Simplified Search Engine

Information Retrieval

AUTHOR: Parmenion Charistos

PID: 3173

DATE: September 2021

Introduction

This project comprises a simulation of the fundamental functionality of a search engine. The search engine consists of three main components, a **web crawler**, an **indexer** and a **query processor**.

No web interface was designed for this particular application due to technical issues when using the *spark* micro framework. Instead, due to time limitation, it was later altered to function as a solid console application.

Java 8 was used for the implementation of all subsystems.

The components have been designed to run concurrent processes via multithreading.

Project Structure

Main

The Controller class. Execution begins here. User input is collected for the initial setup, before it's utilized to create the application's tweaked individual components. From the static context of this class, all execution threads are managed, as it serves as the application's execution flow blueprint.

In a glance, the main class sets up the web crawler and indexer threads, starting them sequentially when each necessary subroutine has been completed for the next one to take place. Once enough pages have been crawled and indexed, the query processor takes the lead, processing user requests until they exit the application.

Crawler

The Web Crawler is formed by piecing together a simple puzzle. It is initialized with a starting URL, and goes on visiting adjacent pages whilst collecting their content, if that is possible policy-wise. Once the threshold is reached (enough pages have been crawled), the crawler shuts down, having stored all the crucial information in its class variables.

The structure used for the *Frontier* is a linked list (*LinkedList<String> Frontier*). The strategy applied is BFS (Breadth First Search), as the new URLs that occur are, in fact, solely, the immediate children of the parent URL. To rephrase, first in line for visiting are all the adjacent webpages of the starting URL, before proceeding to examine grandchildren first. This allows for analysis of a wide variety of resources immediately connected to the starting URL, instead of tunnel visioning by thoroughly searching few link lineages in depth.

In aid of the main processes updating the *Frontier*, there is a structure saving all links that have already been visited (*HashSet<String> Visited*) and, of course, a structure mapping the crawled pages to their content values (*HashMap<String, String[]>* Data).

The aftereffect is, all *Crawler* threads get to execute a straightforward process concurrently, allowing them to noticeably decrease execution time, whilst maintaining low complexity code. I.e. each crawler thread picks up the link in the head of the *Frontier*, saves its raw content, extracts the containing links and adds them to the tail of the *Frontier*.

The method performing the forementioned operation is ***crawl()***, with the assistance of a couple of static functions which are responsible for calculations and safety checks [*extractLinks()*, *isEmpty()*].

Indexer

Once the *Crawlers* have gathered the required information, the *Indexers* take the lead. Each *Indexer* can perform a range of processes. Their activities are categorized based on the stage of execution that has been reached. The initial function an *Indexer* performs is via the class ***index()*** method. Sequentially, they acquire one link from the *IndexFrontier* structure. Then, their purpose lies in processing the raw content of the selected webpage, which is already stored by the *Crawlers*. *Indexers* identify the different tokens and count their occurrences in the document under examination (*term frequency*).

The various tokens are simultaneously stored in *HashSet<String> Terms*, which is thus used to update the *document frequency* of each term found in the new document (*HashMap<String, Integer> DF*.

These operations keep taking place until the *IndexFrontier* is emptied. When the last *index()* function call is terminated, the application has already stored the *term* and *document frequency* of all terms encountered.

Once indexing has been completed, different *Indexers* running on their separate threads can now begin calculating the terms' *IDFs*, whilst storing them in the according *HashMap<String, Double> IDF*.

The function liable for this task is ***calculateIDF()***. As it progresses and the needed stats are calculated, the final *Indexer* method takes place, led by yet different *Indexers* running concurrently.

The ***vectorize()*** method utilizes the *TF* and *IDF* stats of all terms, to produce the weight vectors of all documents (*tf-idf product*). All the information that has been mined so far, is stored in the respective *Indexer* class variables.

Query Processor

The *QueryProcessor* involves solely static methods, thus no individual objects are instantiated throughout the application's execution. Its goal is simple;

- Ask the user for a query and the number of results to return.
- Calculate the *tf-idf product* of the query.
- Calculate the cosine similarity of the query to the stored documents.
- Sort the angles.
- Illustrate results to the user.
- Repeat until the user input matches the exit code.

These tasks are carried out by a handful of static methods, each performing a clear, well defined function. The ***query()*** method collects user input and calculates the *tf-idf product* of the query.

The ***calculateCosineSimilarity()*** method produces the *tf-idf products* of all documents and proceeds to calculate the angle ϑ, between each document and the given query, based on their respective weight vectors. The results are stored in *HashMap<String, Double> Angles*.

The last objectives listed are met by the according assistant methods [*dotProduct()*, *sortHashMapByValues()*, *printMap()*].

Crawler & Indexer Threads

The concurrence of the application is made feasible due to the *IndexerThread* and *CrawlerThread* objects running in parallel in the background. They are simplistic classes implementing the *Runnable* interface, making sure each Thread performs the desired operation, whilst avoiding conflicts and keeping track of the right timing.

Padlock

This class' objects are fundamental for the concurrency of the application. They perform the lock mechanisms, which prevent parallel threads from colliding when editing static structures.

Conclusion

The application runs smoothly as an IntelliJ IDEA project. The *Jsoup* library is used for the initial data processing. In the future, the implementation of the web interface will be completed with the help of the *Spark* framework. So far, the publicly available source code does not include definitions and comment explanations.

Any questions, requests, or notes can be forwarded to *parmencd@csd.auth.gr*.

Thank you for reading.