Demo Video Link:
https://drive.google.com/file/d/10xEIz74D7MCH7c70pdOdEeNd7GiaOYIN/view?usp=sharing

Introduction:

The main goal of my project, at least at first, was to build a simple search engine. The project was designed to be split up amongst the construction of two components: a web app for users to query the search engine and a web crawler to do the indexing behind establishing the database that this web app would use. Stretch goals at the beginning largely revolved around the app and improving its querying power; website database that the app uses stores only a table of URLs and the titles of the webpages they lead to and the naive, starter implementation of a search algorithm was simply to see if any search terms appeared in the URL or title of a webpage, so the biggest stretch goal was to see what could be done to improve this. Another stretch goal was query based searching; if relevant results for a given query were sparse, the few results that were relevant could be further crawled and hopefully other relevant results could be found.

As I began building this project though, I realized that most of these stretch goals would be hard to attain within the time allotted and decided that the web app portion of the project should be treated as somewhat secondary; building the web app did not require me to use much of what I'd learned in this class whereas building the crawler did, so I decided to dedicate myself more towards building a high quality web crawler than improving upon the basic web app. As of now, both components are working properly and about as I had hoped when I made the decision to pivot more heavily towards the crawler. Woo!

Design/Implementation

To discuss the design of the project, I'll start with an annotated file tree:

```
final-project-Zarquon0
       | - search_engine_app/  # App folder (virtual environment)
              | - search_backend.py  # Main backend file
              | - search.py  # Search functionality
              | - search_db.db  # Sqlite database for URLs
              | ~ [a bunch of other backend helper files]
              | - templates/  # Contains app's HTML file
              | - static/  # Contains app's CSS file
              | - Makefile
              | - crawler  # App's copy of crawler binary
              | ~ [a bunch of virtual environment files and folders]
       | - web_crawler/  # Crawler folder (cargo project)
              | - src/
                     | - main.rs  # Main crawler file
                     | - crawler_utilites.rs  # Crawler helper functions (mostly for
page parsing)
                     | - crawler_datatypes.rs  # Crawler data types
                     | - database_interaction.rs  # Database loading/editing functions
                     | - prelude.rs  # Shared imports file
```

```
                    | - url_tree.rs  # [Legacy] Custom data type that became
irrelevant
              | - crawler  # Crawler's copy of crawler binary
              | - Makefile
              | ~ [cargo workspace files]
       | - Makefile  # Provides functionality for quick setup and running everything
together
```

This tree is a little busy, but the broad strokes are as follows. The search_engine_app folder (a virtual environment) holds all of the files relevant to the functioning of the web app. I used Flask for the backend of the web app, so the structure and contents of this folder follows the framework's standards. The excess of files makes the app seem like a lot more than it truly is; at the end of the day, the app is just a singular page being served via a singular view (route). While running, the app listens for any incoming requests for its index page (its only one) and serves it accordingly. The page contains a single form to accept user search input, and upon the app's reception of this form, a list of relevant links is compiled and relayed back to the user in the form of a scrolling list. The search bar stays present, so users may continue to search after receiving results. The most interesting file in this folder is search.py, which contains all of the logic for parsing search queries and returning a list of relevant links. In brief, upon reception of a query, the app queries search_db.db for all of its URL/title pairs, runs a linear search for URLs or titles that contain any of the terms provided in the original query, and then ranks the results by how many terms they matched on. As discussed earlier, it had been my hope when starting that this search algorithm might be improved, but time ran short and this became a lesser priority for me. Additionally, the app has access to a binary of the web crawler; originally, I wanted the app to have the ability to call this crawler to find more links in response to a user's query, but this also became infeasible due to time constraints.

The web_crawler folder (a cargo project) holds all of the files relevant to the web crawler, which I built in Rust. Building the web crawler is where I spent the majority of my time on this project and requires a more in depth explanation than the web app. At its most basic, the crawler accepts a URL (or list of URLs) as an argument, which it uses as its start point (or starting points). It then spawns a series of worker threads (crabs, as I like to call them) that proceed to crawl until a maximum number of URLs have been obtained.

