Project 7 Report

Zhaosong Zhu | CS 314H

Assignment Goals

In the *WebCrawler* project, the objectives are to implement *WebCrawler* supported by the *Handler*, design an index with Java data structures, support *WebQueryEngine* search with a set of given grammar rules, and perform a comprehensive testing on my implementations. Through the process of designing the WebCrawler, I am to develop a relatively comprehensive understanding of how to tokenize a query input and use proper data structures to efficiently store/retrieve information of a webpage. My personal goals include picking efficient data structure that allows fast data retrieval, knowing the basics about HTML language, and finishing bonus Karma to make my crawler more useful.

Solution Design

The design of the *WebCrawler* project involves three aspects: *CrawlingMarkupHandler*, which the *WebCrawler* class relies on to obtain information from each URL given, *WebIndex*, which uses internal Java data structures to store useful information crawled by the Handler, and *WebQueryEngine*, which is responsible for parsing query inputs and search inside a given index. The *CrawlingMarkupHandler* class uses the given library to handle the opening and closing of a page/tag and crawl words inside each tag. The *WebIndex* is a self-designed class that heavily relied on HashMap to store information under each URL page and supports the four types of searches outlined in the assignment description. The *WebQueryEngine* is both the parser or query inputs and calls methods inside *WebIndex* class to obtain proper results. To see if all these classes are working, we also design a test harness for each of these classes.

A. High-level details on implemented classes and methods

1. ***Word/URL Crawling from Each Given URL***
   With the given Attoparser library, we handle all elements in a given URL webpage by their tags.

   The *handlerOpenElement* method will be executed each time when it finds an open tag, such as A, which contains a path to another webpage. To allow the crawler to go through almost all pages that are linked together, whenever the parser finds an element named A or a, it will obtain a string that contains *.html* from the attributes using a key *href*. Then, the URLs found will be added to an ArrayList, unless it's already stored in the Queue.

   The *handleText* method helps to obtain all words within each tag. Since a start index and the length is given, we will append everything within the scope to a string. When the traverse is finished, I used Regex to filter out all the white spaces and get individual words, which are added to the index file with a counter that denotes the position of this word within this webpage.

2. ***Query Language and Token Design***

   As specified in the assignment instruction, a valid Query can contain the *word, !word, &(explicit and implicit),* |, *and ().* Thus, I designed a *Token* class, which contains seven boolean variables to denote what kind of operator, word. Or phrase it is. Two string variables are used to store either the word or the phrase.

   The token class makes it easier to perform parsing and even postfix calculations. During parsing, we can put () as tokens into a stack to ensure expression of postfix expression. Thus, during postfix calculation, my stack only needs to support two data types - Token and ArrayList, which stores search results.

3. ***Page object***

   A page object is used to represent a given URL page. The information it contains, besides the URL of the page, includes the title of a page, the excerpt of related words, and a score of this page, which is used to rank all pages from a given search.

   A HashMap called *wordScore* is used to keep track of which query words are used to obtain this page and how many times does each word show up on this given page. With this information, we can give a score for this page - which is the addition of the occurrence of each word and whether this word shows up in the title. If any given the word shows up in the title, the score +3 each time.

B. Abstractions

1. ***HashMap and HashSet***

   HashMap and HashSet are frequently used throughout the components of the WebCrawler. They are mainly used for two purposes: check the existence of a variable and quickly retrieve information.

   HashSet is used mainly inside *WebCrawler* and Crawling*MarkUpHandler* classes. Since new URLs are discovered when a page is scanned, and the objective is to prevent crawling the same page twice, I add whatever page scanned into a HashSet. Thus, whenever a new URL is found, I can perform an O(1) action to determine whether this URL is already added/ scanned. In doing so, the URL adding and searching process will be less operationally expensive.
   HashMap is mainly used inside *WebIndex* class, which supports key-value pair. Two Maps are used for this functionality - *URLPositionMap,* and *URLWordMap.* Both maps have a URL as the key, while they have a different HashMap for their key. *URLPositionMap* uses a key of integer, which denotes the position of a word on the page, and a String as the value. This map helps determine if several words are consecutive on a webpage. *URLWordMap,* instead, uses a String as the key and use an ArrayList representing all positions that the key occurs on the page. This map helps determine whether a word does exist inside a page and what are the positions to check.

2. ***Regex and Whitespace***

Regex is used to fix whitespace in the string pasted by the handler or a phrase obtained when parsing a query. A pattern [a-zA-Z]+ is used to extract word by word from the string. Then, this pattern is passed to a matcher, which outputs the words.

