Custom Search Engine 2

2020901160

fawaz al-ghzawi

2024

**Introduction**

This project focuses on designing and implementing a search engine for “.txt” files. It’s built to handle large datasets, provide information about searched queries, and suggest related queries. I made it by integrating various algorithms and concepts that we learned throughout this course.

**Objective**

the project aims to create an advanced and feature-rich search function that can index and search through text files effectively. It uses algorithms such as recursion, divide and conquer, dynammic programming, greedy algorithm, and graph algorithm to give you an effictive and relevent search results.  
This tool is designed to handle large volumes of data, correct misspelled queries, give you the line number and number of occerances, and give you related search suggestions

**Core Functionalities**

1. **Recursive File Indexing (Recursion and Backtracking)**
   1. **Traverse Directories:** The search engine navigates through directories and subdirectories to locate all text files.
   2. **Tokenization:** It breaks down text files into individual words, ignoring case and punctuation.
   3. **Inverted Indexing:** An inverted index is created, mapping each word to the files and specific line numbers where it appears.
2. **Divide and Conquer for Large File Processing (Divide and Conquer)**
   1. **Chunking Large Files:** Large text files are split into smaller, manageable chunks.
   2. **Parallel Processing:** Each chunk is processed separately to build partial indexes.
   3. **Merging Results:** The partial indexes from all chunks are combined into a single comprehensive index.
3. **Query Suggestion with Dynamic Programming (Dynamic Programming)**
   1. **Edit Distance Calculation:** The search engine calculates the similarity between the user's query and words in the index to suggest corrections for misspelled queries.
   2. **Caching Mechanism:** Intermediate results are stored to improve efficiency, especially when handling multiple queries.
4. **Greedy Algorithms for Ranking Results (Greedy Algorithms)**
   1. **Ranking by Frequency:** Search results are ranked based on how frequently the search terms appear in each file.
   2. **Proximity Scoring:** The engine considers how close the search terms are to each other within the text to determine relevance.
   3. **Simple Scoring Function:** A straightforward scoring system is implemented to rank the results effectively.
5. **Ranking by Frequency: Search results are ranked based on how frequently the search terms appear in each file.**
   1. **Proximity Scoring:** The engine considers how close the search terms are to each other within the text to determine relevance.
   2. **Scoring Function:** A straightforward scoring system is implemented to rank the results effectively based on the number of occurrences.

**Methodology**

This section details the implementation of each core functionality, explaining the role of specific algorithms and how they contribute to the overall functionality of the search engine.

1. **Recursive File Indexing “Recursion and Backtracking”**
2. **Directory Traversal:** A recursive function explores each directory and its subdirectories to locate all text files. This method ensures that every relevant text file is discovered without manually specifying paths.

def find\_text\_files(root\_dir):

text\_files = []

for entry in os.scandir(root\_dir):

if entry.is\_dir():

text\_files.extend(find\_text\_files(entry.path))

elif entry.is\_file() and entry.name.endswith('.txt'):

text\_files.append(entry.path)

return text\_files

1. **Tokenization:** For each text file found, the content is read, and words are extracted by removing punctuation and converting all characters to lowercase

def tokenize(text):

return re.findall(r'\b\w+\b', text.lower())

1. **Inverted Index Creation:** An inverted index is built where each word maps to a list of tuples containing the file path and the line number where the word appears. This structure allows for quick retrieval of files containing specific words.

def build\_inverted\_index(file\_list):

inverted\_index = {}

for file\_path in file\_list:

with open(file\_path, 'r', encoding='utf-8') as file:

for line\_num, line in enumerate(file, 1):

words = tokenize(line)

for word in words:

inverted\_index.setdefault(word, []).append((file\_path, line\_num))

return inverted\_index

1. **Divide and Conquer for Large File Processing**
2. **File Chunking:**Large text files are divided into smaller chunks to manage memory usage and enhance processing speed. Each chunk is processed independently to build partial inverted indexes.

def read\_file\_in\_chunks(file\_object, chunk\_size=124\*124):

