Programming Assignment #7: WebCrawler

Ryan O'Leary
CS314H

Assignment Overview

This assignment asks the programmer to create an implementation of a WebCrawler, essentially a web search engine that retrieves data from web pages and indexes them accordingly. A WebCrawler consists of an indexing structure to store the data from different web pages, a Query Engine that handles the parsing of user-input queries, utilizing the indexing structure (once populated by crawling the web) to find all appropriate web-pages (in this implementation containing the words specified in said queries). A WebCrawler also requires some sort of Server that creates a GUI, allowing for users to search the web based on their queries, but in this assignment the Server was already implemented and required little modification. This assignment also asked the programmer to support numerous different query operations, allowing the user to search for phrases or specify their search by including different operators such as ANDs, ORs, and negations. Thus, it was the job of the programmer to determine how to efficiently store data retrieved from web pages, parse and then represent numerous types of queries, and finally retrieve the desired web pages in a reasonable amount of time.

Goals

My goals when beginning this assignment were to create an extremely efficient implementation of WebCrawler in order to improve my optimization skills and better understand the processes that occur to facilitate major search engines like Google or Bing and the large amounts of data they handle. This assignment also provided the opportunity to learn more about data structures like HashSets and their use for efficiently storing data when indexing. WebCrawler is a very useful assignment because the programmer is exposed to indexing, storing huge amounts of data, and other languages like HTML and CSS, so one of my main goals was just to learn more about each of those topics. I also set out to improve my white-box and integration testing skills with this assignment, writing comprehensive tests to observe the functions of the WebCrawler and the numerous possible edge cases associated with different types of queries. Finally, since this assignment allowed for ample amounts of Karma, I attempted to improve my WebCrawler by relaxing the restrictions on parentheses in queries, applying negations to entire queries instead of only words, optimizing the speed of my search engine, and more.

Solution Design

Overview:

To implement a working version of WebCrawler it was necessary to design three classes, WebIndex, CrawlingMarkupHandler and WebQueryEngine, while modifying two already implemented classes, WebCrawler and WebServer. WebIndex implemented the Index interface, allowing for the loading and saving of an index from a file, and primarily served to store the words and new URLs found when traversing different web pages. CrawlingMarkupHandler handled the actual crawling of the web pages, implementing AbstractMarkupHandler to provide logic for the different methods that the parser would call when looking at an html file. Because the parser handled all the work of traversing the page, the job of CrawlingMarkupHandler was simply to provide logic that would appropriately store data to a web index when encountering tags and text from an html file, i.e. words and any new URLs found. WebCrawler used CrawlingMarkupHandler to traverse the "web" of connected pages, storing all the words and URLs possible to a WebIndex. Then, WebQueryEngine handled user-input queries by

representing said query and allowing for different operations to occur, returning the desired set of web pages from the index generated when crawling. Finally, WebServer provided for a GUI passing user input to the query engine and displaying all desired links found.

WebIndex

- **getWords():**
  - Returns a HashMap containing all words found as keys and HashSets of their corresponding URLs as values
- **getAttributes():**
  - Returns a HashMap containing attributes found as keys and HashSets of their corresponding URLs as values
  - Since the only attribute necessary to store is “href”, this effectively returns all the URLs found on a page
- **getUrlWords():**
  - Returns a HashMap containing all the URLs found as keys with corresponding ArrayLists of the words found on that page as values
  - Used for phrase queries by storing the order of the words found on a page
- **storeWords:**
  - Takes in an array of strings to store, putting them in the HashMap of words with the corresponding URL of the page where they were found
  - Also stores the URL of the current page with all corresponding words found on the page in order
- **storeAttributes():**
  - Checks if an attribute name and value are valid, and if the attribute is an href stores it in the HashMap of attributes after generating the correct URL from the current absolute path and attribute value (relative path)

CrawlingMarkupHandler

- **CrawlingMarkupHandler()**
  - Default constructor, instantiating a new current index, LinkedList of new URLs, HashSet of previously visited URLs, and Stack to keep track of the current html element
- **getIndex()**
  - Returns the current WebIndex object stored as a class variable
- **newURLs()**
  - Parses through the URLs found by the current index in its attribute HashMap, adding any previously unvisited URLs to the list of new URLs and returning that list
  - Clears the list of new URLs
- **handleDocumentStart()**
  - Initializes the StringBuilder class variable documentWords
  - Pushes the first tag (“html”) to the Stack of elements
- **handleDocumentEnd()**
  - Clears the Stack of page elements
  - Adds all the alphanumeric words found (stored in documentWords) to the current index by calling storeWords() with the current page URL
- **handleOpenElement()**
  - Pushes the new element name to the top of the elements Stack
  - Parses through the attributes HashMap and adds the key-value pairs to the current index by calling storeAttributes()
- **handleCloseElement()**

    - Pops the top element from the Stack of elements
- **handleText()**
    - If the current page element is not “script” or “style”, appends all alphanumeric characters found in the character array to the documentWords class variable

Helper Methods

- **setCurrentURL(URL url)**
    - Sets the currentUrl class variable to the specified URL parameter

