WebCrawler Report

Arvind Raghavan

December 2017

1 Introduction

This project has two main components. The first is to build a WebCrawler to crawl a small, downloaded portion of the web and build an index of the words on each page. The second part was to build a query parser that could understand logical queries and return websites that matched the given queries. I thought this project gave me a unique opportunity to learn how search engines actually crawl the web, and to learn how to effectively parse logical queries. My main goal was to fully understand how both of this things worked as I completed the project. By the end of the project, I'd like to be able to build a WebCrawler from scratch, *without* using the stub code given to us.

2 Solution Design

2.1 Assumptions

As usual, I tried to make this assignment with as few assumptions as possible. However, I realized soon that many implementation details with storing words were left completely up to us. The first assumption that I made was for punctuation. Of course, periods, commas, and exclamation points shouldn't be indexed as part of the word, as nobody really cares where in the sentence these words are showing up. However, punctuation within the words were a different matter. I decided to leave apostrophes in the word, as people are much more likely to search for words like "don't" with the apostrophes. However, I decided to remove hyphens altogether and index their words separately. I made this decision because, from my experience with search engines, I feel like people are much less likely to search for hypenated words. In addition, it makes sense for words like "anti-intellectual" to be returned for a search for the word "intellectual".

Speaking more generally, I also made the assumption that all parts of my code were to be used with each other – that is my Page class would not be exchanged for another Page class in testing. While I typically try to do modular programming, because there were no abstract methods that specified implementation for, say, WebIndex, I wrote my WebQueryEngine in such a way that takes

advantage of my exact WebIndex's implementation. This was very liberating in comparison to previous projects, though I still made each class as modular and well-commented as possible.

The final assumption that I made was that we only had to crawl URLs ending in `.html` and `.htm`. I initially caught several parse exceptions, but I realized quickly that, for whatever reason, some images could successfully be parsed without failing. This meant that a simple negation query, say *!and*, would return links to images, which didn't seem correct. Thus, I decided to optimized the crawling to only URLs which I knew I could parse, catching parse exceptions only for broken HTML pages.

2.2 Abstractions

- **Valid Word** - In one of the first discussions, Tres mentioned that there was no reason Chinese characters couldn't be given to us... To account for this, I made a static method in my WebIndex called isWord, which uses Java's built-in Character.isLetter() method to determine if each character in a word is a letter. Abstracting this identification to one method allowed me to do white-box testing once, and then reuse the method again for validating words in queries.

- **Query Building** - My WebIndex has several data structures which are used to efficiently return results to different types of queries. Instead of having to manually update all of these data structures in CrawlingMarkupHandler, I abstracted query building to a single add method in WebIndex. When I expanded the data structures in WebIndex, I was able to to simply modify the add method in WebIndex, without having to worry about changing the crawler.

- **PostFix** - The packet described a recursive query parsing system with tokens, which I considered implementing. However, I realized that its implementation would heavily rely on a valid query being input – that is, invalid queries would be more like to crash the system instead of returning an invalid result. Because I knew beforehand that for invalid queries, all we had to do was make sure they didn't crash the program, I decided to convert the queries to postfix using the Shunting-Yard algorithm and parse them from there.

- **Query Parsing** - The assignment packet structured the query parsing in layers, adding features in each section. At each section, the query had to be modified additionally in some way. I abstracted the query modifications to a single method, called getPostFix, which called several helper methods to reformat the query, and then converted the query to postfix. Putting all of these different reformatting methods into helper methods allowed me to do white-box testing on each step and easily find at which point the error was occurring while using a debugger.

- **Lower Case** - The entire assignment is case-insensitive. To make sure of this, most methods that take String parameters start by converting them to lower case.

- **Sets** - In the project, we are rarely concerned with duplicates. If a word is found on a page multiple times, we don't want that same URL to be stored multiple times within that word's indexing – we want it stored once. To account for this, I opted to use Sets instead of Lists for almost every data structure. Because we stored web pages in Page objects, I made sure that every page was only associated with one page object, which was initialized once in CrawlingMarkupHandler. This allowed Java's built-in Sets to remove the duplicates for me.

2.3 Web Crawling

The WebCrawler made use of a library called AttoParser, which provided methods for parsing HTML. To use it, our CrawlerMarkupHandler class just had to extend the AbstractSimpleMarkupHandler class. Then, when the a webpage was passed in, the inherited abstract methods were each called by another class whenever a certain part of the HTML was being parsed, and the text found was passed to those methods as parameters. This abstracted the job of reading the HTML itself and made it easier to find the needed information.

