Course Code: COMP4321

Student Name: KAN, Wan On

Student ID: 20701753

COMP4321 Course Project Report

**Introduction**

In this undergraduate project, we were tasked with developing a web-based search engine with various requirements and functions. Although the project was originally instructed to use Java and JSP, with permission I decided to use Python and Flask to build the web-based search engine.

I chose Python and Flask because of my familiarity with the language and its ease of use in web development. Additionally, Flask is a lightweight and flexible framework that allowed me to quickly create a functional web interface for our search engine.

**Overall System Design**

The search engine follows a three-step process of spidering, indexing, and retrieval to crawl and index web pages. The spider function uses a breadth-first search strategy to fetch pages from a given website and checks if the URL exists in the inverted index or if the last modification date of the URL is later than that recorded in the index. It also handles cyclic links by maintaining a set of URLs that have already been visited and not revisiting them. The spider extracts all the hyperlinks from each page fetched into the local system and builds a file structure containing the parent/child link relation.

The indexer extracts keywords from a page using Porter's algorithm to transform words into stems and inserts them into an inverted file and a forward file, both raw and processed. The indexes support phrase search and are stored in file structures that utilize JSON. The indexer removes stop words from the text using a list of stop words stored in a separate text file. The retrieval function compares a list of query terms against the inverted file and returns the top documents, up to a maximum of 50, to the user in a ranked order according to the vector space model. The function favors matches in the title to boost the rank of a page. The retrieval function utilizes NumPy array computation to calculate the result of cosine similarity across a significant amount of query and returns up to 5 most frequent stemmed keywords (excluding stop words) in the page together with their occurrence frequencies.

The web interface is implemented using Flask, a lightweight and flexible web framework that allows for quick creation of a functional web interface for the search engine. The interface allows users to enter queries and displays the returned results in a clear and organized format, with colored blocks. All of the results (up to 50 web pages) can be displayed on one page.

**File Structure**

.

├── HITS.py

├── app.py

├── db

│ ├── child\_index.json

│ ├── content\_forward\_index.json

│ ├── content\_inverted\_index.json

│ ├── page\_info.json

│ ├── pageid\_to\_url.json

│ ├── parent\_index.json

│ ├── query.json

│ ├── raw\_content\_forward\_index.json

│ ├── raw\_content\_inverted\_index.json

│ ├── raw\_title\_forward\_index.json

│ ├── raw\_title\_inverted\_index.json

│ ├── stem\_to\_stemid.json

│ ├── stemid\_to\_stem.json

│ ├── title\_forward\_index.json

│ ├── title\_inverted\_index.json

│ ├── url\_to\_pageid.json

│ ├── word\_to\_wordid.json

│ └── wordid\_to\_word.json

├── db\_functions.py

├── display.py

├── indexer.py

├── spider.py

├── static

│ ├── css

│ │ └── style.css

│ └── images

│ └── background.jpg

├── stopwords.txt

├── templates

│ ├── index.html

│ └── search\_results.html

└── search\_engine.py

The top-level directory contains several script files, directories, and text files necessary for the search engine to function properly. The HITS.py file is a Python script file that implements the HITS algorithm for ranking web pages. The Project Description.pdf file is a PDF file that provides a description of the project. The pycache directory contains compiled Python bytecode files for the scripts in the top-level directory. The app.py file is a Python script file that contains the Flask application for the web interface. The db directory contains several JSON files that store the database information for the web pages, including child-parent relationships, forward indexes, inverted indexes, page IDs, URL mappings, and stem/word ID mappings.

The display.py file is a Python script file that contains functions for displaying search results on the web interface. The indexer.py file is a Python script file that contains functions for indexing web pages, including extracting keywords from web pages using Porter's algorithm, removing stop words, and constructing inverted indexes and forward indexes for the content and titles of web pages. The spider.py file is a Python script file that contains functions for crawling web pages, including fetching web pages from a given website using a breadth-first search strategy, checking for cyclic links, and building a file structure containing the parent/child link relation.

The static directory contains subdirectories for storing static files used in the web interface, including CSS files for styling the web interface and images used in the web interface. The templates directory contains HTML templates for the web interface, including the main page of the search engine (index.html) and the page displaying search results (search\_results.html).

