Web Crawler Specifications

By Kevin Pick

Contents

Overview

The assignment is to create a program capable of crawling the web in an efficient manner. The programmer must be concerned with his data structure of choice as well as the manner in which pages are parsed. To aid in parsing the pages, a library called attoparser is given to the programmer.

Goals

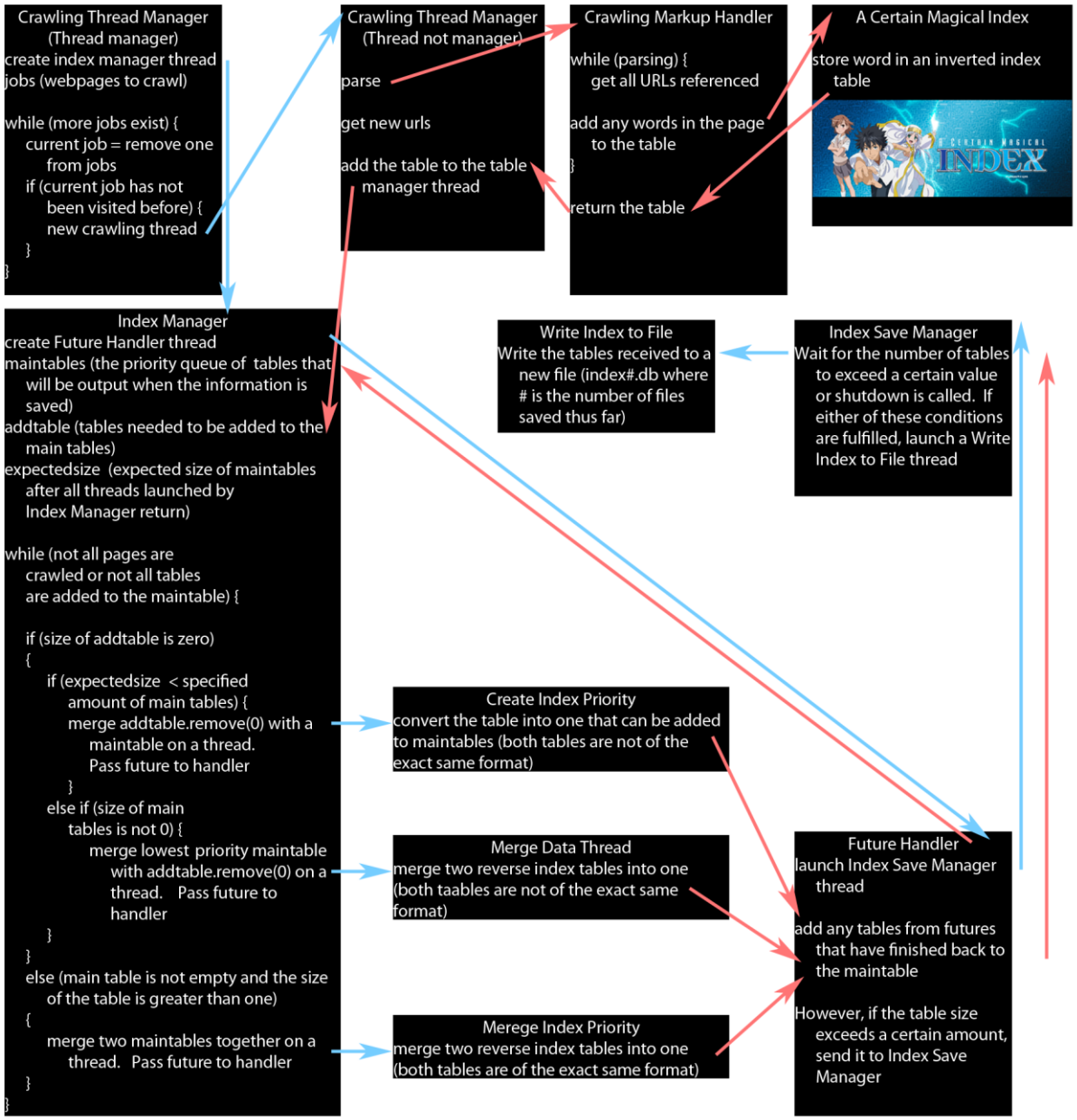
Below are the following goals I had for Web Crawler:

- Utilize multithreading to speed up the process of crawling the webpages.
- Speed up saving and loading from Java's Serializable.
- Make a program that is easily scalable and intended for large inputs.
- Consider needs of the web index apart from the assignment.
- Try to create an original data structure rather than relying on ones presented in class or taken from previous assignments (ex. Trie, HashMap, etc.).

Flow Chart of the Program

Here is the flow chart of the entire program which was completed prior to creating the program:

Blue arrows signify that a thread is launched and red arrows signify a method from an instance of an object is called. Start looking at the chart from *Crawling Thread Manager* as all other threads are spawned as a result of this class' calls.



Pair

This class pairs two objects together. The class contains *equals*, *compareTo*, and *hashCode* overrides to permit various uses of the class. The *hashCode* override uses exclusive or on the hash codes of the first and second elements in the pair to preserve the equal distribution of hash codes.

Command Line Arguments

BTreeHashSet

A *BTreeHashSet* was created for large websites so that visited URLs could be offloaded to the hard drive to save on runtime memory. Elements passed into the add method are added to a *HashMap* that keeps track of the frequency each element has been used. Additionally, there exists a *PriorityQueue* from frequency to URL to log the most frequently accessed URLs. Once the size of the *HashMap* equals to *MAX_HASH_SET_SIZE*, when add is called on a new value, the lowest priority element in the *HashMap* is sent to a *BTree* and the new value is added to the *HashMap* with an initial frequency of *INIT_FREQ* to try to give the URL a chance to grow its frequency before it is removed. This behavior is disabled by default and, instead, the default behavior is to make the *HashMap* run in the same manner as a *HashSet*. To enable *BTreeHashSet*, type "--BTreeHashSet" as an argument in the command line input. To specify the maximum size of the *HashMap* and *PriorityQueue*, type "--BTreeMaxHash #" (replace # with the maximum size number).

File Reader

Three types of File Readers exist for passing information to the parser. The first file reader (the default) uses an *openStream*. The second file reader is faster than an *InputStreamReader* and is called a *BufferedReader*. While the second method does a decent job at loading files, it is slower than allocating a *ByteBuffer* and reading the file in segments. Because of this, I created a third method, *CustomFileReader*, that reads a *CHUNK* at a time and splits up requests into multiple reads. The concept is that reading 4000 bytes at a time, as stated in lecture, is faster to do than reading 1000 bytes four times since data is read in 4000 byte chunks by the hard drive. After loading the data, the data is converted and stored in a character array for attoparser to use. While working on this part of the assignment, I realized that attoparser requests approximately 4000 bytes per read, making its request and received value similar to what the *CustomFileReader* enforces. To change the file reader, type "--FileReader#" where # is replaced with the number of the file reading method.

