Text Technologies – Exercise 1

s0840449

1 Crawler Design

The first design decision I made for my crawler was to use a distributed design. Given a lack of hundreds of machines to distribute the crawl over, I opted to settle for multithreading, using a cyclic producer-consumer approach. 'Fetch' threads pop urls from the frontier, downloading their contents and placing them onto a data queue. A set of 'parsing' threads pop off and scan the contents, extracting any urls. New urls are then placed onto the frontier, completing the cycle.

The frontier is implemented as a queue[1] of priority queues[1], where each nested priority queue stores the urls for a specific domain, based on the coursework-specified priority. While this provides little benefit for this coursework, in a proper crawler this would allow threads to cycle through domains to find available urls. It also allows a simple implementation of the crawl-delay directive when using multiple threads, as each thread pops the domains priority queue when crawling. This means that only one thread can access the domain at a time, and so the crawl-delay is not broken by multiple threads.

As parsing generic HTML with regular expressions is generally frowned upon, I choose to use the HTMLParser[2] package to extract urls. Although this does use regular expressions internally, it *should* be a fairly robust implementation. Unfortunately, it turned out that the Python 2.6 version actually has a bug that caused it to fail in parsing many of the coursework web pages[3]. To fix this, I backported the fix for the bug from Python 2.7, and additionally added a regular expression fallback in case the parser failed again. The regular expression, constructed from various online resources, was fairly standard for matching anchor links:

```
<a.*?href=["\'](["^]+[.\s]*?)["\'].*?>[^<]+[.\s]*?</a>
```

This matches the opening tag, followed by anything up to the href attribute, then captures the linking url, followed by matching the rest of the tag details to finish.

To track previously seen urls I used a dictionary. While more sophisticated than needed, the overhead of using a dictionary is not too significant. One improvement that I didnt implement was to guard access with a bloom filter, which allows for very quick rejection of non-members.

To determine if a url had the correct domain I compared the domain - extracted using the urlparse package - with a whitelist of allowed domains (which for our purposes only contained one domain of course.)

Finally, to parse the robots.txt file I used the robotexclusionrulesparser[4] package. Since the given crawl delay is quite high (1 second between requests, causing a full crawl to take around 13 minutes), I allow the user to selectively choose whether to obey the directive via a flag.

2 Crawl Results

During execution, my crawler tracks all required statistics, plus some additional ones. Table 1 shows the results of three crawls - one that obeyed the crawl delay, one that did not, and a crawl where I used the regex-based parser. Each crawl used 3 fetch and 3 parse threads when running.

	Obeyed Crawl Directive	**Disobeyed Crawl Directive**	**Regular Expression Crawl**
Time Taken	1144.82s	41.25s	N/A
Pages Crawled	811	811	811
Total URLs Found	103,740	103,740	104,329
CONTENT URLs Found	9811 (855)	9811 (855)	9876 (855)
Other URLs Found	93,929 (60,889)	93,929 (60,889)	94,454 (61,090)
Fetch Errors	4; all 404s	4; all 404s	4; all 404s
Parse Errors	0	0	0
Disallowed Domain URLs	0	0	0
Robots.txt Blocked	513 (45)	513 (45)	516 (45)

Table 1: Statistics from 3 crawls. Figures shown in brackets are distinct counts.

As expected, the only difference between the first two crawls is time taken - disobeying the crawl delay gives a 28-times speedup. The regular-expression based crawl actually found more urls (total) of each type than the HTMLParser based ones. This indicates that either the HTMLParser implementation for Python 2.6 is not very good, or there is a mistake in my regular expression syntax, causing it to find more links than it should.

I also compared how the number of unique urls (in the *ir.inf.ed.ac.uk* domain) discovered related to the predictions made by Heaps law. I did not expect the result to closely approximate Heaps, due to the finite nature of the task, but I thought that the initial discovery might match. As can be seen from Figure 1, this was correct - for the first 300 crawled pages the rate of new urls followed Heaps law, but then dropped lower. Indeed, by the 375th page no more unique urls were left.

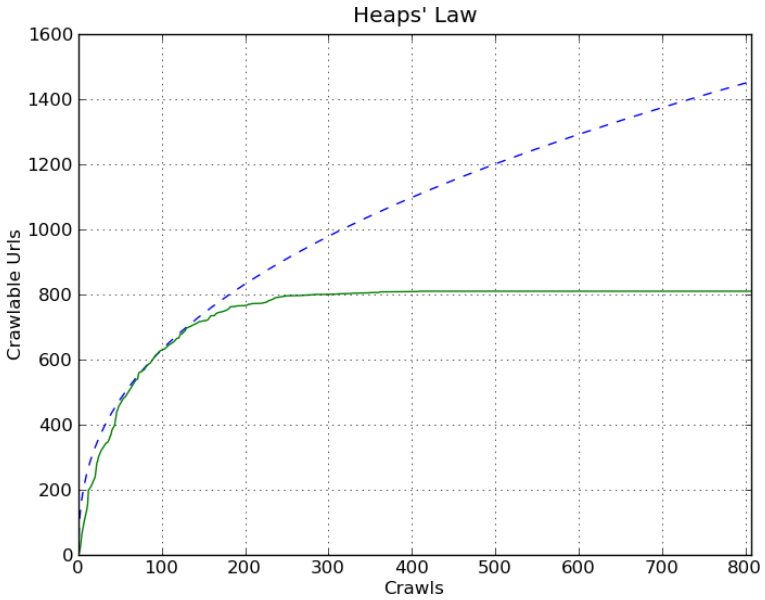

Figure 1: How closely the rate of new urls corresponds to Heaps' law, with $k = 100$, $\beta = 0.4$.

References

[1] Python Queue module. http://docs.python.org/library/queue.html

[2] Python HTMLParser module. http://docs.python.org/library/htmlparser.html

[3] Issue 670664: more robust SCRIPT tag parsing http://bugs.python.org/issue670664

[4] Robotexclusionrulesparser module, http://nikitathespider.com/python/rerp/