Overall, the files and directories in the top-level directory of the search engine serve different purposes, such as implementing the HITS algorithm, storing database information, extracting keywords from web pages, crawling web pages, and displaying search results on the web interface. The subdirectories in the static and templates directories contain files necessary for styling the web interface and displaying the search results in a clear and organized format.

**Algorithm Analysis**

The spider.py script is designed to crawl web pages in a fast and efficient manner using a breadth-first search algorithm. The algorithm ensures that URLs are visited in the order of their depth in the graph, which is an efficient way to explore the graph. The script uses a deque data structure to store the URLs for BFS traversal, which allows for efficient appending and popping of URLs from both ends of the queue. A visited set is used to keep track of the URLs that have already been visited, which ensures that the same URL is not visited multiple times, reducing the chances of infinite loops and wastage of computational resources.

The script uses the requests library to send HTTP requests to web servers and the BeautifulSoup library to parse HTML and extract information from web pages. This allows for efficient fetching and parsing of web pages. To optimize the crawling process, the script checks if the website has been fetched before by looking up the page\_info.json and url\_to\_pageid.json files. This reduces the number of unnecessary HTTP requests and database operations, which can improve the crawling speed and reduce computational resources. The script also extracts only the necessary information from the web pages, such as the page title, URL, last modified date, download size, text data, and child links. This reduces the amount of data that needs to be stored in the database, which can improve the database performance and reduce storage requirements.

The indexer function generates various indexes for a search engine. It takes as input a set of documents and generates forward and inverted indexes for the document content and title. The algorithm begins by loading a set of stopwords and creating empty JSON files for the different indexes that it will generate. It then iterates through each document and generates the corresponding forward and inverted indexes. For each document, the algorithm generates a forward index of the words in the document content and title, and applies the Porter stemming algorithm to each word to obtain its stem. It then checks whether the stem is a stopword, and if it is not, it adds it to the inverted index along with the document id and position of the word in the document.

The algorithm also generates forward indexes for the content and title, where each word in the forward index is replaced with its corresponding stem. The resulting forward indexes map each document id to a list of integer ids representing the stems in the document. Finally, the algorithm dumps the generated indexes to their respective JSON files.

The search engine takes advantage of the generated indexes to quickly retrieve documents that contain the desired query term or set of terms. The content and title of the documents are processed independently before being merged for the final aggregation. The inverted index maps each stem to a set of document ids that contain that stem, making it a breeze to retrieve documents that contain a given query term. On the other hand, the forward indexes map each document id to a list of integer ids representing the stems in the document, allowing for efficient scoring of documents based on their relevance to a given query.

After computing the similarity score of both the content and title to the query, the scores are merged for each document id with weighted up the title similarity in the factor of **3.0.** This means that the title similarity score is given three times as much importance as the content similarity score.

The search engine begins by parsing the user's query into a list of stemmed terms and phrases, removing any stopwords, and handling exact phrase matching using quotes. It then counts the number of occurrences of each term and phrase in the query. For each unique term and phrase in the query, the search engine calculates a TF-IDF score for that element with respect to each document.

The TF-IDF score takes into account the frequency of the query element within the document (the term frequency or TF), as well as the rarity of that element across all documents (the inverse document frequency or IDF). The scores for each query element are stored in a dictionary, with document IDs as keys and score arrays as values.

For exact phrases, the search engine applies a phrase boost of **3.0** to the scores of documents containing that **exact phrase**. It is important to note that the system has no length limitation on the phrase search, so the user can input the phrase with double quotation marks to retrieve more accurate and specific results.

Once a score has been calculated for each query element, the search engine computes the cosine similarity between the query vector - containing the scores for each unique element - and each document's score vector. This results in an overall relevance score between 0 and 1 for each document with respect to the entire query. The documents are then sorted in descending order of relevance score, and the top 50 most relevant documents are returned as the search results.

|  |  |
| --- | --- |
| Phrase Boost Factor | 3.0 |
| Title Boost Factor | 3.0 |

Table 1. A summary table of the phrase boost and title boost factor.

In summary, the spider.py script efficiently crawls web pages using a BFS algorithm and generates indexes for a search engine using a combination of forward and inverted indexes. The search engine uses these indexes to calculate TF-IDF scores and cosine similarity to determine the relevance of documents to the user's query. Query expansion and more-like-this techniques are then used to enhance the diversity and usefulness of the results. The entire process is designed to be efficient, accurate, and optimized for speed and performance.