Thread Management

To set the maximum number of threads for the *CrawlerThreadManager*, type "--MaxThreadManager #" (replace # with the maximum size number). To set the maximum number of threads for *IndexManger*, type "--MaxThreadIndex #" (replace # with the maximum size number).

Jobs Per File

To specify the number of tables considered to be one job, type "--TablesPerJob #" (replace # with the maximum size number). To specify the number of *IndexPriorities* saved per file, type "--JobsPerFile #" (replace # with the maximum size number).

Disclaimer Regarding Below Sections

All assumptions about what constitutes a word, punctuation, etc. are listed in the section of the code that makes the assumption.

CrawlerThreadManager (Manager)

This class was created to both manage and run threads to monitor the crawling process. The method is called from the *WebCrawler* class to act as a thread manager. Note that the URL passed in is assumed to be a URL pointing to a position on the hard drive and not the internet. If it is not, the program will parse the input URLs but not attempt to look at URLs they link to. The constructor, thus, sets the current URL being looked at to null. *WebCrawler* sets the initial set of URLs to crawl by calling *startingJobs()*. To start the crawler, the crawlerThreadManager() is invoked. This method launches an *IndexManager* thread which has the main purpose of dealing with the storage and combining of inverted index tables. It is described in further detail in the *IndexManager* section. The method maintains a buffer of URLs to be looked at called jobs. All URLs in this buffer are not necessarily new URLs, but are URLs to files that exist and were found on a webpage. This is done to avoid using a *ConcurrentHashmap* for the worker threads to access. Jobs is a *Bounded Buffer* to minimize locking due to multiple threads attempting to add new URLs at once. *Bounded Buffer* acts strictly according to its definition with the exception of the *addAll* method. The *addAll* method breaks the job requested into multiple chunks so that a single thread does not request too many permits from a semaphore at once. This helps to speed up the threads as a whole as blocking for writing is chopped up so that threads with smaller jobs terminate earlier than those with larger ones. All of the *crawlingThreadManager (Worker Threads)* are grouped into a *Thread Pool* to reuse terminated threads since the lifetime of a thread is very short. The *crawlerThreadManager* method loops until all threads terminate and the number of elements in the *jobs* buffer is zero. At this time, the *Thread Pool* is also terminated. If the *jobs* buffer is not zero, a new job is created if either the number of jobs currently running is less than a constant *MAX_THREADS* or if there is gridlock in writing to the jobs *BoundingBuffer*. To execute a job, a job is removed from *jobs* and checked against a *BTreeHashSet*of all URLs previously looked at. If it is not contained in that set, a new *crawlingThreadManager (Worker Threads)* is created and sent the task of fulfilling that job via its constructor. The *BTreeHashSet* is then updated to account for the new job. An *AtomicInteger* counter storing the number of jobs running is incremented by one. Although *Java's Thread Pool* keeps track of the number of threads running, it only gives an approximation. By manually keeping track of the number of threads running, the program has an exact amount rather than an approximation. *Atomics* are used so that multiple threads accessing this variable at once does not create race conditions or corrupt the variable's data. After the *Thread Pool* is terminated, the method signals for the *IndexManager* thread to shut down and waits for it to finish.

CrawlerThreadManager (Worker Threads)

When this worker thread receives a job, it passes the name of the job (a URL) to a *CrawlingMarkupHandler* in order to start parsing the page. If the parsing of the page throws an error, the number of jobs running is decremented by one to signal the fact that the thread has ended. After parsing the page, the list of URLs found on the page is converted to an array and added via *addAll* to the jobs *BoundingBuffer* of *CrawlingThreadManager (Manager)*. The index created by crawling the page, *ACertainMagicalIndex*, is retrieved and combined with the newly created *ArrayList* in a *Pair* to be added

to the *combineData BoundingBuffer* (another jobs buffer) for *IndexManager*. At the end of the threads life, it decrements the AtomicInteger of the number of threads running to signify that the job was completed.

CrawlingMarkupHandler

The following characters are considered to be punctuation:

\\	/	“	\n
\r	\t	,	:
[	]	.	;
!	?	(	)
{	}	<	>
%			

Hyphens are not considered punctuation as words can be hyphenated. Regex is not used for parsing since Regex is slow and makes several unnecessary passes through words. It is assumed for the below methods attoparser works as described in its documentation.

storeURL

Removes anchor tags and queries from the website passed into it by iterating through the website from left to right and stopping when a ‘#’, ‘?’, or the end of the website string is reached. Each character found is added to a *newWebsite* string so that only one pass needs to be made through the string. The new URL is then constructed by using the URL relative constructer to the current URL being looked at. The new URL is checked to see if it is a valid file on the hard drive and, lastly, adds it to the list of all URLs found on the page that exist.

searchTag

Attempts to store a tag attribute, known as variable *type*, using *storeURL* or using *ACertainMagicalIndex’s addRefPhrase* depending on whether or not the *URL boolean* is set true. The former method interprets the attribute’s value as a URL if the *boolean* is true and the latter as text within a tag if it is false.

handleOpenElement

This method is called when a tag opens. The method stores any remaining characters that were not processed from a previous *handleText* as a word since a tag is assumed to signify the end of a word. First the *attributes Map* is checked to see if it is null. If it is, the elementName is checked to see if it is equal to either “script,” “style,” “SCRIPT,” or “STYLE.” If it does, ignoreText is set to true as any text found before these tags close is assumed to be text describing these tags and not text displayed on the screen. If attributes is not null, element name is checked against “a,” “A,” “iframe,” and “IFRAME.” If element name is equal to any of these, *searchTag* is called twice with a *URL boolean* of true and a *type* of “href” and “HREF” if it is equal to “a” or “A.” If it is equal to “iframe” or “IFRAME”, “src” and “SRC” are the *types*. The method does not support any other capitalizations or combinations of letters to denote href and a tags.

handleStandaloneElement

This method stores any remaining characters that were not processed from a previous *handleText* as a word since a tag is assumed to signify the end of a word. The method returns if attributes is null. If it is not, *elementName* is checked to be either "img" or "IMG." If it is, *searchTag* is called twice with a *URL boolean* of false and types "title" and "TITLE." The method does not support any other capitalizations or combinations of letters to denote img tags.

