Crawler4j: A Comprehensive Analysis

12 December 2019

NAME	STUDENT ID
JULIA LEWANDOWSKI	
TANISH DAGAR	
RASHEED KHAN	
DAJEL BOURQUE	

Overview

The open-source software selected by our team was: Crawler4j. Put simply, Crawler4j is a form of web crawler, specifically designed for use with Java, which scrapes the web to collect the desired content given by a user. Crawler4j is composed primarily of 6 main components: the Crawler, the Fetcher, the Frontier, the Parser, Robotstxt, and URL. Here is a brief explanation of each component and how they work together to form the Crawler:

1) **Crawler**: Contains the main .java files related to the Crawler itself as well as two subfolders; they are Authentication and Exceptions.
    a) **Authentication**: Contains *AuthInfo.java* then 3 other classes which extend this file. In general, these files store a username and password of a user to allow the Crawler access to password protected sites which you have authorization to.
    b) **Exceptions**: Contains *ContentFetchException.java*, *PageBiggerThanMaxSizeException.java*, *ParseException.java*. All of these classes are exception checks which deal with: a problem fetching content, a page fetched which is larger than the specified max size, and an error when parsing the page, respectively.

    The main .java files contained in Crawler are *CrawlConfig.java*, *CrawlController.java*, *Page.java*, *WebCrawler.java*. There is also *Configureable.java* but this is only an abstract class to be extended by components of the Crawler and it is also tagged with @deprecated stating that it may be removed without notice. In general, *CrawlConfig* sets the rules and conditions which the Crawler will follow (e.g. Setting the destination folder where scraped data will be stored, Setting max pages to be fetched, etc.). *CrawlController* acts as the manager during the execution of the Crawl, creating and monitoring each thread and using a Logger to log events throughout the session. *Page* is rather straightforward, it contains the data of a fetched and parsed page. Finally, *WebCrawler* which instantiates objects of the aforementioned classes, as well as classes from components yet to be mentioned, and is the runnable class that each thread of the crawler executes.

2) **Fetcher**: Contains .java files related to the actual fetching process during the Crawl. There is *BasicAuthHttpRequestInterceptor.java* which implements *HttpRequestInterceptor.java* and does as stated: intercepts a given http request to allow for verification processing. Second, is the *IdleConnectionMonitorThread.java* which shuts down thread if connection is idle for a set amount of time. Next, there is *PageFetcher.java* and *PageFetcherResult.java*: the first fetches the page and can utilize the Authentication classes to use given credentials on a site, if necessary, then the second checks and stores the result; i.e. it stores current URL and URL which thread moved to next, checks status code and whether or not the process should be halted if an error has occurred. The last two .java files *SniPoolingHttpClientConnectionManager* and *SniSSLConnectionSocketFactory* are exception catchers which attempt to reconfigure or remove a hostname when it causes errors in connecting to a server due to improper configuration with Server Indication Name.

3) **Frontier**: In general, the Frontier acts as a scheduler or priority queue for the Crawler. It tracks and stores the URLs of pages which have been crawled, pages currently being processed, and pages to crawl. And it can set certain pages to be visited as higher in priority. *Elaborated in* ***Data Structures section***.

4) **Parser**: Contains several .java files which parse different parts of data from a given page (e.g. HTML, CSS, etc.) as well as *NotAllowedContentException.java* which ensures that any data specified as "not allowed", by the user, will not be parsed.

5) **Robotstxt**: This folder contains .java files which define the crawler's behaviours toward robots.txt files (Robots.txt is a text file specifying which user-agents are affected by its contents followed by a Disallow condition which can block the affected robots from visiting any of the site contents or indicate which file types they are barred access to). In specific, Robotstxt includes *HostDirectives.java* which checks if the current path is allowed by the host and stores the page directives, *PathRule.java* which matches patterns defined in a robots.txt file to a path, and *UserAgentDirectives.java* which stores the configuration for a single user agent defined in the robots.txt file.

6) **URL**: Contains .java files which store and process elements of the url. In specific, *TLDList.java* contains a list of Top-Level Domains (.ca, .org, etc.) to determine if domain name is public/private. *URLCanoncicalizer.java* which determines the best URL to follow when presented with several URLs (e.g. choosing the homepage URL rather than a URL which links to a subsection of website). And *URLResolver.java* which resolves a relative URL given a base URL.

Lastly, before moving into data structures of this software, we must address some minor caveats with Crawler4j. In order to implement this Crawler you must write a couple new classes; some of which will extend or implement objects from pre-existing classes in Crawler4j.

