Concurrent Thread-based Web Crawler

Final Report

One Semester Individual Project
CM3203
40 Credits

Jack Parsons

Supervised by: David W Walker

Moderated by: Paul Rosin

Contents

0

Project Description

The Internet is a vast resource, and since its inception many people and institutions attempted to crawl and index it; the most popular of which is Google. Since the birth of Google, it has grown to accommodate over a trillion queries per year,[1] which far surpasses the number of queries using its biggest competitor. However, even Google, with its many servers estimates that relatively little of the web is contained in its index. This is partly due to websites blocking certain URLs using their respective robots.txts, explained further in the report, however it is also due to the exponential growth of the web. This means that the current approach to crawling websites needs to be improved to cope with the future growth of the Internet; which is what this project aims to do: it aims to produce a multithreaded web crawler which only downloads webpages it hasn't visited before, or pages which have definitely changed since the last time it was crawled. The project was originally to produce the crawler in Haskell, but was later changed to C++ due to difficulties encountered during development which will be covered later.

[1] Danny Sullivan. *Google Still Doing At Least 1 Trillion Searches Per Year*. 2015. URL: `http://searchengineland.com/google-1-trillion-searches-per-year-212940`.

1

Background

The aim of this project is to implement a multithreaded web crawler which aims to maximise utilization of available bandwidth and CPU through the use of HTTP compression, among other methods. The final product should be runnable through use of a Linux/Unix command line.

With the internet being as large as it is, it is important that any web crawlers may traverse the web as quickly and efficiently as possible. This project will aim to improve upon existing solutions for crawling the web quickly.

Bodies which may be interested in the outcome of this project may be Google, Yahoo, and Microsoft, along with any other company which utilises web crawling software. Google's own GoogleBot may be similar to this in its multithreaded nature according to some studies.[1]

Whilst developing this project, politeness and other ethical considerations had to be kept in mind. The project was constrained by needing to adhere to the robots.txt protocol.

1.1 Existing Crawlers

Three of the main crawlers are listed below, with their parent company's market share currently at 13.28%, 67.78% and 8.14% respectively as of 26th April 2016.[2]

Bingbot

This web crawler was brought to the field by Microsoft in May 2009. Specific implementation details of this crawler are hard to come by due to its proprietary nature. This crawler will obey the robots.txt file under the directives 'bingbot' and 'msnbot.'

Googlebot

Originally coined 'Backrub' when it was released in 1997, this web crawler was originally written in Python. One of the most advanced crawlers in its field, it is increasingly evident that it may be able to execute JavaScript and parse content generated by AJAX calls as well as being able to

[1] Joshua Giardino. *Googlebot is Chrome*. 2011. URL: `http://ipullrank.com/googlebot-is-chrome/`.

[2] *Web Crawler Market Share*. 2016. URL: `https://www.netmarketshare.com/search-engine-market-share.aspx?qprid=4&qpcustomd=0`.

follow HREF and SRC tags.[3]This crawler revolutionised the crawling landscape. One downside of Googlebot is that it appears to take up a lot of bandwidth, so Google provides website owners a method of reducing the crawl rate. Although this crawler is also closed source software, as Google is so prolific in the search field a lot of research has gone into attempting to reverse engineer their crawler. Some research papers even suggest that the crawler is simply a headless Chrome browser.[4] This crawler may be limited by using the directive 'Googlebot' in the robots.txt file.

Yahoo Slurp

Yahoo used to outsource their crawling to Google until 2003, when they announced the launch of their own crawler, 'Yahoo Slurp,' and cut their ties with Google to become their direct competitor. Yahoo's crawler obeyed the directive 'Slurp' in the robots.txt file. As of July 2009, Yahoo has been powered by Bingbot.[5]

[3] *Googlebot.* URL: `https://support.google.com/webmasters/answer/182072?hl=en`.

[4] Giardino, *Googlebot is Chrome*, op. cit.

[5] Danny Sullivan. *The Microsoft-Yahoo Search Deal, In Simple Terms.* 2009. URL: `http://searchengineland.com/microsoft-yahoo-search-deal-simplified-23299`.

2

Specification and Design

2.1 Requirements

The project is built around the following requirements and acceptance criteria. They build upon some of the points brought up in the initial plan.

1. The project should produce a multithreaded web crawler.

– Acceptance criteria: The project must use several threads to crawl the internet concurrently.

2. The crawler should adhere to the robots exclusion standard - the crawler must only crawl sections of a site which are either explicitly defined as allowed or not specifically disallowed in the robots.txt file of that domain. It must also obey the 'crawl-delay' directive; it must wait a specified amount of time before requesting again from the server if the robots.txt specifies it.

– Acceptance Criteria: The user should be able to specify at the command line the default wait time between each crawl request on a domain which does not already specify a time to wait between each crawl.

3. The project should prevent the repeat download of a webpage.

– Acceptance Criteria: The crawler must not download a page if it has already encountered the web page on that crawl.

4. The crawler should not use an excessive volume of bandwidth.

– Acceptance Criteria: The crawler must save as much bandwidth as possible, ensuring that the website owners are not inconvenienced by its bandwidth used while at the same time providing the maximum usage of the bandwidth provided on the host computer, downloading from as many different websites as possible.

5. The crawler must provide the user with a way of limiting the crawl.

– Acceptance Criteria: The crawler must be able to be limited by depth or time on the command line.

Before embarking upon the project I had decided to develop this piece of software using the waterfall development model. This is because it is the development model with which I was most familiar, and the project was relatively short, at a maximum of fifteen weeks including the Easter session. Another reason for choosing the waterfall model is that there would only be one release of the project; there was no need for further support to be provided after its completion. I drafted the requirements before starting any development in order to have a clear idea of what work needed to be completed.

