Program 7 - Web Crawler

1 Overview and Goals

1.1 Overview

The intent of this project is to build a robust and efficient Web Crawler that can extract useful information from web networks of hundreds of thousands of pages, store the data in a strucutred manner, and support queries of a specified grammar on the web network. This assignment allowed me to make key design decisions regarding data strcutures used for data storage and algorithms used for query parsing with the hopes of minimizing overall space and time complexities.

1.2 Goals

My goals for this project included making sound design decisions regarding choice of data structures and algorithms to optimally balance the trade-offs between space and time complexity in an effort to enhance the search engine's user experience. I also strove to augment the user experience by expanding on the required components and implementing some interesting features of real-world search engines (this is discussed in the Karma section).

2 Description

2.1 Solution Design

The essential modules for this assignment are the classes. implements the overarching logic to perform a Breadth First Search through the entire web network and save the generated index to for later access while uses the library to parse each individual webpage and call abstracted methods to populate the index.

The class is used to encapsulate data regarding a given page, and the class creates an efficient structuring of the crawled network for later retrieval and abstracts the querying process. This data includes a mapping from a word to all the locations where the word was found (which can be uniquely described by a instance and a position). Lastly, the implements the *Shunting Yard Algorithm* to generate the postfix notation of a given query and compute the set of s that satisfy it using set union and intersection.

2.2 Assumptions

Below are some of the assumptions tha tmy implementation makes

- `Source s are Absolute: It is assumed that the original source s are absolute.`
- `AttoParser correctness: It is assumed that the attoparser library correctly parses all the html on each page and does not introduce an extra level of variability.`

3 Discussion

3.1 Scope and Quality of Solution

Below are some of the notable limitations of my implementation.

- **Memory Limits:** In order to support full-text searches, storage of comprehensive content for each page in the network is required in some form. As such, reliance on memory becomes a significant concern at network sizes of high orders of magnitude.
- **Limited Grammar:** The requisite grammar is quite restrictive due to its requirements of proper parenthesization, spacing, and operator notation. While my Karma implementations do loosen the restrictions slightly, they are still not as freely defined as those of real world search engines.
- **Time Limitations:** Once again, full-text searches restrict the response time of the engine on large network sizes. This includes both indexing time as well as query time since a larger proportion of the network must be accessed each time. Other factors such as rudimentary data storage/accessing methods, tools/hardware also contribute to this gap.
- Anything else?

While my engine surely doesn't compare to those in the real world, it attempts to optimize space and time complexities within the restrictions presented by the assignment. Key design decisions regarding data structures, algorithms for query parsing, and abstractions of different functions are essential to my solution.

3.2 Web Crawler

The class performs the high level breadth first search crawl of the web network. Most of the logic for this was provided in the starter code, but two key changes were made with regards to handling malformed s and setting the current being parsed.

3.2.1 Handling Malformed s

The starter code exits when an invalid is found while doing a graph traversal. I altered the program to throw an , ignore the invalid and continue with the traversal. This was done by shifting the `try/catch block inside the while loop and removing any system exits.`

3.2.2 Setting Current Page

`In order to correctly map words to their respetctive pages in the , I added a function setCurrentPage() in the that was called when a new was being parsed.`

3.3 Web Index

`The class provides a framework for data storage of the indexed web pages that will be saved and accessed during the user querying process. Because the query result time is of paramount importance in search engine optimization, many of the key design decision made here aim to trade-off memory for optimized time of the full-text searches at query time. That said, there were also key design decisions made to reduce memory consumption as best as possible without sacrificing too much query time. The key design decisions are outlined below.`

3.3.1 Full-Text Storage Initial Ideas

`Due to the nature of full-text searches, every word on every page must be accounted for and reliably found during query time. As such, I had a few different ideas as to how the crawled data should be stored in order to optimize both space and time efficiencies.`

1. `Full-Text Document: One of my initial solution ideas was to store the entire document of text for each page. This would necessitate a string matching algorithm (such as KMP or Rabin-Karp) to parse through the pre-processed reductions of every single document at query time to produce the query results. While these string matching algorithms are quite efficient asymptotically for dynamically matching new strings at runtime, the space complexity`

2. `Inverted Index: My next idea was to store a mapping from a word to a HashSet<location>, where location is an object container that stores the and position on the page (as defined by the number of words before it) where the word is located. In this way, all instances of a particular word can be found`

quickly. To find a particular phrase, one would then need to check which pages contain the specific sequence of words in a contiguous fashion (as indicated by the positions). This method seemed to be quite promising, except that storing the location instances in a posed some signficant issues with regards to duplicates. Using a HashSet to store location objects is heavily preferred to enable quick lookup time for phrase queries, but the hashCode definition for non-standard instances checks for equality using memory address. Thus, I thought to override it by numbering my instances and creating a unique using the position number and page number. After some research, I found that the Cantor Pairing Function does this, but it scales very quickly.

$$\pi(k_1, k_2) = (k_1 + k_2)(k_1 + k_2 + 1) + k_2$$

Thus, because the must stay within the bounds of an int, the sum of number of pages and length of each page must stay on the order of 10^4. This was simply infeasible based on the web sizes that needed to be indexed by my program.

