Search Documents Algorithm
Information Retrieval

Session: 2021 – 2025

Submitted by:

M. Jawad Haider 2021-CS-149

Supervised by:

Dr. Syed Khaldoon Khurshid

Department of Computer Science

University of Engineering and Technology Lahore

Pakistan

Contents

1 Introduction

The search engine algorithm is designed to provide a reliable and efficient search mechanism for document files stored in a directory. Its purpose is to simplify the process of finding documents based on specific keywords or phrases, making it especially useful for environments that require fast access to large collections of text documents. This system pre processes document content, builds an index, and provides a user-friendly interface to perform quick searches by leveraging optimized data structures and techniques for high performance.

2 Goals

The primary goals of the searchEngine algorithm are:

1. Efficient Indexing: Build a fast and lightweight index of document content to support quick keyword searches.
2. Scalable Search Mechanism: Design the system to handle an increasing number of documents without significant performance degradation.
3. Flexible Query Processing: Support various types of search queries, such as case-insensitive matches, by normalizing content during indexing and querying.
4. Extensibility: Ensure the system can be easily extended to handle other file types, like PDFs and web pages, with minor adjustments.
5. Performance Optimization: Track execution time for search queries to identify bottlenecks and optimize the algorithm as needed.

3 Implementation Plan

The implementation of the searchEngine algorithm follows these steps:

1. Step 1: Define the structure of the search engine class, including attributes like indexes (a dictionary to hold keywords and their respective documents) and stopWords (common words to exclude from indexing).
2. Step 2: Develop methods for reading and processing files, including handling punctuation, case normalization, and word tokenization.
3. Step 3: Build the createIndexes method to iterate through all documents, preprocess content, and populate the indexes dictionary.

4. Step 4: Implement the searchDocuments method to allow users to search for keywords and retrieve a list of matching documents.

5. Step 5: Integrate and test the entire system, checking functionality with sample text files and ensuring accurate and fast retrieval.

4 Approach

- **Indexing Strategy**:
    1. Use a dictionary structure to map keywords to documents, allowing quick lookups.
    2. Keywords are extracted from both file names and file contents, providing a broad base for keyword searches.
- **Data Processing**:
    1. Convert document content to lowercase and remove unnecessary punctuation to standardize the indexing.
    2. Filter out common stop words to reduce the index size and focus only on meaningful keywords.
- **Data Structure**:
    1. The dictionary-based indexes structure enables efficient storage and retrieval of keyword-document mappings, ensuring O(1) average time complexity for lookups.

5 Backend Concept

5.1 Index Creation

The createIndexes function is responsible for building the index by processing each document as follows:

1. Read the document content and tokenize it into individual words.

2. Remove stop words and punctuation from tokens, normalizing them into lowercase to ensure case-insensitive matching.

3. Use the insertInDictionary method to add each unique, relevant keyword to the indexes dictionary, with each keyword mapped to a list of documents containing that keyword.

5.2 Search Functionality

The searchDocuments method processes search queries by:

1. Splitting and normalizing the search terms in the query.
2. Checking each term against the indexes dictionary to find matching documents.
3. Aggregating and returning results, displaying all documents that contain any of the queried keywords.
4. Logging execution time to facilitate performance tracking and optimization.