First, you will need to write your own Crawler class (e.g. "NewCrawler.java") which extends the WebCrawler class, sets a final static instance variable which can either detail files you wish to visit or wish not to visit, and overrides some of the WebCrawler methods. Specifically, the *shouldVisit* and *visit* methods of the WebCrawler class need to be overridden:

1) **@ override**
   boolean shouldVisit(Page refPage, WebURL url) - This method determines whether or not the given URL should be crawled by utilizing the static instance variable specifying file types and returning a boolean that denotes whether or not these file types exist within the given page.

2) **@ override**
   void visit(Page page) - This method is called after it has been determined that the page should be visited (meaning **shouldVisit** returned True) and allows for easy processing of the downloaded page through parsing of text, links, html, etc. It is in this method where you would specify which parts of the page you wish to scrape.

With the static variable initialized and these two methods overridden, your Crawler class is complete.

Second, you will need to write a Controller class which specifies seeds (a list of URLs to be visited), a destination folder to store data obtained during Crawl, and an integer signifying the number of simultaneously active threads. This class will instantiate an object of type CrawlController which is the component of Crawler4j that actively manages a crawling session by creating and monitoring all threads during the execution. The seeds specified in the Controller class will be added to the CrawlController object so that the crawler has a starting point (first page[s]) to fetch. From the provided page, it can continue to follow relevant links (determined by **shouldVisit**) that exist within this page.

This concludes the overview of Crawler4j, next onto Data Structures.

Data Structures

From the overview of Crawler4j, it appears to be a highly interesting and applicable software, but how does it actually work? How does it store data? How does it give priority to one type of data over another? Although it is recognized that even a string is a data structure, the key data structures which were central to the program are discussed.

1. The ***page class*** in the crawler folder is used to store data of a fetched or a parsed page.
    a. It makes an array giving information about what is the content of the stored data (for example, if it is an image or .PNG file).

2. In the frontier folder of the crawler there is a class called ***work queues*** which generally gives a key to a URL. The key chooses which URL to crawl first; the lower the value of the key, the higher the priority of the URL. This makes this class a priority queue class. O(n)

3. In the ***CrawlConfig*** class of the Crawler folder, hash sets are used for various purposes. For example, one use involves returning a copy of the default header collection and creating copies of the provided headers. It stores an unordered collection containing unique items due to which every index in the hash set has a unique value. The algorithm complexity of the hash set would be O(1).

4. An ArrayList is used in the ***CrawlController*** class of the Crawler folder, which essentially collects the local data of the crawler threads and stores them in the list. O(n)

    protected List<Object> crawlersLocalData = new ArrayList<>();

5. In some cases in the ***CrawlController*** class, generics are being used too as some methods are extending the WebCrawler class. Certain methods return a generic type value.

6. ***HostDirectives.java*** implements a TreeSet of UserAgentDirective objects in its void method setUserAgent. Each UserAgentDirective object has variables pertaining to the path rules, crawl delay, sitemap, and other factors specific to a given user agent. The TreeSet data structure allows for a set of type Tree to exist so that for each UserAgentDirective tree node, the relationship between them is preserved.

7. ***UserAgentDirectives.java*** utilizes Sets of userAgent names (String), HashSets of PathRules, Lists of sitemaps (String), etc.

8. In the Crawler folder, ***Page.java*** has a method called fetchResponseHeader which returns headers which were present in the response of the fetch request. The headers are stored in the form of array and the return value is also extracted from the array.

This is how all the data structures work together and result in the execution of the web crawler. While they all produced a functional software, we found that in general they also allowed crawler4j to run efficiently due to minimization of time complexities when possible as we will discuss. We looked at the data structures in this section, in the next section we are going to talk about algorithms.

Algorithms

The algorithmic complexity, in terms of time complexity, of this Crawler should be O(nx) given the aforementioned Data Structures and linear probing of n pages with x elements on a given page used in this program. More than two-thirds of the entire program mostly consists of various mutator/accessor methods, various sections of overridden and overloaded code, and sections of customizable code. There are sections of important code, but out of all, only certain, less-complex, important algorithms have been highlighted in this section to maintain continuity and simplicity some in our analysis.

CrawlConfig.java:

- **Lines 243-258: validate method.** This method validates the configuration specified by the instance it is paired with.

CrawlController.java:

- **Lines: start method.** Starts crawling session and waits for it to finish.
- **Lines: waitUntilFinish method.** Waits until a crawling session finishes.

- **Lines: sleep method.** This method allows controller to collect local data of crawler threads and stores them.
- **Lines 649-654: shutdown method.** This method sets the current crawling session set to a 'shutdown'. The Crawler threads (from the Thread objects) will monitor the shutdown flag; if it is set to true, it will call the *shutDown* method from *pageFetcher* and the *finish* method from *frontier*.

