HW1

Junwei Li

Design of my crawler: pros and cons.

I have designed 3 spiders, cc_gatech.py, cc_gatech_DFS.py, cc_gatech_BFS.py. They are all under /spiders folder, and the final submitted version is cc_gatech.py. Other twos are using DFS and BFS algorithm.

All spiders start by visiting the website "https://www.cc.gatech.edu/" and follows all the links it finds on that page. It then extracts the title of each page it visits and counts the number of unique keywords that appear in the titles. It also keeps track of the number of URLs visited.

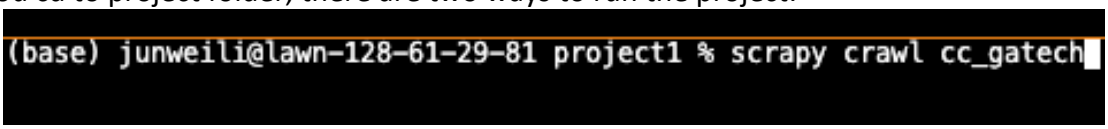
- cc_gatech.py:
    - Pros:
        1. Crawler is well-organized.
           parse( ): Handle the response from crawled page; Extract the title from each page and split into keyword stored into a dictionary for indexing; Extract all links in every page and use response.follow() for directing to another pages until reach 1000 pages.
           closed( ): Handle the output when the spider is finished. The Output contains three parts: Save Keyword dictionary into a CSV file; Plot crawl statistics: (number of keywords)/(number of pages crawled); Print pages/min.
           setting.py: Use CLOSESPIDER_PAGECOUNT=1000 to handle the max number of pages to crawl.
        2. Solve duplication problem.
           Handle the duplicated URLs during crawling. Avoid crawl the page that has been crawled.
        3. Refine the keywords from output.
           Only allow crawl the domain containing “gatech.edu” to avoid collect unrelated data.
        4. Get crawl statistics easily
           The crawl speed is tracked by counting the number of URLs visited and the number of keywords extracted and then plotted in a graph. This can be useful in monitoring the performance of the crawler.

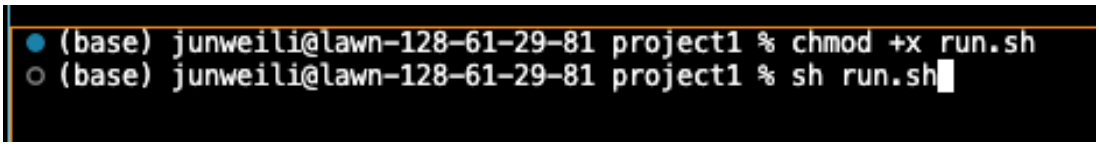
    - Cons:
        1. The code does not implement any rate-limiting for requests, which can lead to the bot being blocked by the website.
        2. The code does not handle any error or exception
        3. The keyword-URLs data is written to a csv file after the crawl is finished. Since it is not written to the file during the crawl, if the crawler crashes before it finishes, the data can be lost.

4. The code uses self.keywords_dict and self.keywords, self.urls to store the keyword-url and their counts, this can lead to huge memory usage if the number of urls are high.

ScreenShots of Web Crawler's Command lines or GUIs.

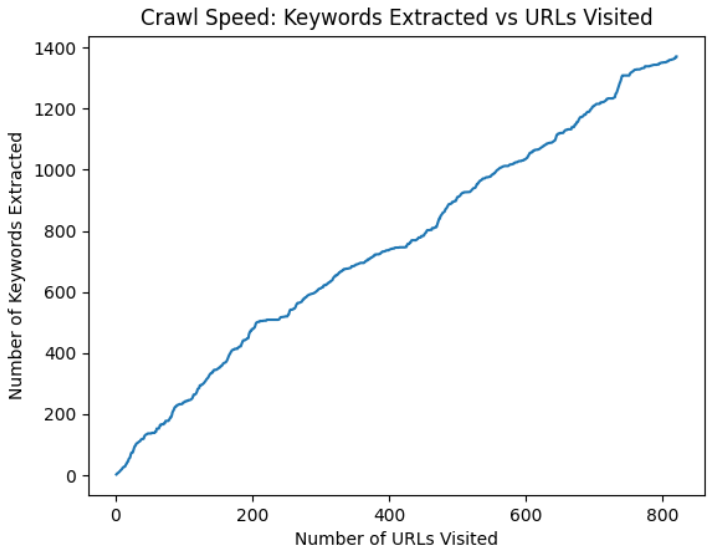
After you cd to project folder, there are two ways to run the project:

1. 