3.3.2 Full-Text Storage Final Decision

After the initial brainstorming, decided to stucture my index as an inverted index that essentially maps a given word to all the pages it can be found and the positions of the word on each respective page. In Java, the structure can be defined as

$$\texttt{HashMap} < \texttt{String}, \texttt{HashMap} < \texttt{Page}, \texttt{HashSet} < \texttt{Integer} >>>$$

This structure solves the issues of redundant String and Page storage and does not pose any issues with respect to equality, since the structure is defined to have the same as its unique . Thus, this structure can account for webs on the order of 10^8 pages which is much more reliable for the project's purposes. Furthermore, because everything is stored as either a or , there is $O(1)$ access to all data.

3.3.3 Insertion

An important design decision in the is the abstraction of the insertion functionalities away from and into the instance. While each page is being parsed, new words must be inserted into the instance with the proper structure, thus abstracting it into the instance allows

for cleaner code and direct access.

The insert() method simply takes in the current word, , and position, and places the key/value pair into the inverted index. Special considerations must be taken for first time entries, but the logic is quite straightforward.

3.3.4 Querying

An important design decision made in the is the abstraction of query functionalities away from the and into the instance. This allows for more direct access to the inverted index and compartmentalizes the set operations in (which is discussed in section...). I modularized the following two functions inside of .

wordQuery()
This function returns all the s that contain the specified word. This can be done in $O(1)$ time by simply retrieving the associated with word and returning the keySet. The keySet represents the set of unique pages that contain word which is exactly what is required.

phraseQuery()
This function returns all the s that contain the specified phrase. Here I make use of the fact that the positions for each word are indexed, by narrowing down our return set with each new word that is part of the phrase. In other words, suppose we have the following phrase

$$"w_1 w_2 w_3"$$

The algorithm first starts with the set of pages that contain w_1. In order for the phrase to be contained within a given , it must also then be the case that the location of w_2 is 1 more than that of w_1. The algorithm then checks whether any of the still viable s has w_2 right after w_1. It is important to note, here, that all locations of w_1 and w_2 are checked everytime, in order to ensure that all combinations of sequences are accounted for. Once this is complete, the set of viable pages has either stayed the same or been cut down. The algorithm then performs the same procedure for w_3, filtering for pages where w_3 can be seen directly after w_2.

With this abstracted querying in place, the query() function in can cleanly perform set operations on the component sets from wordQuery() and phraseQuery().

3.4 Crawling Markup Handler

The class handles parsing of the html files and population of the during the crawling stage. The general strategy I implemented was to keep track of the current page and position while parsing and buld the local instance by inserting the word, page, position triple. Below, I describe the component functions in more detail.

- - Here the location variable which keeps track of the current position on the page is reset to 0.
- - Any leftover word from previously unclosed tags is added into the index, and the a tags are searched for any new s. Additionally, if the current tag is or , a boolean flag is set to indicate that this tag should be skipped.
- - Each character
- - Any leftover word from this tag is added into the index

3.5 Issues

Debugging and testing wre big parts of this project due to the volume of source code and logic heavy implementations. Some of the notable issues I came across include:

- Implicit AND wasn't working properly (some unhandled cases)
- postfix representation generation a bit wonky

3.6 Edge Cases

During my testing and debugging process, I considered a few edge cases that had the potential to negatively affect the functionality of my program.

- Last word of an Element: As a byproduct of the word definition described in section 2.2, one has to take special care to ensure the last word of a tag is inserted into the index. This is because there may be no space after it, so it may not be considered a complete word and the parsing may continue without adding it in. I handle this by inserting any leftover word in handleCloseElement().
- Unclosed Tags: Some tags such as <p> may not need to be closed. Thus, may not be called, and the words from the last tag may not be added in. I handle this by inserting any leftover word in at the onset of the next opened tag.
- Last Tag is Unclosed: If the last tag is unclosed, there is no subsequent call, and the words in that tag may be lost. I handle this by inserting any leftover word in .
- Phrases Across Tags: It is possible that words across different tags could be queried as a phrase. My interpretation of this edge case is that phrases across different tags are considered valid. Thus, there is no special care required to handle this case.
- Words with Special Characters: Most web pages will contain some form of special characters within words such as punctuation. In accordance with the word definition described in section 2.2, I handle this by ommitting any special characters from the parsed words and then inserting them into the index.