WebCrawler.java:

- **Lines 313-353: run method**. This method provides the algorithm for an actual run of the crawler. It initializes a boolean *halt* as False and opens a while loop which continues until *halt* is True. In this loop, the ArrayList of provided URLs *assignedURLs* is initialized with a size equal to the batch read size of urls while a boolean value *isWaitingForNewURLs* is True, and proceeds to call the getNextURLS function from the priority queue in *Frontier.java*. Each visited URL is then passed to the processPage function.

- **Lines 413-585: processPage method.** This algorithm is what allows a fetched page to be parsed. It runs through several condition checks ensuring that: the fetched page is valid, the customized method shouldVisit (see **overview section**) returns boolean True, and then performs several more exception checks, including permissions granted from Robotstxt. Finally when this page is deemed appropriate for the given user specifications, it is parsed by passing *page* and *currentURL* to a Parser object's parse method.

Parser.java:

- **Lines 64-134: parse method.** Performs various conditional checks to determine which parsing objects should be instantiated for the page (e.g. HtmlParser, CssParseData, etc.) and passes the page through these parsers to collect relevant data.

Page.java:

- **Lines 163-190: load method.** This algorithm loads the content of this page from a fetched HttpEntity and throws an IOException when the load fails. It will read in the HttpEntity and the max limit of the number of bytes it is allowed to read in. It sets the attributes *contentType* and *contentEncoding* of the Page object to null and then reads in the entity's *contentType* and *contentEncoding* but only if it is NOT null. On the off-chance that the charset of the *contentType* is not set, there will be a warning stating parsing failure and it will resort to default (uses "try" and "catch").

HostDirectives.java:

- **Lines 62-64: allows method.** Determines if host directives allow path to be visited
- **Lines 88-90: disallows method.** Checks if host directives directly disallow visiting path.

- **Lines 98-127: checkAccess method.** Checks if any rules say anything about the specified path.

As previously stated, by no means is this even half the list of algorithms that are available; this list was compiled with the intention of analysing some core java files only and does not take a look at the various XML documents, test and examples that are meant to be paired with the main code. Upon analysis, the code has been deemed efficient for its purpose. With our overview, data structures and analysis in the picture, we move to our conclusions containing some critiques and suggestions.

Conclusions: Critiques & Suggestions

In general, the software for the 4jscrawler is well configured and its classes are organized in a logical hierarchy. Using the SOLID principles established by software engineer Robert C. Martin, the crawler4j software can be evaluated against each principle individually. This allows for the evaluation of its flexibility, maintainability, and potential to maintain minimal time complexity even as users build on the open software for future applications.

1) **Single Responsibility Principle:** A class should have just one reason to change.

Different aspects of the web crawler's functionality are handled by individual classes working together (e.g WebCrawler.java informs how page data should be physically extracted and how to act when the program encounters certain exceptions, Config.java informs specifically which pages and file types the crawler should extract as well as sets parameters for how many server requests are sent at a time, different classes defined for each required authentication page that the crawler encounters, etc.).

Complexity of the actual program structure is reduced thanks to having clearly defined classes with specific functions: different behaviours are organized into different classes. This is beneficial because it reduces unnecessary complexity in the software and therefore makes it easier to contain. For example, if a user wanted to modify how the crawler interacted with website form authentication requests, they could modify the FormAuthInfo.java subclass designed specifically for that scenario without changing anything in the subclasses that handle other types of authentication the crawler might encounter, maintaining simplicity and keeping the code organized with minimal risk of the user unnecessarily changing things they may not have intended to change in the first place.

2) **Open/Closed Principle:** Classes should be open for extension but closed for modification.

All instance variables in the classes are either private or protected where the author has deemed necessary. The appropriate Accessor and Mutator methods exist in each Class to allow other Classes access to the Instance Variables they require from an Instance of a given Object in the program.

3) **Liskov Substitution Principle:** Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.

In general, following this principle is beneficial for open software packets, as their authors are handing over their code frameworks to the general public who will write code for them that the original author cannot directly access and change. Therefore, they should make sure that if subclasses are overridden, their behaviour will still remain consistent with the constraints of the superclass. This principle is followed with crawler4j, especially as the author specifically instructs the users to override functions such as shouldVisit and visit, as well as create a crawler that extends the WebCrawler.

4) **Interface Segregation Principle:** Users should not be forced to depend on methods they do not use.

