Arab International University

Faculty of Informatics and Communication Engineering

Final Project Search Engine

Submitted to

Department of Informatics Engineering

Submitted by

Hamza al aedi

IRS

Under the Supervision of

DR. Lana Alabdallah

January 2025

Contents

IRS Project Documentation 3

Project Overview 3

This search engine is designed with three major components: a crawler, a preprocessor, and an indexer. It performs advanced searches, including word-based TF-IDF ranking and phrase searches using a positional inverted index. 3

Features 3

1. Crawler 3

Preprocessing: 3

Indexer: 3

System Design 4

Demo and Example Usage: 6

Execution Steps: 6

Code 6

Conclusion 9

IRS Project Documentation

Project Overview

This search engine is designed with three major components: a **crawler**, a **preprocessor**, and an **indexer**. It performs advanced searches, including word-based TF-IDF ranking and phrase searches using a positional inverted index.

Features

1. Crawler

The crawler fetches web pages and extracts links for further exploration. It ensures that the pages are parsed and prepared for indexing.

Key Functions:

Crawler:

- Fetches the content of a given URL.
- Handles errors gracefully.
- Extracts all links from a web page and ensures they are absolute URLs.
- Crawls web pages starting from a root URL up to a specified depth.

Preprocessing:

Prepares the web page content for indexing by normalizing, tokenizing, and removing stop words.

Key Functions:

- Converts text to lowercase.
- Tokenizes into words.
- Removes stop words.
- Stems words using the Porter Stemmer.

Indexer:

The engine builds two types of indexes:

- **TF-IDF Index**: Ranks documents based on term importance.
- **Positional Inverted Index**: Tracks the positions of terms in each document.

TF-IDF Index:

- Computes **Term Frequency (TF)** and **Inverse Document Frequency (IDF)**.
- Ranks documents based on TF-IDF scores.

Positional Inverted Index:

- Tracks the exact positions of terms in documents.
- Enables efficient phrase searches.

Key Functions:

- Updates both the TF-IDF and positional indexes for a given document.
- Computes the IDF values for all terms in the corpus.
- Displays the structure and contents of the positional inverted index.

System Design

Main Script Logic

1. **Prompts the user to input the root URL and crawl depth.**
2. **Instantiates the SearchEngine class.**
3. **Calls crawl to start crawling the web and building indexes.**
4. **Displays the positional index using display_positional_index.**
5. **Invokes run_demo for user interaction.**

Methods:

- **__init__** :**Purpose**: Initializes the search engine instance with essential data structures and objects like to_visit, visited, inverted_index, tfidf_index, positional_index, and NLP utilities (stop_words, stemmer).
- **Fetch()** :**Purpose**: Fetches the content of a given URL using the requests library.
  - **Parameters**:
  - url: The URL to fetch.**Returns**: The content of the webpage or an empty byte string if there's an error.
- **get_links(content, base_url)**: Extracts and normalizes links from the HTML content.

- **tokenize_and_preprocess(text)**: Tokenizes the text and applies preprocessing steps like stopword removal and stemming.
- **update_indexes() Purpose**: Updates the three indexes (inverted_index, positional_index, tfidf_index) for the given content and URL.
  - **Parameters**:
  - content: The HTML content of a webpage.
  - url: The URL of the page being processed.
  - **Returns**: None (updates indexes).

- **compute_idf()** :**Purpose**: Calculates the Inverse Document Frequency (IDF) for terms in theTF-IDF index and updates their scores.
  - **Parameters**: None.
  - **Returns**: None (modifies tfidf_index).

- **search_word_tfidf() Purpose**: Searches for a single word using the TF-IDF index and ranks results by score.
- **Parameters**:
  - **word**: The word to search.
  - **Returns**: A sorted list of tuples (url, tf-idf score).

- **search_phrase() Purpose**: Searches for an exact phrase using the positional inverted index.
  - **Parameters**:
  - phrase: The phrase to search.
  - **Returns**: A list of URLs where the phrase appears.