- Inconsistent Spacing on WebPage: It is possible that the webpage has inconsistent whitespace. I handle this by simply starting a new word once a non-space character is found and ending a word when a space has been found. Thus, any contiguous sequence of spaces is left unaccounted for.
- Inconsistent Spacing in Query: It is possible that the passed query has inconsistent whitespace. I handle this by replacing any multiple space sequence with a single space (using regex).
- Style and Script Tags: and tags contain text and attributes that are not relevant to a full-text search since they describe the page styling and functionality with CSS and JavaScript respectively. Thus, I handle this case by skipping over them by setting a boolean flag in .
- References and Queries in : It is possible that identical webpages have different s due to the query and reference components of the . As such, the same may be parsed repeatedly and would lead to redundant results. To handle this, I keep track of the visited paths rather than the visited s in order to ensure that only unique paths are parsed.
- Case Sensitivity: It is possible that queries, webpage text, and tags have inconsistent casing. As such, I handle this by normalizing everything to lower case to ensure case insensitive searching.
- .htm and .html: Need to do!

3.7 Karma

3.7.1 AutoCorrect

I realized that an analysis similar to that in prog2 would be quite helpful!

4 Testing

This assignment relied on proper testing to ensure that each component/module was functioning properly. In order to perform thorough testing on our implementation, I employed different forms of rigorous deterministic testing, made use of the provided test webs for frequent progress

checks, and thoroughly evaluated the crawler on my own mini test web for a controlled test environment.

Talk about general timeline for testing

4.1 Black Box Testing

Black box testing using both the ß and test web networks was an immensely effective tool to ensure that all modules were functioning correctly together at a high level. By visually confirming that my implementation was producing the correct results in queries, I gained valuable insight into whether I was moving in the right direction.

4.1.1 Black Box Testing and

Performing black box testing on and consisted mainly of passing in different combinations of arguments to my crawler and wholistically evaluating its performance. The majority of these tests were on the test webs ß and where different source files and numbers of source files were passed in as the root(s) of the network. Some specific points of evaluation and their usefulness are described below:

- Termination: One of the most important aspects I tested on included termination of the crawling. Early versions of my program struggled with this issue, so frequent execution of after fixes was a valuable component to understand whether I was moving in the right direction.
- Handling Invalid s: The starter code for terminates the program at the onset of any invalid . This was quickly recognized by an initial crawling test on ß, so I was able to implement a more reasonable exception handling procedure.
- Number of WebPages Crawled: One very valuable test included counting the number of web pages crawled on smaller test sites such as ß or my own (as described in white box testing). I added a small code snippet to to keep a count, and ensured after every execution that the network size is what was expected. This also allowed to me to ensure correctness of disconnected networks and smaller connected components within the larger network.
- Crawling Time: Once the termination issue was corrected (through a BFS style traversal), I used the time taken to crawl

the entire web as a ballpark estimation of how my program was performing. In some early solutions, the crawling time was a lot longer than expected due to an excess of data parsing and storage. This helped me realize that a different indexing method may be more ideal, and allowed me to rethink my solution early on in the process.

4.1.2 Black Box Testing and

Performing black box testing on and consisted of a gradual progression towards complicated query structures and ensuring that the correct s were returned in . Smaller sites such as ß and my own test web were most useful for this type of testing to allow for more flexibility in the type of queries that could be verified. Less specific queries with common words wouldn't be human verifiable thus, testing with larger webs such as was limited to specific and complicated queries. The general strategy and progression is noted below.

1. Basic Queries: Here, simple queries of just words or usage of the or operators were used to ensure basic functionality. A variety of words including c and operations. Some specific examples of this on the ß net are (!news & spoof), Phrase Qu

4.1.3 Problems Found During Black Box Testing

Black box testing proved to be quite useful in ensuring basic functionality of the different components. Some specific problems that were found during the black box testing process are described below.

- Inconsistent Implicit AND: As described in section 3, implicit AND queries are covered using casework in my solution. It quickly became apparent to me after some casework in and testing that many of the cases were unhandled or were incorrectly handled. For example, after investigation, I realized that !word !word was transformed to !&word & !&word instead of !word & !word. Cases such as "phrase" (word | word) and word "phrase" were simply unhandled because I failed to exhaustively recognize all cases in my initial solution.

- Infinite Crawling Over Network: This was a significant issue in my earliest solutions that was immediately realized during my first test execution. Although I implemented a breadth first search to account for an infinite traversal, I found the issue to be in my storage and tracking of the previously visited pages. I was using HashSet<Page> which tested for

"equality" based on the hashCode of the object. Since I had not overridden the hashCode() function to implement my unique definition of equality, Java defaulted to comparisons based on the memory address which resulted in repeat links being added in. Although I eventually shifted to storing visited paths as HashSet<String>, black box testing enabled me to quickly recognize the issue and fix it for the time being.

- Most points of evaluation in 4.1.1 and 4.1.2 required some form of revision. One notable one was a bug in that was found due to some basic queries being parsed correctly but still returning the incorrect set of pages (it was found via tracing that the parsing was correct). I could thus narrow the problem down to extracting the correct information from my index and correctly performing the union/intersection operations. The bug had to do with a simple indexing issue and was fixed accordingly.

4.2 White Box Testing

White box testing allowed me to be much more thorough in my testing and examine cases that were difficult to create otherwise. Using JUnit, I implemented my own unit tests to individually test different functionalities. I will outline my process for testing those functionalities and some more specific cases here.