Implement Structure Guided-Browsing

Instructor

Prof. Syed Khaldoon Khurshid

Submitted by

M. Shahzaib Ijaz 2021-CS-75

Department of Computer Science
University of Engineering and Technology
Lahore Pakistan

Contents

1 Introduction

The exponential growth of digital information has led to a significant increase in the need for effective search engines. Researchers, educators, and professionals often require access to relevant information from vast repositories of documents. This project aims to address this need by developing a search engine tailored for research articles. By leveraging the set-theoretic model, the search engine prioritizes the relevance of search results, making it easier for users to find the information they need. The added feature of structured guided browsing provides users with a more interactive experience by allowing them to navigate research articles with an intuitive Table of Contents, improving content accessibility.

2 Objective

The objective of this project is to create a search engine capable of indexing research articles and providing users with a robust search experience while integrating structured guided browsing. Users should be able to:

- Search for specific words within documents.
- Find documents related to specific topics.
- Receive keyword suggestions based on indexed data.
- Navigate through research articles efficiently with the structure-guided browsing feature, enabling them to explore sections and subsections via an interactive Table of Contents.

3 Structured Guided Browsing

Structured Guided Browsing is a feature that enhances user experience by providing an interactive, hierarchical view of research articles. It allows users to navigate through sections and subsections of the document easily by displaying a Table of Contents (TOC) on the left side of the reader interface. Each section and subsection can be clicked to instantly navigate to the corresponding content, enabling users to focus on specific parts of the article without manually scrolling. This feature enhances content accessibility and improves the overall efficiency of navigating long and complex research papers.

3.1 Structured Guided Browsing Overview

Definition: The Structured Guided Browsing functionality provides a dynamic and interactive structure to a research article, where the content is divided into sections and subsections. These sections are represented in a Table of Contents, allowing users to directly access any part of the document. The content is displayed in a split-screen format, with the Table of Contents on the left and the article's content on the right. This setup allows for seamless browsing and enhanced user interaction.

3.2 Structured Guided Browsing Features

Explanation: The key features of the Structured Guided Browsing include:

- **Table of Contents**: A hierarchical structure on the left side, displaying all sections and subsections of the document.
- **Clickable Navigation**: Users can click on any section or subsection in the TOC, instantly navigating to that part of the document.
- **Real-Time Content Display**: As users click on sections or subsections, the content corresponding to that part is displayed in real-time on the right side of the screen.
- **Seamless Navigation**: Allows users to quickly move between different parts of a research article without the need for scrolling or searching.

3.3 User Interaction in Structured Guided Browsing

Procedure:

- The research article is parsed into sections and subsections, and these are displayed in a Table of Contents.
- Users can click on any item in the Table of Contents to directly jump to that section or subsection of the document.
- The content related to the clicked section or subsection is displayed on the right side of the screen, enabling users to read and explore the document more efficiently.

3.4 Example

Example: In a research paper on "climate change," the Table of Contents might include sections such as "Introduction," "Methodology," and "Results." Each of these sections would contain further subsections like "Impact of Climate Change" or "Data Analysis." Clicking on any subsection, such as "Impact of Climate Change," would immediately take the user to that part of the paper, enhancing readability and usability, especially for longer documents.

4 Key Features

- **Keyword Suggestions**
  - Detects typos in user queries and provides corrected keyword suggestions.
  - Enhances user experience and ensures accurate searches.
- **Result Details**
  - Displays the following for each query:
    - **Title** of the document.
    - **Author Name(s)**.
    - **Snippet** of the document's context.
- **Filters**
  - Allows users to refine their search by applying filters:
    - **Search by Title**: Matches query against document titles.
    - **Search by Author**: Filters results based on author names.
    - **Search by Context (Default)**: Searches within the document's full content.
- **Ranking**
  - Ranks documents based on **set-theoretic model**.
  - Ensures relevant documents appear higher in search results.

