COMP4321 Final Project

Group 7

CHAN Sheung Yin, CHEUNG Chi Shan Jennifer, SHING Ho Hin

SPRING 2025

Content

1. Overall Design

a. requirements.txt

- This file includes the list of required Python packages and their versions. After activating the virtual environment, the user can install the necessary packages by using the command "*pip install—r requirements.txt.*"

b. stopwords.txt

- This file lists all the common stopwords. Some stopwords are removed from the queries during indexing.

c. spider.py

- Contains functions to retrieve webpages using Requests and BeautifulSoup.
- Uses a breadth-first search (BFS) strategy to traverse hyperlinks and avoid cycles.
- Generates unique page IDs (using CRC32) and stores page information (e.g., title, modification date, content size) in the database.

d. indexer.py

- Reads the crawled page data from the database and extracts text from both the page body and title.
- Applies stopword filtering (using the provided stopwords.txt) and stemming (via NLTK's PorterStemmer).
- Constructs two inverted indexes: one for page bodies and one for titles.
- Builds forward indexes (aggregated word counts) and calculates ranking scores by creating and iteratively updating an adjacency matrix (simulating a PageRank-like mechanism).

e. retrieval.py

- Provides functions for converting a user query into numeric vectors and comparing them with document vectors using cosine similarity.
- Implements TF-IDF weighting for search results.
- Handles phrase detection and word stemming for better search accuracy.

f. **app.py**
   - Implements a Flask web application that renders the search interface and displays search results.
   - Accepts input from the frontend and submits queries to the retrieval engine.
   - Displays search results (including page title, URL, modification date, page size, keywords, and parent/child links) in JSON format.

g. **utils.py**
   - Contains utility functions used across different modules, such as string encoding.

2. File Structures

a. **Root**

○ The main logic responsible for the search engine. Contains the backend part of the application and the files stated above.

b. **Frontend**

○ The frontend part of the application displays a UI that the user can interact with. Connects to the backend of the application. Built with Vite and React for a responsive user interface.

3. Algorithms Used

a. **Breath-First Searching (BFS)**
   - A breadth-first search algorithm is used by the spider module to methodically crawl web pages. BFS makes sure that before going on to the next level, the crawler visits every link at a specific depth. A queue data structure is utilized in the implementation of this:
     1. Place a seed URL in the queue first.
     2. As long as the queue is not empty:

- Dequeue the first URL.
- Retrieve and parse the webpage (extract links and content).
- Enqueue all child links that have not been visited.

3. Continue until a certain limit is achieved or the line is empty.

This approach ensures comprehensive coverage of the website while avoiding cycles by tracking visited URLs.

b. <u>Cosine Similarity</u>

- Implemented in retrieval.py to calculate the similarity between query vectors and document vectors. The formula is given by:

  cosine_similarity(q, d) = (q · d) / (|q| × |d|)

  Where: q and d are the query and document vectors, respectively, q · d is the dot product, |q| and |d| are the magnitudes of the vectors.

- This function is implemented in retrieval.py for scoring each searched result. The title and the text are weighted at 70% and 30% respectively.

c. <u>TFxIDF</u>

- Term Frequency (TF) and Inverse Document Frequency (IDF) are used to calculate the importance of terms:

  TF(t, d) = frequency(t, d) / max(frequencies in d)

  IDF(t) = log(total documents / documents containing t)

  TF-IDF(t, d) = TF(t, d) × IDF(t)

- This weighting scheme gives higher values to terms that are frequent in a specific document but rare across the collection, helping to identify distinctive terms for each document.

d. Phrase detection

- We used spaCy to detect noun keywords present in the document. The list of keywords generated is filtered to include 2-word and 3-word phrases only. This allows for more sophisticated search capabilities beyond single-word matching.

e. PageRank-like Algorithm

- The indexer implements a PageRank-like algorithm that:
    1. Creates an adjacency matrix from the crawled page network.
    2. Iteratively updates ranking scores based on link relationships.
    3. Applies a teleportation probability to simulate random jumps in the network.
    4. Continues until the scores converge or a maximum number of iterations is reached.
- This provides a way to rank documents based on their importance in the link structure, complementing the content-based TF-IDF ranking.