Each crab keeps its own store of links that it has discovered. At the start of each crab's event loop, a given crab will remove a link from this list, check that no other crab has searched this link (more on both of those actions in a sec), and then send a GET request for the web page associated with it. Upon receiving this web page, the file is canvassed: any links found in anchor elements are added to the crab's local store of links and if a title was found, it gets paired with the original link. Having been properly searched, the original link (and its associated title, if it has one) are stored, and the cycle repeats itself.

The crabs are synchronized by two shared objects: a SiteMap and a LinkList. The LinkList provides a shared cache of a small number (equal to the number of crabs) of links that crabs can pull from if they run out of links in their local store to crawl. If the LinkList ever contains a number of links that is below its set capacity, crabs will add links they discover to it until it reaches capacity again, and the input starting point links provide the first contents of this list. In practice, the LinkList is never accessed after a few seconds of crawling, as each crab will have built up a formidable local store of links, but this shared object is crucial to proper startup.

The SiteMap is where links that have been searched by crabs go for storage till the end of the crawling process and is also the object that allows crabs to check whether a given link has already been searched or if the target number of links to search has been reached. A hashmap of URL/title pairs as well as a hashset for links that caused crabs to encounter errors and a hashset of previously searched links (when hooked up to database; will be discussed later) allow for efficient storage and a constant time check of whether or not a link has already been searched for. The length of this hashmap of URL/title pairs is kept track of as well, allowing the SiteMap to notify crabs when the target number of links has been met.

There exists one final important object in the crawler's implementation: LocalLinks. The LocalLinks object is what each crab instantiates to hold their local collection of links. A custom data type is used here instead of a queue or vector because I discovered early on that continuously crawling whatever the first link in this local store happened to be did not lead to the most diverse results. Web pages often have groupings of links that all lead to pages from one particular site or domain (lots of random and uninteresting github links in particular), so I designed the next() method of LocalLinks to skip around a little bit when it detects these groupings. Additionally, these local stores of links will grow to the hundreds of thousands if given time and it makes more sense every once in a while to explore a completely different section of this store; LocalLinks::next() will also jump to a random index in its internal store of links on occasion to shake things up.

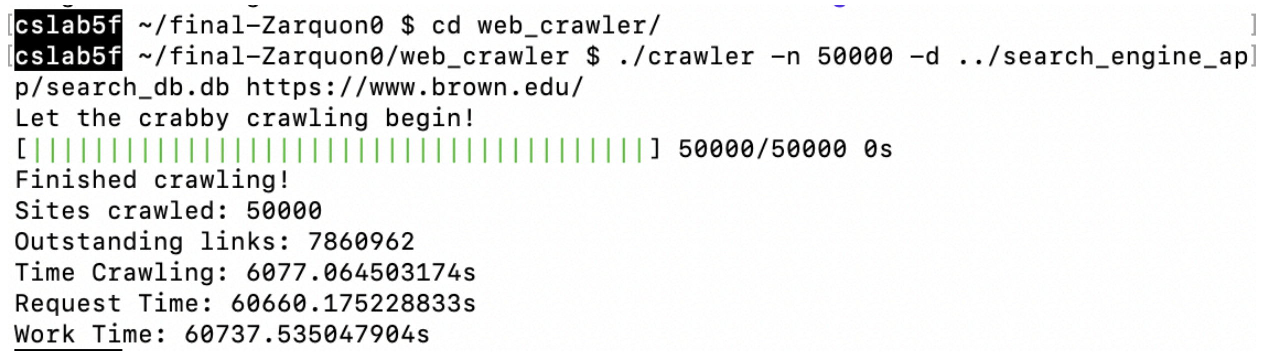
Once the target number of links is reached, each crab will terminate and a handful of statistics related to the searching process will be printed. The crawler will do nothing with its accumulated data by default, however, if a user specifies the path to a properly configured database when running the crawler, this data will be written into it. The other operational difference invoked by running the crawler with a database is that the crawler will read all entries in that database at startup and prevent any of its crabs from searching links that already appear there.

The crawler does not have a REPL, but it comes with a robust set of options that may be specified upon execution. Options include specifying the number of target URLs (default is 100), specifying the number of workers (default is 10), providing the path to a database (as previously discussed), enabling strict mode (any malformed starting point URL will trigger a panic), and specifying a log-level (verbosity). The default log-level is 1, which prints a startup message, a progress bar, and a handful of stats upon completion. A log-level of 2 will cause each individual crab to print individual stats upon their own termination, a log-level of 3 will cause messages to be printed every time a link is searched (either a success or error message), and a log-level of 0 will suppress all printed output.