5 Implementation

5.1 Document Handling

5.1.1 extract_text_from_documents function

1. Reads a .docx file using the Document class from the python-docx library.
2. Extracts key sections:
   (a) **Title:** The first non-empty paragraph is assumed to be the document's title.
   (b) **Author:** A paragraph starting with "Author:" is processed to retrieve the author's name.
   (c) **Content:** All other text is stored as the main content.
3. Returns a structured dictionary with title, author, and content fields.

app.py

```python
# Function to extract doc_data from specific docx file
    def extract_text_from_documents(self, file_path):
        document = Document(file_path)
        doc_data = {'title': "", 'author': "", "content": ""}

        paragraphs = (p.text.strip() for p in document.paragraphs if p.text.strip())
        for text in paragraphs:
            if not doc_data['title']:
                doc_data['title'] = text
            elif text.startswith("Author:"):
                doc_data['author'] = text[7:].strip()
            else:
                doc_data['content'] += f"{text}"

        return doc_data

    # Function to extract all content from the Document folder
    def document_extractor(self):
        documents = []

        for file_name in os.listdir(self.folder_path):
            file_path = os.path.join(self.folder_path, file_name)

            if not (os.path.isfile(file_path) and file_name.endswith('.docx')):
                continue

            doc_data = self.extract_text_from_documents(file_path)
            doc_data['file_name'] = file_path

            documents.append(doc_data)

        return documents
```

Figure 1: Document Handling In Search Engine.

5.1.2 document_extractor function

1. Iterates through all files in a specified folder (self.folder_path).
2. Filters for valid .docx files.
3. Calls extract_text_from_documents to extract data from each file. Adds the file path for traceability and appends the structured data to a list.
4. Returns a collection of extracted data for all .docx files in the folder.

5.2 Text Preprocessing

5.2.1 stem function

1. A list of common suffixes (suffixes) is defined.
2. Iterates over the suffixes and checks if the word ends with one of them using word.endswith(suffix).
3. If a match is found, it removes the suffix by slicing the string up to -len(suffix) and returns the modified word.
4. If no suffix matches, the original word is returned unchanged.

Figure 2: Text Preprocessing In Search Engine.

5.2.2 preprocess_text function

1. Converts the text to lowercase using .lower().
2. Removes punctuation with .translate(translator).
3. Splits the text into individual words using .split().
4. Filters out stopwords and applies the stem function to each remaining word.
5. Returns the processed words as a list.

5.3 Partial Matching

5.3.1 calculate_similarity function

1. Uses the SequenceMatcher class from the difflib module to compute the similarity ratio between two terms.
2. The None parameter is for ignoring junk elements (which isn't needed here).
3. The function returns a similarity score (between 0 and 1) representing how similar term1 and term2 are.

5.3.2 find_partial_matches function

1. Iterates through a list of terms and compares each term with the query_term using the calculate_similarity function.
2. For each comparison, it calculates the similarity score.
3. If the similarity score is greater than or equal to the provided threshold (default is 0.5), the term is added to the matches dictionary with its similarity score.
4. Finally, it returns the matches dictionary, which contains terms that have a sufficient similarity to the query_term.

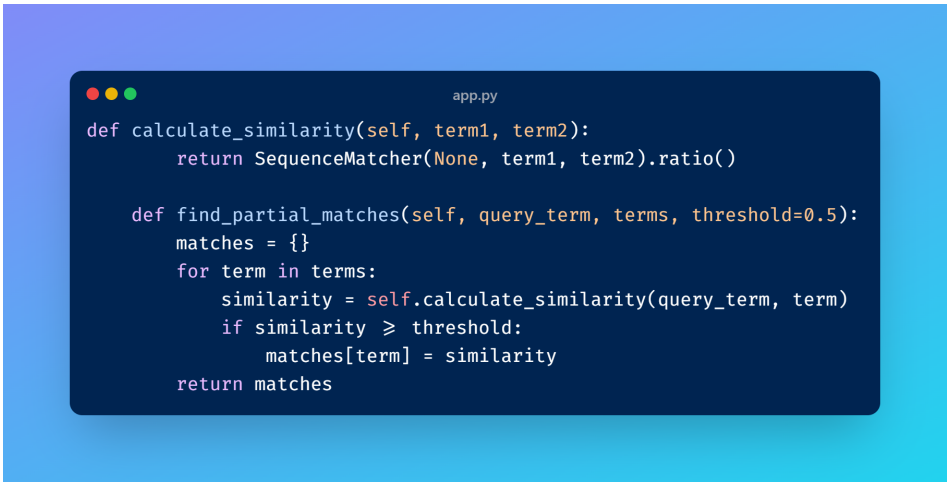

Figure 3: Partial Matching In Search Engine.