The other two cases of whitespace lay in query inputs. One case would be whitespace at the front/end of a query, which should be deleted. The other case is whitespace between a token - we will keep going forward until a token can be met.

Project Discussions

A. Assumptions, Designs, Strength, and Weakness

As we write up codes for the WebCrawler project, one important consideration, as mentioned in the instructions, is to correctly and efficiently paste queries and perform the search in the *WebIndex* created.

Assumptions

1. ***Regular Query Input***

   Since my query engine supports the case where no bracket exists in a given query, an assumption made is that as long as a given query contains any operator(& or |), at least a bracket, either left or right, should present inside the query. This is because the checker of regular compounded queries can only check if a query contains bracket. Without bracket, it can be treated as a karma supported query.

   Another related assumption is that a regular query given should not contain extra brackets. For example, a query like *((ivy & !liang) "wealth fame"   sanity | (happiness & joy))* does contain equal pairs of brackets while *sanity | (happiness & joy)* is indeed an invalid query. Compared to *(ivy & !liang) "wealth fame" (sanity | (happiness & joy)),* the query above will give different results, although both of them can be calculated.

2. ***URL Paths***

   In my JUnit test, I used several webpages from rhf folder to test various functions of my *WebIndex* and *WebCrawler* classes. However, each URL given is based on where the file is at on my PC. This may cause failure of test cases due to an invalid path. So, the assumption is that all URLs given in the test cases, which are used for index testing, are valid.

Class Design Highlights

1. ***WebIndex* Design**

   My WebIndex class contains three HashMaps and eight methods to support the query and search. With a URL as the first key, *URLPositionMap* uses a key of integer, which denotes the position of a word on the page, and a String as the value. This map helps determine if several words are consecutive on a webpage. *URLWordMap*, instead, uses a String as the key and use an ArrayList representing all positions that the key occurs on the page. These two maps allow the *WebIndex* class to perform a phrase search. Besides, a *tilemap* is used to store the title for each given URL.

During phrase search, each word from a phrase will be taken and converted into an ArrayList. To start off, we will search the first word and obtain all occurrences of this word within each URL. Then, using each position, we can use the URLPositionMap that helps us determine if the consecutive words match up with all the words within the ArrayList, with correct ordering. If a URL can pass all the tests for any of its occurrences, it can be returned as the desired result.

Besides, the *WebIndex* can also help a page form it's highlighted excerpt. Since each page object contains a set of words/phrases that were used in a given query to obtain this page, the *pagePreview()* method can find a given the word and phrase, provide two words before and after the word, and form a preview whose searched word is highlighted. The method processes each used word/phrase and accumulates previews found from each one as a complete preview, which is used for Karma.

2. **Shunting Yard Algorithm**

   The shunting yard algorithm is used to obtain results based on the query. Having translated a regular expression into a postfix expression, the *QueryEngine* uses the shunting yard algorithm to process the translated query. The shunting yard algorithm calculates a postfix expression by doing the following:

   1. If the current token represents a word/!word/phrase, it will be pushed into a stack;
   2. If there's only one token exists in the end, we will use methods provided for single token within the *WebIndex* class to obtain a query result. Otherwise, if a query is valid, there should be at least three elements.
   3. Whenever an operator is encountered, we will check if there are at least two elements inside the stack; if yes, pop them out, perform a search, and push the result as an ArrayList inside the stack again;
   4. Perform the following steps until we reach the end of all tokens. Then, pop out the element left inside the stack, which will be the final result.

Postfix for query *(a & (b & c))*, for example, will be *a b c & &*. When calculating, *a b c* will be put in the stack first. When the first & is encountered, we will calculate b & c and place the results back into the stack. Then, when we encounter the second &, we will calculate (a & (b & c)), which is the desired result.

This algorithm is helpful because of two reasons. First, shunting yard algorithm eliminates the need for parenthesis, which denotes priority of calculation, since all expressions are post-fixed. Second, this algorithm also allows us to process expressions without any parenthesis at all because it's parenthesis independent. Thus, this algorithm earns a Karma. However, we do need to define the priority of & | operator, which will be discussed in the Karma section.

Weakness and possible improvements

I believe my project design has at least two possible weakness.

First, the negation search is in general not very efficient. During a *!word* search, my search methods inside the *WebIndex* class needs to almost traverse through all the URLs stored to determine whether a given the word doesn't exist on the webpage. Although checking whether a word exists insides a HashMap is an O(1) operation, it does spend roughly O(n) time to traverse. A similar situation happens with the full support of negation search.