URL links are typically stored as hyperlinks in HTML a tags. However, as described in Section 2.2, I already knew that I only had to parse HTML files. As such, I decided simply check every element in every tag for something ending in HTML, and from there try to resolve the String to a URL. Because I am somewhat unfamiliar with HTML, checking element was the best way for me to ensure that all URLs were parsed. Because I only used a try-catch for URLs ending in HTML (where it would be unlikely to have a broken link), I felt like this solution was a good balance between full coverage and optimization.

For parsing text, I first used the println statements given to us in CrawlingMarkupHandler to look at the text in each tag. After looking through about 20 different files in superspoof, I came to the conclusion that both the *style* and *script* tags always contained gibberish. The rest usually seemed to have coherent text that was worth indexing. To find valid words, I first built a text String by appending a given character if it was a valid letter or apostrophe, and a space if it wasn't. This allowed me to get only the valid words from a String, with spaces properly dividing each word. I then formatted the String by replacing all 1 or more spaces with a single space, and then trimming white space off of the beginning and end of the String. This gave me a String with 1-space separated words for me to parse and add to the index.

2.4 WebIndex

My WebIndex used three main data structures. The first was a HashSet of all the visited pages, which was useful for determining in constant time if a page

had been visited before. The other two are described below.

2.4.1 Mapping Words to Page

I initially wanted to create a HashMap that linked page URLs to a HashSet of all the words that page contained. For a simple lookup operator, say on the word `and`, this would take $O(n)$ time, where n is the total number of URLs. For more complicated queries, say `and & or`, this would take $O(n)$ time to find the pages with `and`, and then $O(k)$ time to find the pages with `or` (where k is the number of pages with and in it).

This seemed okay at first, but I decided to change my design in the spirit of splay trees. Splay trees optimize for the most common actions, and as such I wanted to do the same for queries. Simple queries such as and, need not take $O(n)$ time, especially since when this project is scaled up, searching every possible web page for such a simple query is just not realistic. Thus I decided to reverse the index to map word Strings to a HashSet of URLS. This allowed me to retrieve the Set of all websites with `and` in them in constant time. This however, potentially worsens the runtime of a query such as `and & or`. If, for example, there were 20 websites with `or` but 100 million websites with `and`, simply taking the set intersection could still be a computationally intensive task, depending on the strength of the hashing function. However, for the purposes of algorithm analysis assuming the set contains method is $O(n)$, the task would be $O(k)$, where k is the amount of pages with `or`. This is still better than the $O(n + k)$ solution described above, and thus I decided to store my words in a HashMap of String to set of URLs.

2.4.2 Phrase Queries

I realized quickly that phrase queries couldn't be implemented with a simple HashMap structure, because the whole point of the HashMap was that the Strings were unordered. However, I couldn't give up the constant time from hashing, and because this project was done on a small scale, I decided to implement another data structure, which linked all of the words on a website together. To parse a phrase query, I first find the set of all websites with every word in the query, and then for each website, I search through the linked list linearly to find the phrase. This solution runs in $O(nk)$ time, where n is the amount of website with all the words and k is the amount of words on each page. This solution works well on both superspoof and RHF, but it won't scale well with larger sections of the web. Even with the optimization of only looking through pages with all the words from the query, on a web engine the size of a Google, a linear search through their millions of web pages would be extremely intensive.

2.5 Query Parsing

Instead of doing the recursive parsing described in the packet, I decided to convert the query into postfix. Note that the and operator could be seem as a

multiply, and the or operator could be seen as an add. Thus, the algorithm can be converted just like any mathematical expression into postfix, just as it's done on graphing calculators. To do this, I used the Shunting-Yard algorithm. From there, I parsed the postfix, adding terms to a stack, popping from the stack when an operator was found, and then pushing back to the stack. At any point, if I'm trying to pop from the stack when they are no more elements, I know that the infix notation given to me was invalid, and I can throw the appropriate exception. An important note is that the Stack contains objects, and the terms pushed to it can be Strings or sets. Thus, I have multiple if statements checking the instances when performing the `and` and `or` operations. In some scenarios, this involves using set intersections, say for example in the query `( a | b ) & ( c | d )`. You must first `or` a, b and c, d, and then `and` the sets together.

3 Reflection

3.1 Personal Thoughts

I started this project early, so I didn't find it as difficult as people had said it would be. I really enjoyed being able to implement something which an actual practical purpose – something that people really used in industry (in contrast to Treaps...). I also enjoyed being freed from constricting abstract classes. Although I understand the modularity for the purposes of testing, having the reign to implement the data structures I want in the way that I want was very liberating. If we had done this at the beginning of the year, I probably would have put everything in the main method. But now that I have some good object oriented principles, it was nice to be able to implement them freely.