4. Installation Procedure

Before running the program, the user is required to download several Python packages in the designated version and set up a virtual environment.

a. Setup Instructions

1. Create a Virtual Environment

```
python -m venv .venv
```

2. Activate the Virtual Environment
    a. On Windows:

        .venv\Scripts\activate

    b. On macOS/Linux:

        source .venv/bin/activate

3. Install Dependencies

    pip install -r requirements.txt

b. Running the code

4. Build the database

    python spider.py

    python indexer.py

5. Start the server

    python app.py

6. Start the frontend

    cd frontend

    npm install

    npm run dev

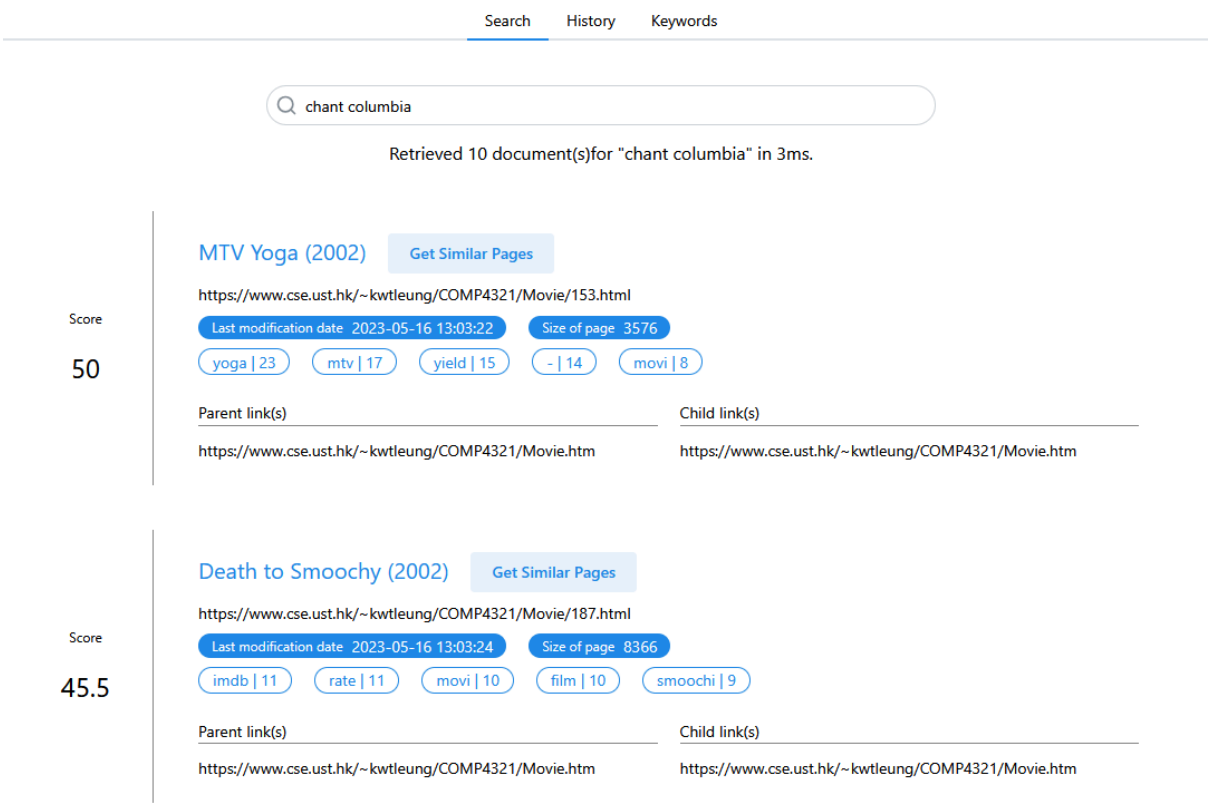
7. Open the search engine

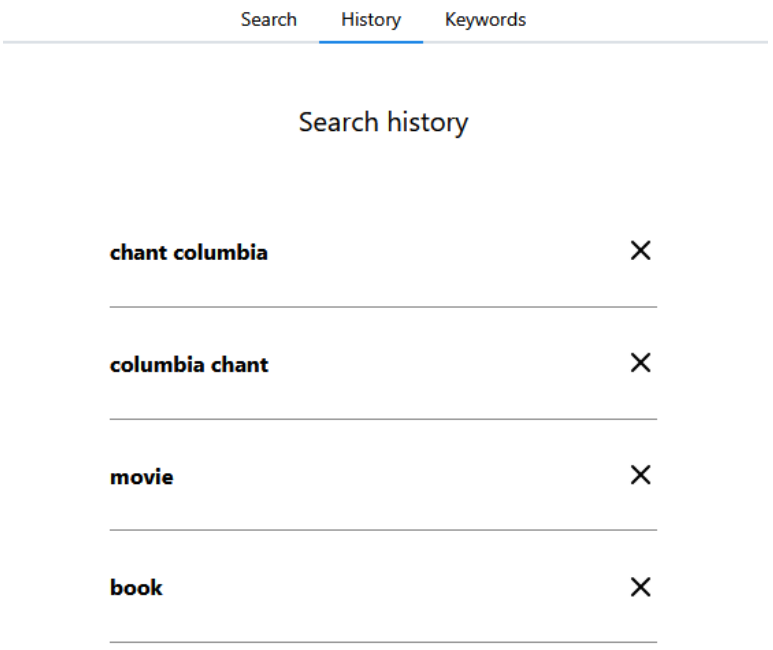
    http://localhost:5173

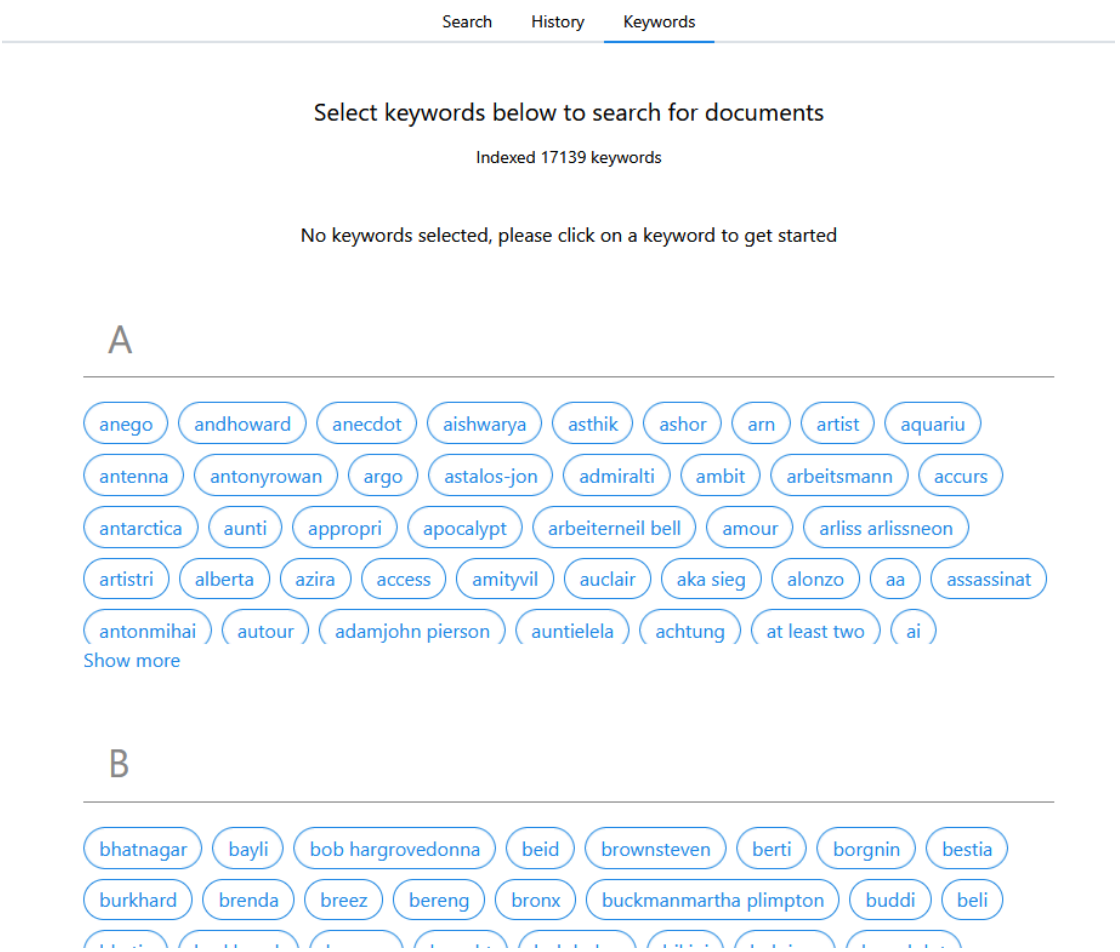
    Click it and open the browser. You can search now!