5.4 Indexing

1. The function creates an index of words and n-grams (phrases) for a list of documents.
2. The input is a list of documents, and the output is an index and a count of search terms.
3. It initializes an empty index and a dictionary to count word frequencies.
4. For each document, it preprocesses the content by converting it to lowercase and removing unwanted characters.
5. It indexes each unique word in the document, adding the document ID to the index if the word appears.
6. It also counts how many times each word or n-gram (phrases of 1 to 4 words) appears in the documents.
7. The function returns the index of words and n-grams along with the term frequency counts for all terms.

Figure 4: Indexing In Search Engine.

5.5 Search Query

1. The function preprocesses the query to get individual terms.
2. It calculates the weight for each term based on its frequency in the query.
3. For each term, if it exists in membership_degrees, it checks if the document has a high enough membership value to add a score to that document.
4. If the term doesn't exist in membership_degrees, it finds similar terms and adds a weighted score for documents where those similar terms have a high enough membership.
5. After scoring, the function normalizes the document scores by dividing them by the highest score.
6. The function returns the normalized document scores based on the fuzzy query.

app.py

```
def process_fuzzy_query(self, query, membership_degrees, fuzziness_threshold=0.30):
        query_terms = self.preprocess_text(query)
        doc_scores = defaultdict(float)
        term_weights = {term: query_terms.count(term) for term in query_terms}

        for term in query_terms:
            if term in membership_degrees:
                for doc_id, membership in membership_degrees[term].items():
                    if membership ⩾ fuzziness_threshold:
                        doc_scores[doc_id] += membership * term_weights[term]
            else:
                partial_matches = self.find_partial_matches(term, membership_degrees.keys(), fuzziness_threshold)
                for match_term, similarity in partial_matches.items():
                    for doc_id, membership in membership_degrees[match_term].items():
                        if membership * similarity ⩾ fuzziness_threshold:
                            doc_scores[doc_id] += membership * term_weights[term] * similarity

        max_score = max(doc_scores.values()) if doc_scores else 1
        for doc_id in doc_scores:
            doc_scores[doc_id] /= max_score

        return doc_scores
```

Figure 5: Searching In Search Engine.

5.6 Context Hierarchy Extraction

1. The function reads a DOCX file and extracts the title, author, abstract, and sections.
2. It processes each paragraph of the document.
3. The title is assumed to be the first non-heading paragraph that isn't empty.
4. The author is extracted from paragraphs containing the word "author".
5. The abstract starts when a paragraph labeled "Abstract" is found and ends when a blank line is encountered.
6. Sections are identified by paragraphs with heading styles. Heading level 1 indicates a main section, while heading level 2 indicates a subsection.
7. The content under each heading is collected and added to the corresponding section or subsection.
8. The function returns a dictionary with the extracted information, including the title, author, abstract, and structured content.

