CS685 HW 3 Report – Josh Wilson

I found that designing and implementing a simple crawler was a fun and thought-provoking task. In this report, I'll outline the design, implementation/optimization, and analysis of data for said crawler. I will also discuss the challenges faced along the way.

Design

I knew that I wanted to design this project with C# data types and functions in mind, as it's a language I commonly use to write backend systems. My initial architecture for this assignment included a main function that would assign discovered URLs to separate asynchronous threads for maximum performance, but given time constraints and complexity of implementation, I shelved that design for potential use at a later date. What I ended up settling on was a synchronous design, at the heart of which was a queue for processing URLs and two dictionaries (hashmaps) that would track discovered files and visited HTML pages. The queue would be of type string, the file dictionary would map strings to strings (i.e. urls to file types), and the dictionary for visited pages would map strings to DateTime objects (i.e. urls to date fetched/parsed). My biggest regret with this initial data design was not converting strings to URI objects from the very beginning, as that would have made many of the parses/checks more uniform.

In terms of functionality, the program would start by reading the next URL from the queue, fetching its HTML content as a string, and then extracting all links from said HTML content. The links would then be processed to check what kind of content they represent: they'd be lumped into HTML pages, or some variety of file. This is also the point where the program would check to ensure that the links were within the engr.uky.edu domain. Next, it would perform two checks on the links: one to ensure that the links are not in the URL queue, and another to ensure that the links/files were not present in their respective dictionaries. If the checks for a file pass, the file would be added to the file dictionary. If the checks for a link pass, then the link would be added to the URL queue. After all this, the "parent" URL from which the set of files/links were parsed would be added to the visited URL dictionary. This process would then loop until the link queue becomes empty.

Lastly, in terms of dashboarding, I did something that I believe aligns with the spirit of HTTP. I designed a Rest API in express JS that would receive POST messages from the C# application. Then, GET messages could be used to view live counts of pages crawled, errors encountered, and files discovered.

Implementation

This project was implemented almost entirely with native dotnet functionality. Functions and data types from System.Net, Systems.Collections.Generic, and System.Text namespaces were enough to cover ~90% of the described functionality. The only piece of the project where this wasn't the case was the actual parsing of links from HTML content, for which I relied on a nuget package called HtmlAgilityPack. This package allowed me to use xpath expressions to select specific node properties from the fetched HTML documents. The specific node properties which I set said expressions to pursue

were the “href” properties of <a> and <link> tags, and the “src” properties of <img>, <script>, and <source> tags. I’m not going to make the claim that these sought properties covered the entirety of the domain’s link structure, but I would anticipate that they represented the vast majority.

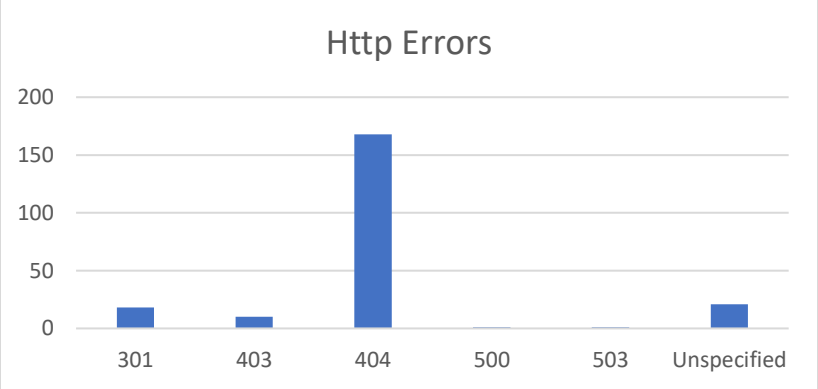
In terms of running checks on link structure and ensuring that they were within the top level domain and conformed to the valid structure of a URL in general, I simply made use of native string functions and regular expressions, as well as Uri objects. Making sure that links were not represented in other forms was a challenge that certainly presented a few bugs to me. For example, the link “https://engr.uky.edu” could also be represented as “http://www.engr.uky.edu”, which might cause said link to be inserted into the URL queue twice. Properly handling these link format edge cases was one of the more difficult parts of doing this implementation from scratch.

Another challenge faced was the fact that certain pages presented HTTP errors, as well as a few errors that fell outside the bounds of HTTP. I used try/catch logic to combat this, and having taken an interest in the counts of these errors, I also implemented a dictionary that would log data about them. I chose to include these errored out pages among the “visited,” since they were technically discovered and crawled, even if the resource did not pan out. However, having typed that, I am not completely sure that I agree with my decision after the fact.

