COMP 4321 Project Report

Group 30

LEE Pak Nin, NG Lok Chun, WIJAYA Henry

1. Overall Design of System

In this project, we developed a web-based search engine based on the Vector Space Model, built with Python 3.10, and React for the frontend.

1.1. System Components

1.1.1. Crawler

The crawler recursively fetches all pages under the base URL using Breadth First Search algorithm. In total, we crawled 297 pages and they are then passed to the indexer for indexing and storing.

1.1.2. Indexer

The indexer extracts titles and keywords from the crawled pages, then performs stop words removal with the stopword list provided, and stemming using Porter's algorithm. Meanwhile we create forward and inverted indexes, and finally compute the term weight for each title and keyword separately, using TF-IDF, and store them in database tables.

1.1.3. Retrieval Function

The search query received is first parsed into separate words, removed stop words, and converted to its stemmed form. The function then compares the vector of query keywords against the vectorized documents using Cosine similarity, and returns the most similar documents. To support phrase search, we used regex to search for the quoted words. There's also a mechanism to favour title matches which will be illustrated in Section 3.

1.1.4. Web Interface

We used React and Typescript framework, and Flask packages to coordinate backend calls. The web interface accepts query inputs, and provides helpful features such as relevance feedback and query histories.

1.2. Packages used

Package	**Used in**	**Justification**
requests	Crawler	To send HTTP requests and fetch web page content from the base URL and its subpages
beautifulsoup4	Crawler	To parse HTML content, extract titles, keywords, and hyperlinks for indexing
nltk	Indexer & Retrieval function	For stemming using Porter's algorithm
Flask	Web Interface	For handling API routing and cross-origin requests between the React frontend and Flask server
Sqlite3 (built-in)	Indexer, Retrieval function	Manage database with SQL
re	Retrieval function	Use regular expression to search for matching phrases in each document

2. File Structure of Database

Our database is a relational database implemented on Sqlite3 packages, here is the database scheme,

Here is a detailed description of each table,

Table	**Justification**
page_information	4 columns, each store a page's information, including title, last modification date, and size
page_relationships	2 columns, parent_url and children_url, store the relationship (parent-child) between 2 urls, if any
urls	Store all the urls crawled, here we used string-id
titles	Store all the stemmed titles, here we used string-id
words	Store all the stemmed words, here we used string-id
tokens	Store all the stemmed uni-, bi-, and tri-grams, here we used string-id
forward_index	Store all content words of each document as a string in one column
inverted_index	Store the term frequency of each stemmed words inside each document as a string in one column
title_forward_index	Store all title words of each document as a string in one column
title_inverted_index	Store the term frequency of each stemmed title words from each document as a string in one column
word_statistics	Store the precomputed TF-IDF of each stemmed words inside each document
title_statistics	Store the precomputed TF-IDF of each stemmed title words from each document
page_stemmed_title	Similar to title_forward_index, but the title words in stemmed form
page_stemmed_word	Similar to forward_index, but the content words in stemmed form

3. Algorithms

3.1. Crawler

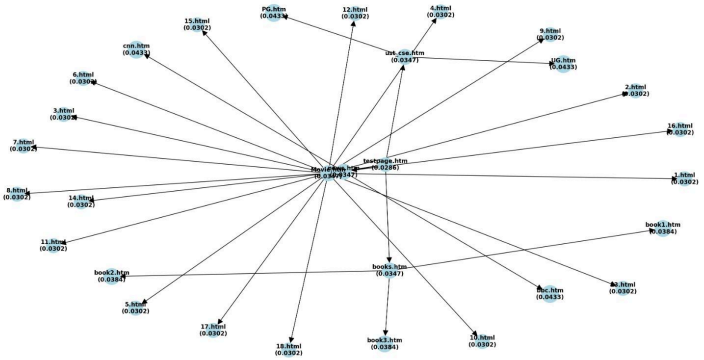

Figure 1. Graph illustration of BFS of pages in this project with Page Rank score (Not normalized)

We used Breadth First Search (BFS) to explore web pages level by level and ensure that the crawler prioritizes breadth over depth. The crawler begins with a URL, which serves as the starting point. From there, BFS operates using a queue-based mechanism, where each URL is processed in the order it is discovered. This guarantees that all links on the current page are visited before moving on to links found on subsequent pages, effectively exploring the web in layers. To prevent redundant crawling and infinite loops, a visited set tracks already processed URLs, ensuring no page is crawled more than once, e.g. for movie subpages, we simply ignore the back link to the movie index page.