**Installation Procedure**

This project is under conda-based environment. It is recommended to use MacOS to test the program. Please follow the command line procedures below step by step.

1. Please install either anaconda or miniconda. After changing directory to project directory, the user can create a conda environment my\_env in the python version=3.9 with the required packages listed in requirements.txt. Remember to activate the environment myenv.

$ conda create --name my\_env --file requirements.txt

$ conda activate my\_env

1. Run the spider program first. It will also call the indexer functions.

$ python spider.py

1. Run the app.py to host the flask-based web interface.

python app.py

**Highlighted Features Beyond The Required Specification**

The search engine system has successfully implemented all of the suggested features, including relevance feedback, a list of indexed keywords, a user-friendly interface, query history tracking, and link-based result ranking. By utilizing special implementation techniques, the system achieves an exceedingly good speed and provides a comprehensive and efficient search experience for the user. These features enhance the accuracy, speed, and usability of the search system, offering users a more efficient and effective search experience.

The search engine's implementation of relevance feedback allows users to view similar pages by clicking on a button next to each search result. The system retrieves all relevant results for the top 50 webpages immediately after the query search, and preloads them in the webpage. Clicking the "show/hide similar page" button displays the preloaded similar pages without delay. This feature is integrated into the existing search algorithm, where the "search\_engine" function takes the user's query and returns the search results. It calls the "match\_with\_other\_similarity" function to retrieve similar pages for a specific search result. The "match\_with\_other\_similarity" function extracts the top 5 most frequent keywords (excluding stop words) from the page content and adds them to the query. It then executes the search algorithm with the new query and returns the top similar pages.

To enhance the search experience, the system allows users to view a list of all the stemmed keywords indexed in the database. Users can browse through the list and select the keywords they are interested in to create a vector-space query for the search engine. To achieve this, the system uses an input system that displays a list of stemmed words. Users can select a word by clicking on it, and the system will add it to the existing query. The datalist of stemmed words is populated from a JSON database using AJAX. This ensures that the datalist is always up-to-date with the latest keywords indexed in the database. Overall, this feature allows users to create more accurate queries to improve the search results.

The system utilizes AJAX to improve the user experience in several ways. It loads stemmed words and query keys from JSON files asynchronously, ensuring that users always have access to the most up-to-date keywords and their previous search queries. Additionally, the system uses AJAX to fetch search results from the server without requiring a page refresh, providing a seamless and responsive user interface. These features improve the accuracy, speed, and usability of the search system, offering users a more efficient and effective search experience.

The system provides users with an application-based interface that enables them to view and operate on their previous search queries. The "loadQueryKeys" function fetches query history data asynchronously from a JSON file using AJAX and populates the datalist of query keys using the "populateDatalist" function. With this feature, users can easily access their previous queries by selecting the corresponding query key from the datalist when searching for new topics. Additionally, the search engine performs query expansion using the most similar historical query in its logs to improve result diversity. It takes a weighted sum of the scores of the top results of that historical query with the original results and finds 5 other highly similar documents by using just the content or title terms of each document as a new query. This "More like this" feature enhances the user experience by providing additional relevant results and increasing the diversity of search results.

The system's user interface is designed to be visually appealing and easy to use, with a background image and color palettes used to distinguish between different types of search results. The interface also includes functional features such as a pop-up window when the user clicks on an external link and a "More like this" section that suggests similar websites based on the user's search query. These features help to improve the user experience and make the process of searching for information more efficient and effective. Overall, the combination of aesthetic design and functional features in the system's user interface creates a positive and user-friendly experience for the user.

Moreover, the system implements two algorithms, HITS and PageRank, for ranking search results based on the quality and quantity of links to a webpage. HITS algorithm calculates two scores, authority and hub, for each webpage. The authority score measures the quality of the content on a webpage, while the hub score measures the number of links from high-quality webpages to the webpage in question. The hits\_algorithm function initializes the scores for all webpages, constructs the adjacency matrix of the links between webpages, and then uses the calculate\_auth\_hub\_scores function to calculate the final authority and hub scores. The function then combines the authority scores with the similarity scores to rank the pages using the combine\_similarity\_pagerank function.

