

Web Crawler Description

Architecture

Since I am new to this, I wanted to build my own crawler to more deeply understand the process. My architecture is of a central thread that manages the shared resources of URLs to search, explored URLs, and the file used for performance monitoring. I chose to make a BFS wikipedia crawler as there are lots of easily parsable links and the category system gives each page a set of keywords. This would massively simplify my parsing code and let me focus more on the multithreading and URL management aspects of building a crawler.

I must admit that the last feature that I implemented was the performance monitor, so I do not have numbers to back up my claims here. When I started this project, I wanted to use a simple database library for Python called TinyDB, but some design choices made it incredibly slow. Every time I would add a URL in the list of URLs I would check for duplicates which made my system far too slow. I removed this requirement and only checked for duplicates of pages I have already visited when I popped that URL off the queue. This change sped up my program by about three times. This is when I started on some premature optimizations. I used a thread-safe linked list implementation built into the python standard library for $O(1)$ appends and pops, and used an external bloom filter library to speed up checking if I had already traversed a URL. The bloom filter was a little overkill, but it works well with not checking for duplicates in the frontier URL list. Since the filter has a probability of giving false positives, then there is a high chance that a future duplicate of the same URL would get past the filter.

My parsing code is incredibly simple, since it is only for the subcase of wikipedia pages. For new pages, I check for hrefs with relative links. For keywords, I use the list of categories that the page belongs to which exists at the bottom of the page, and only exists for content pages instead of navigation pages.

This design has a couple of massive benefits. Firstly, my parse code and performance monitoring is in a medium length function. The central managing class also makes it easy to handle multiple threads accessing the same resources as they are locked behind the mutexes held within the main manager class. Also, the managing class only has 4 functions, those being: add, pop, record, and run. The use of the bloom filter with $O(1)$ pops and appends to the list of new URLs means that my crawler scales well as the number of pages explored increases. For the most part, it looks like my pages explored per minute number remained pretty linear.

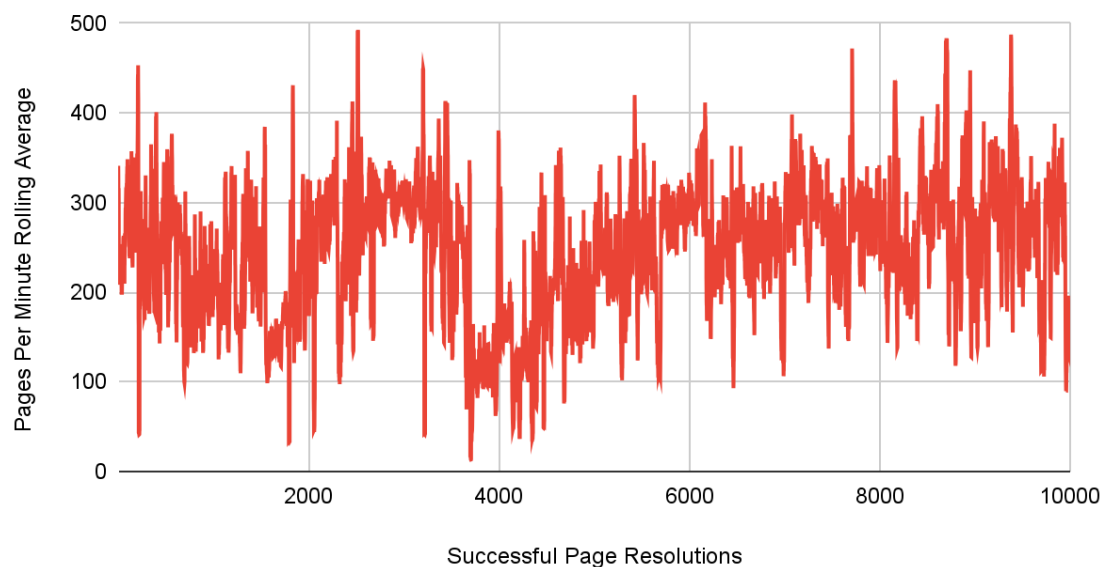
The downsides for my design are in part because of its simplicity. I do not record my list of URLs to explore in a file which means that my heap usage can explode, and this crawler is not resistant to crashes. My code was resilient enough and confined enough where this was not a problem for me. I write to my database file every time a crawler finishes processing a page which is a source of possible improvements. In hindsight, I think that trying to use a database was a bad idea, and I should have used pickle. Pickle is a library for serializing python data structures to files, which would be more efficient and give me the ability to store chunks of data and make my crawler crash resistant. Another issue is that I allow duplicate URLs in my list of URLs that I need to travel. This means that over time the list of linked URLs would become largely ones that I have already visited which makes my discovery of new URLs less efficient. This process of popping URLs off the queue is handled by the main thread, so this could bottleneck the crawling threads in extreme cases.