- **display_positional_index() Purpose**: Displays the positional inverted index for debugging or visualization.
  - **Parameters**: None.
  - **Returns**: None (prints positional index to the console)

- **Crawl()** :**Purpose**: Crawls the web starting from a root URL up to a specified depth, extracting links and updating indexes.
  - Parameters:
  - root_url: The starting URL for crawling.

- depth: The number of pages to crawl.
- Returns: None (updates visited and to_visit).

- **run_demo() Purpose**: Provides an interactive demo of the search engine, allowing the user to search for a word or a phrase.
  - **Parameters**: None.
  - **Returns**: None (handles user input and prints results).

Demo and Example Usage:

The user is prompted to choose from various search options:

1. **Search for a single word**: Retrieve all URLs containing the specified word.
2. **Search for a** phrase

The system then performs the search and displays the results, with an option to save them.

Example Workflow

1. Crawl the root URL (https://en.wikipedia.org/wiki/Web_search_engine) with a depth of 3.
2. Index the pages and preprocess the content.
3. Search for the word "…..".crawl
4. Save the results in a file.

Execution Steps:

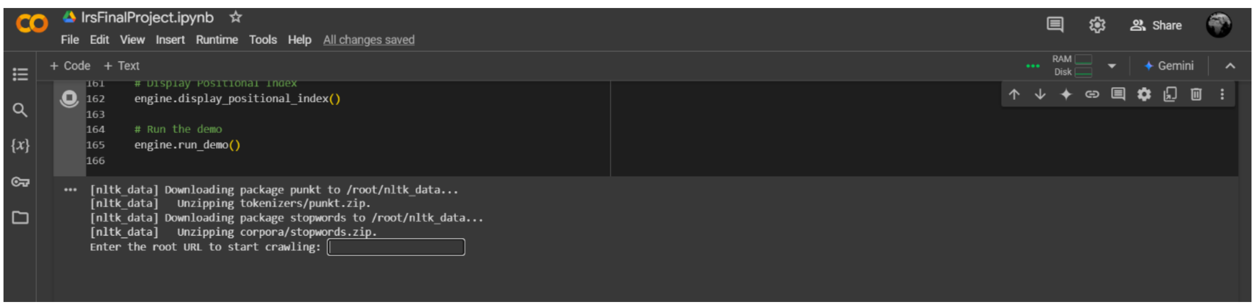
1. **Crawl**:
   - Specify the root URL and depth (5 e.g).
   - The system fetches the content and indexes the pages.
2. **Search for a Word**:
   - Choose the search option and input the search terms.
   - Enter a word to search: algorithm
     1. **View the Results**

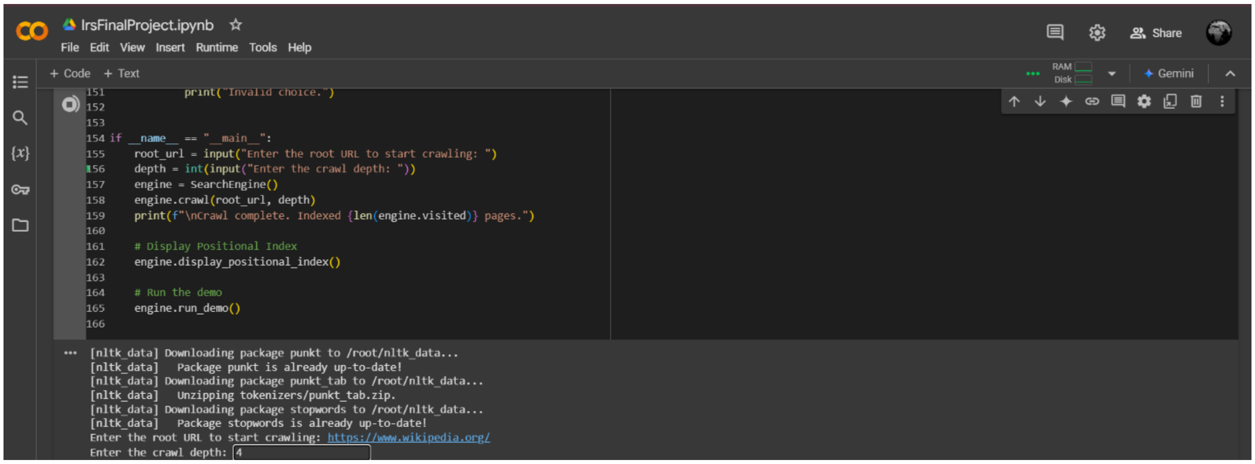
   - Verify the TF-IDF Index
   - The system returns the relevant URLs.
3. Search for a Phrase:
   - Choose the search option and input the search terms.
   - Enter a two words or more to search algorithm
   - **View the Results**