On the other hand, the PageRank algorithm calculates a score for each webpage based on the number and quality of links to that webpage from other webpages. The pagerank function constructs the transition matrix of the links between webpages and then iteratively calculates the PageRank scores until convergence. Finally, the function returns a dictionary of webpage scores. The combine\_similarity\_pagerank function combines the similarity scores and PageRank scores to rank the webpages. It first calculates the PageRank scores using the pagerank function and then combines these scores with the similarity scores. Finally, it sorts the combined scores and returns a list of ranked webpages.

Overall, these algorithms provide a robust ranking system that considers various factors, such as the quality and quantity of links, to produce relevant and high-quality search results for the user. The hits\_algorithm and pagerank functions work in tandem with the combine\_similarity\_pagerank function to provide a comprehensive ranking of search results. The implementation of these algorithms enhances the search experience for the user and helps them to find the information they need more efficiently and effectively.

To achieve high speed and efficiency in a web search system, several techniques can be implemented. Beautiful Soup with BFS techniques is used for efficient data scraping, while NumPy is used for efficient manipulation of large multi-dimensional arrays and matrices. The Porter stemming algorithm from NLTK is used to reduce the size of the data and improve processing speed. JSON is used as a lightweight and easy-to-use database for efficient storage, retrieval, and manipulation of data. These techniques are already existing features that are widely used in search systems to retrieve and analyze data quickly and accurately, providing a comprehensive and efficient experience for the user.

**Testing and Evaluation**

To evaluate the performance of the search engine system, we conducted testing on various aspects of the system. First, we conducted a survey to gather feedback on the user interface of the system from the target audience, which consisted of students. The survey results showed that over 90% of users found the system's design to be very appealing but clean and intuitive. They also found the system easy to use, indicating that the design was successful in meeting the needs of the target audience.

一張含有 雪, 螢幕擷取畫面, 樹狀, 橫向 的圖片

自動產生的描述

Figure. 1 A screenshot of the web interface of the starting page.

一張含有 文字, 螢幕擷取畫面, 網頁, 網站 的圖片

自動產生的描述

Figure. 2 Another screenshot of the web interface that shows the retrieved results.

一張含有 文字, 螢幕擷取畫面, 軟體, 字型 的圖片

自動產生的描述

Figure. 3. The screenshot that illustrates the input system what includes stemmed word list.

Next, we evaluated the system's performance in terms of speed and efficiency. We tested the system's ability to process large amounts of data by fetching 300 pages into the database. The system was able to complete this task in just 52 seconds, indicating its ability to handle large amounts of data efficiently. We also tested the indexer, which works by analyzing the content of each page and indexing the relevant keywords. The indexer was able to complete its task in just 0.89 seconds, which is a testament to the efficiency of the system.

一張含有 文字, 螢幕擷取畫面, 字型 的圖片

自動產生的描述

Figure 4. A screenshot of the CMD result in spider function

一張含有 文字, 螢幕擷取畫面, 字型 的圖片

自動產生的描述

Figure 5. A screenshot of the CMD result in the server side

Finally, we tested the retrieval function of the system, which is responsible for returning search results to the user. We tested the retrieval function on a set of queries and measured the response time. The retrieval function was able to return search results in just 0.191 seconds, indicating a fast and responsive system.

Overall, the testing and evaluation of the search engine system have shown that it is a high-performance, user-friendly, and efficient system that delivers accurate and relevant search results to users. Its sleek and intuitive design, small size, and fast response time make it an ideal tool for users seeking a comprehensive and efficient search experience.

**Conclusion**

Strengths of the search engine system include the implementation of relevance feedback, a list of indexed keywords, and link-based result ranking. The user-friendly interface, fast response time, and efficient processing of large amounts of data also contribute to the system's effectiveness. However, weaknesses include the potential limitations of the HITS and PageRank algorithms for link-based result ranking and the system's reliance on past indexing for search results. If re-implementing the system, improvements could include conducting more extensive user research, incorporating additional ranking algorithms, and adding features such as NLP capabilities, image and video search, personalization, integration with social media, and voice search. It should also be noted that the original system received a low score in the first phase due to the use of an external library that was not allowed. However, the system was revised to avoid such libraries, and the developer expressed a willingness to improve and learn from the experience.