WebCrawler

- I chose to retain the given implementation of WebCrawler for my solution, simply adding in a call to setCurrentURL on the CrawlingMarkupHandler so that I could know which page I was currently on
- I also added in error checking / exception handling such that any bad URLs found would simply print an informative error message while continuing iteration

WebQueryEngine

- **fromIndex()**
    - Returns a WebQueryEngine object backed by the specified WebIndex
- **query()**
    - Given a string query, finds all desired URLs through the method handleQuery() and returns the list of results

Helper Methods

- **handleQuery()**
    - Converts the String query to appropriate form (converts to postfix, removes unnecessary white space, and creates a String array)
    - Parses through the postfix representation of the query and performs all specified query operations, returning a HashSet of the desired web pages
- **findWords()**
    - Returns the HashSet of URLs associated with the key (String word) from the wordPages HashMap object in the current index
- **findPhraseURLS()**
    - Finds the intersection of all the words contained in the double quotation marks (pages for which all desired words occur on) and then utilizes the urlPages HashMap to check whether those words are contiguous and sequential on the page, returning the HashSet of valid pages
- **findNegation()**
    - Finds all web pages contained in the current indexes wordPages HashMap that do not contain the specified query
- **findIntersection()**
    - Finds all web pages that contain both queries, utilizing findWords() on both HashSets of URLs and then returning the common pages
- **findEither()**
    - Finds all web pages that contain either query, returning the HashSet of valid URLs
- **postFixConverter()**
    - Converts the given String to postfix form, specifically with the operators AND, OR, and negation while accounting for parentheses (parentheses not required as a result of converting to postfix but can be included)

- **formatImplicitAnds()**
    - Inserts implicit ANDs wherever they ought to occur in the given String query
- **getTokenPrecedence()**
    - Returns the token precedence of the given character, with tokens "&" and "|" holding a precedence of 0 while the token "!" holds a precedence of 1

WebServer

- My solution only made minimal changes to the given implementation of WebServer, adding in better exception handling and null cheer request would not cause the page to crash

Page

- For my implementation I did not find it necessary to modify the Page class in any way, the basic implementation provided worked well with the decision choices I had made

Error Checking:

The error checking that I conducted consisted of utilizing try-catch statements when calling openStream() on URLs, when handling Stacks or conducting other operations that might throw an error, and in WebServer such that a bad server request would not cause the whole engine to fail. In these statements, I printed informative System.err messages rather than exiting from the system. Finally, added null checking when appropriate to avoid null pointer exceptions.

Evaluation:

Overall, this project proved significantly challenging given the sheer amount of code we were required to implement, the numerous helper methods I created in order to compartmentalize my code and improve the efficiency of my solution, and the numerous design decisions that left much of the choice in how their solution functioned up to the individual programmer. As a result, this project extensively tested my OOP skills while allowing for improvement in optimization and writing tests for large, integrated programs such as this one.

Scope:

The scope of solution for WebCrawler can be judged by both how effectively the crawler traverses and stores the data from web pages and the different types of queries supported by the query engine. On the first measure, this implementation of WebCrawler was extremely robust, with error checking ensuring that no data found on the web pages would cause error while the crawler efficiently stored the desired data using HashSets and Maps, allowing fast lookup time and relatively low space complexity. The use of an InflaterInputStream and DeflaterOutputStream in WebServer also helped this solution prove very efficient, crawling thousands of web pages with fairly low time costs. However, the scope was limited by the fact that the crawler only traverses html files whereas search engines like Google can traverse any number of types of pages. On the second measure, this implementation of WebQueryEngine proved very comprehensive, allowing for all specified query operations (while also allowing for operations with no parentheses and negations on entire queries) while returning the desired results in relatively low time cost. However, this too was limited in its scope because my WebQueryEngine only supports English, alphanumeric queries and would not work correctly if passed incorrectly formatted queries/invalid characters (escape characters, etc.).

Quality of Solution:

This solution to WebCrawler proved very comprehensive, functioning for all possible edge-cases (within our assumptions for queries) and passing/storing large amounts of data with relatively low time and space complexity. By storing the URLs and words in HashSets / Maps, the crawler was able to achieve low lookup cost which was the primary operation needed to be performed when implementing a search engine. We did take up some memory allocation by storing class variables such as a Stack of elements, but the trade off was that our solution functioned more comprehensively and achieved the correct results in all cases. I also minimized my redundant code by efficiently organizing different functions such as AND/OR operations and the storing of words vs. urls into different helper methods, allowing for increased code readability while allowing errors to be easily identified.

Some improvements that could have been made include adding more error checking to the WebQueryEngine and Crawler such that invalid queries could be handled in a more comprehensive manner. I also could have tried to add support for different languages in the Crawler rather than exclusively storing English words. Finally, I could have reduced space complexity by storing the positions of words found on a page in the same data structure as the words and the URLs, rather than creating an entirely new HashMap just to perform phrase queries.