Considering that my use case was tightly constrained, and my aim was to generate the smartest naive crawler that I could, I feel like I succeeded.

Performance and Monitoring

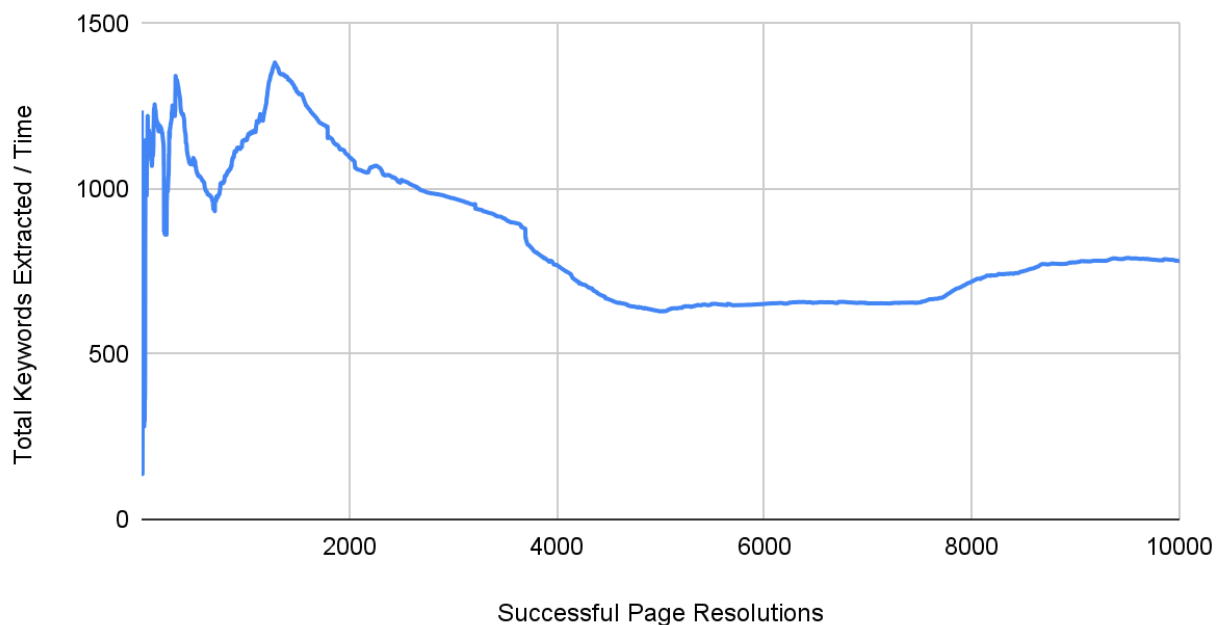
As for the elephant in the room, my average number of pages explored per minute was ~244 and this was for a trial of 10,000 pages. This number was generated by extrapolating from the rolling average of the last 10 page resolutions.

Pages Traversed Per Minute



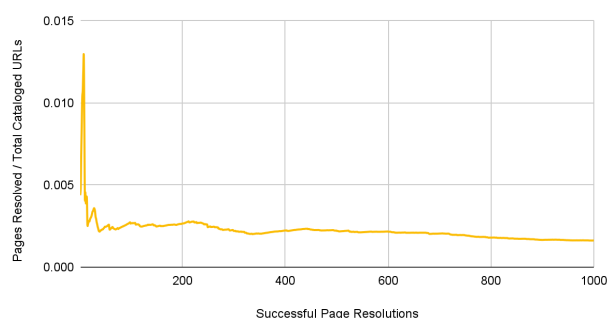
There is a large amount of variance, but that is expected as wikipedia page sizes can drastically change sizes and my laptop was on wifi. Although 10,000 pages is not massive for this size and more testing is needed, I was happy at how the page traversal response did not noticeably slow down over time.