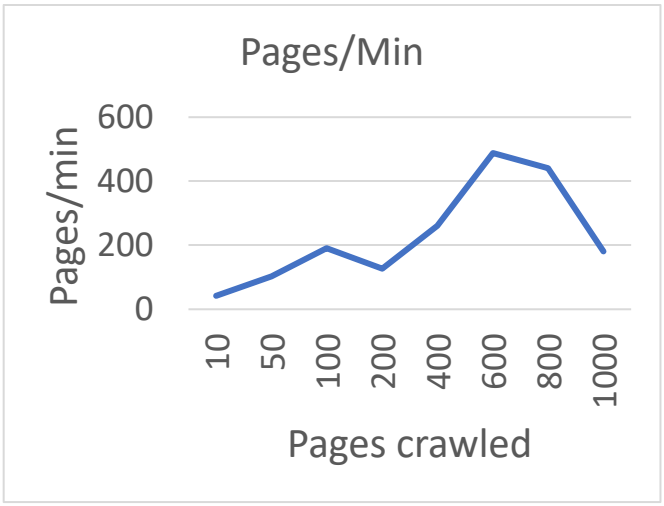
2. 

Crawl Statistics

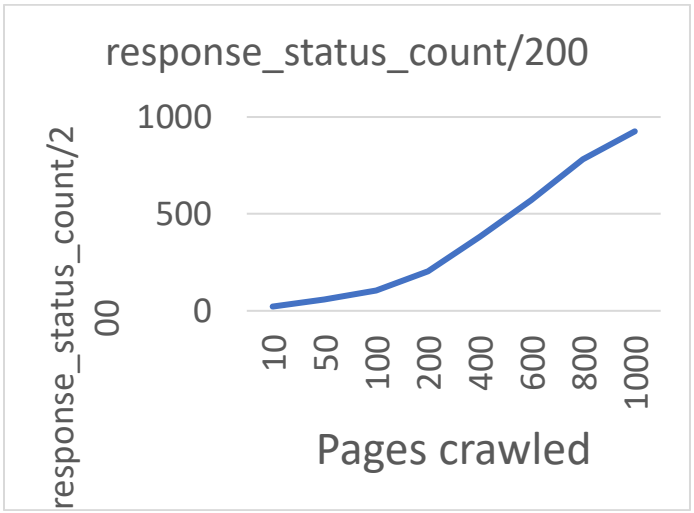
1. (#Keywords)/(#Pages)



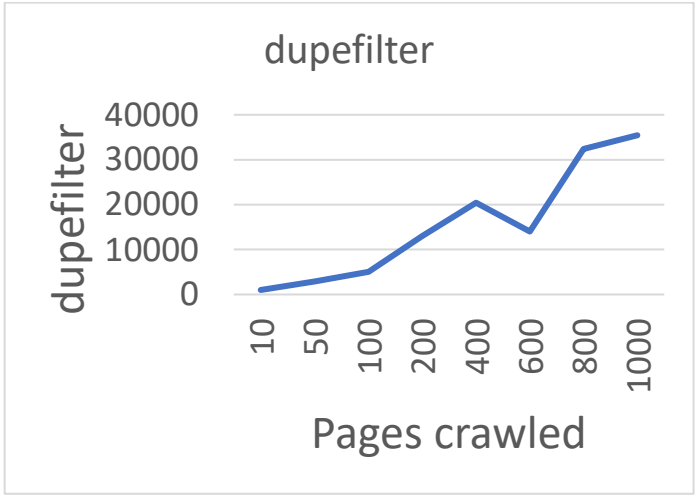
2. (Pages/Min)/(#Pages)



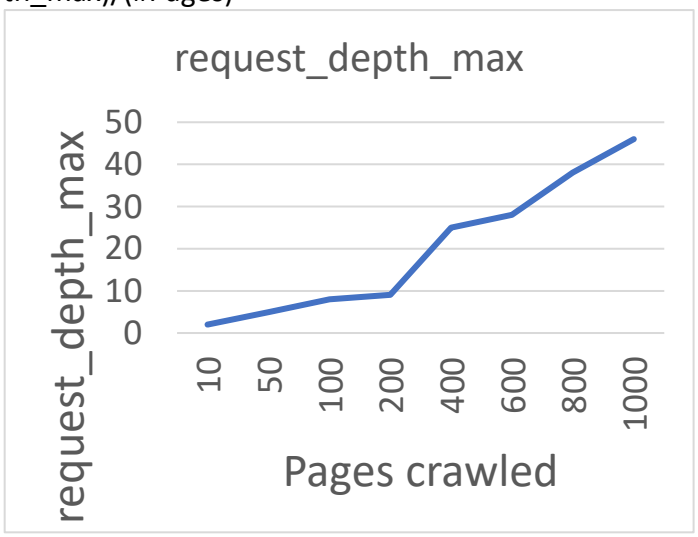
3. (response_status_count/200)/(#Pages)