3.2. Indexer

The search engine's indexer plays a crucial role in organizing and processing textual data to enable efficient retrieval of relevant web pages. The implementation begins by initializing the indexer with a set of stopwords—common words like "the," "and," or "is" that are filtered out during processing—loaded from a predefined file. Additionally, the indexer employs the Porter Stemmer from the Natural Language Toolkit (NLTK) to reduce words to their root forms, ensuring that different inflections (e.g., "running," "runs") are treated as the same term.

When indexing a web page, the indexer processes both the title and body text. The tokenization step involves splitting the text into individual words while removing punctuation and converting everything to lowercase. Stopwords are then filtered out to reduce noise and improve the relevance of search results. The remaining tokens are stemmed to consolidate variations of the same word. For example, "computers," "computing," and "computed" might all be stemmed to "comput".

The indexer maintains two separate inverted indexes: one for the titles of web pages and another for their body content. Each stemmed term in these indexes maps to a list of page IDs (URLs) where the term appears. This structure allows the search engine to quickly locate all pages containing a given query term. By separating title and body indexes, the system can later prioritize matches in titles, which are often more indicative of a page's relevance. Overall, this indexing process ensures that the search engine can efficiently retrieve and rank pages based on user queries.

3.3. Retrieval Function

In the Vector Space Model, each document is represented by a bag of keywords plus their statistical information.

3.3.1. Term Weights Calculation

We used the following normalized TF-IDF formula for each term weight, w_{ij},

$$w_{ij} = \frac{tf_{ij} * idf_j}{max\,(tf_{ij})}$$

where idf_j is defined as,

$$idf_j = log_2(\frac{N}{df_j})$$

tf_{ij} refers to the term frequency of word j in document i, df_j refers to the document frequency of word j, which means the number of documents containing this word, and N equals the number of documents in this search engine. Here, idf_j is the inverted document frequency, which penalises frequently occuring words as they process less powerful discriminative power. Note that we handle title and content words separately in 2 tables for favouring title matches, so as their term weights. All term weights are computed after the page was crawled and indexed. This helps to enhance efficiency when retrieving relevant documents.

3.3.2. Query Processing

As we have applied stop word removal and stemming to the indexed documents, we have to do the same for query terms so as to perform valid search. To support phrase search, we handle single words and phrases separately. We first search for exact matches of phrases in each document, if such phrases cannot be found, we simply ignore this document. If a phrase is found, we then consider each phrase as a list of single words, then perform standard similarity computation, i.e. the phrase “Hong Kong” will be considered as “Hong Kong”, “Hong”, and “Kong”.

3.3.3. Similarity Calculation

After the above steps, we present the query as a vector with each query term having equal weight, so a user can increase the weight of a word by repeating it. Before comparing the query to each document, we check if the document, including title, contains any of the query words, if not, we skip this document to enhance efficiency. The similarity function we used is Cosine similarity, defined as,

$$sim(D_i, Q) = \frac{D_i \circ Q}{|D_i|\,|Q|} = \frac{\sum_{k=1}^{t} d_{ik} * q_k}{\sqrt{\sum_{k=1}^{t} d_{ik}^2}\sqrt{\sum_{k=1}^{t} q_k^2}}$$

Each query vector is compared with each document's title and content vector separately using Cosine distance, the scores are then combined to form a list of similarity scores. Then we rank it in decreasing order and extract at most 50 top scoring documents as required.

3.3. Favour Title Match

Note that we compute document similarity by handling title match and content match separately. We derived a formula to combine these two scores,

$$Sim(D_i, Q) = \alpha * Sim_{title}(Dt_i, Q) + \beta * Sim_{content}(Dc_i, Q)$$

where Dt_i and Dc_i refer to the title and content vector of document i respectively, and α and β is the weights for title match and content match. We set $\alpha = 0.7$ and $\beta = 0.3$ to favour the title match.