Second, although my WebIndex only has three HashMaps, the *.db* file created is usually pretty large - around 80MB for rhf. The large size of the *.db* file makes it slow for the *Serializable* interface to convert a *.db* file to a *WebIndex*. A solid 20 seconds is required to load the database file for rhf, which is somewhat undesirable. However, due to the slow nature of the interface, we can't do much about it.

B. Edge cases

This is a list of edge cases that I found. Please refer to the testing section for handling.

a. Test if we can find the two URLs used above after obtained them;
b. Test if we can add the same URL twice
c. Test if we can add the same URL with #
d. Query with irregular characters
e. Check empty query or empty inside a phrase
f. Crawl URL doesn't end with .html
g. A webengine with a not initialized webindex

C. Karma & Personal Reflections

1. Karma discussion

I have finished four Karma options from this project - search result excerpts, removal of parenthesis restriction, webpage ranking by meaningful factors, and full negative query.

The first karma is search result excerpts. Since a query can be as simple as a word, or as complex as numerous nested queries, I let the result excerpts use the first five words(in case there are more) used in a query to be the highlighted word/phrase. For example, if a query is (a & (b & (c & d))) and a webpage contains all four words, then the excerpt should use all four words inside the excerpt. Then, using words related to this webpage showed up in the query, we will find each word/phrase's first occurrence, highlight this word, and assume it has position k, the excerpt from a single word would be (k-2) (k-1) (k) (k+1) (k+2) and the similar idea applies to the phrase. Finally, we append all the excerpts formed together as the excerpt from the page. One possible drawback is when doing an (a | b) search, whose result may contain both words but only one would be shown in an excerpt.

The second karma is parenthesis restriction removal. By using the shunting yard algorithm, once an expression is translated into postfix expression, we can easily calculate the results using the steps outlined in *Class Design Highlights* section. Under the situation of no parenthesis, all single word is similar to a number in an expression. !word or phrase are dealt with similarly, except that when the postfix calculator encountered them, it will use a method from the *WebIndex* class to calculate out results as an ArrayList of *Page*. In terms of operators, I defined 4 priority from low to high: 0 = (, 1 = |, 2= &, 3=). All the words will be put into output directly when translated to postfix expression. The operators, however, will be put into a stack with the following rules: the upmost operator in the stack must have a priority <= that of the current operator; if not the case, pop operators out to output until either the condition meets or the stack is empty; (can be added in regardless of the priority; whenever (is found for), we just pop (out without putting them into output; when all tokens are processed, pop everything out from the stack to the output.

The third karma is ranking the pages by meaningful factors. I used two factors to roughly evaluate a webpage - the number of times that each related query word shows up on this webpage and the similarity between query word and page title. As mentioned earlier, each *Page* object contains a HashMap that stores key as a word/phrase and value as an ArrayList whose length representing the occurrence of that word/phrase within the page. Thus, to calculate the rating of a page, the procedure will be:

```
for each Key:
    Add the occurrence to an integer score counter;
    If the key can be found in the title, +3 to the score;
return final score
```

Before showing all the pages as query results, a comparator is used to rank all pages, with the highest score in the front and lowest at the end.

The fourth karma is supporting negations on all query types other than just *word*. According to the grammar provided in the assignment description, the negative query can be the following situations:

1. *! word*
2. *! "this is a phrase."*
3. *! Query containing & | ()*

Thus, to support all the situations above, I added to the Token class a boolean variable *isQuery* and an ArrayList to store Tokens that forms a prefix query. When parsing a query, if an ! is encountered, the program will determine which situation is it and output a negative Token that contains either a word, a phrase or a Query. If a word exists, the *negationSearch* method from the *WebIndex* class will be invoked. If a phrase is found, we will first obtain all webpage that contains the phrase and invoke *negationSearchOnResult* method to get the negation. If a query exists, we will evaluate the query first and call on the same method to obtain the negation. The negation is given as a result placed in the result stack.

2. **What I learned from this project and future improvements**
   On the one hand, this project is the climax of various data structures we have learned in this class. As denoted in the abstraction section, HashMap and HashSet are two data structures frequently used. By using these data structures inside actual projects, I am becoming more knowledgeable about their usages and situations to use maps and sets. Indeed, they are great data structures to help store and retrieve data relatively fast. Since these data structures are even common data structures showing up during the interview, I feel somewhat more competent now.

   On the other hand, this project exposes me to basics of HTML. When using the Attoparser, I inspected the source code of webpages provided and got familiar with some tags - <A, <title. These tags contain key information for page search in this project. For example, the <a tag will usually provide relative/absolute path for the links on this page linking to other pages.