To make use of the metrics dashboarding API, I simply sent asynchronous POST requests from the C# application to respective endpoints upon starting the crawl, finding a page/error/file, and ending the crawl. GET requests could then be sent by an end user to the ‘metrics’ endpoint, which would return the start and end times as well as the counts and contents of each respective dictionary as json.

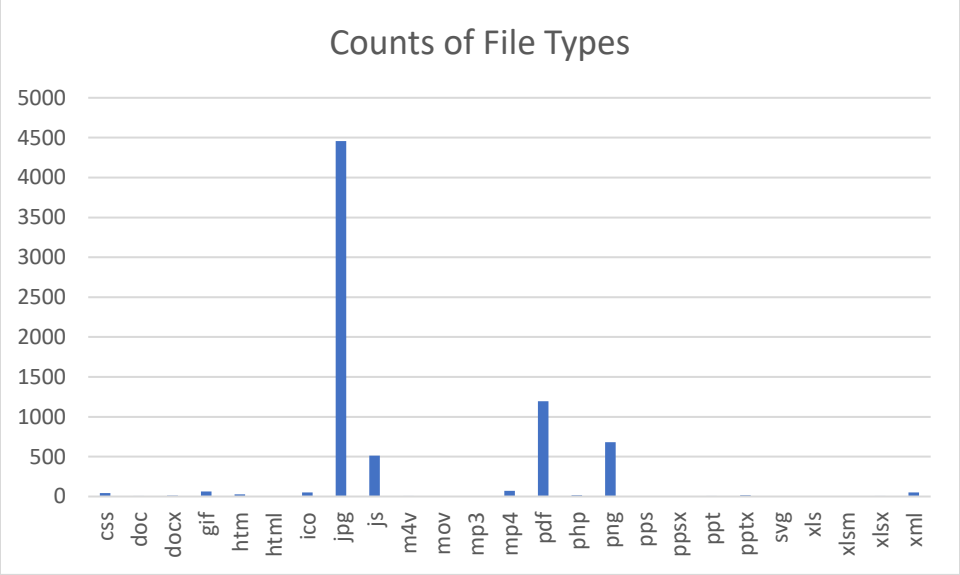
Analysis

For your reference, the raw data from the crawl can be found in the “output” folder of this submission. In terms of baseline statistics, the crawler took roughly 25.5 minutes, and attempted to visit a total of 7178 html pages. Of the pages visited, 219 pages returned errors, and 6959 pages were successfully crawled. In addition, 7241 files were discovered within these pages’ links. All in all, this leads to a crawl speed of roughly 4.69 pages per second. This is not nearly enough for a crawler that would operate at a larger scale, but it could be improved with threading and more asynchronous functionality.



The most frequently encountered HTTP error was a 404, which of course indicates that a page wasn’t found. I also was denied access from ten pages (403) and a few pages were reported as moved

(301). Only two pages returned errors in servers/functionality (500/503), which is honestly quite impressive given the scope of the pages crawled. There were also a number of unspecified errors, which simply are errors that fell outside of the realm of HTTP errors. I did not dig into these on any greater level than that.



The file type data clearly demonstrates that .jpg(/jpeg/JPEG/etc) files dominate the competition within the engr.uky.edu domain. Other common web files such as .js scripts, pdfs, and .png images were also well represented. Of note is that a very small number of html pages did sneak into these counts, so a minor bug is likely causing that to happen.

I also did some digging into the number of subdomains within the greater engr.uky.edu domain. From this crawl, I was able to locate a total of 64 subdomains. The raw data does lay out a count of 66 subdomains, but I forgot to exclude engr.uky.edu from said list, and it shows up there a couple of times.

Furthermore, the dashboarding metrics API worked perfectly, and the numbers aligned exactly with what the console output. On top of this, I was able to use GET requests to watch the numbers of pages, errors, and files steadily increase throughout the crawl. Additionally, this allowed me to do live monitoring of which links were getting sent where, which was invaluable for debugging.

Conclusions

All in all, I'm quite pleased with how this project turned out. I found it to be a challenging and fun problem to solve, and I did not end up regretting my choice of tooling in the end. I also enjoyed keeping my use of outside libraries to a minimum, as I feel it gave me a greater understanding of exactly what the application was doing and where it could be optimized. Future work to be done on this implementation would include further tightening up of the filters and a lift/shift to a threaded, asynchronous implementation.