# HomeWork-3 Report Template

1. **Crawling [ Note down steps implemented for each of the below]**
    a. **Any changes in the seed URLs provided?**
    b. **URL Canonicalization –**
    c. **Frontier Management**
    d. **Politeness Policy**
    e. **Document Processing**

a. seed_urls = ["http://en.wikipedia.org/wiki/Cold_War",
"http://www.historylearningsite.co.uk/coldwar.htm",
"http://en.wikipedia.org/wiki/Sino-Soviet_split",
"https://www.marxists.org/history/international/comintern/sino-soviet-split/"]
Add one more wiki Cold_war URL

b. Create url blacklist for: set([
".jpg", ".svg", ".png", ".pdf", ".gif",
"youtube", "edit", "footer", "sidebar", "cite",
"special", "mailto", "books.google", "tel:",
"javascript", "www.vatican.va", ".ogv", "amazon",
".webm", ".mp3", ".mp4", ".avi", ".mov",
".zip", ".bin", ".dmg", ".pptx", ".xls", ".ppt", "xlsx"
])
Converting the scheme and host to lowercase, removing the "www" prefix, omitting standard ports for HTTP and HTTPS, and cleaning up the path to avoid duplicate slashes.

c. Frontier management in my web crawler is handled through a priority queue, which prioritizes URLs to be crawled based on several factors: The crawler uses a PriorityQueue from Python's queue module for managing the frontier, where the frontier consists of URLs that have been discovered but not yet visited. URLs are selected from the frontier for crawling based on their priority. Each URL in the frontier is wrapped in a

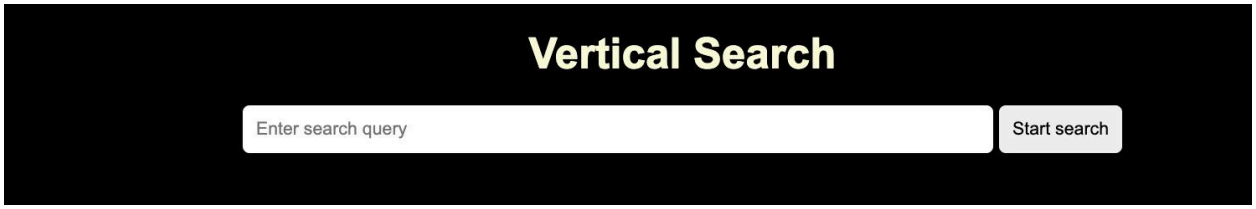
FrontierItem object, which contains metadata about the URL such as its score, wave number, domain, and whether it's a seed URL. The score is used to prioritize URLs, with higher scores being crawled first.
The scoring of a FrontierItem is based on several factors: In-link Count, Keyword Matches, Seed Status, Preferred Domains, Wave Number, Timestamp. URLs are added to the frontier when new links are found during the crawling process. Each URL is added with its calculated score, and the priority queue ensures that URLs with higher scores are processed first.