When starting the project, I implemented the command line options. However I wished to hold some familiarity with other GNU/Linux command line tools. To do this, I adhered to the GNU Standards for Command Line Interfaces.[1]

This project is hard coded to specify a delay of 5 seconds if a 'crawl-delay' is not specified in the robots.txt file. This is in an attempt to conform to a politeness policy while also attempting to remain competitive.

In order to save bandwidth and meet acceptance criteria X, HTTP compression was investigated. This may only be accessed if the server supports it, through use of the header 'Accept-Encoding: gzip.'

One method of saving bandwidth which was quickly discarded was the idea of utilising a proxy server to download and cache compressed versions of these pages. This idea is not new, with Google and Opera[2] offering this service for users with slow internet connections, or users wishing to save data on their mobile data plan. This idea was discarded because it would not save the website owners bandwidth, and instead moves the bandwidth to a different service. Another reason for discarding this idea was that in both cases mentioned above, there were clauses in the terms of service specifically prohibiting using them for external applications[3].[4] Because of this, it would have meant extra development time spent coding a new caching and compressing server for minimal benefit, as the resources were not available to run the compressing server off site, to show true bandwidth savings. I also did not wish for it to encroach on time spent developing the actual crawler, so I opted not to develop this avenue of research any further.

Another method of bandwidth saving which was investigated was to use the mobile version of a site if it is available and instead take links from that. Some of the downsides to this is that while the mobile sites do take up less bandwidth, often they contain fewer links which therefore gives less scope for crawling the web. Furthermore, with the advent of responsive design, many websites do not provide a version of their site especially tailored to mobile devices, negating any possible benefit from the extra work.

[1] *Standards for Command Line Interfaces.* URL: `https://www.gnu.org/prep/standards/html_node/Command_002dLine-Interfaces.html`.

[2] *Data savings and turbo mode.* URL: `http://www.opera.com/turbo`.

[3] *Google Terms of Service.* URL: `https://www.google.com/intl/en/policies/terms/`.

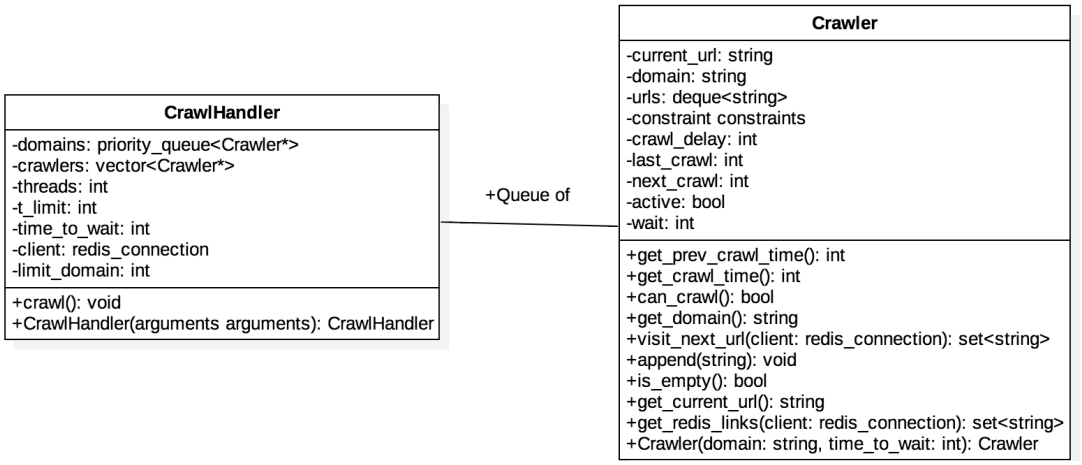
[4] *Opera Software End-User License Agreement for Opera Desktop Browser.* URL: `http://www.operasoftware.com/eula/browser`.

2.2 Risk Analysis

Risk	Likelihood/Impact	How to minimise disruption
Change to program structure	Moderate / Moderate	Document every change and justification so that if somehow an improved method is discovered, the justification will be apparent to future readers.
Hardware failure	Low / High	Back up the project to an external device, and utilise version control to ensure that every small change is backed up in two places.
Unable to meet time constraints	Low / Moderate	Scale back the project to a more manageable scope.
Unable to meet with supervisor as often as planned	Low / Low	As long as I keep up to date with milestones then I don't see this being a massive issue.

2.3 UML

To ease creation of the project, a UML diagram was created before the project was started. The UML diagram however does not illustrate the project very well, as helper functions were used liberally throughout the project to reduce code repetition. Classes were only used where they were deemed necessary, however this would change were I to continue this project - see section 6: Future Work and Reflection.



As the above UML diagram shows, object oriented principles were not applied throughout the project. Actions such as downloading a web page, or breaking the URL down into its component parts were not deemed necessary to use a class for. I do however feel that the constraints found in robots.txt could be represented better.

When investigating which data storage solution to utilise in this project, after finding that Redis would be easier to use for my use case, a method of storage of URLs had to be chosen. The data was chosen to be stored as a hash set. This is because the hash sets may contain user defined fields which could be iterated through as the fields could be an integer. Therefore, for this application the key for the data was the URL being stored in the database, the time the URL was crawled was stored in the field 'time,' and the URLs linked to from the original URL are stored in the field 1 to n, where n is the last URL found on the web page. To reduce memory usage, the expire time could have been set on the key, so that after the defined number of seconds the key would be removed from the data store. This would have reduced the amount of work for the

processor as well, as there would not have needed to be a call to the current object's member function to get the last time the domain was crawled. This expire time was not set in the program as I was unaware of its existence at the time Redis integration was implemented.

As this project is multithreaded, one of the problems faced is that if left unchecked, the crawler could potentially access the same URL in two or more threads at the same time, or access a URL which is on the same domain as another URL which was just accessed. Even in the sequential implementation, the former is a problem which all web crawlers must address.

To shorten development time, only one scheduler option was explored: always scrape the domain which has not been crawled in the longest time. This method seemed to me to be the most efficient algorithm to implement while still being reasonably efficient during crawling.