# chunk size set to 124 x 124 for testing, it should be at least 1MB

while True:

data = file\_object.read(chunk\_size)

if not data:

break

yield data

1. **Parallel Processing:** Each chunk is processed to tokenize words and update a partial inverted index. This parallelism helps in efficiently handling large files by breaking down the problem into smaller parts
2. **Index Merging:** After processing all chunks, partial inverted indexes are merged into a single inverted index.

def merge\_indexes(partial\_indexes):

merged\_index = {}

for index in partial\_indexes:

for word, occurrences in index.items():

merged\_index.setdefault(word, []).extend(occurrences)

return merged\_index

1. **Query Suggestion with Dynamic Programming**
2. **Edit Distance Calculation:** The search engine calculates the edit distance between the user's query and each word in the inverted index. Words with the smallest edit distances are suggested as corrections for misspelled queries. It uses the **the levenshtein algorithm** to do so.

def edit\_distance(word1, word2):

m, n = len(word1), len(word2)

dp = [[0] \* (n + 1) for \_ in range(m + 1)]

for i in range(m + 1):

for j in range(n + 1):

if i == 0:

dp[i][j] = j

elif j == 0:

dp[i][j] = i

elif word1[i-1] == word2[j-1]:

dp[i][j] = dp[i-1][j-1]

else:

dp[i][j] = 1 + min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1])

return dp[m][n]

1. **Caching Mechanism:** To enhance efficiency, results of edit distance calculations are cached. This reduces redundant computations, especially for frequent or repeated queries.

cache = {}

def suggest\_corrections(words, vocabulary, cache):

suggestions = []

for word in words:

if word in vocabulary:

suggestions.append(word)

else:

if word in cache:

suggested\_word = cache[word]

else:

min\_distance = float('inf')

suggested\_word = word

for vocab\_word in vocabulary:

distance = edit\_distance(word, vocab\_word)

if distance < min\_distance:

min\_distance = distance

suggested\_word = vocab\_word

cache[word] = suggested\_word

suggestions.append(suggested\_word)

return suggestions

1. **Greedy Algorithms for Ranking Results**
2. **Frequency-Based Ranking:** The search engine ranks files based on how frequently the search terms appear. Files with higher frequencies are deemed more relevant.
3. **Proximity Scoring:** In addition to frequency, the engine considers the proximity of search terms within the text. Terms that appear closer together indicate higher relevance.
4. **Proximity Scoring:** In addition to frequency, the engine considers the proximity of search terms within the text. Terms that appear closer together indicate higher relevance.

def rank\_results(results, inverted\_index, words):

file\_scores = {}

# Extract unique file paths from results

unique\_files = set(file\_path for file\_path, \_ in results)

# Calculate score and track lines for each unique file

for file\_path in unique\_files:

if file\_path not in file\_scores:

file\_scores[file\_path] = {'score': 0, 'lines': set()}

for word in words:

if word in inverted\_index:

# Get all occurrences of the word in the current file

occurrences = [loc for loc in inverted\_index[word] if loc[0] == file\_path]

# Increment score by the number of occurrences

file\_scores[file\_path]['score'] += len(occurrences)

# Add line numbers to the set

for loc in occurrences:

file\_scores[file\_path]['lines'].add(loc[1])

# Convert to a list of tuples and sort by score in descending order

ranked\_results = sorted(file\_scores.items(), key=lambda x: x[1]['score'], reverse=True)

return ranked\_results

1. **Graph-Based Related Query Search**
2. **Proximity Scoring:** In addition to frequency, the engine considers the proximity of search terms within the text. Terms that appear closer together indicate higher relevance.

from collections import defaultdict

class WordGraph:

def \_\_init\_\_(self):

self.graph = defaultdict(dict)

def add\_cooccurrence(self, word1, word2, count=1):

if word2 in self.graph[word1]:

self.graph[word1][word2] += count

else:

self.graph[word1][word2] = count

def build\_graph(self, inverted\_index, window\_size=5):

# Organize occurrences by file and line for efficient processing

file\_line\_map = defaultdict(lambda: defaultdict(list)) # file -> line -> list of positions and words

for word, occurrences in inverted\_index.items():

for loc in occurrences:

file\_path, line\_num, pos = loc

file\_line\_map[file\_path][line\_num].append((pos, word))