Certain subclasses extend an interface but must include thrown exceptions to account for potential exceptions that arise from the superclass. For example, the WebCrawlerFactory is a generic public interface defined in CrawlController.java which extends the WebCrawler class. The SingleInstanceFactory and the DefaultWebCrawlerFactory both contain overrides where either an exception is caught or an exception is tried and then caught, where in both cases the issue stems from the superclass. It is possible that the software could have benefitted from the author using this principle more and avoiding the necessity for this error-checking in the first place.

5) **Dependency Inversion Principle:** High-level classes shouldn't depend on low-level classes. Both should depend on abstractions. Abstractions shouldn't depend on details. Details should depend on abstractions.

In general, many of crawler4j's higher level classes that guide its direct interactions with web page content utilize successful abstractions. For example, the PageFetcher class utilizes the CrawlConfig interface generated, allowing for the addition of lower classes without having to worry about changes to the PageFetcher class affecting them (as they will similarly depend on the abstract class). This can also be observed with the WebCrawler class, which becomes an abstraction for the user's implemented Crawler.

Some key positive areas of the software:

- **Speed and efficiency.** Crawler4j is a very computationally powerful crawler, able to crawl 200 Wikipedia pages per second at max efficiency.
- **Language neutral.** The crawler is not limited to English alone, and can extract data from other languages as specified as it is only dependent on ASCII.
- **Offers scalability options.** The depth of crawling is configurable, and the user can play around with various conditionals to further increase or decrease the scale of the crawl (for instance, by deciding whether or not to include binary content).
- **Error checking**. Error checking is often done within the program without having to implicate the user via try-catch code blocks (ex: CrawlController.java lines 415-425). This allows the program to run more smoothly with minimal disruptions.

- **Flexible and configurable.** While called a web crawler, crawler4j is technically a web scraper and allows its users to configure exactly what kinds of file types they want to retrieve. The depth of crawling can be customized as mentioned, as can other factors such as the politeness delay of the crawler, the max connections allowed per host and more under the CrawlConfig.java class. Similar customizability is observed in classes dealing with how the crawler interacts with the robots.txt of the pages it encounters.

Some specific areas the software could be improved:

- **Certain larger classes are severely lacking in commenting**. While in general the author provides helpful header comments for most pivotal methods, and occasional line by line commentary when appropriate, certain classes have a severe lack of commenting that would offer much clarity. For example, CrawlController.java is partially commented but still lacking in explanations for some key aspects such as the WebCrawlerFactory interface and its ensuing classes that implement it).
- **Certain exceptions that are caught or tried may not be needed.** It is plausible that the author could have restructured certain classes to be less dependent on their superclasses, resulting in less exceptions.
- **Doesn't exhibit characteristics of intelligent recrawling.** While crawler4j exhibits many positive characteristics for both short term crawls and long term crawls, there is no visible algorithm that analyzes the frequency at which pages that are being crawled get updated. This would be relevant if the crawler was used for the long term goal of indexing a site, making it beneficial to be able to prioritize pages that get updates more frequently when scraping rather than having to brute force scrape every single page during each update. Being able to detect that a particular targeted page is regularly updated on average every three days, for instance, would allow the crawler to regularly update this page alone every three days, rather than having to update every single page and resulting in pages without any new edits being unnecessarily scraped.

All Java File Observations

crawler4j\src\main\java\edu\uci\ics\crawler4j\crawler:

Authentication Folder:

AuthInfo.java:

- Basic object
- Abstract class
- Essentially gathers login info from a user to authenticate access
- Gets proper url to direct user to authorized domain
- Is extended by other .java files in Authentication folder which are specific authentication form types:
    - BasicAuthInfo.java (basic authentication where the server requests that the computer send authentication info directly)
    - FormAuthInfo.java (form authentication where the server requests that the user inputs some authentication information)
    - NtAuthInfo.java

BasicAuthInfo.java:

- Extends AuthInfo so it performs similar operations
- Only for Basic authentication - doesn't really specify what "Basic" means, but presumably for any non-admin user.