To prevent the same URL being crawled multiple times, a cycle checking algorithm was used. To prevent a URL from being crawled multiple times, the URL is added to the Redis data store upon the time of its first crawl. The time of this is also stored, as well as any URLs which may be children of the original URL. Every time a URL is crawled, its presence is checked for as a key in the Redis data store. If a match is returned, the first entry of the associated data is returned, which is the UNIX timestamp of when the site was originally crawled. A modified GET request is then performed on the URL, utilising the 'if-modified-since' header. If the domain has been crawled already and a specified time has not passed, it may be ignored.

In the initial plan of the project, ethical concerns were outlined. One of the issues found while considering ethical implications of the project was how the crawler would deal with data while crawling a social media site. To circumvent this issue, it was proposed that the crawler should avoid any URLs containing the character '?.' This proved to be impractical as there are so many valid URLs the crawler would miss that it is not worth including, especially as there are so many ways to pass a query to a website, despite there only being one RFC approved method of passing a query according to RFC 3986.[5] Trying to find links which do not contain any dynamic content was therefore considered beyond the scope of the project. To tackle the other ethical concerns raised, instead of allowing the user to specify the crawl delay, the flag was omitted. This was because this flag could have allowed a user to misuse the software, even with a limit set on how frequently a request could be sent.

[5] et al. Berners-Lee. *Uniform Resource Identifier (URI): Generic Syntax*. 2005. URL: `http://www.rfc-base.org/rfc-3986.html`.

3

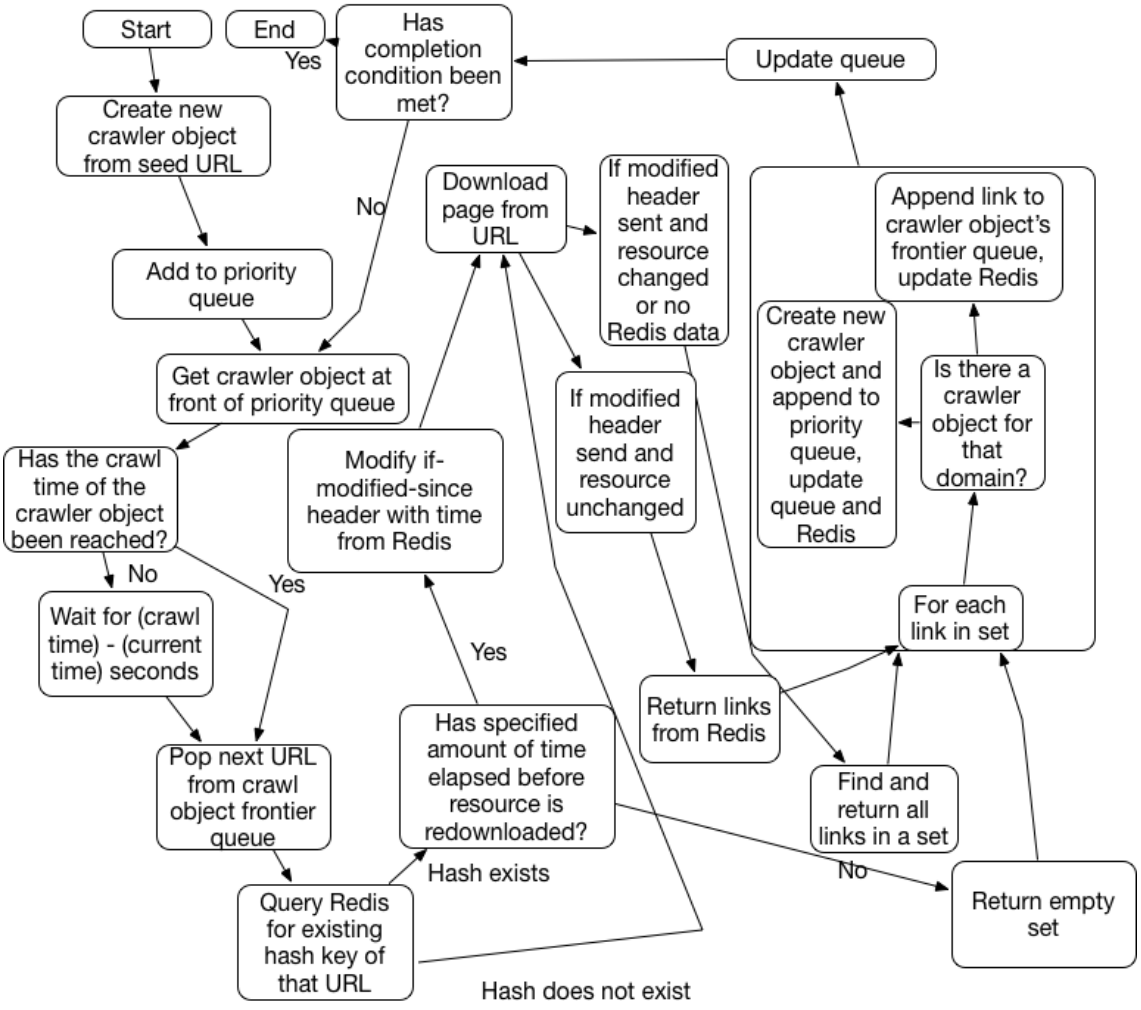
Implementation

This project is programmed to the C++11 standard. This section assumes that the reader is familiar with C/C++ and some common data structures. The flow of the program is detailed in the following diagram.

The project depends on the following libraries in order to compile: libcurl, Qt, argp, redis3m, hiredis, gumbo, and OpenMP. To run, the project requires a running Redis server on the same system as the system attempting to run the project. Without this, the project will throw an exception as soon as it is started.