handleCloseElement

elementName is checked to equal "script," "style," "SCRIPT," or "STYLE." If it does, *ignoreText* is set to false so that text will be recorded again since these tags are closed.

handleText

If *ignoreText* is true, the method immediately returns and no text is recorded. The block of text passed into the method by *attoparser* is iterated over character by character from left to right from *start* to *start + length*. If the current character is not '&' (for escape characters), not punctuation, or not a space, it is converted to lower case and added to a string. This string is stored as a global so that information is not lost or misinterpreted as a word so that *handleText* will behave correctly when it is called back to back (this is possible because attoparse parses text at approximately 4000 bytes at a time meaning that *handleText* may not include the entire text block. I discovered this when reading through the attoparser source code). If the current character is a '&,' punctuation, or a space, the string is considered to be a word. *addWord* is then called on *ACertainMagicalIndex* and the string is cleared. If the current character is a '&', the string is set equal to it to account for escape characters. Thus, this code assumes that a word is any consecutive sequence of characters not including punctuation or a space. Escape characters are considered to be a word as well and are the only exception to the above definition.

handleDocumentEnd

Adds any lingering words to *ACertainMagicalIndex* via its *addWord*. The method then calls its *finishPorcessing* method to add in any text collected from the "title" attribute of "img."

ACertainMagicalIndex

This class assembles an inverted index table using a *TreeMap* to sort the elements by string value. An inverted index stores a mapping from the content, which in this case is the word, to a data set describing the location of the data. The data structure for the *TreeMap* is shown below:

TreeMap<String, Pair<Pair<Integer, Integer>, ArrayList<IndexTable>>>

The key of the *TreeMap* represents the word in the table.

The value of the *TreeMap* stores the position and URL data. *Pair<Integer, Integer>* acts as the header information of the *ArrayList<IndexTable>* block. Since *ACertainMagicalIndex* only stores the inverted index of one URL, the header information, *Pair<Integer, Integer>*, is set to Pair<0, 0>. The first element in the pair represents the number of the URL in the table which is further explained in *IndexManager*. For the purpose of this class, the number is 0 since only one URL is in the table. The second element represents the location of the start of the data representing the *IndexTable's* of the page. Since only one page is stored in the *ArrayList*, the number is also 0. As will be explained in the *IndexManager* class, the header is important when there are multiple pages stored in the *IndexTable* array. The *IndexTable*

object only stores the position of the word on the page and is sorted from least to greatest based on the position value.

addWord

It is assumed that the words entered into the page are entered in sorted order based on position. The *CrawlingMarkupHandler* assures this in this specific implementation as attoparser reads text in sequential chunks. A word and a position are passed in. If the word is an escape character, it is converted to its appropriate character as defined by the University of Texas at Austin: https://www.utexas.edu/learn/html/spchar.html. If the *String* does not exist in the table, it is added with the header set to *Pair<0, 0>*. The word is added to the appropriate *String* location in the table and is appended to the *ArrayList* of *IndexTables*. The *lastPosition* stored is updated for the *finishProcessing* method.

addRefPhrase

Add the phrase to an *ArrayList* for post processing in *finishProcessing*. Phrases that are entered into this method are from tags and exist outside the text on the page as far as locations are concerned.

finishProcessing

Processes the phrases stored by *addRefPhrase* by looping through each phrase and setting the starting position of the phrase to two more than the last position stored in the table. This way these phrases cannot be considered as a part of a phrase search for words not in the phrase as these words are not directly next to any other words. The phrase is parsed by the same punctuation, space, and escape character rules as *handleText*. The words are added to the table via *addWord* like *handleText* as well.

Big O Runtime

Since a *TreeMap* is used to store the data, the worst case scenario would occur when new nodes need to be added for each word in the data structure. Thus, Big O is the same as inserting n elements into a Red-Black tree:

$$O(n \log n)$$

Big O Space Consumption

The worst case scenario occurs when the number of words inserted are equal to the number of nodes in the *TreeMap*. Thus, the amount of data reflected should be the number of words stored multiplied by two since both words and positions are stored. Thus, Big O is:

$$O(n)$$

IndexManager

This class manages the inverted index tables that will eventually be saved. These tables are slightly different than those found in *ACertainMagicalIndex* since they can support multiple webpages per table due to the fact that the header (*Pair<Integer, Integer>*) is an *ArrayList<Pair<Integer, Integer>*. This new type of inverted index table is known as *IndexPriority* and also has a *HashMap* of *Integers* to *URLs* which has a key referring to the number of URLs in the table prior to the addition of this URL. This way, each key has a unique ID in the table. *IndexPriority* also maintains a count of the number of bytes it will take to save the table. This ensures that all URLs in the table have a unique mapping to an Integer and

permits data to be compressed. Instead of storing an entire URL in the header, an *Integer* is only needed to represent it. The value pair for the URL, thus, refers to the starting index in the positions *ArrayList* of the position data that describes this URL. The data block is thus segmented by sorted sections of positions. When a thread of *IndexManager* is spawned, a thread of the class *FutureHandler* is also spawned. *FutureHandler* monitors when threads finish and adds futures into the table or saves them. For a more in depth explanation of the class, please visit the *FutureHandler* section.

How the Method Operates

A *Thread Pool* monitors all threads spawned to perform operations on a set of inverted index tables. The method maintains at most *STARTING_CAP* number of reverse index tables. The maximum number of reverse index tables allowed to exist in the *Priority Queue* signified by *segmentCap*. *tableSize* is an *AtomicInteger* that measures the projected number of tables that will be in the *PriorityQueue* after all threads currently running terminate. The jobs the manager must delegated are defined as a *BoundedBuffer<Pair<URL, ArrayList<Pair<String, Pair<Pair<Integer, Integer>, ArrayList<IndexTable>>>>>>* called *combineData* which is the *URL* of the data from *ACertainMagicalIndex* with the *TreeMap* converted to an *ArrayList* by *CrawlerThreadManager*. The converted data is stored in *WebIndexPriorityQueue webIndex* which is a *Priority Queue* of *IndexPriority* objects. The *Priority Queue* is sorted by the number of words in the list so that smaller tables will have data added to them to keep the size of all of the tables approximately equal. The manner in which data is added to the *Priority Queue* will be discussed in the *FutureHandler* section. If the number of jobs currently being executed is less than *MAX_THREADS* or *combineData* is not overloaded with tasks (meaning that it does not have any permits to give out), *webIndex* will be modified. This is done to force the manager to wait for threads to finish instead of spawning new ones so that the computer is not overloaded. If *combineData* has jobs, it will delegate them. If *tableSize* is less than *segmentCap*, an element of *combineData* is converted into an *IndexPriority* by launching a thread called *CreateIndexPriorityThread*. The thread returns an *IndexPriority* and is added to the *ArrayList* of *Futures* in *FutureHandler*. Otherwise, if the size of the *PriorityQueue* is not zero, the lowest priority element is polled from the queue and combined with an element from *combineData* by launching a thread called *MergeDataThread* and adding its output, an *IndexPriority*, to the *ArrayList* of *Futures* in *FutureHandler*. If *combineData* does not have a job and the size of the Priority Queue is greater than one, the two lowest elements in the queue are polled and combined via the *MergeIndexPriorityThread*. The output from the thread is an *IndexPriority* which is added to the *ArrayList* of *Futures* in *FutureHandler*. If none of the requirements for adding or merging data are meant, the loop continues until it is signaled to shutdown, meaning that web page parsing is complete. When the signal to shutdown is sent to the method, *segmentCap* is set to the *ENDING_CAP* and tables in the *Priority Queue* are combined until *tableSize* equals the cap. After the cap is met, the thread signals for *FutureHandler* to shutdown and waits until it does so before ending.