Dependencies

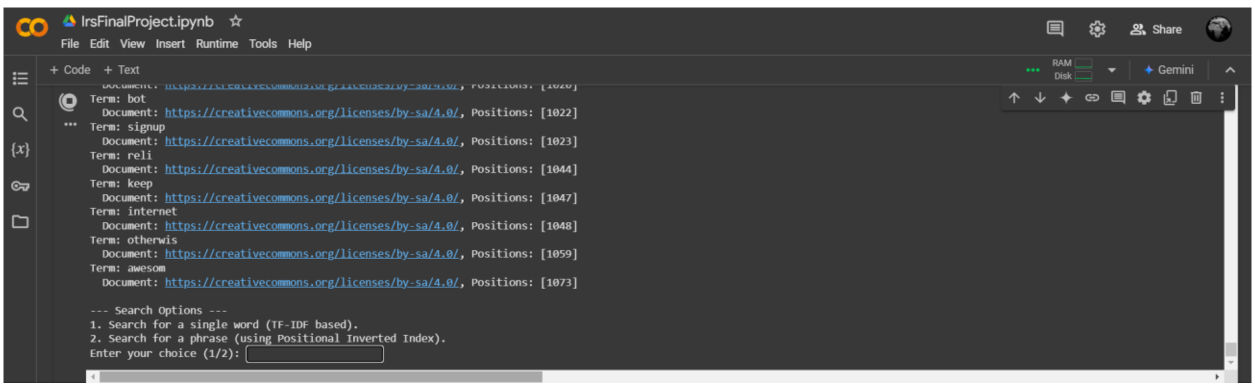
- **requests**: For fetching web content.
- **nltk**: For natural language processing tasks like tokenization, stopword removal, and stemming.
- **re**: For extracting links from the HTML content.
- **defaultdict**: For maintaining the inverted index.
- Download necessary NLTK data
    - download('punkt')
    - download('stopwords')

ScreenShots:

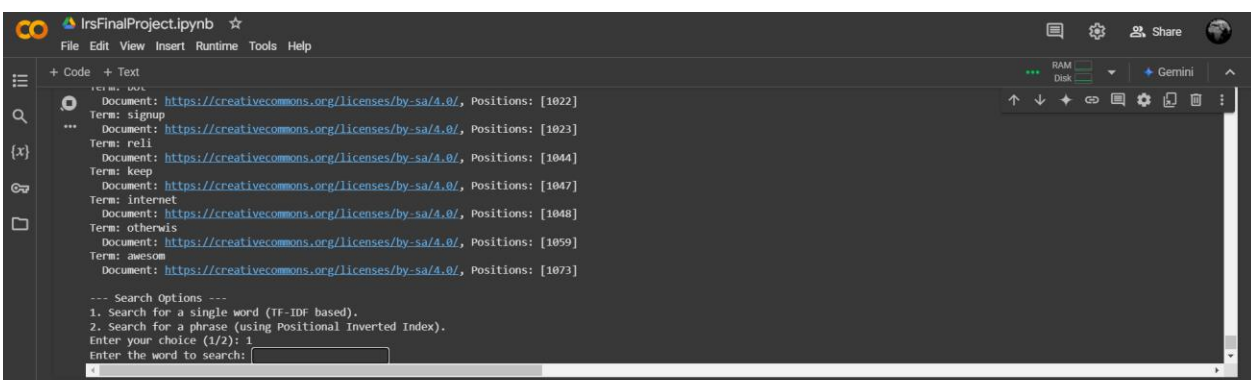




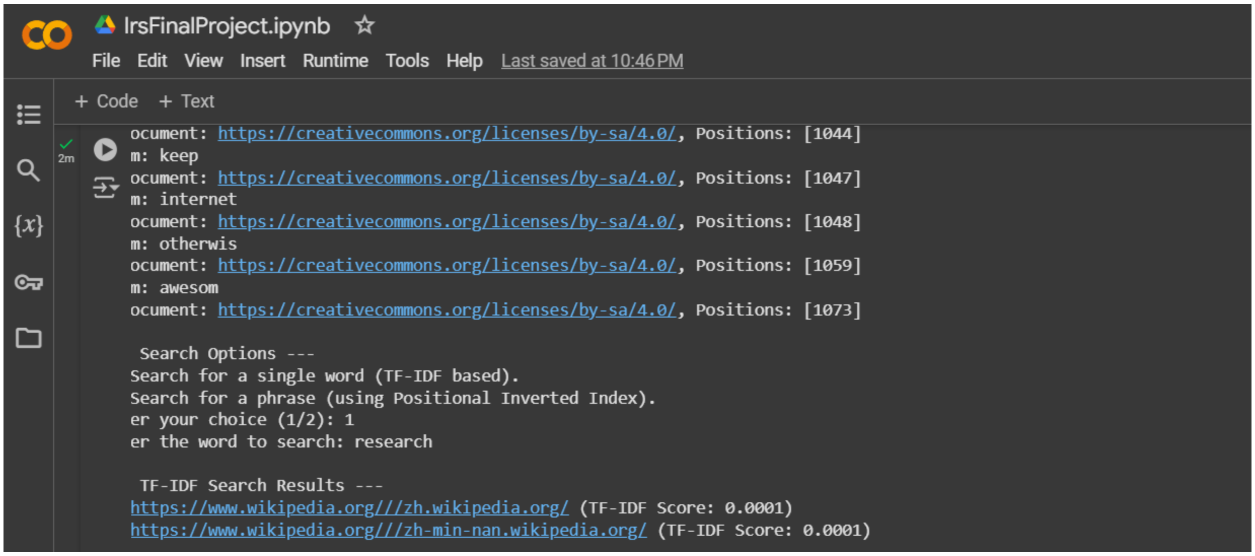
```
            print("Invalid choice.")


if __name__ == "__main__":
    root_url = input("Enter the root URL to start crawling: ")
    depth = int(input("Enter the crawl depth: "))
    engine = SearchEngine()
    engine.crawl(root_url, depth)
    print(f"\nCrawl complete. Indexed {len(engine.visited)} pages.")

    # Display Positional Index
    engine.display_positional_index()

    # Run the demo
    engine.run_demo()

[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package punkt_tab to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt_tab.zip.
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
Enter the root URL to start crawling: https://www.wikipedia.org/
Enter the crawl depth: 4
```



```
Term: bot
  Document: https://creativecommons.org/licenses/by-sa/4.0/, Positions: [1022]
Term: signup
  Document: https://creativecommons.org/licenses/by-sa/4.0/, Positions: [1023]
Term: reli
  Document: https://creativecommons.org/licenses/by-sa/4.0/, Positions: [1044]
Term: keep
  Document: https://creativecommons.org/licenses/by-sa/4.0/, Positions: [1047]
Term: internet
  Document: https://creativecommons.org/licenses/by-sa/4.0/, Positions: [1048]
Term: otherwis
  Document: https://creativecommons.org/licenses/by-sa/4.0/, Positions: [1059]
Term: awesom
  Document: https://creativecommons.org/licenses/by-sa/4.0/, Positions: [1073]

--- Search Options ---
1. Search for a single word (TF-IDF based).
2. Search for a phrase (using Positional Inverted Index).
Enter your choice (1/2):
```



```
  Document: https://creativecommons.org/licenses/by-sa/4.0/, Positions: [1022]
Term: signup
  Document: https://creativecommons.org/licenses/by-sa/4.0/, Positions: [1023]
Term: reli
  Document: https://creativecommons.org/licenses/by-sa/4.0/, Positions: [1044]
Term: keep
  Document: https://creativecommons.org/licenses/by-sa/4.0/, Positions: [1047]
Term: internet
  Document: https://creativecommons.org/licenses/by-sa/4.0/, Positions: [1048]
Term: otherwis
  Document: https://creativecommons.org/licenses/by-sa/4.0/, Positions: [1059]
Term: awesom
  Document: https://creativecommons.org/licenses/by-sa/4.0/, Positions: [1073]

--- Search Options ---
1. Search for a single word (TF-IDF based).
2. Search for a phrase (using Positional Inverted Index).
Enter your choice (1/2): 1
Enter the word to search:
```