3.1 Program flow

3.1.1 Flow chart



3.1.2 Pseudocode

```
Require: Domain
  New crawler object with Domain
  Add object to priority queue
  while object in queue does not have an empty URL queue OR the empty counter is not equal
  to the number of objects in the queue do
    if front queue item may be crawled then
      Pop next URL from crawler URL queue
      if URL exists as key in Redis store then
        if specified time elapsed then
          Get page with if-modified-since time from Redis data store
          if Page unchanged since last crawl then
            Add links from Redis associated with the current URL to the list of links
          else
            Add links from page to list of links
          end if
        else
          Leave list of links empty
        end if
      else
        Get page
        Add links from page to list of links
      end if
      for all link in list do
        if crawler object for link domain exists then
          Append link to crawler object's queue
          Update Redis
        else
          Create new crawler object
          Append object to priority queue
          Update Redis
        end if
      end for
    end if
  end while
```

3.2 Implementation specifics

The priority queue is used to ensure that the domain with the longest time since having been crawled is the next domain to be crawled. Any new domain object is initialised with the time since last crawled set to zero, to ensure that it will be the next domain be crawled. Justification: There needs to be a way to ensure that a domain is not constantly accessed, to prevent the crawler's connections from being dropped because of it being labelled as a denial of service attack. The priority queue provides an elegant method to do this.

Redis is utilised to ensure that if a website has been crawled before, then it is not downloaded unnecessarily if the website has not changed. Justification: Redis is faster than a relational database

such as MySQL,[1] as it stores everything in memory. It is also thread safe as it is single threaded; while one command is executing, no other command may be executed.[2] The library Redis3m was used as it was well recommended on various websites visited during Redis client research. The library first used produced some odd errors during compile time. Trying to fix these errors lost a day of development time so the library was changed to prevent the loss of more development time.

The following code is contained within the 'compare.hpp' class.

```
class comparison
{
    public:
        bool operator() ( Crawler* lhs,  Crawler* rhs) const
        {
            return lhs->get_crawl_time() > rhs->get_crawl_time();
        }
};
```

This code takes two crawler object pointers as arguments and returns true if the first argument's crawl time is greater than the second argument's crawl time. This ensures that the priority queue is ordered with the smallest next crawl time as the item nearest the top of the queue.

To compile the project, the GNU Compiler Collection was used as it is readily available on the platforms which I use and is free to use, as long as the source code of the project is made available which I plan to do.

OpenMP was used instead of Pthreads as it is higher level and so simpler to implement. I also have prior experience using OpenMP which simplifies the implementation. One of the disadvantages to using OpenMP is that it lacks the flexibility of Pthreads, however I think the familiarity and the ease of use of OpenMP is a good reason for choosing it over the alternative. As OpenMP is an API, the license of its implementation depends on the compiler used; therefore in this instance it is freely available to use under the GNU Public License.

Gumbo-query was used to parse the HTML retrieved. This library was used as it provides an easy to use abstraction layer to the Gumbo parser library built by Google for use with the C++ language. One downside to using this was the implementation of gumbo-query used appeared to include a file which would not compile, causing the project to be slightly delayed while I dealt with the file causing the errors. The documentation for this project was also sparse; due to this I had to read the source code of the library to find the correct usage of it. One negative of using this library is that it states in the overview of the project that it does not place speed of parsing as the highest priority. The parsing library itself, 'gumbo-parser,' is available freely under the Apache version 2.0 license,[3] while the interface to C++ is available under the MIT license[4]

Libcurl was used to retrieve the HTML pages. This choice was made as it is a mature and thread safe library used to download online resources, which also provides a simple way to modify the headers of HTTP requests. For example, using libcurl massively simplifies the downloading of compressed resources as it handles the decompression of gzipped resources so that it only requires

[1] Ruturaj Vartak. *Redis, Memcached, Tokyo Tyrant and MySQL comparision.* 2009. URL: `http://ruturaj.net/redis-memcached-tokyo-tyrant-and-mysql-comparision/`.

[2] Karl Seguin. *The Little Redis Book.* 2015. URL: `http://openmymind.net/redis.pdf`.

[3] *COPYING [gumbo-parser].* URL: `https://github.com/google/gumbo-parser/blob/master/COPYING`.

[4] *LICENSE [gumbo-query].* URL: `https://github.com/lazytiger/gumbo-query/blob/master/LICENSE`.

one additional line of code. Similarly, the implementation of modifying the 'if-modified-since' header only required an additional 5-10 lines of code. Libcurl is available under a modified version of the MIT license, which may be found on the project's website.[5] This makes it a good choice for use with this software.

The parallelisation was completed using the section directive. This directive is used to create task based parallelism. The theory of this implementation was to have a thread crawl a website, and while that thread added domains to the queue, they would be picked up by other waiting threads which did the same. This works well while the crawler is run with only a constraint on the amount of time it runs for or without a constraint at all, but when the crawler is run with a limitation on the domain the crawler's additional threads continue looping, checking to see if the only crawler object is free to crawl. Because of this, I introduced a half a second sleep every time the threads find a crawler object at the top of the queue which is busy or may not yet be crawled. The code for this logic is listed below:

[5] *Copyright - License.* URL: `https://curl.haxx.se/docs/copyright.html`.

```
#pragma omp parallel num_threads(threads)
        {
            #pragma omp sections firstprivate(c) nowait
            {
                #pragma omp section
                {
                    if (!domains.top()->can_crawl()){
                        nanosleep(&ts, NULL);
                        c = false;
                    }
                    if (c && domains.top()->can_crawl()){
                       //Crawler code

                    }
                    #pragma omp critical
                    {
                        domains.push(a);
                    }
                }
            }
        }
```

The code above has been shortened for brevity. The first 'sections' pragma denotes what follows is structured code which may be run independently, with the 'nowait' directive ensuring that the threads do not wait for the previous section to complete before working on the next section. The 'firstprivate' clause is put in place to ensure that each thread will have its own copy of 'c.' While this could have been accomplished with the 'private' clause, I felt it important that the variable took the value assigned to it before the start of the parallel clause. The following 'section' directive contains the first code block which should be run in parallel; whether or not the top queue item is ready to be crawled. As this section is run by one thread, other threads run alongside it checking the conditions, and crawling the web if the conditions are met. Contained in that block is a 'critical' directive, in an attempt to minimise problems which might arise from the queue being updated by several threads at the same time. The critical directive ensures that the code encapsulated by the critical block will only be run by a single thread at a time - in this case, ensuring only one thread will push things to the queue at a time.