How Merging Works

Since both tables are sorted by keys, merging the tables is similar to that of Merge Sort. Two integers are set to zero. The keys corresponding to these integers in the table are compared. The lesser of the two is added to the bottom of the new table. If they are equal, both are combined and added to the bottom of the new table. If the integer representing the key position is greater than the length of the tree, the rest of the other table is added to the bottom of the new table. A new table is always created

during merging so that, if the algorithm were used in practice, users accessing tables during the merge process would not lock the data from merging.

Advantages of this Method

This class makes smart decisions regarding how data should be managed. Since it is faster to convert data and not combine blocks, the class prefers to convert data and add it to the *Priority Queue* immediately if the size of the *Queue* is less than a threshold. Once the threshold is exceeded, data will be combined. While the thread is not being given jobs, it will start to combine tables from the Priority Queue to compress that data and ultimately allow for faster searches. The *ENDING_CAP* allows for the thread to strictly focus on combining *Priority Queue* data prior to saving when the cost of combining is greatly outweighed by the cost of writing a larger amount of data to the disk. Thus, this class helps to balance performance and compression in a strategic manner to create an output that can be saved to disk.

Big O

Since all of the threads launched combine two index tables, the time complexity depends on the key, value pairs in both tables. Let m represents the number of keys in the first table, n represents the sum of the number of URL Integers and block headings for each row, and k represents the sum of number of position data for each row. Let $_0$ denote the first table and $_1$ denote the second table. In all scenarios, the number of operations to merge two lists into a new data set is equal to the summation of all elements in both lists. Therefore, to merge the tables, Big O is:

$$O((m_0 + n_0 + k_0) + (m_1 + n_1 + k_1))$$

FutureHandler

A thread of *IndexSaveManager* is launched on the creation of this class. The class is intended to monitor the *Futures* sent over to it from *IndexManager* and decide whether or not to re-add them to the *Priority Queue* in *IndexManager* or send them to the *IndexSaveManager*. The method repeatedly loops through all *Futures* in its *ArrayList* and checks whether or not the method the *Future* belongs to has finished. If it has, it looks at whether or not the number of words in the table is less than *MAX_DATA_SIZE*. If it is, the *IndexPriority* returned by the Future is re-added to the *Priority Queue*. If it is not, it is added to the jobs list of *IndexSaveManager* and the predicted size of the table (*tableSize*) in *IndexManager* is decremented by one since it was incorrect. When the class is signaled to shutdown, it passes the signal on to *IndexSaveManger* and waits for that class to shutdown.

Advantages of this Method

The *FutureHandler* allows for *IndexManager* to focus specifically on allocating jobs and frees it up from bottlenecks caused by checking if *Futures* have finished. Additionally, this class offloads data to be saved in order to allow for large pages to not overload the runtime memory of the program.

IndexSaveManager

This method receives jobs in the form of *IndexPriority Objects*. The jobs are stored in a *BoundedBuffer<IndexPriority>* called jobs. Once jobs exceeds *JOBS_PER_FILE*, the first *JOBS_PER_FILE* IndexPriorities from the *BoundedBuffer* are sent a new thread that is created called *WriteIndextoFile*. Once the *IndexSaveManager* is signaled to shutdown, it sends all remaining jobs to a new

WriteIndextoFIle thread and waits for all *WriteIndextoFile* threads to execute. These threads are managed in a *Thread Pool*.

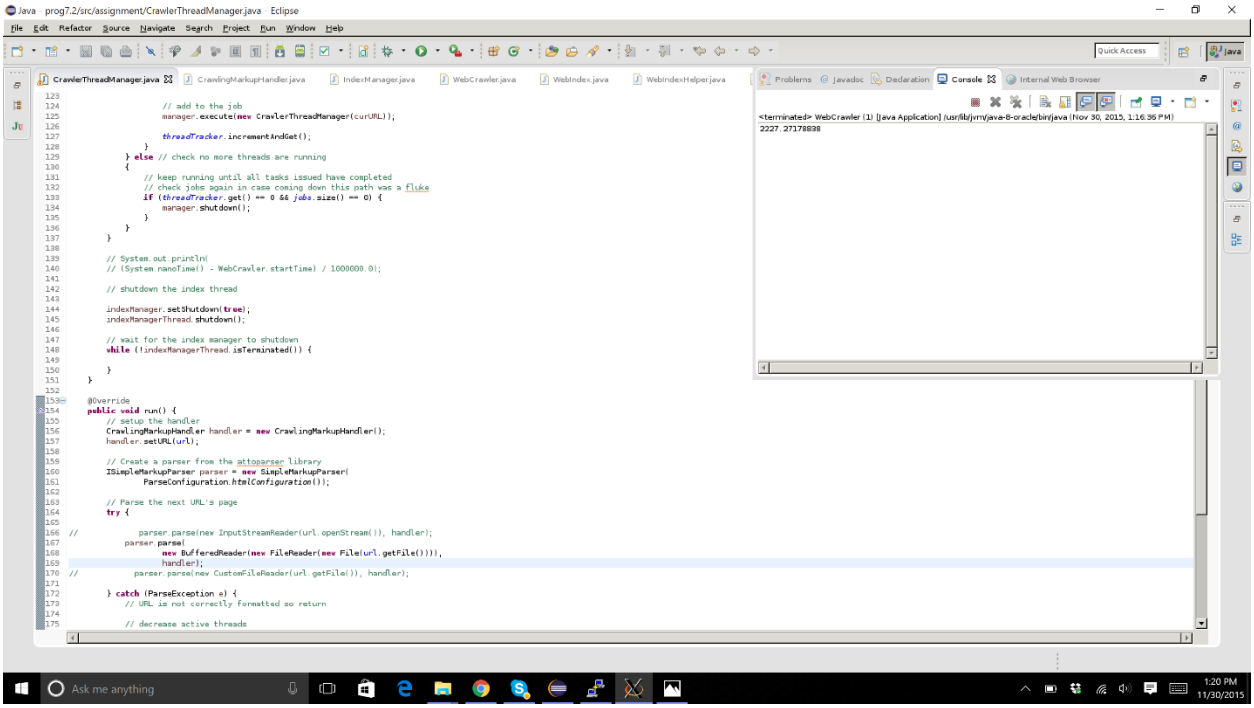
WriteIndextoFile

Stores all *IndexPriorities* sent to the class in a *WebIndex*. *WebIndex* save method is then called with a unique number that is equal to the number files saved before it to create "index#.db" where # is the number.