Now finish taking screenshots of the final implementation and demo.

Code Implementation:

```
import requests
import re
from collections import defaultdict
import math
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
from nltk import download

# Download necessary NLTK data
download('punkt')
download('stopwords')

class SearchEngine:
    def __init__(self):
        self.to_visit = []
        self.visited = set()
        self.inverted_index = defaultdict(list)
        self.tfidf_index = defaultdict(lambda: defaultdict(float))
        self.positional_index = defaultdict(lambda: defaultdict(list))
        self.stop_words = set(stopwords.words('english'))
        self.stemmer = PorterStemmer()
        self.document_lengths = {}

    def fetch(self, url):
        try:
            res = requests.get(url)
            if res.status_code == 200:
                return res.content
        except requests.RequestException as e:
            print(f"Error fetching {url}: {e}")
        return b''

    def get_links(self, content, base_url):
        links = re.findall(r'<a href="([^"]+)"', str(content))
        for link in links:
            if link.startswith('/'):
                link = base_url + link
            if link.startswith('http') and link not in self.visited:
                self.to_visit.append(link)
```

```
    def tokenize_and_preprocess(self, text):
        tokens = word_tokenize(text.lower())
        return [
            self.stemmer.stem(word)
            for word in tokens if word.isalnum() and word not in
self.stop_words
        ]

    def update_indexes(self, content, url):
        text = content.decode('utf-8', errors='ignore')
        tokens = self.tokenize_and_preprocess(text)
        self.document_lengths[url] = len(tokens)

        # Positional Index
        for position, token in enumerate(tokens):
            self.positional_index[token][url].append(position)

        # Inverted Index
        for token in set(tokens):
            self.inverted_index[token].append(url)

        # TF-IDF Index
        token_counts = defaultdict(int)
        for token in tokens:
            token_counts[token] += 1
        for token, count in token_counts.items():
            tf = count / len(tokens)
            self.tfidf_index[token][url] = tf

    def compute_idf(self):
        total_docs = len(self.visited)
        for token, doc_dict in self.tfidf_index.items():
            idf = math.log(total_docs / len(doc_dict))
            for url in doc_dict:
                self.tfidf_index[token][url] *= idf

    def search_word_tfidf(self, word):
        word = self.stemmer.stem(word.lower())
        return sorted(
            self.tfidf_index.get(word, {}).items(),
            key=lambda x: x[1],
            reverse=True
        )
```

```
    def search_phrase(self, phrase):
        tokens = self.tokenize_and_preprocess(phrase)
        if not tokens:
            return []

        result_urls = set(self.positional_index[tokens[0]])
        for token in tokens[1:]:
            result_urls.intersection_update(self.positional_index[token])

        phrase_results = []
        for url in result_urls:
            positions = [
                self.positional_index[token][url] for token in tokens
            ]
            for start_pos in positions[0]:
                if all((start_pos + i) in positions[i] for i in range(1,
len(tokens))):
                    phrase_results.append(url)
                    break
        return phrase_results

    def display_positional_index(self):
        print("\n--- Positional Inverted Index ---")
        for term, doc_positions in self.positional_index.items():
            print(f"Term: {term}")
            for doc, positions in doc_positions.items():
                print(f"  Document: {doc}, Positions: {positions}")

    def crawl(self, root_url, depth):
        self.to_visit.append(root_url)
        while len(self.visited) < depth and self.to_visit:
            current_url = self.to_visit.pop(0)
            if current_url in self.visited:
                continue
            content = self.fetch(current_url)
            if content:
                self.visited.add(current_url)
                self.get_links(content, root_url)
                self.update_indexes(content, current_url)
        self.compute_idf()

    def run_demo(self):
        print("\n--- Search Options ---")
        print("1. Search for a single word (TF-IDF based).")
        print("2. Search for a phrase (using Positional Inverted Index).")
```

```
        choice = int(input("Enter your choice (1/2): "))
        if choice == 1:
            word = input("Enter the word to search: ")
            results = self.search_word_tfidf(word)
            if results:
                print("\n--- TF-IDF Search Results ---")
                for idx, (url, score) in enumerate(results, 1):
                    print(f"{idx}. {url} (TF-IDF Score: {score:.4f})")
            else:
                print("\nNo results found for the word.")
        elif choice == 2:
            phrase = input("Enter the phrase to search: ")
            results = self.search_phrase(phrase)
            if results:
                print("\n--- Phrase Search Results (Positional Index) ---
")

                for idx, url in enumerate(results, 1):
                    print(f"{idx}. {url}")
            else:
                print("\nNo results found for the phrase.")
        else:
            print("Invalid choice.")


if __name__ == "__main__":
    root_url = input("Enter the root URL to start crawling: ")
    depth = int(input("Enter the crawl depth: "))
    engine = SearchEngine()
    engine.crawl(root_url, depth)
    print(f"\nCrawl complete. Indexed {len(engine.visited)} pages.")

    # Display Positional Index
    engine.display_positional_index()

    # Run the demo
    engine.run_demo()
```