The structure of the program not included in the UML diagram is like so: The robots.cpp file included all functions related to the robots.txt files, including breaking the file into its constituent lines and checking if a given URL is contained in the directives of a given constraint data structure, which is also defined in the file. The utils/util.cpp file contains generic functions used throughout the program for recognising parts of URLs, constructing URLs from absolute and relative paths, and the required files for downloading the specified web resources. It also included the function for modifying the 'if-modified-since' header. The arguments.hpp file includes the data structure for the command line arguments, so that the structure may be passed into the CrawlHandler class without having to include main.cpp in the CrawlHandler class. Finally, the comparison class provides a method of comparing the Crawler classes so they may be correctly ordered in the priority queue.

When the CrawlHandler class is instantiated, the class then instantiates the first instance

of the crawler class. The crawler class is programmed so that when it is instantiated it fetches the robots.txt. Because of this request to its domain, using the data from fetching and parsing robots.txt, its next crawl time is then set to the current time plus the crawl delay, to prevent number of too-frequent requests to the domain.

3.3 Problems Encountered

During the first three weeks of the project, the language used was Haskell. Unfortunately the paradigm was less familiar to me than I had first thought. It was with my supervisor's guidance that I then changed the language of the project to C++. The C++ language provides the object oriented paradigm which I am familiar with. It also provides a wealth of libraries with which to interface. It allows access to the powerful OpenMP library for multithreading, among a wealth of libraries written in C. Other libraries utilised in this project which are written in C include the 'sys/time.h' library and the libcurl library. The prolific nature of the C++ language also means there is a large community from which to draw from should any errors be encountered, unlike Haskell which has a relatively small user base.

When first attempting to crawl the web, a crude implementation of a URL handler was written. The implementation did not cope very well with malformed URLs, often not forming absolute paths from relative paths correctly, causing the crawler to go to an error page instead of the correct destination, corrupting the results of the crawl. To correct this, the C++ library Qt[6] was used, specifically 'Qurl.h.' Qt was chosen as it was the easiest C++ URL library to use, and it provided the most accurate results after testing it against various sites. Another benefit of using Qt is that its URL parser supports unicode, and it is compliant to the RFC 3986 standard. It is also freely available under the GNU Lesser General Public License.[7] However, as Qt is quite a large library to include in a project such as this, the URL parser was rewritten using regular expressions.

While testing the crawler, I found that it treated subdomains as completely separate websites. As subdomains are usually used for aesthetic purposes, I tested to see if the subdomains were associated with the same IP addresses as the URL without any subdomains. To test this, I utilised the Linux command 'nslookup' on google.com and support.google.com, among other subdomains. The resolved IP addresses were similar but not exactly the same. This could be down to load balancing on the server side redirecting the request to different servers, however there will still be traffic on the site's internal network. To prevent using too much bandwidth or being falsely flagged as a denial of service attack on the site's internal network, these subdomains were assigned to the same crawler object as the URL without the subdomain. For example, google.com and support.google.com were assigned to the same crawler object, and therefore share the same crawl delay property. When further research was conducted, the subdomains were found to have their own robots.txt files which would be ignored, as they were only treated with the constraints placed upon them by the robots.txt file of the main domain. The change was then reverted such that the subdomains had their own crawler object so their constraints may be respected.

The crawler objects originally used a queue data structure to store the next URLs to be crawled. However when it came to testing the crawler on various sites, I found multiple pages of a website may link to the same sites; for example the site navigation on some pages will always be

[6] *QUrl Class.* URL: `https://doc.qt.io/qt-5/qurl.html`.
[7] *GNU Lesser General Public License (LGPL).* URL: `https://doc.qt.io/qt-5/lgpl.html`.

present. This would have potentially wasted the crawl delay time as the site would have returned an HTTP error 304 unmodified if the pages were accessed in quick succession. To correct this behaviour I needed to be able to iterate over items in the queue to check for the presence of a link ready to be pushed to the queue. The queue data structure in C++ does not allow you to do this, however the structure from which it is derived, deque, does. Therefore the queue instance was replaced with an instance of deque.

The priority queue is ensured to be updated in the while loop by popping the topmost element from the queue when it becomes busy. The queue is updated by inserting items, and then the current item is added again, ensuring that its position in the queue is consistent with the other elements. Previously I was simply calling the top method from the `priority_queue` class and creating a pointer to the element returned by that method. This caused errors in the program where a domain would continue to be flagged for crawling, but it had no URLs in its queue, simply returning an empty set of links. The crawler would continue with this until it was interrupted. This problem was solved in two ways: If a domain is present in the priority queue which has no URLs in its queue when it is the next domain to be crawled, its crawl time is simply increased by its specified crawl delay and it returns an empty set. As insertion into a priority queue has complexity $O(\log n)$, it is an efficient method of keeping an ordered list of domains.

While testing the crawler, a problem was encountered where a crawl would end abruptly as soon as it began as the given seed URL would be found in the Redis database. To work around this initially, the 'redis-cli flushall' command was run after every test crawl, however after it became apparent that the results of a previous crawl would interfere with the results of the current crawl, the line detailed below was added.

```
client->run(redis3m::command("FLUSHALL"));
```

This ensures that the next crawl will start with an empty dataset so that the previous results will not interfere with the next crawl's results.