WebIndex

The save and load methods are optimized to run as fast as possible and are substantial improvements on *Serializable*. Originally, I looked into using *FileWriter* to create the output but, after looking at the class, I realized that this *Object* made several unnecessary copies of the data. Therefore, the speed of saving large files would be compromised. I also attempted to map the *ByteBuffer* to a *FileChannel*. Although this method is the fastest Java offers, I realized there is a bug with this kind of mapping in Java. Currently there is no way to unmap the *ByteBuffer*, causing for the file to be locked until the *ByteBuffer* is deleted by the *Garbage Collector*. Thus, the data will exist indefinitely as the Garbage Collector cannot be controlled by the user without using *Unsafe*. Thus, I chose the next best option which was to use *ByteBuffer.allocate* to create memory that, when compiled, is accessed via *MOV* commands. To make this method even faster, the exact number of bytes needed to save the file can be allocated since *IndexPriority* stores the number of bytes needed to save a table. Thus, calculating the file size needed is quick and easy. The loader uses a *BufferedInputStream* so that each byte can be read in at a time. Methods to interpret bytes as *Strings* and integers were created to support this form of loading.

Overall Speed



The web crawler was averaged over 100 consecutive runs of rhf (9360 pages as confirmed by more than 6 classmates) on the lab machine Skipper. The average time was 2.227 seconds meaning that with all of the speedups of multithreading as well as rewriting saving methods, the web crawler is able to load, index, and save 4,203 pages per second. Thus, the methodology of the design as well as the speedups implemented are justified by the overall output.

File Size

The overall file size for *rhf* was shrunk via these changes from 87,751 KB using *Serializable* to 26,008 KB using the saving algorithm presented in the assignment. This proves that the method presented is

Data Structure Justification

The reason this data structure is advantageous to use is for the reasons listed below:

- Threads can be viewed as servers allowing for multiple servers to take on some of the processing power for crawling, storage, and saving
- The data structure is multithreading friendly as each step can be accomplished by delegating tasks
- Balances performance and storage cost to give optimal runtime
- The ability to support multiple inverted index tables allows for shorter re-indexing costs if the algorithm were to be used in the real world
    - Re-indexing a website means looking at a URL again to update the data on the URL to reflect the changes that occurred to the webpage since the time it was last visited. Instead of altering one large table, the user has the ability to have several tables to lower the cost of altering the table which is O(m) (m = number of elements in the table)

instead of O(n). The Big-O computations come from the fact that each word in the table must be looked at in order to perform re-indexing.

Queries

Note

Phrase queries are sensitive to double spaces and will return nothing if one is found within them.

Parsing

The query parser is based on the Shunting-yard algorithm to convert infix notation to postfix notation. My implementation of the algorithm was based on the set of rules it must abide by as outlined on https://en.wikipedia.org/wiki/Shunting-yard_algorithm. Thus, any query that has &, |, !, word, (, and) can be entered and parsed correctly. Precedents are taken into account so parentheses are not needed for queries. Punctuation, as defined by *CrawlerThreadManager*, is filtered out of words and & is only allowed in a word if it is an escape character in a phrase query. Implicit & between parentheses is supported as well as between words. This is done by tracking the previous character and storing whether or not it was an operator or part of a word. Phrase queries are supported by reading until a closing question mark is found when an opening one is seen. If there is an error in the query entered, the program will print an error to the console and return a new *Collection<Page>*.

Execution

Since the query is now in postfix notation, it can be interpreted as reverse-poles notation and read by traversing the output from left to right and adding elements to a Stack. When an operator is found, a certain number of elements are popped from the stack, processed, and re-added as described here https://en.wikipedia.org/wiki/Reverse_Polish_notation (was also talked about in section). Prior to processing the logic, a first pass is done through the data to get all terms that need to be searched for. This information is then used to create run a *SearchManager* thread for each search that needs to be run. This thread returns a *Future* with *ConcurrentHashMap<Pair<Integer, Pair<Integer, Integer>>, Object>>*. The *ConcurrentHashMap's* key represents the *Pair<TabeNumber, Pair<TableID, URLNumber>>*. The *TableNumber* represents which file the set of tables in which the URL is contained was found. *TableID* refers to the specific ID of the table in the file *TableNumber* represents. Lastly, *URLNumber* refers to the URL in the *HashMap* of *Integers* mapped to URLs contained within the table. This data is then used for when Reverse-Poles notation is interpreted. During the interpretation of the data, the following subcases make up the entirety of finding the output:

A & B

!A & B

A & !B

!A & !B

A | B

!A | B

A | !B

!A | !B

For A & B as well as A | B, the program loops through one set of data and compares it the existence of the other. For A & B, if an element is found in B and not in A, the element is stored and then deleted from B afterwards. The new version of B is then returned as the representation of A & B. For A | B, A is looped through and, if an element is found in it and not in B, it is added to B. The new version of B is then returned as the representation of A | B. To deal with the ! operator, the *ConcurrentHashMap<Pair<Integer, Pair<Integer, Integer>>, Object>>* is *Paired* with a *Boolean* that, when false, means the representation of the set is A and, when true, means !A. This way, the data sets do not have to be flipped by ! until the processing of the query is complete. Thus, the pass over the data to flip it is limited to once rather than every time the ! operator is found. If the ! operator is found while reading reverse-Poles, the *Boolean* of the element on the top of the *Stack* is flipped. This makes queries like !A & !B and !A | !B immensely easier. Using De Morgan's Law, these equations can be converted to !(A | B) and !(A & B). Meaning that the logic for interpreting A & B as well as A | B applies, with the exception that the output *Boolean* in the *Pair* must be true. !A & B and A & !B can be considered the same case since & is transitive (I will represent the case as !C & D). To interpret this query, C is iterated over and compared to D. If a key is in both sets, it is removed from D. D is returned as the representation of !C & D. By similar logic, !A | B and A | !B can be interpreted as !C | D. !C | D can be evaluated by iterating through D and checking for elements that are in both sets C and D. These elements are removed from D and !D is returned. Each of these problems were solved by using truth tables to evaluate each case and coming up with a method that would produce the desired truth table answer. After processing finishes, if the *Boolean* of the last term on the *Stack* is true, all URL's stored are iterated through and, if they are not represented in the *ConcurrentHashMap*, are returned as a part of the *Collection* of *Pages*. Otherwise, the URLs that the *ConcurrentHashMap* represents are found and returned.