3.4. Page Rank

Our search engine incorporates a custom implementation of the PageRank algorithm to determine the importance of web pages in our index, as illustrated in Figure 1. PageRank is a link analysis algorithm that assigns numerical weights to each element of a hyperlinked set of documents, with the purpose of measuring its relative importance within the set.

3.4.1 Implementation

Our implementation follows the classic PageRank formula with a damping factor,

$$PR(A) = \frac{(1-d)}{N} + d * (\frac{PR(T_1)}{C(T_1)} + \ldots + \frac{PR(T_n)}{C(T_n)})$$

where PR(A) is the PageRank of page A; d is the damping factor (typically 0.85); N is the total number of pages; T_1 to T□ are pages linking to page A and C(T□) is the number of outbound links from page T□.

3.4.2 Mathematical Details

Initialization:

$$\forall\, node \in G:\ PR_0(node) = 1/N$$

For each iteration until convergence:

$$\forall\, node \in G:\ PR_{k+1}(node) = \frac{(1-d)}{N} + d * \sum(\frac{PR_k(T)}{C(T)}\ for\ T \in in\_links(node))$$

For nodes with no outgoing links:

$$\forall\, node \in G:\ PR_{k+1}(node) \mathrel{+}= d * \frac{PR_k(dangling)}{N}$$

Convergence Criterion:

$$\Sigma\, |PR□_{+1}(node) - PR□(node)| < tolerance$$

3.4.3 Key Features of Our Implementation

1. Dangling Node Handling: As mentioned in Section 3.1, we simply ignore cyclic links, e.g. movie subpages have no out-going link back to movie index page. These dangling nodes distribute their PageRank evenly across all pages.
2. Convergence Check: The algorithm stops when the difference between iterations falls below a tolerance threshold (1.0e-6).
3. Normalization: For easy search score calculation, scores are normalized using L2 norm (Euclidean distance).

4. Installation process

Here is a brief instruction, for more details please visit the README.md file in the code base.

Step 1. Download the code base.

Step 2. Create a python virtual environment by typing the following commands into the terminal.

- For Windows: `python -m venv venv` ; For Unix: `python3 -m venv venv`.

Step 3. Activate the python virtual machine.

- For Windows: `venv\Scripts\activate` ; For Unix: `source venv/bin/activate`.

Step 4. Go into the backend folder by typing `cd backend` into the terminal.

Step 5. Install the dependencies by typing `pip install -r requirements.txt`.

Step 6. Run *crawl_and_save.py* to perform crawling and indexing

- For Windows: `python crawl_and_save.py`; For Unix: `python3 craw;_and_save.py`.

Step 7. Run `start_server.py` to start the backend server

- For Windows: `python start_server.py` ; For Unix: `python3 start_server.py`.

Step 8. Open another terminal and go into the frontend folder by typing `cd frontend` into the terminal.

Step 9. Install the required packages by running `npm i`.

Step 10. Run the command `npm run dev`.

Step 11. Go to http://localhost:5173 to use the search engine.

Step 12. After using the search engine, deactivate the python virtual machine by typing `deactivate` into the terminal.

5. Bonus Features

5.1. Get Similar Page

We implemented a relevance feedback feature “Get Similar Page” for each searched result. When a user clicks on the button, it extracts the top 5 most frequent stemmed words, rewrites the query, and does the search automatically to retrieve similar pages under the same topic.

5.2. View Stemmed Words

We allow users to view the list of all stemmed words from the database (more than 13K words), they can then choose the keywords they wanted to search, add them to the query and perform the search.

5.3. User-friendly Interface

Our web interface took reference from the aesthetic design of the Perplexity, a popular AI-powered answer engine. We provide 2 themes, light and dark mode, users can switch between them easily. The interface is easy-to-use and straightforward, searching is done in the middle search bar, history and settings can be done on the left side bar.

5.4. Voice Input

Apart from query input by typing, we also support voice input. Simply click the microphone icon and users can input a query by speaking.