app.py

```
def extract_docx_content(docx_file):
    doc = Document(docx_file)

    content = {
        "title": "",
        "author": "",
        "abstract": "",
        "sections": {}
    }
    current_section = None
    current_subsection = None

    # Flags for extracting title, author, and abstract
    title_extracted = False
    extracted_author = False
    extracted_abstract = False
    abstract_start = False  # Initialize the abstract_start flag

    for para in doc.paragraphs:
        # Extract title: Assume it's the first non-empty paragraph that's not a heading
        if not title_extracted and para.text.strip() and not para.style.name.startswith('Heading'):
            content["title"] = para.text.strip()
            title_extracted = True
            continue  # Skip the title from further processing

        # Extract author: Assume the author's name follows a specific format or keyword
        if not extracted_author and "author" in para.text.lower():
            content["author"] = para.text.strip().replace("Author:", "").replace("author:", "").strip()
            extracted_author = True
            continue

        # Extract abstract (assuming it's labeled as "Abstract" or a similar heading)
        if not extracted_abstract and para.text.strip().lower() == "abstract":
            abstract_start = True  # Start extracting the abstract
            continue
        elif abstract_start:
            content["abstract"] += para.text.strip() + "\n"
            if para.text.strip() == "":
                abstract_start = False  # End of abstract when a blank line is encountered
            continue

        # Check if the paragraph is a heading (usually marked as bold)
        if para.style.name.startswith('Heading'):
            heading_level = int(para.style.name.split()[-1])  # Extract heading level
            heading_text = para.text.strip()

            if heading_level == 1:  # Main section
                current_section = heading_text
                content["sections"][current_section] = ""  # Initialize as an empty string for the section content
                current_subsection = None
            elif heading_level == 2:  # Subsection
                current_subsection = heading_text
                if current_section:  # Ensure current_section is initialized
                    # If the section is not a dictionary, convert it into one
                    if isinstance(content["sections"][current_section], str):  # Main section is currently a string
                        content["sections"][current_section] = {}  # Convert it into a dictionary for subsections
                    content["sections"][current_section][current_subsection] = ""  # Initialize empty string for subsection
content
            else:
                continue  # For further levels, you can extend logic here if needed

        else:
            # Otherwise, it's content under the current section or subsection
            if current_section:
                if current_subsection:
                    # Append content to the subsection (it's a string)
                    content["sections"][current_section][current_subsection] += para.text.strip() + "\n"
                else:
                    # Append content to the main section (it's a string)
                    content["sections"][current_section] += para.text.strip() + "\n"

    return content
```

Figure 6: Hierarchy Maker In Search Engine.

5.7 Keyword Suggestions

1. Converts the user's input to lowercase: This makes the search case-insensitive, so the function works regardless of whether the user types in uppercase or lowercase.

2. Finds terms that start with the user's input: It checks all available search terms and selects the ones that begin with the text the user has entered so far.

3. Sorts the matching terms by popularity: The terms are ranked based on how often they have been searched or used, with the most popular ones appearing first.

4. Limits the suggestions: It only shows a fixed number of the top matches, making the suggestions manageable and easy to read.

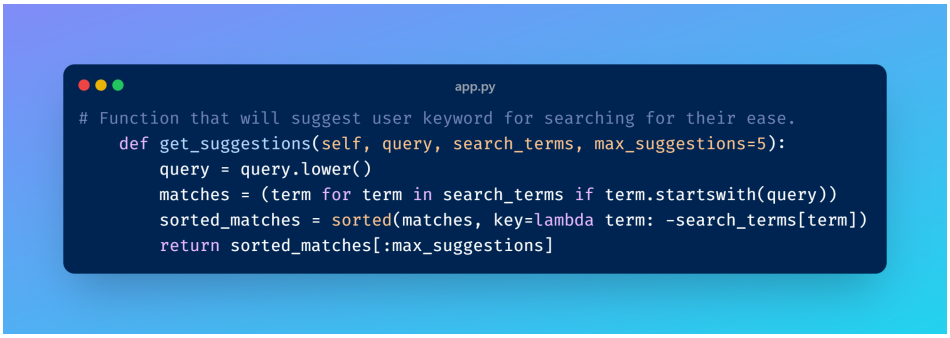

Figure 7: Keyword In Search Engine.

6 API Endpoints

6.1 `/suggestions`

- **Method**: `GET`
- **Description**: Provides keyword suggestions for the input query.
- **Parameters**: `query` (string)
- **Response**:

```
["keyword1", "keyword2", "keyword3"]
```

6.2 `/api/v8/search/fulltext`

- **Method**: `POST`
- **Description**: Searches the full text of documents.
- **Parameters**: `query` (string)
- **Response**:

```
[
    {
        "title": "Title of Document",
        "author": "Author Name",
        "snippet": "Snippet of content...",
        "file_path": "path/to/document.docx"
    }
]
```