Logic Runtime

Since each of the logical operations as outlined above loop through the dataset of one of its inputs once while calling contains on the other (contains is constant time since it's a *ConcurrentHashMap*), the runtime is:

$$O(n)$$

SearchManager

Receives a search phrase or word and sends the phrase or word to one *Search* thread per *IndexPriority* table loaded in. A *ConcurrentHashMap* is passed to each of these threads so the results are combined upon storage. The *ConcurrentHashMap* is returned as the Future of the thread.

Search

The table is traversed through a binary search. To find the first word, a binary search of the words in the table is run to find the row containing the data. When this row is found, the information regarding the position of the word on the page as well as its URL association are stored in a master set. The master set (called *possibleURLs* in the code) is of the form *ArrayList<Pair<Integer, ArrayList<Integer>>>* which means *ArrayList<Pair<URLNumber, ArrayList<Positions>>>*. Then subsequent words are searched for via binary search and the data block (URL and position data) are looked at. For each URL in the master set, the URL is looked for in the new data block via a binary search. Then the position data in the master set is looked for with the offset of i (the offset to account for the number of words after the current one we are looking at). If it is found, the position / URL data goes unchanged. If it is not, the position data is

removed from the master set. If a URL no longer contains positions in it, it is removed from the master set. After all words are looked at, the remaining master set data is converted into the form *Pair<TabeNumber, Pair<TableID, URLNumber>>* (the meaning of this form is described in the *Parsing* section) and added to the *ConcurrentHashMap*. The *ConcurrentHashMap* helps to manager writes and ensures that no two threads write to blocked sections of the *ConcurrentHashMap* at once.

Big O for Search

Assume that the algorithm is running under worst case scenario for a phrase search. The algorithm would first run a $log(n)$ search to find the first word in the phrase. n stands for the number of words in the table. After running the search, the value of the Pair, *Pair<ArrayList<Pair<URLNumber, StartofDataBlock>>, Positions>*, is found. This data is looped through and converted into the form *ArrayList<Pair<Integer, ArrayList<Integer>>>* which represents an *ArrayList<Pair<URLNumber, ArrayList<Positions>>>*. Since the values are iterated over once to copy them, $m * \frac{p}{w}$ represents the cost of this process where m is, in the worst case, the number of URLs in the table, p is, in the worst case, the average number of word position of all of the words in each URL. p is divided by w, the number of words in the query, to equally distribute the positions over each word in the query to search. Since the number of URLs is multiplied by the average number of positions per URL, $m * \frac{p}{w} = \frac{n}{w}$. For ease of calculation, since$w \leq n$, assume $O\left(\frac{n}{w}\right) = O(n)$. This is possible since $\frac{\mathrm{n}}{\mathrm{w}} \leq \mathrm{n}$ and, in Big O, using a higher bound is acceptable. Thus, $O(m * \frac{p}{w}) = O(n)$ for this Big O calculation. Since $n > logn$, $\mathrm{O(logn)}$ (the cost of the initial binary search) can be ignored. Thus, for finding one word, Big O is:

$$Eq\ 1: O(n)$$

Although that is the Big O for the first search, the subsequent searches have a different cost. Assume that after each subsequent word looked at, no elements are removed from the master set (signifying the worst case scenario). The first binary search is $\log(n)$ as it was before. This results in m (the maximum number of URLs in the table) URLs being found with p / w positions per URL. The next search must first locate the word so it is $\log(n)$. Then m binary searches are run on m *Pairs* of *Pair<URLNumber, StartofDataBlock>* to check if the word being looked at contains the URLs from the master set. Additionally, for each of the m binary searches $mlog(m)$. For each of the $mlog(m)$ binary searches, p/wbinary searches are run on a block size of $\frac{p}{w}$. This process runs for $w - 1$ times since this process describes all searches other than the first. w represents the length of the string. Thus, Big-O can be expressed as shown below:

$$Eq\ 2: O((w-1)(log(n) + m * log(m) * \left(\frac{p}{w}\right) * log\left(\frac{p}{w}\right)))$$

As discussed before, m *p = n. Thus, the equation can be rewritten as:

$$O((w-1)(log(n) + \frac{np}{w} * log(m) log\left(\frac{p}{w}\right)))$$

Now distribute w – 1.

$$O(wlog(n) - log(n) + np * log(m) log\left(\frac{p}{w}\right) - \frac{np}{w} * log(m)\ log\left(\frac{p}{w}\right))$$

Since w represents the length of the word, thus meaning it is a non-negative number, $wlog(n) > -log(n)$ and $np * log(m)log(p) > -\frac{np}{w} * log(m)log\left(\frac{p}{w}\right)$. Since it is Big O notation, $-log(n)$ and $-\frac{np}{w} * log(m)log\left(\frac{p}{w}\right)$ can be ignored:

$$O(\text{wlog}(n) + np * log(m)log\left(\frac{p}{w}\right))$$

Now that both Eq 1 and Eq 2 are simplified, they can be combined to find the overall Big O of searching.

$$O(n + wlog(n) + np * log(m)\log\left(\frac{p}{w}\right))$$

Big O Memory Usage for Search

The amount of memory needed to run a phrase query is equal to the amount of data found by the first word search since this data is copied to the master set. For more information about where this equation comes from, look at the *Big O for Search* . Thus, the Big O for memory usage is:

$$O(m * \frac{p}{w})$$

Big O for SearchManager

Since *SearchManager* is multithreaded, its Big O is the worst runtime of *Search* assuming all threads run at the same time and do not slow down the computer.