   Besides, since the *WebCrawler* project is comprised of many parts, it is very important for me to have a clear idea about what each part are responsible for. This is important not only because the different parts work together to support a certain function but also because knowing each part clearly may help the debugging process - being able to think like a stack to trace an issue from class A to class B. The *WebCrawler* project gives me a better idea of designing a relatively big piece of software.

Testing

A. Testing Methodology

The JUnit testing for this project includes both integration and unit tests for the following classes: *CrawlingMarkUpHandler*, *WebQueryEngine (Token* parser, *Query* checker, infix to postfix expression, postfix calculation), *WebIndex* (basic functionalities and search methods), *Page* (basic functionalities and integration with a search on actual webpages). Below are the specifics of both the unit tests and integration tests I performed.

***CrawlingMarkUpHandler* Test:**

The Handler helps the WebCrawler to go through each webpage and crawl down useful information, including URLs on the page, words on the page, the title of the page. Since most of the method implementations are provided by the given library, my test mainly focuses on adding/processing URLs without repetition and some edge cases. Since there are relatively few items to be tested, this is a single test.

The items tested are listed below:

- Parse two randomly chosen URL and obtain new URLs from the handler;
  - EXPECTED: should obtain two ArrayList of URLs
- Test if we can find the two URLs used above after obtained them;
  - EXPECTED: shouldn’t be found
- Test if a newly added URL can be found;

    - EXPECTED: TRUE
- Test if we can add the same URL twice
    - EXPECTED: FALSE
- Test if we can add the same URL with #
    - EXPECTED: FALSE
- Test if we can get any URL at the beginning of the crawl
    - EXPECTED: *FALSE*
- Test the *wordCounter* value at the beginning of the crawl;
    - EXPECTED: *wordCounter = 0*

I used JUnit 4 for running all our unit tests. For the most part, I used the assertEquals and assertArrayEquals methods to check if the actual values matched with the expected values. As for negative cases, We used a couple of asserts (false, a==b) statements to make sure that the test harness detected that we were passing in the wrong inputs and that it wasn't just passing all the cases. The negative cases are the following:
- Test if we can find the two URLs used above after obtained them;
    - EXPECTED: shouldn't be found
- Test if we can add the same URL twice
    - EXPECTED: FALSE
- Test if we can add the same URL with #
    - EXPECTED: FALSE
- Test if we can get any URL at the beginning of the crawl
    - EXPECTED: *FALSE*

***WebQueryEngine* Test:**

WebQueryEngine is one of the most complex classes in the project, as it includes *Query* checker, *Token* parser, infix to postfix expression, postfix calculation. For each of the functionalities, I designed both unit test and integration test to cover various cases.

- The **Query checker** is comprised of two methods: *check characters*, which checks if there are unexpected punctuations in the expression and *checkQuery*, which checks the correctness. To check if these two methods can effectively filter out invalid queries that I can think of, I designed a group of edge cases to do testing. The items tested are listed below:

- Query with irregular characters besides valid operators specified in assignment
    - EXPECTED: *FALSE*
- Check the situation where parenthesis exists in a given query
    - EXPECTED: assuming no extra brackets, # of pairs of brackets should match up with # of & or | be *TRUE. else FALSE*
- Check empty query or empty inside a phrase
    - EXPECTED: *FALSE*
- Check implicit and
    - EXPECTED: at least one space is required for implicit, and all conditions above should satisfy to be *TRUE; else FALSE*

- Check support for negation Query
    - EXPECTED: space can exists between ! and (or word or “a phrase.”
- All valid cases used for Token parser
    - EXPECTED: all of them should be *TRUE*

Besides the simple cases given above, an integrated test on the mixture of complex situations is also given with various white spaces mixed inside the input to check if the checker can successfully identify them. The complex cases involve & and | on words or query, implicit and two or more queries. We used a couple of asserts (false, a==b) statements to make sure that the test harness detected that we were passing in the wrong inputs and that it wasn’t just passing all the cases. The negative cases are the following:

- Query containing weird characters
- # of parenthesis pairs doesn’t match with # of & or | operator
- Empty query or query with empty item

● The **Token parser** reads a valid query and whenever a Token is found it will be returned. As specified earlier, the Token can be () & | word “phrase” !. I passed in single token cases, which are just a word, phrase, or !word, and more complex cases. Below is a list of tests I performed to test the parser.