In order to comply with the various open source licenses included in the projects utilised, the source code of this web crawler shall be made pubicly available online at the time of publication of this paper.

4

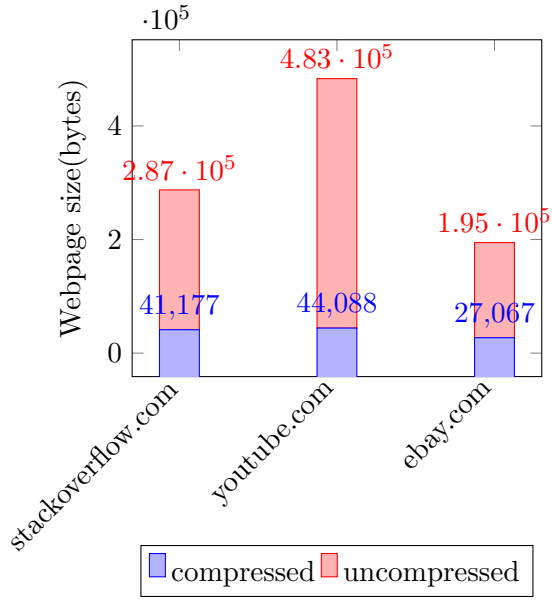
Testing

The web crawler was tested on an up to date distribution of Arch Linux. This computer had 8GB RAM, an Intel Xeon E3-1230, 240GB of solid state storage and a gigabit connection to the local area network, to ensure that the network hardware would not limit the tests. The maximum internet connection speed was 80Mb. The version of libcurl used was 7.48.0. The zlib version was 1.2.8. Gumbo-parser was, at the time of writing and use of it in the project, at commit 597b7df on its Github repository.[1] Gumbo-query was at commit 9a269cd at the point of use. The Redis interface library redis3m was at commit 950209f when it was used. The project and its component community projects were compiled using GCC version 5.3.0. The shell used to run the program was zsh version 5.2. Qt version 5.6 was used. To ensure prior tests did not conflict with the results of the following tests, the Redis database was cleared of all keys before the tests were conducted where applicable. Each of these tests was run 10 times for each property change and averages were taken of the results.

4.1 HTTP compression bandwidth usage

Individual webpages were tested to see the potential savings using HTTP compression from using the 'Accept-Encoding' header. These results were obtained using the curl command. The results of three of these tests are shown below.

[1] *gumbo-parser latest commit.* URL: `https://github.com/google/gumbo-parser/commit/597b7df0e4937afb6c727368fb7f500dce9ee943`.

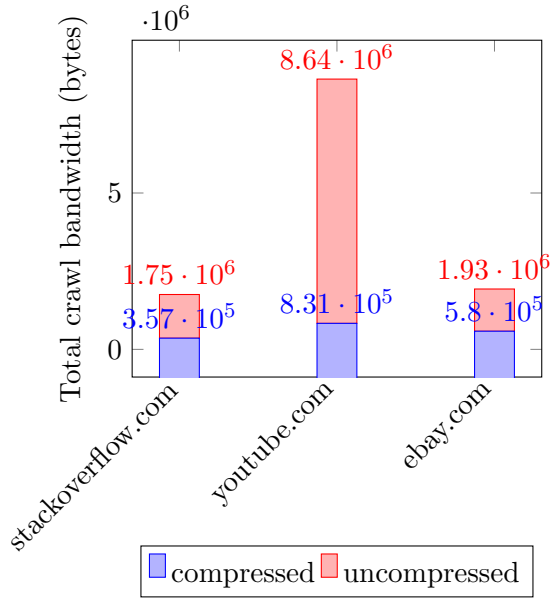


These results show that these three popular websites all utilise the server side option of compressing the content before it is served to the user. The average saving of using compression for these three sites is 246875 bytes. The potential savings when using this method with a web crawl are huge.

This test for the web crawl was conducted using the flag to limit the domain of the crawl to the input domain only, and with the threads flag set to one. To measure the saving during a crawl of the internet, the same seed URL was changed but the crawl was then no longer limited to just one domain. To prevent the crawl from lasting too long a time and preventing any further tests being conducted, the crawls were both limited to one hundred seconds. During these two tests, the only property of the tests changed between the two tests was the lines enabling HTTP compression were commented out; the test also left the Redis connection enabled. The network utilisation was monitored by modifying the program to print the size of every transfer in bites; this was done using libcurl. To get these results easily, the following command was run, with slight variations.

```
for i in {1..10};
do time ./crawler google.com --domain -t 100 -T 1 >>
compression/domain/$i.txt;
done
```

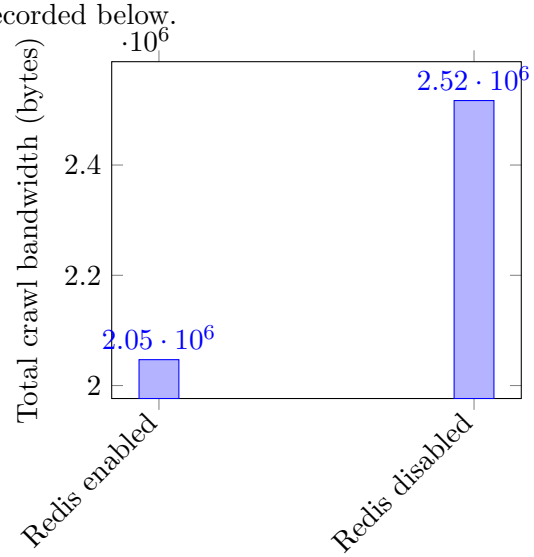
The results of this test are shown below.



The crawler crawled eighteen webpages in the 100 seconds defined at the command line. This graph shows that disabling HTTP compression on a web server which otherwise supports HTTP compression may lead to using nearly four times more bandwidth than leaving HTTP compression on. The results of the total bandwidth used experiment are expected, as the ratio of the uncompressed versus the compressed sizes are roughly the same in the individual webpage and crawling the domain test.