Interesting Results:

The interesting results I saw during this project included how much faster you could make your Crawler simply by compressing the input/output streams with DeflaterOutputStream and InflaterInputStream. It was also interesting to learn about postfix notation and how much it easier it makes the computation of operations such as the ones we wanted to perform on queries. Creating different, complicated, queries and seeing the results that were garnered was a very interesting part of the assignment and ultimately helped me improve my WebQueryEngine. Finally, since this project necessitated working with such large streams of data, it was intriguing to see how small changes in code such as using one relatively similar data structure over another could drastically change the time cost it took to run the program.

Design Decisions (Red = Karma):

- Represented queries in postfix notation rather than using a parsetree because it allowed me to very easily perform query operations while being easy to construct
- Utilized HashMaps and HashSets in WebIndex because of fast lookup times and the fact that they did not store duplicates (also had useful functions such as removeAll(), retainAll(), that helped with query operations)
- By using postfix I chose to allow my query engine to accept queries not formatted in parentheses – any parentheses added simply gave precedence to the operations stored inside
- Utilized InflaterInputStream and DeflaterOutputStream to compress data being passed in and much more quickly crawl web pages, decreasing time cost
- Allowed negations to be called on entire queries, essentially applying DeMorgan's law to get the correct result

Problems Encountered:

- I had an extremely persistent issue where certain words were being appended to each other rather than leaving a space between them, resulting in fewer than expected results when searching for certain words

- Initially I attempted to structure relative paths into absolute ones manually, resulting in a lot of headache until I found out you could instantiate a new URL with a relative and absolute path and have the creation of a valid URL handled automatically
- I had issues with WebCrawler taking a long time traverse large directories such as RHF until I optimized the input/output streams and decided to start using HashMaps and Sets

Assumptions:

- Queries are in English, alphanumeric, and any operations are formatted correctly
- WebCrawler is only expected to crawl html pages

Testing Methodology

Overview:

To comprehensively test WebCrawler, I conducted a majority black-box and integration testing while utilizing white-box JUnit tests for certain functions such as those contained within WebIndex and CrawlingMarkupHandler. For the most part, I found exhaustive black-box testing sufficient for WebQueryEngine due to the fact that I could create simple queries for which I knew the results, and then increasingly make them more complicated to determine whether my support for query operations was working correctly.

White-box Testing:

The white-box testing I conducted on WebCrawler consisted of testing the WebIndex class to determine whether its functions were executing correctly. I found this to be the necessary extent of white-box testing on this project because the functionality of WebQueryEngine and WebCrawler could be easily observed through black-boxing input, especially on smaller directories like superspoof. Thus, by simply ensuring that the basic functionality of WebIndex worked correctly (i.e. not missing any words, storing everything efficiently and storing all URLS), I could be confident that by implementation was working correctly after black-box testing when running the program as a whole.

- TestStoreWords()
    - Checks whether an index, given a long dictionary of words, could efficiently store all of them in a WebIndex with the appropriate URL
- TestStoreAttributes()
    - Checks whether an index, given several href keys and URL values, can effectively store all the given URLs
- TestGetWords()
    - Checks whether a populated index can correctly return all words passed to it, with the appropriate corresponding URLs
- TestGetAttributes()
    - Asserts that the attributes HashMap can be returned containing all URLs passed to it with the key “href”
- TestStoreInvalidWords()
    - Checks whether non-aphanumeric words will correctly be neglected from being stored in the index

Black-box Testing

After performing the above white-box tests, I was confident in the basic functionality of indexing structures. Thus, since CrawlingMarkupHandler simply passes values to the index I was able to black-box check whether all desired values were being stored. I did this by printing out all the text found at the end of a document and then comparing the results to what was actually stored in the index. This was easy to achieve through simple print statements and showed me that only the desired, alpha-numeric text within the correct tags was being stored to my index. I could then move on to testing WebQueryEngine, for which the primary goal was to see whether simple queries could be made more and more complex while achieving the correct returned web pages. I did this by starting with a one-word query, for example "jedi", for which I previously confirmed that there were only 7 webpages containing said query. After confirming that the correct number of results were returned when running WebServer, I could make the query more complicated, adding in basic operations like AND (jedi & star correctly only returned 3 results) and OR (jedi | star correctly returned 228 results, the number of pages returned for Star plus the number of results returned for jedi minus the number of pages containing both words). Through this process, I made the queries more and more complex, confirming the results with numbers I had counted/confirmed independently until I was confident that all possible queries and combinations of queries would work correctly for my implementation of WebCrawler. I was also able to test the optimization of my code through a similar process, searching for queries such as "and" for which I know there are a little over 7000 results before and after changing an aspect of my code (i.e. which data structure to use) and observing the time cost it took to run WebCrawler, allowing me to increase the efficiency of my code in this way.

Conclusion:

Overall, I believe I implemented an extremely comprehensive and efficient solution to WebCrawler, utilizing JUnit white-box testing to confirm the basic functionality of my code and then black-box and integration testing on my own to determine whether the integration of my program was working correctly. Since all input was done by the user, it was easy to check edge-cases by simply inputting them myself and then seeing whether my code worked correctly, especially when inputting small directories for which I could confirm the correct number of results (e.g. superspoof, etc.). As a result, I was able to achieve a very robust solution that functions for all possible edge cases and ultimately passed every test I could throw at it.