# Iterate through each file and line to find co-occurring words

for file\_path, lines in file\_line\_map.items():

for line\_num, word\_positions in lines.items():

# Sort words in the line based on position

sorted\_words = sorted(word\_positions, key=lambda x: x[0])

words\_in\_line = [word for pos, word in sorted\_words]

# Slide a window over the words to find co-occurrences

for i in range(len(words\_in\_line)):

window = words\_in\_line[i+1:i+window\_size+1]

word1 = words\_in\_line[i]

for word2 in window:

if word1 != word2:

self.add\_cooccurrence(word1, word2)

def get\_related\_words(self, word, top\_n=5):

if word not in self.graph:

return []

related = sorted(self.graph[word].items(), key=lambda item: item[1], reverse=True)

return [w for w, count in related[:top\_n]]

**Code Documentation**

**File: indexing.py**

Function: find\_text\_files(root\_dir)

Description: Recursively traverses the specified root directory to locate all .txt files.

Parameters: root\_dir (str) – The path to the root directory.

Returns: list – A list of file paths to text files.

Function: tokenize(text)

Description: Converts a string of text into a list of lowercase words, removing punctuation.

Parameters: text (str) – The text to tokenize.

Returns: list – A list of tokenized words.

Function: build\_inverted\_index(file\_list)

Description: Constructs an inverted index mapping each word to the files and line numbers where it appears.

Parameters: file\_list (list) – A list of file paths to index.

Returns: dict – The inverted index.

**File: ranking.py**

Function: rank\_results(results, inverted\_index, words)

Description: Calculates and ranks the relevance of search results based on the frequency of search terms.

Parameters:

results (set) – A set of tuples containing file paths and line numbers.

inverted\_index (dict) – The inverted index.

words (list) – The list of search query words.

Returns: list – A sorted list of tuples containing file paths and their corresponding details (score and lines).

**File: query\_suggestion.py**

Purpose: Suggests corrections for misspelled queries.

Function: edit\_distance(word1, word2)

Description: Computes the edit distance between two words using dynamic programming.

Parameters:

word1 (str) – The first word.

word2 (str) – The second word.

Returns: int – The edit distance.

Function: suggest\_corrections(words, vocabulary, cache)

Description: Provides suggestions for misspelled words based on the closest matches in the vocabulary.

Parameters:

words (list) – The list of words to check.

vocabulary (set) – The set of valid words.

cache (dict) – A cache to store previously computed suggestions.

Returns: list – A list of suggested words.

**File: graph.py**

Purpose: Builds and manages a word relationship graph for related query suggestions.

Class: WordGraph

Attributes:

graph (defaultdict) – A graph where each key is a word, and the value is a dictionary of related words with their co-occurrence counts.

Method: add\_cooccurrence(word1, word2, count=1)

Description: Adds or updates the co-occurrence count between two words in the graph.

Parameters:

word1 (str) – The first word.

word2 (str) – The second word.

count (int) – The number of times the words co-occur.

Method: build\_graph(inverted\_index, window\_size=5)

Description: Constructs the word relationship graph based on word co-occurrence within a specified window size.

Parameters:

inverted\_index (dict) – The inverted index.

window\_size (int) – The number of words to consider for co-occurrence.

Method: get\_related\_words(word, top\_n=5)

Description: Retrieves the top related words for a given word based on co-occurrence frequency.

Parameters:

word (str) – The target word.

top\_n (int) – The number of related words to retrieve.

Returns: list – A list of related words.

**File: search\_engine.py**

Purpose: Implements the core search functionality, integrating indexing, querying, ranking, and related query suggestions.

Function: search(inverted\_index, query, vocabulary, cache, word\_graph, proximity=1)

Description: Handles user queries by searching the inverted index, ranking results, and suggesting related queries.

Parameters:

inverted\_index (dict) – The inverted index.

query (str) – The user's search query.

vocabulary (set) – The set of valid words.

cache (dict) – A cache for query suggestions.

word\_graph (WordGraph) – The word relationship graph.

proximity (int) – The proximity window size for ranking.

Function: perform\_search(inverted\_index, words, proximity

Parameters:

inverted\_index (dict) – The inverted index.

words (list) – The list of search query words.

proximity (int) – The proximity window size for ranking.