4. (#depefilter)/ (#Pages)



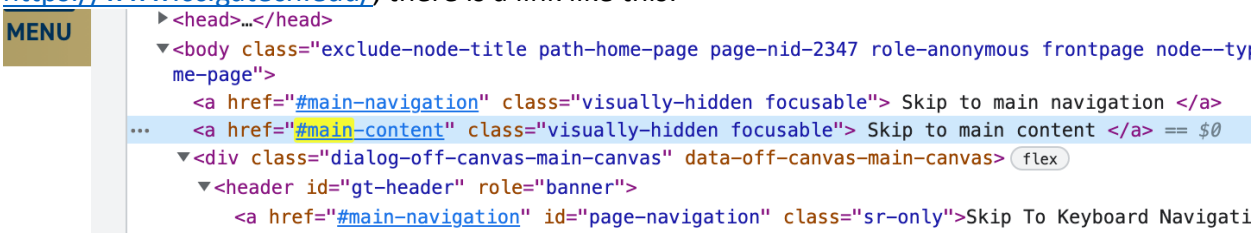
5. (Request_depth_max)/(#Pages)



Discussion

I have designed 3 spiders, cc_gatech.py, cc_gatech_DFS.py, cc_gatech_BFS.py. They are all under /spiders folder, and the final submitted version is cc_gatech.py. Other twos are using DFS and BFS algorithm.

I built cc_gatech_DFS.py first. I found some links in page start with “#”. Like in https://www.cc.gatech.edu/, there is a link like this:



This is the same page with the seed URL. In order to avoid crawl the same page repeatedly, I write this structure for crawl urls starting with ‘http’ or ‘/’:

```
if next_url.startswith('https'):
    next_page = next_url
    yield{'link': next_page}
    #yield{"#": self.page_count}
    yield response.follow(next_page, self.parse)
elif next_url.startswith('/'):
    next_page = urljoin(base_url, next_url)
    yield{'link': next_page}
    #yield{"#": self.page_count}
    yield response.follow(next_page, self.parse)
```

But when running this DFS algorithm, I found the process stuck in a loop. Because it will crawl the first link in every page. If the first link in some pages is the seed link, the process will stuck in loop.

So, I try BFS, and wrote cc_gatech_BFS.py in order to avoiding loop. I use a Queue structure like this:

```
#crawl next URL
for link in response.xpath('*//a/@href').getall():
    self.queue.append(link)
next_url = self.queue.pop(0)
yield{"queue":len(self.queue)}
self.counter+=1
self.keywords.append(len(self.keywords_dict))
self.urls.append(len(self.visited))
self.visited.add(next_url)
```

The idea is, put all links in the page into a queue, and pop it when crawling next page. At the same time, I define a visited list for avoiding crawling the page that has been visited. It works! But only for crawling 100 pages ☹. If crawling more pages, it will not work. Because of I used visited[ ],queue[ ] to store URL, this can lead to huge memory usage if the number of urls are high.
And I found scrapy using DFS with Depth_limit. Which is a better way in web crawl.

Does Scrapy crawl in breadth-first or depth-first order?

By default, Scrapy uses a LIFO queue for storing pending requests, which basically means that it crawls in DFO order. This order is more convenient in most cases.

If you do want to crawl in true BFO order, you can do it by setting the following settings:

```
DEPTH_PRIORITY = 1
SCHEDULER_DISK_QUEUE = 'scrapy.squeues.PickleFifoDiskQueue'
SCHEDULER_MEMORY_QUEUE = 'scrapy.squeues.FifoMemoryQueue'
```

While pending requests are below the configured values of `CONCURRENT_REQUESTS`, `CONCURRENT_REQUESTS_PER_DOMAIN` or `CONCURRENT_REQUESTS_PER_IP`, those requests are sent concurrently. As a result, the first few requests of a crawl rarely follow the desired order. Lowering those settings to `1` enforces the desired order, but it significantly slows down the crawl as a whole.

DEPTH_LIMIT

Default: `0`

Scope: `scrapy.spidermiddlewares.depth.DepthMiddleware`

The maximum depth that will be allowed to crawl for any site. If zero, no limit will be imposed.