3.2 Algorithm Analysis

I prefer to analyze my algorithms as I describe them, so I put my algorithmic analysis in Section 2.

3.3 Karma

Most of the karma that I implemented was for the WebQueryEngine. The potential karma is listed below.

- **Parenthesis** - Because I used Shunting-Yard to convert my queries into postfix notation, I can handle queries without any parenthesis and queries with extra parenthesis.
- **Phrase Negations** - Phrases, in addition to words, can be negated as follows: ! "hello world".
- **Query Validation** - I employed a complex system of query validation to determine whether or not a query is valid. For every step of the validation, I throw exceptions with the appropriate error message. From that, I catch

the exception and display the error message on the search engine for any invalid search.

4 Test Methodology

For general testing, I realized I had two main options. I could either create my own HTML websites to crawl, or I could use a command-line tool like grep to test my results. I decided to do the latter for multiple reasons. First, I am not very familiar with HTML. If I created my own websites, I would probably put all the text within a couple of the most basic tags, and the links would all be referenced in the most standard way. I realized that this would not be a comprehensive testing strategy, because it wouldn't account for all the possible edge cases within the HTML and AttoParser that I couldn't possibly be aware of. Instead of doing that, I decided to verify my searches with case-insensitive grep results. I realized that grep would always return a superset of the words that my WebIndex returned, which would allow me to manually check the discrepancies and verify that my indexing works as expected. Though this was a lot more manual work, I think it allowed for much more comprehensive testing.

4.1 Grep Tests

The first tests that I did were on single words. I chose words with around 100 results in RHF, because I couldn't manually check thousands of files. I took the results of grep for those words, and put them into the text file `grepTest.txt`. From there, I read them from my testing suite and compared the files that they returned with that of WebIndex. Grep would always have extra files, because searching for words like "insane" would return words like "insanely" in grep. For each word, I manually looked through the set difference to figure out whether or not the file should have been returned by WebIndex. If it shouldn't have been, then I could safely remove it from `grepTest.txt`, and if it should've been, then I knew I had a bug to fix. I did this on four or five words to solidifying the testing. Looking through these pages manually helped me understand exactly what my parser and WebIndex were doing, which helped me gain a greater understanding of my own expected behavior.

4.2 Find Testing

In order to make sure all the files are being indexed, I used the command line tool `find`, to recursively get a list of all the .html files in both superspoof and RHF. I then took the set difference between all the files and the list of visited files generated by WebCrawler. Both in superspoof and RHF, there were more HTML files in the directory than I crawled. For each of those files, I grep'd the directory for it and ensured that there was either no link to it, or the link was hidden in some Javascript. Doing this helped me make sure that I was actually parsing all of the files that I should be.

4.3 Helper-Methods Testing

In addition to black-box testing, I did a lot of white-box testing to ensure that my query parsing and WebIndex helper methods worked as expected. A lot of these methods started to fail every time I changed a feature about, say, how I addressed punctuation, which was a good reminder for me to verify the new expected behavior. In addition to helper methods, this was the first project where I wrote some methods directly into the helper class itself just to test how some aspect of Java worked. Being able to use asserts to quickly verify or update my knowledge of Java was really useful, and I will continue to do such testing in the future.

4.4 Integration Testing

My main integration tests came from the list of grep tests described in Section 4.1. Once I verified the single word results of grep with that of WebQueryEngine, I could test full queries using those single words. For each integration test, I loaded each individual word from my grep file, then did the & and | operations on the given sets using Java's built-in `retainAll` and `addAll` methods. Then, I compared the resulting set to the one returned from WebQueryEngine.query(). Doing so actually caught some bugs at the last minute, which surprised me. Up until doing real integration tests, it's very difficult to know if your project is actually working correctly. Doing these integration tests gave me some piece of mind as to whether my project really worked.

5 Conclusion

I found this project very instructional and a good practical application of CS skills. Being able to explain to people that I crawled websites and built and actual web index is (and will be in job interviews) very cool. Though I didn't implement as much of the packet karma as I usually do, I don't feel like any of was too far out of my reach – that is, if I had more time, I could definitely implement it. As a whole, I've really enjoyed the projects in this class. They've pushed me to learn so much more on my own than I otherwise would have, especially the karma. I'm really glad I took this class and joined Turing, and I think that I'm not alone in that judgment. To whoever's reading this, have a great winter break!