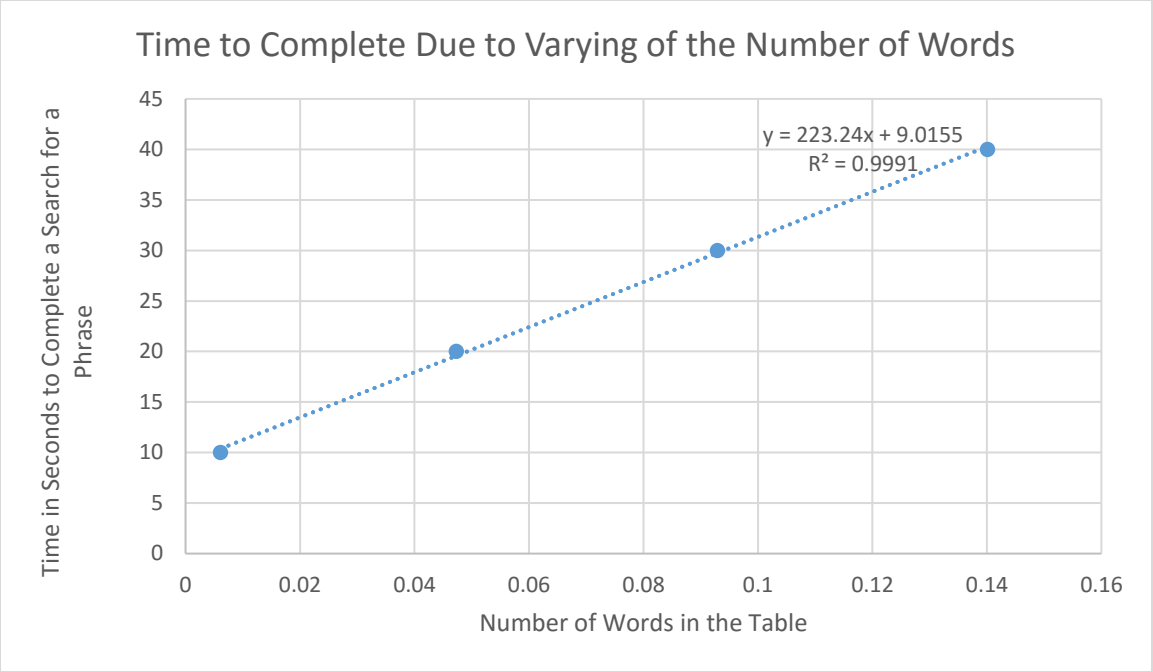
Big O for Searching in General

Since the Big O for searching is larger than the Big O for parsing a query, it will dominate the process. Assume that the n, p, and m presented below are from the thread with the largest number of operations. Thus, the Big O for the entire searching process is:

$$O(n + wlog(n) + np * log(m)\log\left(\frac{p}{w}\right))$$

Checking Big O

To verify the Big O is correct, a *JUnit* test file named *PhraseSeaerchBigOTester.java* was created along with *IndexGenerator.java*. *IndexGenerator* creates a *WebIndex* that fulfills the input properties of m and n. Since $p = \frac{n}{w}$ it can be derived from the inputs. The *WebIndex* is then passed into *Parser* along with a phrase query of length of w where the phrase is contained in the *WebIndex*. *RUNS* searches are run through *Parser* and the time for each search is averaged and output to the console for recording. This process is repeated four different times where one to two variables are changed in each iteration. By holding several variables constant, the values provided in the Big O expression can be verified by the behavior of the runtimes (runtimes are directly proportional to the number of operations performed). NOTE: please take the below tests with a grain of salt as runtime is not always an accurate representation of operations due to the amount of background processes running at a computer at any given time.



m = 5

n = {10, 20, 30, 40}

w = 1

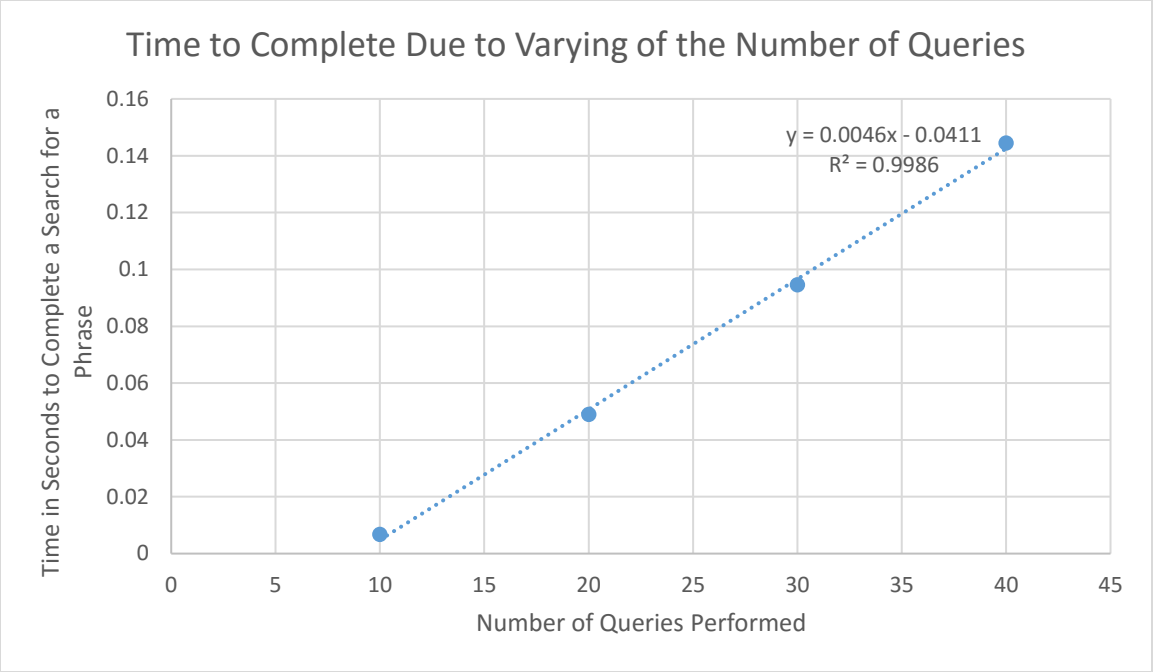
p = n / 5

The value expected for the graph of n v. runtime is linear because, if all other variables are assumed to be constants, the Big O for search would be $O(n + \log(n) + n)$. Since n grows faster than $\log(n)$, it is $O(n)$.

Small values are chosen for each variable in order to keep the influence on the graph from p to a minimum so n's performance can be monitored.

Although p is not kept constant, n dominates it by so much that it does not largely effect timing.