5.5. Real-time Input Suggestions

When a user is inputting the query, we provide in-time word completions and suggestions. These suggestions were generated by retrieving the most similar n-grams given the partially entered query. In this way, the word suggestions are not only completed words, but words or phrases that lead to a valid search.

5.6. Keep track of query history

We made use of the browser's local client-side persistent storage to store a user’s search queries. The queries are stored as JSON strings, users can browse on the left side bar, select a query to view the results or edit on top of it, or remove any histories. User can delete all histories by clicking the rubbish bin icon, or remove a specific history by double tapping it.

5.7. Page Rank

We implemented Page Rank to perform links analysis as mentioned in Section 3.

5.8. Significant Retrieval Speed

The retrieval speed of our search engine is commendable, backed by our approach in skipping similarity calculations with documents that do not contain the query words, and the fact that we pre-computed term weights for all titles and words. Still, the retrieval speed for common words like “movie” is not significant.

6. Testing of System

We tested our projects on both Windows and UNIX-based OS. We followed the instructions in README.md to make sure it is easy-to-follow. We then first tested our web interface by trying all buttons and text boxes, and checking if it connects to the backend server. Afterward we tested with different searching query formats – single word, phrases(bi- and tri-grams), mixture of both, repeated words, etc. We also tested on bonus features including reusing query history, get a similar page, search from the stemmed keyword list, etc. Finally we tested with edge cases like empty query, stopwords, invalid words, etc.

Here are some test cases and screenshots of our search engine.

