**Homework-3 Report**

Name: Rohith Chandra Kandambeth

1. **Crawling [ Note down steps implemented for each of the below]**
   1. **URL Canonicalization –**

* Parse the URL: Break down the full URL into its constituent parts using the urlparse function. This includes the scheme, netloc (which includes domain and port), path, parameters, query, and fragment.
* Lowercase scheme and netloc: Convert the URL's scheme (e.g., 'HTTP', 'HTTPS') and netloc (the domain and optionally the port) to lowercase to eliminate discrepancies due to case sensitivity.
* Remove default ports: If the URL's scheme is 'http' and includes port 80, or if it's 'https' and includes port 443, remove the port number. These are the standard ports for their respective protocols and are therefore redundant.
* Handle relative URLs: Ensure the path of the URL begins with a forward slash to convert any relative URLs to absolute ones, establishing a clear and complete path.
* Eliminate fragments: Remove the fragment portion of the URL, which starts with a '#' symbol, as it usually points to a specific part of a page and is not necessary for the base URL.
* Consolidate duplicate slashes: Replace any instance of double slashes ('//') found in the URL's path with a single slash to avoid treating similar URLs as different due to mere formatting variations.
* Reconstruct the URL: Reassemble the URL into its whole form using the urlunparse function, piecing together the processed components into a standardized, canonical URL
  1. **Frontier Management**

**Initial steps:**

1. Canonicalize the seed URLs
2. Convert the URLs to objects of the URL class defined with attributes – url, wave number, inlinks, outlinks, anchor\_text, score.
3. Add the seed URL objects to a deque called frontier

**Next steps:**

1. Start the crawling process for all the URL objects present in the frontier
   1. Checks if it can fetch the robots.txt file and checks if it produces any error
   2. If no error it continues to parse the pages for links, titles and content after checking if the content-type is html/htm.
   3. Adds the contents along with the outlinks to a dictionary and updates the file
2. Every 500 urls crawled is saved as one file
3. After the frontier has done crawling, the result dictionary is used to update the frontier. It adds the already crawled urls to visited and adds outlinks to the frontier after canonicalization. During the above process the inlinks, anchor texts are updated and ensures the url crawled already is not used again.
4. The frontier now contains the url objects for the next wave. Scores of these objects are calculated based on inlink count, similarity with previous anchor texts and keywords.
5. Based on descending order of scores, the frontier is sorted and ready for crawling again
6. The above steps are repeated until the required number of documents are crawled.

* 1. **Politeness Policy**

1. I instituted a strict rule of making no more than one HTTP request per second to any domain, preventing server overloads and aligning with common courtesy in web data extraction practices.
2. Prior to the initiation of crawling any domain, the robots.txt file was fetched and parsed using a reliable third-party library. This enabled my crawler to respect the site-specific rules delineated within the file, thereby avoiding any areas the site administrator intended to keep off-limits for crawlers.
   1. **Document Processing**
3. The extracted data includes url, title, content, outlinks, inlinks. The url, title and content is stored in the format

<DOC>

<DOCNO>http://www.example.com/something.html</DOCNO>

<HEAD>The page title</HEAD>

<TEXT>The body text from the document</TEXT>

...

</DOC>

1. These are indexed after preprocessing, removal of stopwords and stemming.
2. **Vertical Search**
   1. **Add a Screenshot of your Vertical Search UI**

**A screenshot of a computer

Description automatically generated**

* 1. **Explain briefly how you implemented it.**

We used the Streamlit app functions as a vertical search interface that queries an Elasticsearch index to display relevant search results for a user's input query. It begins by prompting the user to input their search term. Upon entering a query and initiating a search, the app communicates with an Elasticsearch instance using defined access credentials.

The Elasticsearch search method is invoked to execute a multi\_match query against both the title and content fields of documents within the 'crawler' index, aiming to find matches with the user's query and limiting the results to 25 documents.

For each hit returned by Elasticsearch, the app constructs a display list that includes the document's content, author, and title, keyed by the document ID. The search results are then presented to the user in an interactive manner. Each result is encapsulated within an expandable section, displaying the domain extracted from the article's URL, the article ID, and its title.

The content of each article is processed to highlight instances of the search query, wrapping matched terms with HTML span tags to apply a yellow background for visibility. This highlighted content is rendered within the Streamlit interface, allowing HTML through the unsafe\_allow\_html parameter to enable the styling.

If no results are found, the app informs the user accordingly. This search interface is an example of a vertical search engine specialized in a particular segment of the web, as represented by the 'crawler' index in Elasticsearch.

1. **Extra Credits Done [ Note done what was done for each extra credit ]**

EC1 – Crawled 180,000 documents as a team

EC4 – Used concurrent futures to speed up the crawling process