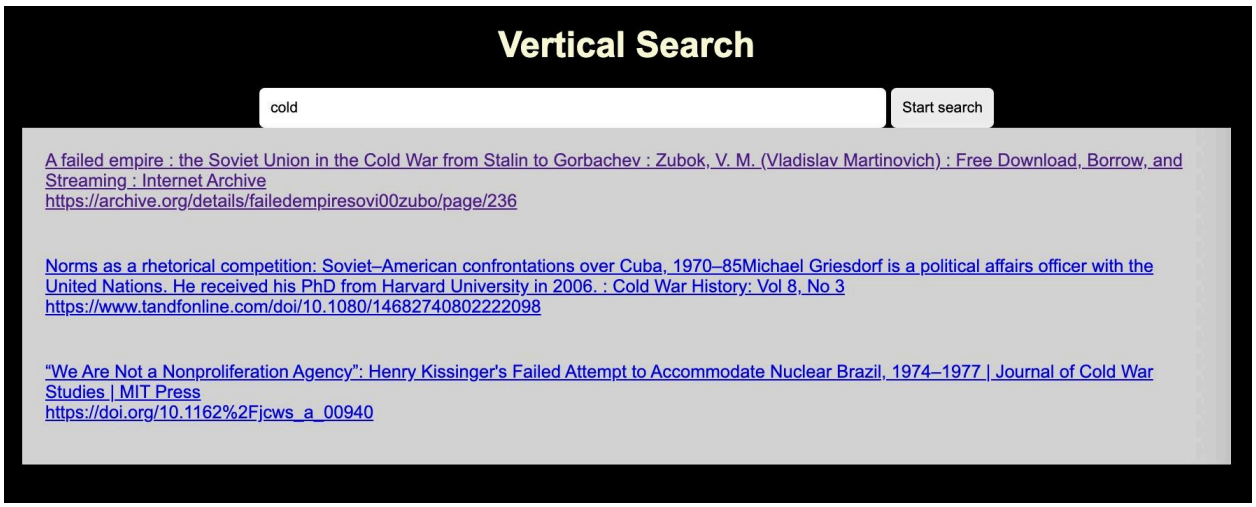
d. The crawler enforces a delay between consecutive requests to the same domain to avoid overloading servers. For each domain, the crawler keeps track of the last request time. Before making a new request to the same domain, it checks how much time has passed since the last request. If the time since the last request to a domain is less than 1 second, the crawler waits until the threshold is met before proceeding with the next request. By this:
```
if time_since_last_request < 1:
        time.sleep(1 - time_since_last_request)
```

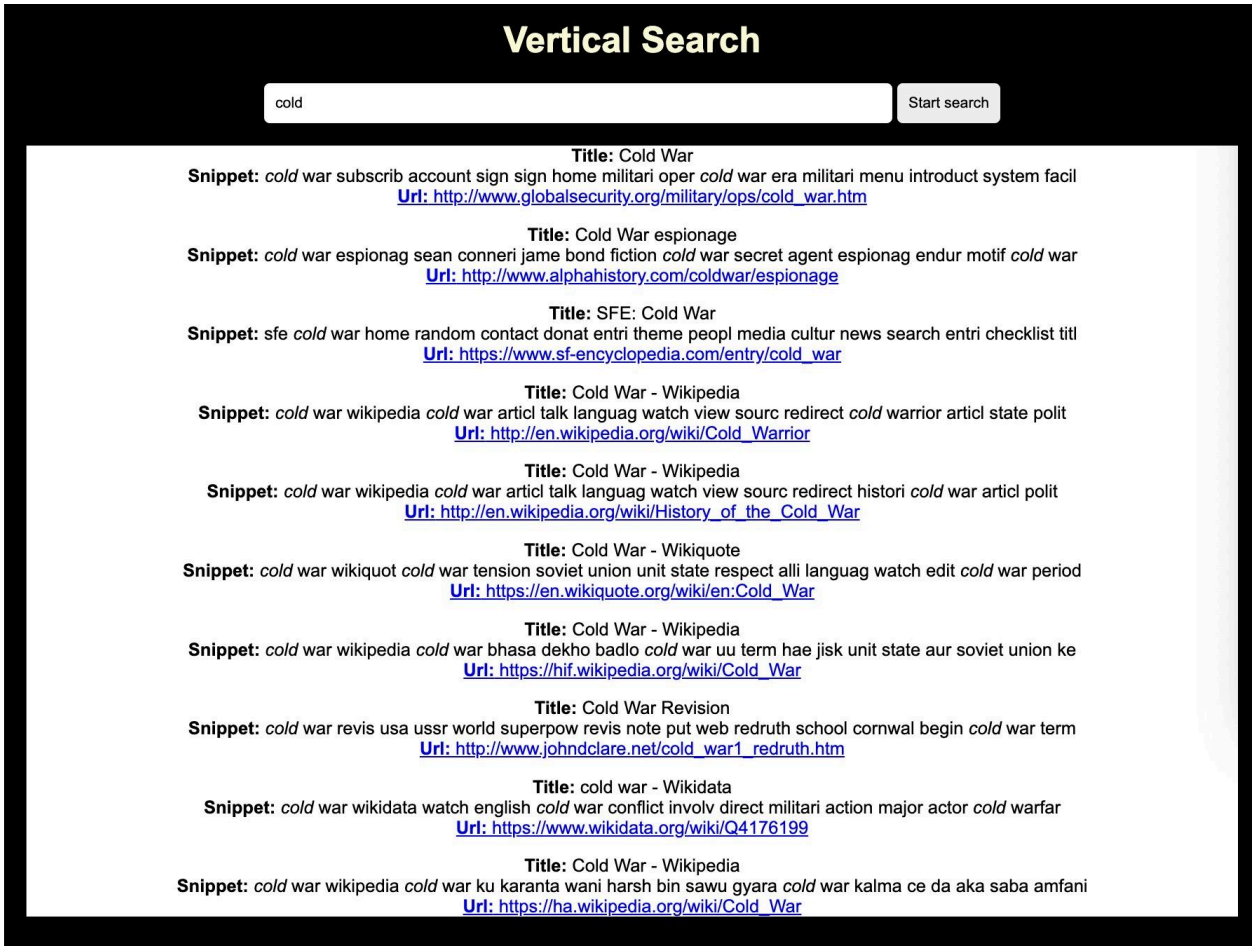
e. Document processing involves several steps designed to extract, store, and organize the content retrieved from crawled web pages. When a web page is successfully fetched, the crawler uses BeautifulSoup, a Python library for parsing HTML documents, to parse the page content. It then extracts the textual content and the title of the page for further processing. The extracted text undergoes normalization to remove any irrelevant content such as script and style elements. The extract_text method is responsible for cleaning up the HTML content and separating the text from the markup. And uses the langdetect library to determine the language of the extracted text. Pages not in the desired language are skipped, streamlining the data collection to relevant content. For each processed web page, the crawler creates a structured document containing the URL, title, and extracted text. This document is stored in a specified format, making it easier to access and analyze the information later. Your crawler uses a custom method, write_ap89_doc, to format and save the document in a structure reminiscent of the AP89 document format commonly used in information retrieval tasks. The process_links_found method extracts and processes links found within the HTML content. This involves canonicalizing URLs and adding them to the frontier if they haven't been visited yet and are not in the blacklist. Before fetching a URL, the crawler checks the site's robots.txt file to ensure compliance with the site's crawling policies. Finally, the crawler logs significant events and errors during the whole process.

