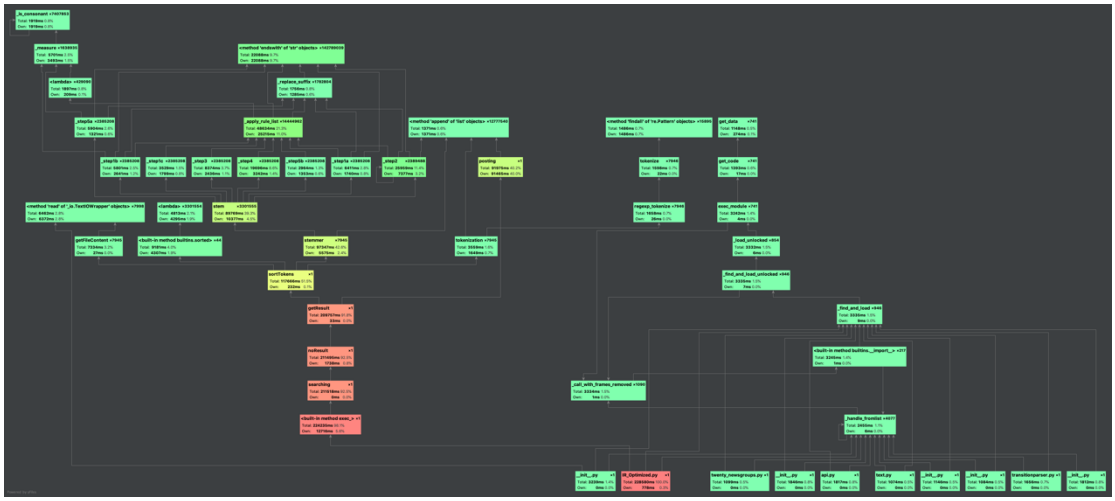
8.3.2. After Optimization:

Time profile result: It takes 228580ms for the whole process.

Name	Call Count	Time (ms)		Own Time (ms)	
IR_Optimized.py	1	228580	100.0%	778	0.3%
<built-in method exec_>	1	224235	98.1%	12716	5.6%
searching	1	211518	92.5%	0	0.0%
noResult	1	211495	92.5%	1738	0.8%
getResult	1	209757	91.8%	33	0.0%
sortTokens	1	117666	51.5%	232	0.1%
stemmer	7945	97347	42.6%	5575	2.4%
posting	1	91975	40.2%	91465	40.0%
stem	3301555	89769	39.3%	10377	4.5%
_apply_rule_list	14444962	48634	21.3%	25215	11.0%
_step2	2389488	25959	11.4%	7377	3.2%
<method 'endswith' of 'str' objects>	142789039	22088	9.7%	22088	9.7%
_step4	2385208	19696	8.6%	3242	1.4%
<built-in method builtins.sorted>	44	9181	4.0%	4307	1.9%
_step3	2385208	8374	3.7%	2436	1.1%
getFileContent	7945	7334	3.2%	27	0.0%
<method 'read' of '_io.TextIOWrapper' c	7998	6462	2.8%	6372	2.8%

As shown in the picture, <built-in method exec_>(Refer to the index establishing time) takes 224235 ms, which is longer than unoptimized one.

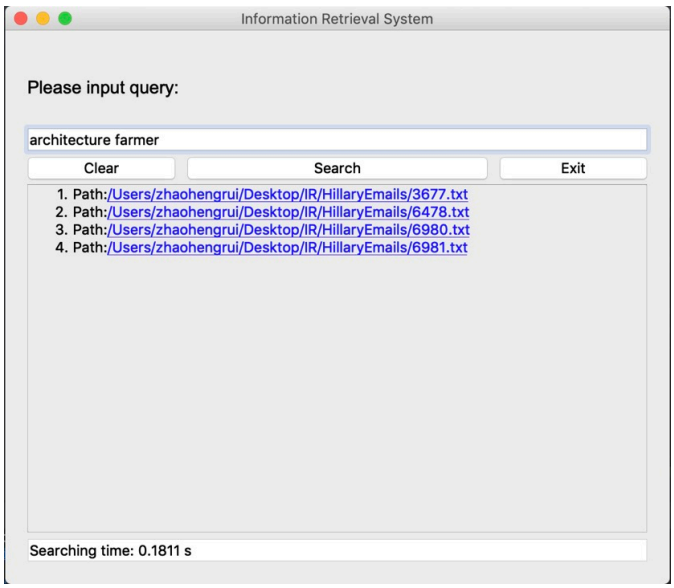
Call graph:



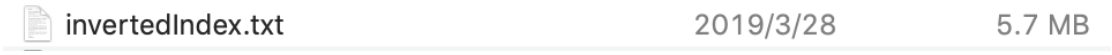
Size Consumption Result: It consumed 1156.8 MiB for the whole process which is slightly less than unoptimized one.

```
Henrys-MBP:IR zhaohengrui$ python -m memory_profiler IR_Optimized.py
Filename: IR_Optimized.py

Line #    Mem usage    Increment   Line Contents
================================================
   141    138.4 MiB    138.4 MiB       @profile
   142                                 def getResult(self, query):
   143    138.4 MiB      0.0 MiB           if os.path.isfile('invertedIndex.txt'):
   144                                         invertedIndex = self.loadInvertedIndex()
   145                                         pathList = invertedIndex['.']
   146                                     else:
   147   1183.6 MiB   1045.2 MiB               sortedToken, pathList = self.sortTokens()
   148   1151.2 MiB      0.0 MiB               invertedIndex = self.posting(sortedToken, pathList)
   149   1156.0 MiB      4.8 MiB               self.dumpInvertedIndex(invertedIndex)
   150   1156.0 MiB      0.0 MiB           queryList = query.split(' OR ')
   151   1156.0 MiB      0.0 MiB           result = []
   152   1156.0 MiB      0.0 MiB           for query in queryList:
   153   1156.0 MiB      0.0 MiB               queries = self.queryProcess(query)
   154   1156.0 MiB      0.0 MiB               matchedPostings = self.postingList(queries, invertedIndex)
   155   1156.0 MiB      0.0 MiB               result += self.postingMerge(matchedPostings)
   156                                     # result = sorted(list(set(result)), key=lambda k: int(re.findall(r'(\d+)\.txt', k)[0]))
   157   1156.0 MiB      0.0 MiB           finalResult = '<ol>'
   158   1156.0 MiB      0.0 MiB           for index in result:
   159   1156.0 MiB      0.0 MiB               finalResult = finalResult + "<li>Path:<a href='file://" + pathList[index] + "'>" + pathList[index] + '</a></li>'
   160   1156.0 MiB      0.0 MiB           finalResult = finalResult + '</ol>'
   161   1156.0 MiB      0.0 MiB           return finalResult
```

Searching time（query case=”farmer architecture”）: Average searching time is 0.1823s (in 5 times), which is slightly faster than unoptimized one.



Index Size:



The optimized invertedIndex size is 5.7 MB which is greatly reduced, helping to save memory resource.

8.4. Optimization Analysis

The results are very unexpected, the test results show that the system before the optimization has less memory consumption and only slightly longer running time. And for searching time, after measuring the search time of the same query and processing the average value, there is no large difference in search time before and after optimization. As for the index size, we found that the size of optimized invertedIndex is greatly reduced. We analyze the possible reasons as follows：

- After optimization, we use the index of pathlist to read and write. (eg. for the tokenization part, we just pass in the index instead of whole pat.). Path is a long string while index is a short string. So, it supposed to be faster to obtain the path after optimization. But according to the language features of Python itself, the list is read based on the offset instead of hash map, which result in lower index efficiency than dictionary and some other languages. Before optimization, we can get the path directly. However, after optimization, we get the real path through index of pathlist, which may take a certain amount of time. So, in a word, the time advantage of optimized system is offset by the time of reading through the indexed list/array of document paths.
- The email dataset we use in this case is relatively small and path length is relatively short, too. This may not cause significant difference or improvement between the systems before and after optimization.
- Because we reduce the repeated storage of long string paths, the resulting reverse index size is much smaller than before optimization.

9. Conclusion

Information Retrieval (IR) is the technique by which a sizably voluminous accumulation of data is represented, stored, and fetched for the purport of cognizance revelation as a result to a utilizer's request or query. The main aim of information retrieval model is to finding relevant knowledge-based information or a document that meet user needs". Through this basic IR system we build, search requests have been successfully fulfilled. Thus, we conclude a basic IR usually employ following processes:

- Preprocessing: Tokenize and turn each document into a list of tokens; produce a list of normalized tokens for indexing.
- Indexing process: the documents are represented in restate content form.
- Searching process: string matching to find the occurrences of a short string inside a (usually much longer) string/text. (Indexed string matching is employed in this case.)
-

References

1. Silva, Catarina, and Bernardete Ribeiro. “The importance of stop word removal on recall values in text categorization.” Neural Networks, 2003. Proceedings of the International Joint Conference on. Vol. 3. IEEE, 2003.
2. Klatt, Benjamin, Klaus Krogmann, and Volker Kuttruff. “Developing Stop Word Lists for Natural Language Program Analysis”. Proceedings of WSRE’14 (2014).
3. Balwinder Saini, Vikram Singh, Satish Kumar. “Information Retrieval Models and Searching Methodologies: Survey”. International Journal of Advance Foundation and Research in Science & Engineering (IJAFRSE) Volume 1, Issue 2, July 2014.
4. H. Dong, F.K. Husain and E. Chang. “A Survey in Traditional Information Retrieval Models”. IEEE International Conference on Digital Ecosystems and technologies, pp. 397-402,2008.
5. Rujia Gao, Danying Li, Wanlong Li, Yaze Dong. “Application of Full Text Search Engine Based on Lucene”. Advances in Internet of Things, October 2012, 2, 106-109.

-