4.2 Tests to find the data saving when using a data store

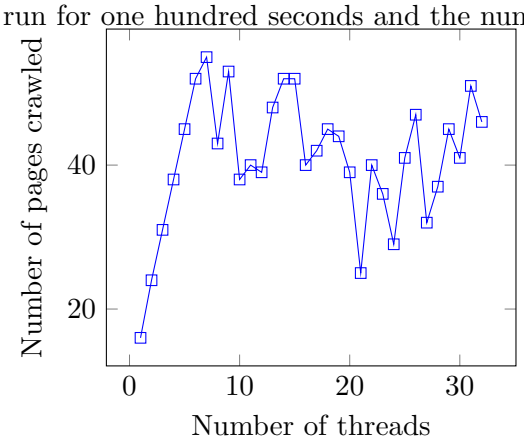
The Redis backend is supposed to save CPU time and network load by passing over items which have been crawled recently. To test the savings, the crawler code was edited to not insert any links into the Redis data store and with the crawler set to only send requests with the 'if-modified-since' header if the item in the database is older than thirty minutes, the default. To save time for other tests, the crawler was run for 5 minutes. This test used 'google.com' as the seed URL. The results are recorded below.



These tests show a considerable data saving of almost 470 kilobytes, and if the test was run for longer I imagine that figure would only increase. The number of repeat web pages avoided because the Redis backend was turned on was only four, however similarly I believe that figure would increase as the overall number of web pages crawled increased.

4.3 Single vs Parallel Implementation

To test the single threaded implementation of the application, the crawler was compiled without the '-fopenmp' flag. This caused the crawler to be compiled with the OpenMP pragmas ignored. The crawler was then compiled with the '-fopenmp' flag, and the number of threads it was run with was incremented from 1 to 32. The Redis backend was enabled for this experiment. The crawls were run for one hundred seconds and the number of links visited was recorded.



4.4 Conclusion

These tests prove that the implementations used for reducing the bandwidth use of a crawler and preventing the crawler from crawling in the same web pages in a loop are very successful, however they also show that the method in which I have implemented the multithreading for the web crawler becomes ineffectual after eight to nine threads are created. I believe that the effect of the Redis data store would be more profound had the tests been run for more than five minutes, however making ten tests of an hour is quite time consuming. Regrettably I did not get a chance to hire servers to run the crawler from, so the power consumption of leaving the machine overnight did factor in to my testing decisions.

5

Limitations

In the initial work plan, it was specified that the web crawler would fetch pages in a breadth first manner. Because of conforming to crawling politeness principles, this was found to not be a practical method of crawling the web, and it was thought that fetching web pages based on their domain of origin would have a better outcome.

The crawler does not currently meet the requirement of being able to be limited by depth. As the project no longer relies on recursion, placing this restraint on the crawler became more difficult because of the structure of the program. As the difficulty for this command line flag increased and it wasn't a main part of the program, this was moved down the list of priorities so focus could be placed on getting the web crawler functioning. It is, however, an option I would like to add to the crawler in future.

Another limitation of this implementation is that a few of the websites the crawler has come across do not use new line characters to delimit lines in their robots.txt file. This causes the crawler to not pick up the file correctly, so it does not obey the rules correctly. One possible solution to this is to find the number of lines in the file and use a space character as a delimiter under certain circumstances.

6

Future Work and Reflection

During the progression of this project I realised that I would not be able to make it able to parse links hidden inside Javascript elements, or parse JSON responses from links. In the future I would consider adding support for this to the project. I would also consider adding MPI support to the project in order for it to compete with the volume of pages able to be parsed by the competing crawlers. In order for the MPI support to function correctly, I would then have to add support for choosing the Redis server the user wished to connect to instead of hard coding the connection to the local server.

The implementation of the web crawler depends on the queue of crawler object pointers and a vector of pointers to the same object. This is due to not being able to iterate over items in the queue. If I were to continue work on the web crawler I would either change the data structure used to store the crawler objects so that two lists of pointers do not have to be maintained, or I would implement a version of the priority queue where not only would you be able to access the top item in the list, you would also be able to iterate over the other elements in the queue. The use of a deque to ensure there are no items in the URL queue is a substitute for a queue, as the queue class does not allow iteration of elements. If I were to continue work on the web crawler, I would attempt to create a class which utilised a set as a queue. The runtime of this would be better than the runtime of the implementation of checking for duplicates in a deque - the insertion of an element into a set has the runtime complexity of $O(logn)$, compared to the worst case runtime complexity of the loop to check for duplicates of $O(n)$.

The crawler's member functions and helper functions mostly rely on copy operations when arguments are passed to them. Copying a data structure can be a costly operation. Because of this extra overhead, in an effort to limit the resources used by the crawler I would refactor the program to only take function arguments as references for efficiency.

The parallelism in this project is only based around the idea that each thread has its own crawler object assigned to it through which it crawls a domain. When crawling a single domain, however, the project becomes limited to a single thread. In the future I would look at making the multiple loops for insertion and checking of URLs concurrent. As mentioned above, subdomains are handled independent of a parent site. If I were to produce further work on this crawler, I would add the robots.txt allows and disallows of the subdomain to the parent site's constraints data structure, and any subdomain would be added to the parent domain crawler's URL queue. This would reduce stress on the domain's internal network.

A feature which I would be very interested in adding to the crawler is some kind of real time

graph output, whether it be in a format compatible with graphing software like Gephi,[1] or in an independent window. This has been researched, however I do not believe there are any up to date plugins which allow Gephi to visualise a graph in real time. Some research was made into this area, and the Gephi library was downloaded, however when an attempt was made at compiling the library, it failed. This may be due to the library apparently not having been updated for seven years.