FormAuthInfo.java:

- Extends AuthInfo so performs similar operations
- States in comments that it is "most common authentication"
- Simply authenticates a username and password submitted through an HTML form

NtAuthInfo.java:

- Extends AuthInfo so performs similar operations
- Authenticates the info for Microsoft Active Directory, a database based system which provides authentication, directory, policy, and other services using LDAP (lightweight directory access protocol) where data is stored in a tree-like structure with leaf nodes containing the actual data. It acts as a central manager for all Windows systems within a domain, providing authorization service for all Users depending on how much access they have within the system.
    - Metaphor: imagine a phonebook which stores people's phone numbers (i.e User Objects), but also information like who they work for, their role, etc.. (i.e Group Objects) as well as more technical and specific information such as details of the printers at their company building (i.e other Objects within the domain).
- Throws MalformedURLException: checks for malformed URL, where a malformed URL is classified as any URL without a legal protocol in the specification String associated with it, and/or

Exceptions Folder:

ContentFetchException.java:

- Extends Exception class

- Throws Exception when there is an issue fetching content

PageBiggerThanMaxSizeException.java:

- Extends Exception class
- Throws Exception when page content exceeds maximum size

ParseException.java:

- Extends Exception class
- Throws Exception when there is an error parsing content on page

Other .java Files:

Configureable.java:

- Several core components of crawler4j extend this class to make them configurable.
- This class is a simple class with one constructor and a protected instance variable of CrawlConfig type and a getter method.
- See CrawlConfig.java for details on this Object

CrawlConfig.java:

- This is a really big class which has a lot of instance variables, getter and setter methods with some other methods like toString.
- INSTANCE VARIABLES:
    - **String crawlStorageFolder** : It has an instance variable of string type which basically acts like a folder where all the data from the crawl will be stored.
    - **Boolean resumableCrawling** : The value of this variable is set to false first but if becomes true it would be able to resume previous stopped/crashed crawling. However it makes the crawling slightly slower.
    - **Long dbLockTimeout**: Initialized at 500. LOOK UP sleepycat DB.
    - **Int maxDepthofCrawling**: Maximum depth for crawling. The value should be set to -1 for unlimited depth.
    - **Int maxPagestoFetch**: Maximum number of pages to be fetched. For unlimited pages the value is set to -1.
    - **String userAgentString**: Generally used for representing the crawler to other web servers. This string contains the location of the source code of the crawler. Which is originally on github.
    - **Collection<Basic header> defaultHeaders**: it is just a collection of basic headers in a hash set.
    - **Int politenessDelay**: specifies the delay in between requests send to servers, as without a delay the crawler's high efficiency could potentially overload servers

and/or anger their owners resulting in loss of privileges. Initially set as 200 milliseconds between requests.

- **Boolean includeHttpsPages**: Determines whether or not Https pages should also be included in crawling process. Initially set to true.
- **Boolean includeBinaryContentInCrawling**: Determines whether binary content like Images, Audio, etc should be fetched during crawling process. Initially set to false.
- **Boolean processBinaryContentInCrawling**: Determines whether binary content should be processed, using TIKA, during crawling process. Initially set to false.
- **Int maxConnectionsPerHost**: Sets the maximum allowed connections per host. Initialized at 100.
- **Int maxTotalConnections**: Sets the maximum allowed total connections.
- Initialized at 100.
- **Int socketTimeout**: Sets the number of milliseconds before timeout occurs while waiting for packet. Initialized at 20000.
- **Int connectionTimeout**: Sets the number of milliseconds before timeout occurs while waiting for server to reply.
- **Int maxOutgoingLinksToFollow**: Sets the maximum number of allowed outgoing links that can be processed from a page. Initialized at 5000.
- **Int maxDownloadSize**: Maximum size of a page; anything larger than this will not be fetched by the crawler. Initialized at 1048576. **(**DOESNT SPECIFY WHAT THIS NUMBER REPRESENTS - kb, mb, gb?**)**
- **Boolean followRedirects**: Specifies whether or not redirects should be followed by the Crawler. Initialized as True.
- **Boolean onlineTldListUpdate**: Specifies whether TLD list should be updated automatically after each Run of the Crawler. TLD means *Top-Level Domain* which refers to the last segment of a Domain name (**e.g.** *.com, .org, .ca*). Initialized as False. Comment above this Var also mentions that the TLD list can alternatively be loaded from an embedded .zip file that is obtained from a link which is stored as a String in the next Instance Variable.
- **String publicSuffixSourceUrl**: Stores the above mentioned Url which contains an embedded file called “tld-names.zip”. Initialized as "https://publicsuffix.org/list/public_suffix_list.dat".
- **String publicSuffixLocalFile**: Presumably the .zip file can also be stored locally after retrieval, but this is initialized as NULL. Will have to see where/if this Variable is used later in methods.
- **Boolean shutdownOnEmptyQueue**: Determines if Crawler should stop when Queue (presumably Priority Queue containing URLs “to be crawled”) is empty. Initialized as True.