Finally, the last thing I'll note about design and implementation is the project's Makefile structure. The idea behind the Makefiles is to allow anyone with a UNIX system to clone the repository, run a single make command to set up the virtual environment, gather all dependencies, and perform all additional setup, and then be able to run one other make command to run a production version of the app on localhost. I am not the make wizard I wish I was, so I cannot say for sure whether or not I succeeded on that front, but this was another design consideration for me.

Discussion/Results:

Overall, I'm pretty happy with the results of this project. While the app itself remains fairly primitive in its functionality, it accomplishes its base functionality seamlessly and forfeiting improvements in the app allowed me to create a crawler of a quality I am proud of. There are no known bugs in either the crawler or the app. As a way of quantifying the efficacy of the web crawler, when running it on a Sunlab computer with the default 10 workers and a target of 50,000 links, it successfully terminated after roughly 100 minutes (see below).



```
cslab5f ~/final-Zarquon0 $ cd web_crawler/
cslab5f ~/final-Zarquon0/web_crawler $ ./crawler -n 50000 -d ../search_engine_ap
p/search_db.db https://www.brown.edu/
Let the crabby crawling begin!
[||||||||||||||||||||||||||||||||||||||||] 50000/50000 0s
Finished crawling!
Sites crawled: 50000
Outstanding links: 7860962
Time Crawling: 6077.064503174s
Request Time: 60660.175228833s
Work Time: 60737.535047904s
```

This may sound a bit slow (and in fact, I am certain it could be made faster), but what I find most encouraging from this capture is the slim difference between the Work Time and Request Time numbers; Work Time is the total amount of time that crabs spent running before terminating while Request Time is the total amount of time that crabs spent simply making GET requests, so the difference between the two numbers is the amount of time the crabs spent parsing pages and interacting with synchronized objects to save a URL/title pair or find out which link to crawl next. It took substantial effort to get this difference to be as low as it is now, and this is a sign that little can be done to more fully optimize the code I have in place. One thing that could be done to speed up the crawler is to incorporate asynchronicity into each crab; each individual crab is sitting around doing nothing while blocking GET requests are sent, so using Tokio to allow each crab to continue to work while GET requests are sent would speed things up substantially. I am, however, content with the current functionality of the crawler and the efficiency of the code that is already in place.

As far as challenges faced along the way goes, I definitely encountered a good handful, mostly with regards to the crawler. The biggest challenge I faced was probably a deadlocking issue that would crop up occasionally while crawling. This bug was particularly challenging because of the breadth of the places it could be happening (at first, I couldn't even be certain that the halting wasn't just a really slow GET request) and its nondeterminism (adding print statements in certain places could cause it to appear less often). After an hour or two of messing around with it, I managed to find where the bug was: earlier, I had used an online implementation of a condition variable for read-write locks that I had desperately needed (see here if curious: https://github.com/Amanieu/parking_lot/issues/165) and a rare edge case in the way I used this makeshift condition variable was causing an absolutely insidious deadlocking bug. One thing you could say I learned from this is not to trust online fixes even if they have only received positive feedback, as doing so cost me a significant amount of time.

Conclusion/Future Work:

Other things I learned from this project besides the trustworthiness of online fixes includes an improved understanding of multithreading and synchronicity, how to leverage HTTP to explore contents of the internet, and how to design and organize a network application in an efficient and intuitive way. I’ve never built an application with this many moving parts of this scale all on my own before, so I learned a lot of little things here and there about project management as well.

If I were to continue to work on this project in the future, the first thing I’d do is incorporate asynchronicity into each crab as discussed earlier. I would then add support in the app for users to request further crawling of the web to find more relevant results to their query and would then have to add a term specific crawling functionality to web crawler. The next thing I’d do is focus on improving my search algorithm, perhaps by collecting more data about a given website than its title (frequency of certain words for instance) and factoring this in. Other improvements could include spiffing up the front end to look a little cleaner (perhaps adding some cool CSS transitions) and other user experience improvements (fuzzy searching?). There remains much that could be added (it’s not quite Google yet), but I do feel a sense of closure at the point where I am currently, so I am happy with what I’ve already produced as it is.