This project has made me aware of a method of development which I was unfamiliar with before - test driven development. If I were to begin another project this is something I would be very interested in learning more about, as it looks like it would allow me to pick up on issues with the project as they were developed if a thorough set of tests was maintained. If I were to continue with this project, I would look at migrating the development model to a test driven model.

When starting this project, it is clear to me that I grossly overestimated my knowledge of functional programming. Before making such a design choice again I would ensure that I knew how I would tackle the problem using the chosen paradigm. This setback cost me three weeks in development time, which made me miss most of the milestones set out in the work plan. Upon the completion of this project, I believe that for a task such as web scraping, a language such as C++ which allows such low level access is not the correct choice as the low level access is wasted, as so much time is spent waiting for the next url in line to be parsed or downloaded. I believe that this project would be better completed in a more abstracted language such as Python, where development time is low and community support is high. I have attempted to utilise C++'s object oriented features by writing class files for the crawler objects and the crawl scheduler object. However were I to continue this project I would create a class file to handle the URLs, so that I could simply call class methods instead of using regular expressions to trim down the URL to its needed components at that time. I would also make the constraints data structure a class instead. I believe that if I was more experienced with programming in C++, I would have been able to make more informed decisions with regard to my library choices, for example the HTML parser utilised is not built with speed in mind. If I were to redo this project, I would program a new HTML parser explicitly built to extract links quickly.

While programming the crawler, the version control software, git,[2] was used. This was an attempt at being able to control the damage of writing code which would not compile, or would not run. Having not used git before, however, I was not in the habit of committing my work to the git repository after every single change. It was used more of a way to synchronise work between machines. This led to several occasions where the project would not compile so I was unable to show my supervisor the work completed that week. In the latter stages of the project I started to utilise it fully however. I feel that in a future project I would be able to more fully utilise version control software to revert any disastrous changes before they create any larger problems.

During the implementation, I feel that I focussed too much on elements of the crawler which did not matter that much. This time imbalance caused a great deal of stress when developing other elements of the crawler for which I did not leave enough time; however I believe that if I were to undertake a project such as this again I would be able to allocate my time more effectively. If possible, I would get the project working to a bare minimum specification before picking on the details of the project, instead of getting caught up in the details before the project is complete.

Another area which I feel could be improved is the testing of the crawler. The testing

[1] *The Open Graph Viz Platform.* URL: `https://gephi.org/`.

[2] *Git.* URL: `https://git-scm.com/`.

performed was only done for short periods of time. To get more meaningful data, I would set aside more time for testing the web crawler, as I think longer crawls would produce far more interesting result than the tests done over periods of seconds. I would also have liked to have run the tests with about an hour between each, so that I could be sure that the web servers which were being crawled were not throttling the crawler's connection to them.

While completing this project, I found I did not conform to the plan put forward in the initial plan; as I encountered challenges which I had not foreseen, and with the initial three week setback of attempting to use Haskell to complete the project. I think some of the time frames I set myself in the initial plan were overly ambitious. When I wrote the initial time plan, it is clear to me that I was unaware of how long certain aspects of the implementation would take, however because of these initial mistakes in planning I feel I am much more realistic now in terms of how much time implementing some things will take.

The skill which has allowed me to accomplish most during this project is my self motivation to learn. Without this, the project would not have progressed nearly as well as it has done. It allowed me to push myself, and research some problem areas thoroughly. However upon completion of the project I think that it has developed my skills in such a way that I would be well placed to complete a similar project.

Bibliography

Berners-Lee, et al. *Uniform Resource Identifier (URI): Generic Syntax*. 2005. URL: http://www.rfc-base.org/rfc-3986.html.

COPYING [gumbo-parser]. URL: https://github.com/google/gumbo-parser/blob/master/COPYING.

Copyright - License. URL: https://curl.haxx.se/docs/copyright.html.

Data savings and turbo mode. URL: http://www.opera.com/turbo.

Giardino, Joshua. *Googlebot is Chrome*. 2011. URL: http://ipullrank.com/googlebot-is-chrome/.

Git. URL: https://git-scm.com/.

GNU Lesser General Public License (LGPL). URL: https://doc.qt.io/qt-5/lgpl.html.

Google Terms of Service. URL: https://www.google.com/intl/en/policies/terms/.

Googlebot. URL: https://support.google.com/webmasters/answer/182072?hl=en.

gumbo-parser latest commit. URL: https://github.com/google/gumbo-parser/commit/597b7df0e4937afb6c727368fb7f500dce9ee943.

LICENSE [gumbo-query]. URL: https://github.com/lazytiger/gumbo-query/blob/master/LICENSE.

Opera Software End-User License Agreement for Opera Desktop Browser. URL: http://www.operasoftware.com/eula/browser.

QUrl Class. URL: https://doc.qt.io/qt-5/qurl.html.

Seguin, Karl. *The Little Redis Book*. 2015. URL: http://openmymind.net/redis.pdf.

Standards for Command Line Interfaces. URL: https://www.gnu.org/prep/standards/html_node/Command_002dLine-Interfaces.html.

Sullivan, Danny. *Google Still Doing At Least 1 Trillion Searches Per Year*. 2015. URL: http://searchengineland.com/google-1-trillion-searches-per-year-212940.

— *The Microsoft-Yahoo Search Deal, In Simple Terms*. 2009. URL: http://searchengineland.com/microsoft-yahoo-search-deal-simplified-23299.

The Open Graph Viz Platform. URL: https://gephi.org/.

Vartak, Ruturaj. *Redis, Memcached, Tokyo Tyrant and MySQL comparision*. 2009. URL: http://ruturaj.net/redis-memcached-tokyo-tyrant-and-mysql-comparision/.

Web Crawler Market Share. 2016. URL: https://www.netmarketshare.com/search-engine-market-share.aspx?qprid=4&qpcustomd=0.