6 Data Flow Diagrams

6.1 Creation of Index:

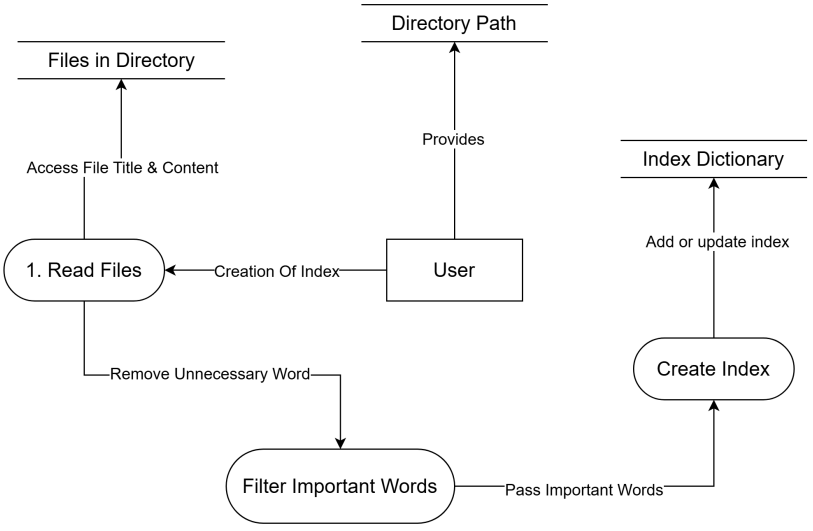

FIGURE 1: DFD: Creation of Index

6.2 Search Functionality:

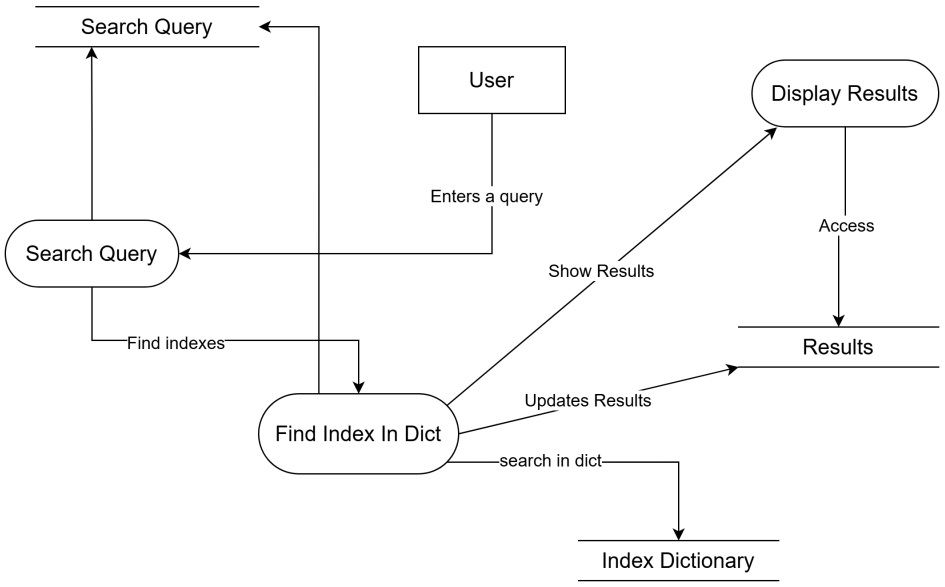

FIGURE 2: DFD: Search Functionality

7 Code

```
import os

class searchEngine:
    def __init__(self):
        self.indexes = {}
        # common Stop words and verb's suffixes to be filtered
   from text
        self.stopWords = {'the', 'is', 'and', 'in', 'to', 'of',
    'a', 'with', 'for', 'on', 'by', 'are', 'an', 'as', 'that'}
        self.verbSuffixes = {'ing', 'ed', 'ate', 'ify', 'ize',
   'en', 'fy'}

    # Add fileName in the dict against its indexedWord
    def addIndexInDict(self, indexWord, value):
        if indexWord not in self.indexes:
            self.indexes[indexWord] = [value]
        elif value not in self.indexes[indexWord]:
            self.indexes[indexWord].append(value)
```

```
    # Take filePath as parameter and returns it's content
    def readFile(self, filePath):
        try:
            with open(filePath, 'r') as file:
                content = file.read()
                return content
        except FileNotFoundError:
            displayMsg(f'The file {filePath} does not exist.')

        return None

    # This method read's the documentsFolderPath, reads the
files content and creates title and content's indexes
    def createIndexes(self, documentsFolderPath):
        try:
            for filename in os.listdir(documentsFolderPath):
                if filename.endswith('.txt'):
                    filePath = os.path.join(documentsFolderPath
, filename)
                    fileContent = self.readFile(filePath)

                    if not fileContent:
                        continue

                    self.createTitleAndContentsIndex(filename,
fileContent)
            return True
        except FileNotFoundError:
            displayMsg(f'The Directory {documentsFolderPath}
does not exist.')
            return False

    # Take fileName and it's content as parameters and creates
its index in the dictionary
    def createTitleAndContentsIndex(self, fileName,
contentWords):
        # Remove .txt if any fileName has
        if 'txt' in fileName:
            fileName = fileName.split('.')[0]
```

```
        filteredFileNamesList = self.filterImportantWords(
fileName)
        filteredContentWordsList = self.filterImportantWords(
contentWords)

        for indexWord in filteredFileNamesList +
filteredContentWordsList:
            self.addIndexInDict(indexWord, fileName)

    # This method take text as parameter and remove stop words
and words with verb suffixes
    def filterImportantWords(self, text):
        wordsList = []
        for word in text.split():
            word = word.lower().strip(',.')

            # Check if the word is a stop word
            if word in self.stopWords:
                continue

            # Check if the word ends with a verb suffix
            if any(word.endswith(suffix) for suffix in self.
verbSuffixes):
                continue

            wordsList.append(word)

        return wordsList

    # This method take query as parameter and returns a list of
    related file names
    def searchDocuments(self, query):
        if query == "" or not query:
            displayMsg('Enter a valid query!')
            return None

        if len(self.indexes) == 0:
            displayMsg('Create Indexes to search a document!')
            return None
```

```
        results = []
        for subQuery in query.split():
            subQuery = subQuery.lower()
            if subQuery in self.indexes:
                results.extend(self.indexes[subQuery])

        if len(results) == 0:
            displayMsg('No Document found against the query!')

        return results

    # Display Index and its values
    def lookupIndexes(self):
        if len(self.indexes) == 0:
            return displayMsg('Create indexes to look up!')

        print('\n-------------------------------------')
        for index in self.indexes:
            print(index, '=>', end=' ')
            for file in self.indexes[index]:
                print('\t', file, end=', ')
            print()
        print('-------------------------------------')

    # Display results of the searched query
    def displayResult(self, query, result):
        print('\n-------------------------------------')
        print('Searched Query: ', query)
        if len(result) == 0:
            print('No Documents found!')
        else:
            print('Results:')
            for index, file in enumerate(result):
                print('\t', index + 1, file)
        print('-------------------------------------')

# Helper function to display messages
def displayMsg(msg):
    print('\n-------------------------------------')
    print(msg)
```

```
    print('-------------------------------------')

if __name__ == '__main__':
    documentsFolderPath = './documents'
    searchEngineInstance = searchEngine()

    while True:
        print('\nMenu:')
        print('1. Change Documents Directory Path (default: ./
    documents)')
        print('2. Create Indexes')
        print('3. Lookup Indexes')
        print('4. Search FileName')
        print('5. Exit')

        choice = input('Enter your choice (1-5): ')

        if choice == '1':
            documentsFolderPath = input('\nEnter Documents
    Directory: ')
            displayMsg('Documents Directory Path updated
    successfully!')
        elif choice == '2':
            indexCreated = searchEngineInstance.createIndexes(
    documentsFolderPath)
            if indexCreated:
                displayMsg('Indexes created successfully!')
        elif choice == '3':
            searchEngineInstance.lookupIndexes()
        elif choice == '4':
            query = input('\nEnter Query: ')
            results = searchEngineInstance.searchDocuments(
    query)
            if results:
                searchEngineInstance.displayResult(query,
    results)
        elif choice == '5':
            print('Exiting program.')
            break
        else:
```

```
            displayMsg('Invalid choice. Please enter a number
    between 1 and 5.')
```

8 Conclusion

The searchEngine algorithm successfully implements an efficient and scalable system for document indexing and searching. It meets key objectives of fast retrieval, case-insensitive search, and extensibility for future improvements. By leveraging a dictionary-based indexing approach and filtering out stop words, the algorithm ensures a manageable index size while maximizing search relevance.

Future enhancements, such as supporting other file types (PDFs, web pages) and adding ranking algorithms for improved relevance, can make this tool even more robust and versatile.