Cosine Similarity and Its Role in Information Retrieval Systems

Definition

Cosine similarity measures the cosine of the angle between two vectors in a multi-dimensional space. In information retrieval (IR) systems, these vectors represent text documents or query terms. The cosine similarity value ranges from -1 to 1, where:

- 1 indicates identical vectors (completely similar).
- 0 indicates orthogonal vectors (no similarity).
- -1 implies completely opposite vectors.

Mathematical Formula

Given two vectors AAA and BBB:

Cosine Similarity=A·B‖A‖×‖B‖\text{Cosine Similarity} = \frac{A \cdot B}{\|A\| \times \|B\|}Cosine Similarity=‖A‖×‖B‖A·B

Where:

- A·BA \cdot BA·B is the dot product of vectors AAA and BBB.
- ‖A‖\|A\|‖A‖ and ‖B‖\|B\|‖B‖ are the magnitudes (Euclidean norms) of the vectors.

Cosine Similarity in Information Retrieval

1. Vector Space Model (VSM):
   Documents and queries are represented as vectors of term weights, typically computed using TF-IDF scores. Each vector component represents the relevance or frequency of a specific term.
2. Relevance Scoring:
   When a user submits a query, cosine similarity determines how closely a document matches the query by computing the angle between their vectors. Higher similarity scores indicate greater relevance.

Example in Practice

- Query: "Artificial Intelligence"
- Document A contains "Artificial Intelligence applications are growing."

- **Document B mentions "Quantum mechanics theories."**

Cosine similarity will likely assign Document A higher score due to the overlap in relevant terms with the query, ranking it higher in the search results.

Advantages

- **Efficient for Sparse Data: Works well even if documents and queries have many zero-valued dimensions.**
- **Scale-Invariant: The similarity score is unaffected by document length.**

Applications

- **Search Engines: Google and other search platforms use cosine similarity for ranking documents.**
- **Recommendation Systems: Suggest items by comparing user preferences or item features.**
- **Document Clustering: Groups similar documents for text analysis or classification tasks.**

In summary, cosine similarity is a foundational tool in IR systems, helping efficiently rank and retrieve the most relevant documents by comparing vectorized representations of text data.

Conclusion

This project provides an efficient way to crawl the web, build an inverted index, and search through it with various logical conditions. The text preprocessing steps ensure that the results are relevant and focused on meaningful terms.