- **Int threadMonitoringDelaySeconds**: Sets the amount of time, in seconds, before active threads should be checked. Initialized at 10.
- **Int threadShutdownDelaySeconds**: Sets the amount of time before shutting down a thread, allowing for verification that this threads task is completed. Initialized at 10. Presumably, works in conjunction with previous Variable, where it would check the current status of the Thread every 10 seconds then another delay is given by this Variable to confirm the inactive status before the actual shutdown process occurs.
- **Int cleanupDelaySeconds**: Initialized at 10. Delay of 10 seconds before allowing Garbage Collection??
- **String proxyHost**: If the Crawler requires a proxy host to run behind, this Variable can specify a proxy host. Initialized as NULL.
- **Int proxyPort**: If a proxy host is used, this Variable specifies the port to be used. Initialized at 80.
- **String proxyUsername**: If username is required for authentication in proxy, this Variable stores the username. Initialized as NULL.
- **String proxyPassword**: Same as previous just for password. Stores it in this variable. Initialized as NULL.
- **List<AuthInfo> authInfos**: List of the potential authentications necessary for the Crawler. Only Declared, nothing initialized. SEE AuthInfo objects.
- **String cookiePolicy**: Initialized as CookieSpecs.STANDARD. This simply sets the specifications for cookies on an HTTP client and they use the Standard specs.
- **Boolean respectNoFollow**: Determines whether “nofollow” flag should be listened to. Initialized as True. NEED MORE INFO ON “nofollow”.
- **Boolean respectNoIndex**: Determines whether “noindex” flag should be listened to. Initialized as True. NEED MORE INFO ON “noindex”.
- **CookieStore cookieStore**: Object which stores and retrieves cookies, aids in providing cookies to Crawler. Declared, nothing initialized.
- **DnsResolver dnsResolver**: Provides the DNS resolver which should be used. Initialized as default SystemDefaultDnsResolver( ).
- **Boolean haltOnError**: Whether or not the process should be stopped when an error is flagged. Initialized as False.
- **Boolean allowSingleLevelDomain**: Determines whether or not Single Level Domains are allowed. Initialized as False. A Single Level Domain or Single Label Domain means using only the service name without prefix or suffix. This is most likely initialized as False as it is bad practice and the general convention is to have .com, .ca, etc after the name.
- **Int batchReadSize**: Sets number of pages to fetched/processed from a database in a single read. Initialized at 50.

-

- METHODS:
  - **void setDnsResolver(final DnsResolver dnsResolver)**: Uses provided DnsResolver object to set/modify current dnsResolver Instance Variable.
  - **DnsResolver getDnsResolver( )**: Returns the current DnsResolver object stored in dnsResolver.
  - **void validate( ) throws Exception**: validates the configurations specified in a given instance. Has several error checking conditions including:
    - If the *crawlStorageFolder* is NULL; returns message stating that "Crawl storage folder is not set in CrawlConfig".
    - If *politenessDelay* is below zero; returns "invalid value".
    - If *maxDepthOfCrawling* is below -1; returns "should be either a positive number or -1 for unlimited."
    - If *maxDepthOfCrawling* has larger value than *Short*.MAX_VALUE; returns "Maximum value for crawl depth is *Short*.MAX_VALUE".
  - **String getCrawlStorageFolder( )**: Returns value in *crawlStorageFolder*.
  - **Void setCrawlStorageFolder(String crawlStorageFolder)**: Sets given String as new value in *crawlStorageFolder*.
  - **Boolean isResumableCrawling( )**: Returns value in *resumableCrawling*.
  - **Void setResumableCrawling( )**: Allows for altering of Boolean value in *resumableCrawling*. If enabled (set to True), then allows for paused or crashed crawl to continue/resume. However, crawling process will be slower.
  - **Void setDbLockTimeout( )**: Sets new value for *dbLockTimeout* (MORE RESEARCH STILL REQUIRED FOR THIS VARIABLE).
  - **Int getDbLockTimeout( )**: Returns value stored in *dbLockTimeOut*.
  - **Int getMaxDepthOfCrawling( )**: Returns value stored in *maxDepthOfCrawling.*
  - **Void setMaxDepthOfCrawling(int maxDepthOfCrawling)**: Sets new value for *maxDepthOfCrawling*. Where depth of 1 = all links on current page. -1 for unlimited depth.
  - **Int getMaxPagesToFetch( )**: Returns value stored in *maxPagesToFetch*.
  - **Void setMaxPagesToFecth(int maxPagesToFetch)**: Sets new value for *maxPagesToFetch*. -1 for unlimited pages.
  - **String getUserAgentString( )**: Returns value stored in *userAgentString*.
  - **Void setUserAgentString(String userAgentString)**: Sets new value for *userAgentString*; this value acts as an identification ID for Crawler to web servers.
  - **Collection<BasicHeader> getDefaultHeaders( )**: Returns HashSet *defaultHeaders* which contains collection of basic headers.