Function: main()

Parameters: None

Returns: None

**Code Structure and Flow**

**Initialization:**

The main() function starts by identifying all text files within the specified directory using find\_text\_files().

It builds the inverted index with build\_inverted\_index().

Initializes the WordGraph and constructs the word relationship graph using build\_graph().

**User Interaction:**

The application enters a loop, prompting the user to enter search queries.

Upon receiving a query, it checks for misspellings and suggests corrections using suggest\_corrections() if necessary.

**Search Execution:**

Valid queries are processed by perform\_search(), which retrieves common results and ranks them using rank\_results().

Ranked results are displayed to the user, showing file paths, line numbers, and the number of occurrences.

The WordGraph provides related query suggestions based on word co-occurrence.

**Termination:**

The user can exit the search engine by typing 'exit', which gracefully terminates the application.

**Test cases**

These tests cover various scenarios to ensure the search engine operates as intended.

We used 8 text files to test the program:

1. Test1.txt
2. Test2.txt
3. Test3.txt
4. Test4.txt
5. Test5.txt
6. Test6.txt
7. Big.txt (Contains a big text sample)
8. emptyText.txt

**Test Case 1: Basic Single-Word Search**

* **Objective:** Verify that the search engine can locate and rank files containing a single search term.
* **Query:** fawaz
* **Output:**

Welcome to the Custom Search Engine!

You can search for words or phrases.

Type 'exit' to quit the search engine.

Enter your search query: fawaz

Results for 'fawaz':

1. File: data/test3.txt

Lines: 4, 9, 11, 12, 13

Number of Occurrences: 9

2. File: data\Deeper folder\test1.txt

Lines: 3, 8, 10

Number of Occurrences: 6

Related queries:

again

Enter your search query:

**Test Case 2: Multi-Word Search With Ranking  
Objective:** Test the ranking mechanism based on word frequency and proximity**.  
Query:** “Driving car”  
**Output:**

Welcome to the Custom Search Engine!

You can search for words or phrases.

Type 'exit' to quit the search engine.

Enter your search query: driving car

Results for 'driving car':

1. File: data\test6.txt

Lines: 1

Number of Occurrences: 2

Related queries:

have, were, she, country, ventilator, driving, new, back, street, not, own, quite, 2007, temple, all, shrunk, fill, now, direction, house, reading, through, baker, there, acquaintance, rapidly, my, from, lexus, gs450h, let, straight, rod, who, those

Enter your search query:

**Test Case 3: Misspelled Query with Suggestions  
Objective:** Ensure that the search engine can suggest corrections for misspelled queries.  
**Query:** fawz  
**Output:**

Enter your search query: fwaz

No exact match found for 'fwaz'.

Did you mean: 'fawaz'?

Press 'y' to search with the suggested query, or any other key to skip: y

Results for 'fawaz':

1. File: data\test3.txt

Lines: 4, 9, 11, 12, 13

Number of Occurrences: 9

2. File: data\Deeper folder\test1.txt

Lines: 3, 8, 10

Number of Occurrences: 6

Related queries:

again

Enter your search query:

**Test Case 4: Query with No Matching Results  
Objective:** Confirm that the search engine appropriately handles queries with no matches.

**Query:** “unexpectedsearchqueryexample”

**Output:**

Enter your search query: unexpectedsearchqueryexample

No exact match found for 'unexpectedsearchqueryexample'.

Did you mean: 'unexpected'?

Press 'y' to search with the suggested query, or any other key to skip: n

No results found.

Enter your search query:

**Test-Case 4: Not Inputting Any Query  
Objective:** Confirm that the search engine appropriately handles a null query  
**Query: “”**

**Output:**

Welcome to the Custom Search Engine!

You can search for words or phrases.

Type 'exit' to quit the search engine.

Enter your search query:

Please enter a non-empty query.

Enter your search query:

**Test-Case 5: Inputting Invalid characters**

**Objective: See how the program reactes to inputting different characters  
Input:** #@!%

**Output:**

Enter your search query: #@!%

Invalid query. Please enter a valid search term.

Enter your search query**:**