5. Highlighted Features

- Screenshots of the final product







- Get similar pages: The user can click on the get similar pages button of a search result after getting a list of results. Then, half of the related document's vector is added to the query vector to get a modified query. The new query is then used to retrieve a new list of results. The original related document is excluded from the final result list.
- List of keywords: A list of keywords is presented to the user in alphabetical order. The user can select a few keywords to build a query, and the query is submitted to the search engine, and a list of results is returned.
- Modern interface: Made with Vite and React. The interface is user-friendly with an application-based interface. Requests are made with the fetch API to enable AJAX.
- Search history: The app keeps track of a list of search history in the browser's local storage. The user can view the result of a previous query or delete a search history item.

- PageRank Integration: The page rank scores of the documents are kept in the database for ranking purposes, enhancing the relevance of search results by considering link structures.
- Performance Metrics: Relevant information, like the search time, is included in the app to provide transparency about search performance.

6. Function Testing

a. <u>Phase 1</u>

In Phase 1, by running the testprogram.py, the indexer.py and spider.py have been tested for correctness. This ensured that:

- The crawler could successfully fetch and process web pages
- The indexer correctly extracted and stored keywords
- The database structure appropriately maintained relationships between pages and words

b. <u>Final Phase</u>

By following the steps in “4. Installation Procedure”, the user should be able to install the correct Python packages, create the virtual environment, and run the complete search engine application. The testing included:

- Ensuring that search queries return relevant results
- Verifying that the BFS crawler works correctly
- Confirming that the TF-IDF and cosine similarity calculations produce accurate rankings
- Testing the phrase detection and matching capabilities
- Validating the UI features like search history and "Get Similar Pages"

7. Conclusion

- Strengths
    - A user-friendly interface that is easy to use
    - Implementation of convenient features like search history
    - Easily readable database design
- What we would have done differently
    - Documentation: We have only finished part of the documentation, so not every function in the backend is documented. We would have done it one by one to enhance the readability of the codebase.
    - SQL accesses: We notice that sometimes the code will get stuck due to improper SQL accesses. We did not have enough time to fix it, and we would love to if we had the time.
- Other interesting features
    - Fuzzy searching: Allow for fuzzy keyword matching to allow searches with typos.
    - Snippet of websites: Display a snippet of a website below the title of every entry, allowing the user to see a small part of the website before navigating to it.

8. Contribution

CHAN Sheung Yin: (33.3%)

- Report writing
- Unit testing
- Integration testing
- Debugging and optimization

CHEUNG Chi Shan Jennifer: (33.3%)

- Report writing
- Phrase detection
- Application frontend
- UI/UX design

SHING Ho Hin: (33.3%)

- Report writing
- Building the basic code framework
- Algorithm implementation
- Backend development