- Parse a query containing a negative query
    - EXPECTED: any query comes right after ! should be parsed as one token with the negation sign, despite spacing in between
- Single-word query
    - EXPECTED: return one query containing only the word
- Phrase query
    - EXPECTED: return one query containing only the phrase, with irregular spacing in between fixed
- Query containing & operator and various elementary token on the sides
    - EXPECTED: each elementary token and & token should be correctly identified with correct order
- Query with implicit and
    - EXPECTED: as long as the format is correct, there can be however many spaces in between
- Integrated test on more complex situations
    - EXPECTED: all tokens should conform rules above with correct order

Besides the simple cases given above, an integrated test on the mixture of complex situations is also given with various white spaces mixed inside the input to check if implicit & and regular tokens can be successfully identified. The complex cases involve & and | on words or query, implicit and two or more queries. Since I implemented full support for negative query, two unit tests are dedicated to testing parsing those token. I used a couple of asserts (false, a==b) statements to make sure that the test harness detected that we were passing in the wrong inputs and that it wasn’t just passing all the cases.

- The **infix and prefix calculations** happen after a String expression is translated into an ArrayList of tokens. I use all the cases from above and see if their postfix calculations translation are correct. Below is a list of tests I performed to test the parser:

- Translate a single token to postfix
    - EXPECTED: obtain the same token
- Simple & and | token
    - EXPECTED: tokens should be returned in the order of *word1 word2 operator*
- Nested & and | token
    - EXPECTED: similar case as above with correct order defined ()
- Implicit and
    - EXPECTED: processed the same way as a regular &
- Query with implicit and
    - EXPECTED: as long as the format is correct, there can be however many spaces in between
- Integrated test on more complex situations
    - EXPECTED: all tokens should conform rules above with correct order

***WebIndex* Test:**

WebIndex is used to store and retrieve information on the webpage, thus performing searches. I crawled five random pages from rhf and formed a WebIndex using these pages. Then, I tested each method inside *WebIndex* with knowledge of what these five pages should return. The tests are specified as below:

- Perform searches on an empty index
    - EXPECTED: 0 results should be returned
- Search on single word query
    - EXPECTED: return pages that contain the word
- Search on phrase query
    - EXPECTED: return pages that contain the phrase
- Search for compounded queries to test & |! search methods
    - EXPECTED: show results as expected

Besides the simple cases given above, an integrated test on the mixture of complex situations is used to see if correct results can be returned. I went to each page and checked how many pages are there out of 5 that satisfy the query.

***Page* Test:**

The Page object stores four types of information: URL, page title, a score of the page, query words used to find this page. To test the basic functions, I manually add in all the information above and see if correct results can be returned. Then, I incorporated all these tests with a 5 URL webindex and various searches to test if correct information can be returned. Below are the tests implemented:

- Set and get a title
    - EXPECTED: should get the same thing

- Add three words with occurrence into a page, with two of which matches with the title
    - EXPECTED: return a correct score = sum of occurrence + 3*2
- Equals method of two pages
    - EXPECTED: *TRUE* if URLs are equal
- Perform score testing on five given pages
    - EXPECTED: show results as expected

With all the information above, the page object helps display excerpts on the TSoogle search result page. However, highlight and excerpt can be tested through black box testing.

B. Black Box Testing:

While I was coding the assignment, I performed various queries over and over to make sure that the particular feature we had just completed worked. This is a list of things I tested:

Check if *WebCrawler* can start off from a page and crawl desirable # of pages
Check if valid query can render a list of pages
Check if invalid query render no result, rather than crashing
Check if excerpt shows up for any single word, phrase, and compound query search
Check if each page has a scoring, with negation searches return 0
Check if all pages are ranked with score high on the top

In addition to searching in the webpage, I also used System.Out.print statements in several places for debugging the code.

C. White box testing:

As explained above, we created both integrated and unit tests for functions in each class.

D. Testing Exceptions:

The exception may happen when trying to crawl invalid webpage and parsing the invalid query. In our testing harness, we have tested the cases below:

1. Crawl webpage doesn't contain .html - should be automatically ignored
2. Crawl webpage doesn't exist - should be automatically ignored

3. Invalid query, whether caught or doesn’t caught at the beginning - throws an exception, catch it and print system.err.print a message
4. A webengine with a not initialized webindex - throws an exception

In each of these cases, we will catch the exception. Since the exception is with a specific message of the error, we will see if the error message with the exception is matched up with our expectations, thus testing exceptions.

E. Strengths and Weaknesses of our Testing Methodology

Strengths:
Testing the code increased our confidence in its correctness. Since we used JUnit, I was able to automate the testing process. Instead of having the program print diagnostic statements and examining those statements to make inferences about the correctness of our program, I can have the JUnit test method check for correctness automatically. Having an automatic checking mechanism shortened the time required for testing individual cases considerably.
Weaknesses:
I came up with a large number of test cases and tried to test as many cases as I could think of, but couldn’t code all of them due to time constraints. However, by analyzing case by case, I feel relatively confident.