2. **Vertical Search (FOR CS6200 only)**
    a. **Add a Screenshot of your Vertical Search UI**
    b. **Explain briefly how you implemented it.**

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






I start by creating a Flask application instance that will handle HTTP requests and responses. Flask routes are defined to respond to specific endpoints, such as the root (/) for the home page and /search for search queries. It provides two key functionalities: performing searches and offering autocomplete suggestions. Searches are conducted through a GET request, utilizing Elasticsearch's multi-match query across specified fields, and results are returned in JSON format. For autocomplete, a similar GET request is made, employing a prefix query to fetch suggestions based on user input, also returned as JSON. Both functionalities utilize Elasticsearch queries to retrieve relevant data from a predefined index, ensuring users can search for and receive suggestions efficiently.

3. EC1&EC2: team crawl to 180,000 documents

## Available indices

indices indices Filter El

| Index name | Index health | Docs count | Ingestion na... | Ingestion me... | Ingestion sta... |
|---|---|---|---|---|---|
| craw_chenxuan_xu_2 | green | 35033 | | API | Connected |
| general_crawler | green | 181540 | | API | Connected |
| general_crawler_for_test | green | 14506 | | API | Connected |
| metrics-endpoint.metadata_current_default | green | 0 | | API | Connected |
| search-chenx | green | 70008 | | API | Connected |
| search-hw3-wy | green | 23142 | | Crawler | Connected |

EC4: My crawler uses a ThreadPoolExecutor with a number of threads equal to the CPU count of the machine. To allow multiple URLs to be processed in parallel, significantly increasing the crawling rate. Since each thread can independently fetch and process a URL, the overall time taken is reduced compared to a single-threaded approach.

```
with ThreadPoolExecutor(max_workers=num_threads) as executor:
        futures = [executor.submit(self.crawl) for _ in range(num_threads)]
        for future in as_completed(futures):
                future.result()
```

EC5: search engine interface.