- **Void setDefaultHeaders(Collection<? extends Header> defaultHeaders)**: Creates new HashSet of BasicHeader objects and sets as new value for *defaultHeaders*.
- **Int getPolitenessDelay( )**: Returns value stored in *politenessDelay*.
- **Void setPolitenessDelay(int politenessDelay)**: Sets new value for *politenessDelay*.
- **Boolean isIncludeHttpsPages( )**: Returns value stored in *includeHttpsPages*.
- **Void setInclueHttpsPages(boolean includeHttpsPages)**: Sets new value for *includeHttpsPages*.
- **Boolean isIncludeBinaryContentInCrawling( )**: Returns value stored in *includeBinaryContentInCrawling*.
- **GETTER and SETTER methods continue for every Instance Variable!! (Tired of writing the same thing for different getters and setters)**

CrawlController.java:

- This class creates the crawler threads and monitors their progress throughout the duration of the Crawlers run - manages the crawling session.
- INSTANCE VARIABLES:
  - **Logger logger**: Utilizes the Logger and LoggerFactory Objects. Essentially logs message for the CrawlController.
  - **CrawlConfig config**: Declares an Instance of CrawlConfig Object.
  - **Object customData**: Used to pass custom crawl configurations to different parts of the Crawler.
  - **List<Object> crawlersLocalData**: Initialized as an ArrayList. Once crawling session has terminated, collects local data from threads and stores in this list.
  - **Boolean finished**: Declared not initialized; determines whether crawling session has terminated.
  - **Throwable error**: Declared not initialized; presumably to throw exceptions when errors occur.
  - **Boolean shuttingDown**: Declared not initialized; notifies Crawler threads of imminent shutdown. Crawler threads monitor the status of this Boolean and once set to True, they stop processing new pages.
  - COME BACK TO THESE ONCE OBJECTS ARE BETTER UNDERSTOOD{
    - **PageFetcher pageFetcher**: Declares PageFetcher object.
    - **RobotstxtServer robotstxtServer**: Declares RotstxtServer object.
    - **Frontier frontier**: Declares Frontier object.

- **DocIDServer docIdServer**: Declares DocIDServer object.
- **TLDList tldList**: Declares TLDList object.
- **Object waitingLock**: Initialized as new Object();
- **Environment env**: Declares Environment object.
- **Parser parser**: Declares Parser object

}

- METHODS:
  - CONSTRUCTORS{
    - **CrawlController(CrawlConfig config, PageFetcher pageFetcher, RobotstxtServer robotstxtServer) throws Exception**: Initializes CrawlController object, however with NULL values for Parser object and TLDList object.
    - **CrawlController(CrawlConfig config, PageFetcher pageFetcher, RobotstxtServer robotstxtServer, TLDList tldList) throws Exception**: Initializes CrawlController object, however with NULL value for Parser object.
    - **CrawlController(CrawlConfig config, PageFetcher pageFetcher, Parser parser, RobotstxtServer robotstxtServer, TLDList tldList) throws Exception**: Initializes CrawlController object with no NULL values. Ensures *config* is valid by using the *validate* method before initializing it as the *config* Var in CrawlController. Creates new File object by getting the *config*'s *CrawlStorgaeFolder*.

      }
  -

Page.java:

WebCrawler.java:

crawler4j\src\main\java\edu\uci\ics\crawler4j\fetcher:

BasicAuthHttpRequestInterceptor.java:

- Implements the Interface HttpRequestInterceptor
- Intercepts Basic Authorization URL requests
- Generally for pre-processing requests before they are executed in case request configurations need to be altered

IdleConnectionMonitorThread.java:

- Extends Thread class
- Shuts down thread if connection is idle for too long
- Most likely to free up space and run time

PageFetcher.java:

-

PageFetchResult.java:

SniPoolingHttpClientConnectionManager.java:

SniSSLConnectionSocketFactory.java:

crawler4j\src\main\java\edu\uci\ics\crawler4j\frontier:

Counters.java:

DocIDServer.java:

Frontier.java:

InProcessPagesDB.java:

WebURLTupleBinding.java:

WorkQueues.java:

crawler4j\src\main\java\edu\uci\ics\crawler4j\parser:

AllTagMapper.java:

BinaryParseData.java:

CssParseData.java:

ExtractedUrlAnchorPair.java:

HtmlContentHandler.java:

HtmlParseData.java:

HtmlParser.java:

NotAllowedContentException.java:

ParseData.java:

Parser.java:

TextParseData.java:

TikaHtmlParser.java:

crawler4j\src\main\java\edu\uci\ics\crawler4j\robotstxt:

HostDirectives.java:

PathRule.java:

RobotstxtConfig.java:

RobotstxtParser.java:

RobotstxtServer.java:

UserAgentDirectives.java:

crawler4j\src\main\java\edu\uci\ics\crawler4j\url:

TLDList.java:

URLCanonicalizer.java:

UrlResolver.java:

WebURL.java

crawler4j\src\main\java\edu\uci\ics\crawler4j\util:

IO.java:

Net.java:

Util.java:

Works Cited

https://github.com/yasserg/crawler4j

https://en.ryte.com/wiki/Crawler?fbclid=IwAR20EBBMAlhROMewHvB-5xbsD51dhGZ78o1cU5kTIrUjqbLC_F-5Vt0zwq8

https://www.baeldung.com/solid-principles

https://docs.oracle.com/javase/7/docs/api/java/net/MalformedURLException.html

https://hc.apache.org/httpcomponents-core-ga/httpcore/apidocs/org/apache/http/HttpRequestInterceptor.html

https://www.techopedia.com/definition/1348/top-level-domain-tld

https://smallbusiness.chron.com/single-label-domain-name-73404.html

http://www.slf4j.org/apidocs/org/slf4j/LoggerFactory.html

https://refactoring.guru/design-patterns/catalog

https://www.robotstxt.org/robotstxt.html

https://www.promptcloud.com/blog/important-qualities-of-good-web-crawler/

(basic methods: set/getDnsResolver [219], set/getCrawlStorageFolder, is/setResumableCrawling [243], set/getDbLockTimeout [294], get/setMaxDepthofCrawling [302], get/setMaxPagetoFetch [315], get/setUserAgentString [333], get/setPolitenessDelay [365], is/setIncludeHttpsPages, is/setIncludeBinaryContentInCrawling [391], isProcessBinaryContentInCrawling [404], get/setMaxConnectionsPerHost [411], get/setMaxTotalConnections [426], get/setSocketTimeout [437], get/setConnectionTimeout, get/setMaxOutgoingLinksToFollow [459], get/setMaxDownloadSize [470], is/setFollowRedirect [482], is/setShutdownOnEmptyQueue[493], is/setOnlineTldListUpdate [504], get/setPublicSuffixSourceUrl [517], get/setPublicSuffixLocalFile [540], get/setProxyHost [544], get/setProxyPort, get/setProxyUsername, get/setProxyPassword [581], get/setAuthInfos [598], get/setThreadMonitorDelaySeconds [617], get/setThreadShutdownDelaySeconds [625], get/setCleanupDelaySeconds [633], get/setCookiePolicy [641], get/setCookieStore [654], is/setRespectNoFollow [670], is/setRespectNoIndex [678], is/setHaltOnError [689], is/setAllowSingleLevelDomain [707], get/setBatchReadSize [727],)

CrawlConfig.java*:*

- **toString [line 736]**: Prints a log of info including name of folder which stores all scraped data, max depth of the crawl, max pages to crawl, etc.
- **validate [line 243]**:
- **get/setDefaultHeaders [line 350]**:

CrawlController.java*:*

- **CrawlController [line 89]**:
- **start [line 269]**:
- **waitUntilFinish [line 429]**:
- **sleep [line 470]**:
- **addSeed [line 488]**:
- **addSeenUrl [line 568]**:
- **shutdown [line 649]**:

Page.java:

- toByteArray [line 120]
- load [line 163]

WebCrawler.: [119] init, [150] onStart, [160] onBeforeExit, mostly override and overloaded methods, [413] processPage

Fetcher contains mostly gets, sets, authentications, fetching various link objects(?)

FRONTIER:

Frontier: [85] scheduleAll, [112] schedule,

InProcessPagesDB: [52] removeURL,

WorkQueue: delete

PARSER

HtmlContentHandlerJava: [149] addToOutgoingURLs

CssParseData: [58] extractUrlInCssText, [84] initializePattern, mostly get methods

Parser: [64] Parse

TikaHtmlParser: [128] chooseEncoding

ROBOTSTXT

HostDirectives: 52 needsRefetch, 62 allows, 88 disallows, 98 checkAccess

PathRule: 26 robotsPatternToRegexp, 107 matchesRobotsPattern

URL

UrlResolver: removeLeadingSlashPoints, resolveUrl, parseUrl, toString,

WebURL: toString, hashCode, just get and set stuff

UTIL

IO: deleteFolderContents

Util: tf? But look again eyy