Keyword Rate

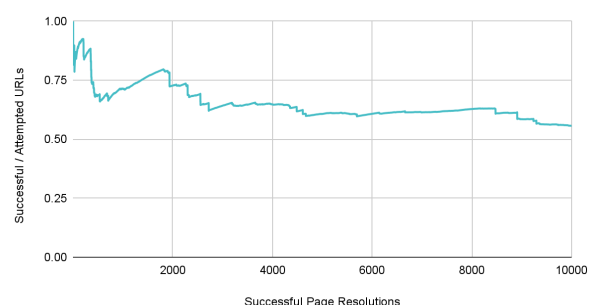


Keyword rate was harder for me to calculate, so I did the total number of keywords resolved over the time the crawler was running as it changed by number of pages resolved. While does not have the instantaneous change of the prior plot, but shows that the keywords per minute converges to 781 over the course of 10,000 pages.

Ratio of URLs Explored



Ratio of URLs Tested to Resolved

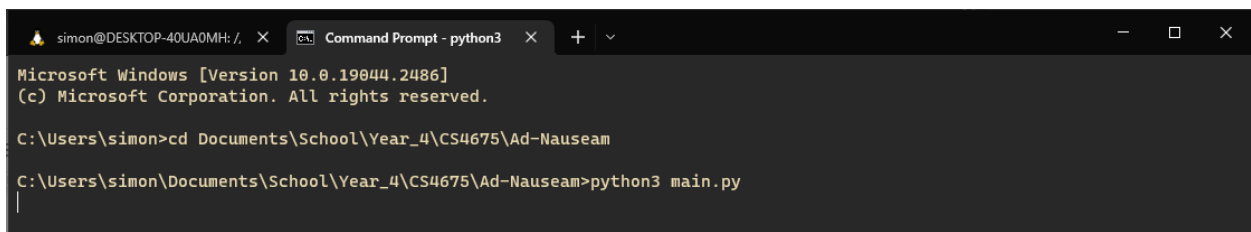


The ratio of URLs explored to total cataloged URLs and Ratio of URLs tested to success are importantly related because of some design choices I made. I allowed for duplicate URLs to appear in my list of URLs that I need to explore, so I would expect that over time there would be more URLs popped from the list that the spider has already visited. This

prediction is true, where around $\frac{1}{3}$ of the URLs popped from the list have already been visited, but the graph seems to be plateauing and list inefficiency does not seem to be a major problem. This also implies that my ratio of URLs explored vs cataloged over predicts the number of links that the spider has not traversed.

Operation

There is not much to say here, I just run the python program. There are no command line arguments for this. Right now, I am just seeding my crawler with a single Wikipedia article, but this has the potential for more. I used to have more print lines when the crawler would go to a new URL, but removed that for speed.



```
simon@DESKTOP-40UA0MH: /  Command Prompt - python3  + - □ ×
Microsoft Windows [Version 10.0.19044.2486]
(c) Microsoft Corporation. All rights reserved.

C:\Users\simon>cd Documents\School\Year_4\CS4675\Ad-Nauseam
C:\Users\simon\Documents\School\Year_4\CS4675\Ad-Nauseam>python3 main.py
|
```

Takeaways

Building something that cleanly scales onto the order of hundreds of thousands is really hard and I am glad that I did not have to. Assuming that I am at the same efficiency, if I wanted to crawl 1 billion wikipedia pages it would take around 7.79 years. The problem is that my URL list would crash my computer or be filled with entirely duplicates long before that is a problem. I think it is really important for me to include a state outside of just running my program. I had to let my program sit for around 1.5 hours a couple of times because I forgot to add the requisite performance monitoring. Also further segmenting my program so that I could do asynchronous reads and writes to disk would make my program much more resilient and may allow better filtering of the URL catalog. What I wrote was a good first step in understanding web crawlers, and as far as naive solutions go, I think it is pretty good, but it highlights the incredible complexity of proper spiders. This project has definitely piqued my interest.