6.3 /api/v8/search/title

- **Method**: POST
- **Description**: Searches for documents by title.
- **Parameters**: query (string)
- **Response**:

```
[
    {
        "title": "Title of Document",
        "author": "Author Name",
        "snippet": "Snippet of content...",
        "file_path": "path/to/document.docx"
    }
]
```

6.4 /api/v8/search/author

- **Method**: POST
- **Description**: Searches for documents by author name.
- **Parameters**: query (string)
- **Response**:

```
[
    {
        "title": "Title of Document",
        "author": "Author Name",
        "snippet": "Snippet of content...",
        "file_path": "path/to/document.docx"
    }
]
```

7 Data Flow Diagram

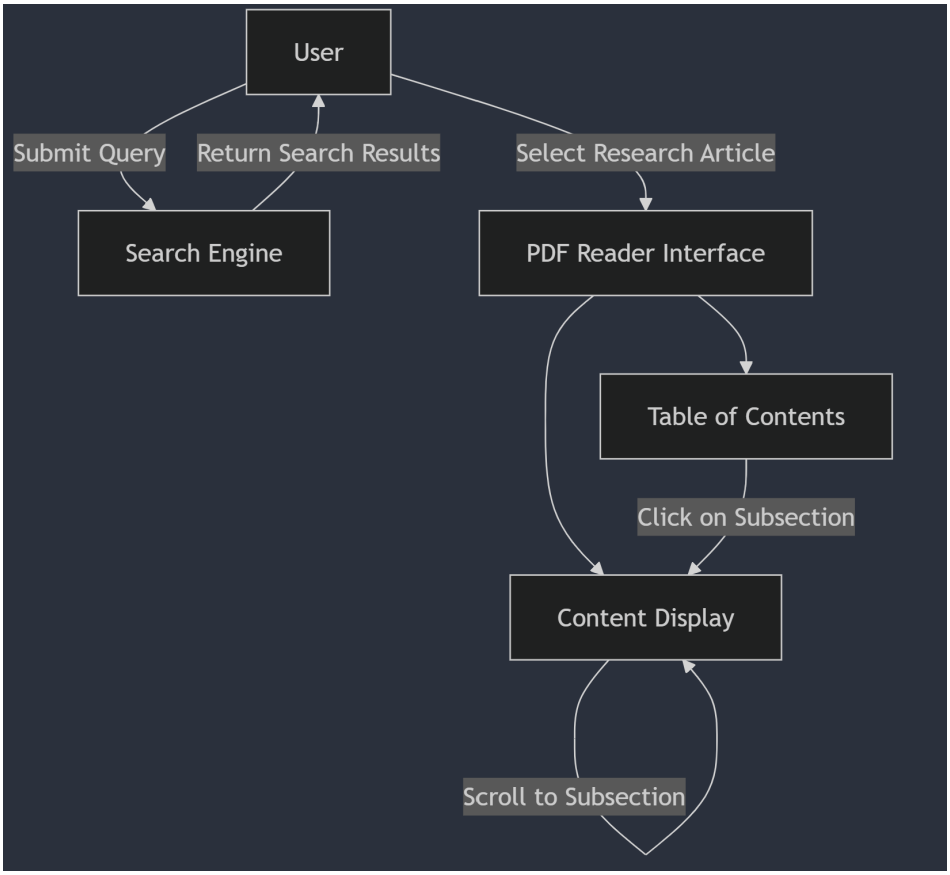

Figure 8: Structured Guided Data Flow Diagram.

8 Results

- The search engine successfully indexes and ranks research articles using the set-theoretic model, ensuring more diverse and relevant search results.
- Keyword suggestions guide users in query formulation, improving the precision of search results.
- API endpoints provide flexible search options for full text, titles, and authors, supporting multiple retrieval models.
- The structure-guided browsing feature enhances the user experience by displaying research articles with a Table of Contents on the left side and content on the right, allowing users to quickly navigate to specific sections or subsections by clicking on them.

9 Future Enhancements

1. Implement word embeddings for ranking to improve relevance, allowing for semantic-based search.
2. Add support for additional file formats, such as PDF, ensuring the search engine is versatile across various document types.
3. Introduce advanced filtering options for search results (e.g., publication date, author affiliation) to refine search outcomes based on user preferences.