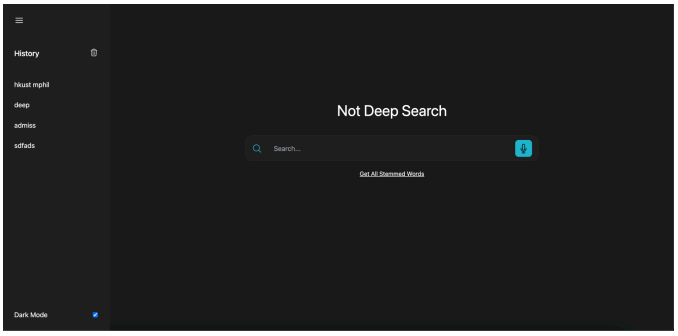

Figure 2. Screenshot of Main Page in Dark Mode

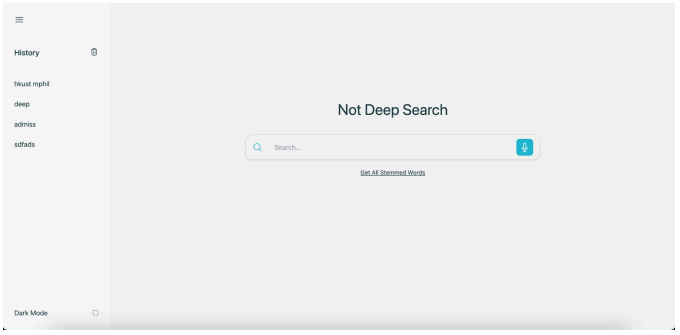

Figure 3. Screenshot of Main Page in Light Mode

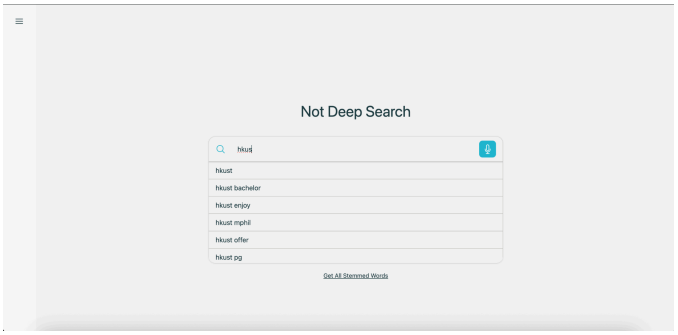

Figure 4. Screenshot of Query Auto Completion/Suggestions

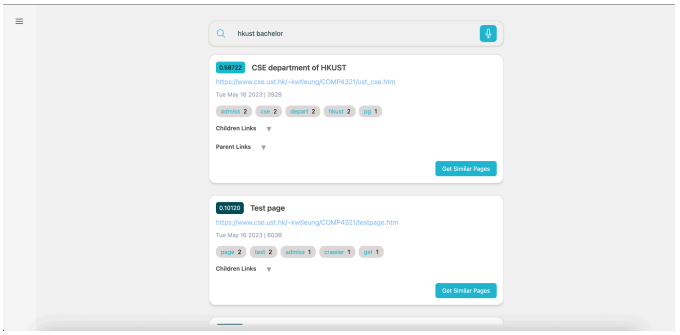

Figure 5. Screenshot of Search Results

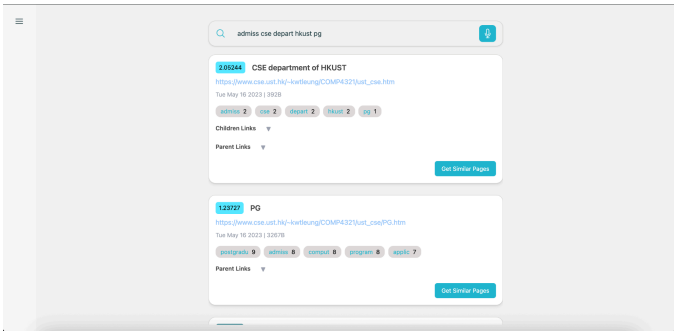

Figure 6. Screenshot of Multi-keyword Search

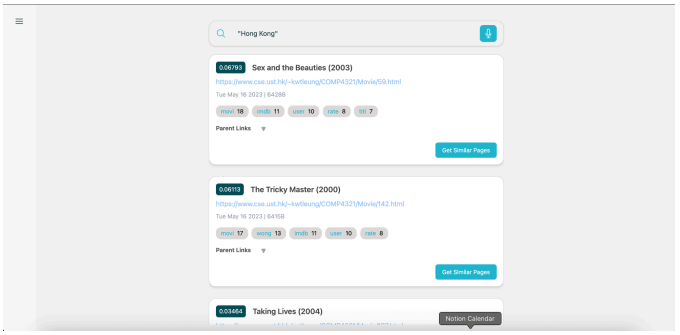

Figure 7. Screenshot of Phrase Search

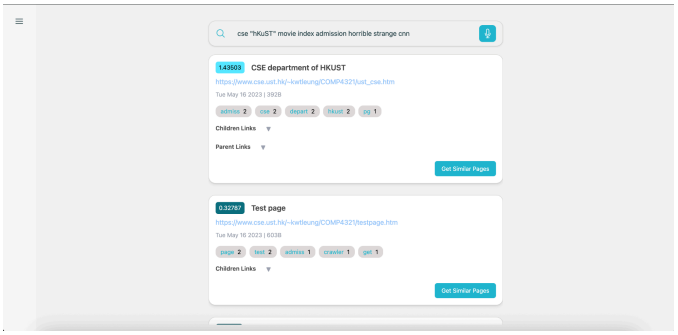

Figure 8. Screenshot of Mixture of Single Words and Phrase Search

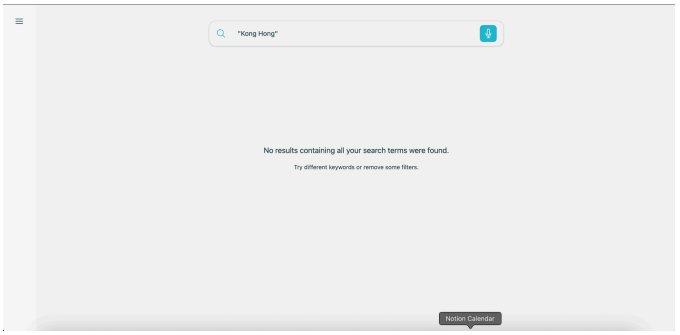

Figure 9. Screenshot of Invalid Phrase Search

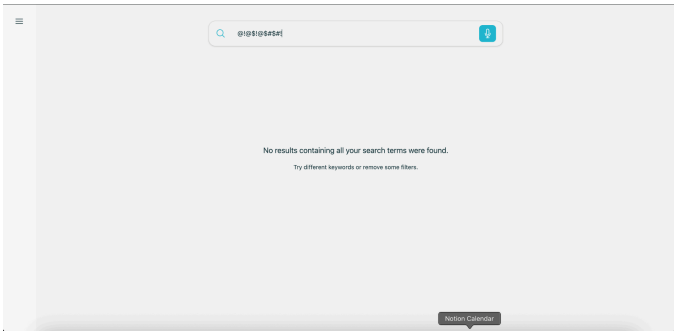

Figure 10. Screenshot of Invalid Query

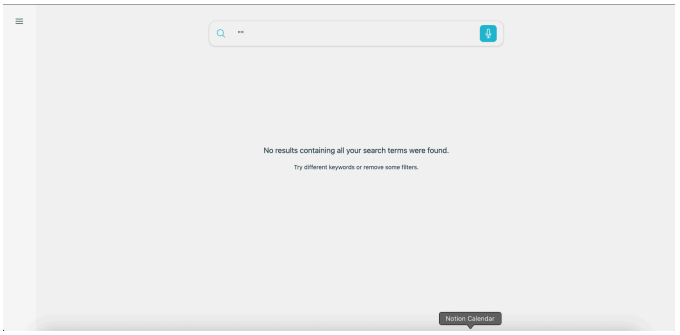

Figure 11. Screenshot of Another Invalid Query

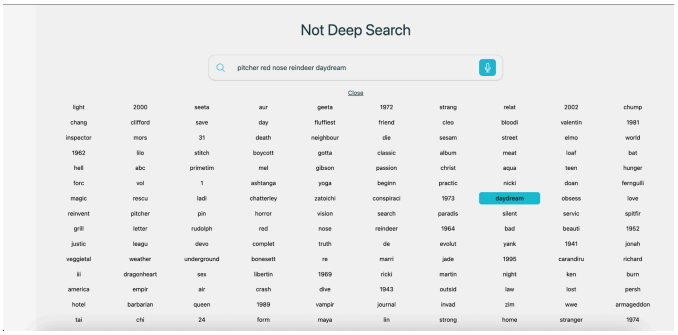

Figure 12. Screenshot of Get All Stemmed Words

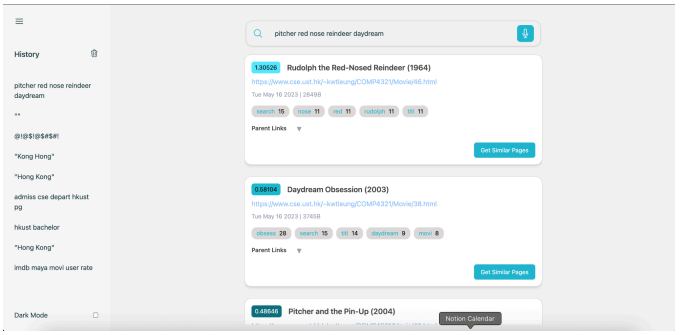

Figure 13. Screenshot of Viewing History (Left)

7. Conclusion

In conclusion, we successfully implement a simple yet effective web-based search engine which can handle query search at a promising speed.

The strength of our system lies in having a good UI, comparable retrieval speed, and included quite a number of additional features to improve user experience like storing history, relevance feedback, and using page rank to improve results.

As for the weakness, our system does not support single character searching, e.g. “A” which will be considered as stop words. We did not implement a dynamic update of the database when there is a page modification, we just simply delete it and crawl the pages again.

If we are re-implementing the system, we would try to use more advanced data structure for the inverted field like hash table or b-tree to enhance the effectiveness of index term retrieval. We did implement fuzzy search in our system, but was eventually removed as it makes the searching very slow (takes 10-20 seconds) as it compares to all possible n-grams to find the most similar one. We may try to optimize this workflow if we have more time.

8. Contribution

Team Member	**Contribution**	**Responsibilities**
Lee Pak Nin	33.3333%	Frontend, bonus features, report, demo video
Ng Lok Chun	33.3333%	Indexer, page rank, handling n-grams, report
WIJAYA Henry	33.3333%	Crawler, retrieval function, report