Since the R^2 value is above .8, it proves that n is linear as was stated in the Big O equation.



m = 10

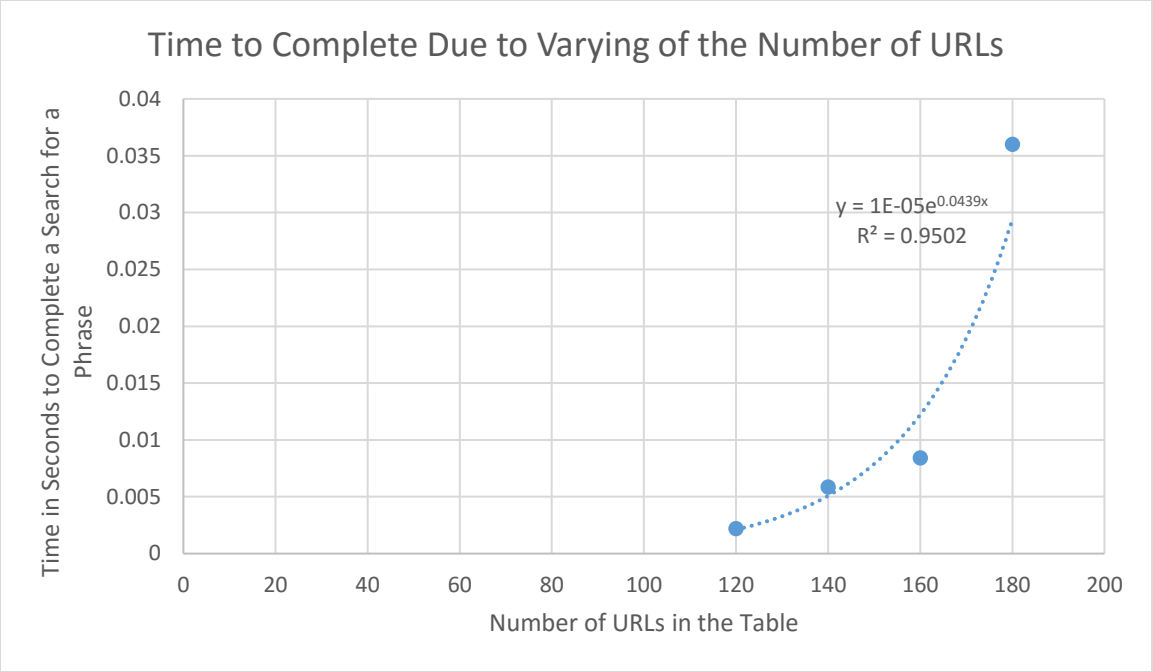
n = 200

w = {10, 20, 30, 40}

p = 20

The graph expected for w v. runtime is linear because, if all other variables are assumed to be constants, the Big O for search would be $O\left(0 + w + \log\left(\frac{1}{w}\right)\right)$. w grows faster than $\log(\frac{1}{w})$ and, thus, dominates Big O making it $O(w)$.

Since the R^2 value is above .8, it proves that w is linear as was stated in the Big O equation.



M = {120, 140, 160, 180}

n = 200

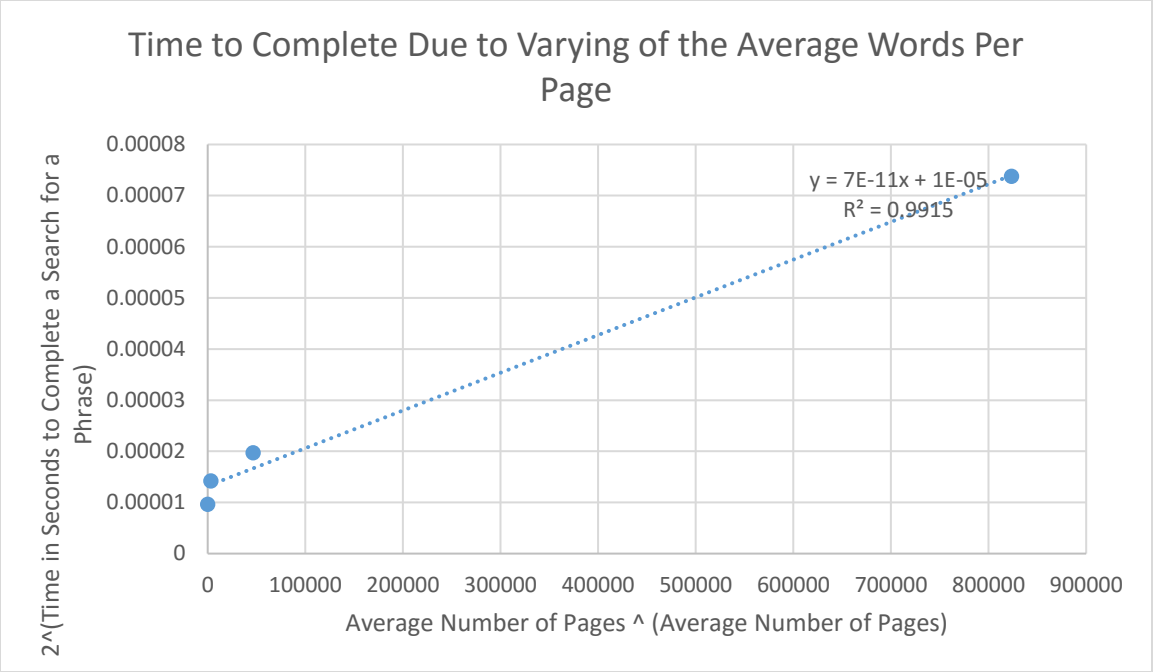
w = 1

p = 200 / m

The graph of m v. runtime is expected to be exponential because, if all other variables are assumed to be constants, the Big O for search would be $O(0 + 0 + \log(m)) = O(\log(m))$. Since, $O(\log(m)) = number\ of\ operations\ perfomred, m = (base\ of\ log)\text{^}(number\ of\ operations\ performed)$

Values close to n are chosen for m in order to keep the influence on the graph from p to a minimum so n's performance can be monitored.

Since the R^2 value is above .8, it proves that m is contained within a logarithm as was stated in the Big O equation.



m = 40

n = {80, 200, 240, 280} each index of this array maps the average words per page being used

w = 1

p = {2, 5, 6, 7}

The graph of of p^p v. $2^{runtime}$ is expected to be linear because, if all other variables are assumed to be constants, the Big O for search would be $O(0 + 0 + \text{plog}(p)) = O(p\text{log}(p))$. Thus, $plog(p) = number\ of\ operations$. This equation can also be written as $\log(p^p) = number\ of\ operations$. Raise both sides to the base of the logarithm (2):

$$p^p = 2^{number\ of\ operations}$$

Thus, the graph of p^p v. $2^{number\ of\ operations}$ should be linear.

Since p varies based on the value of n, n is altered. Small changes in n are chosen to keep the effect of n on Big O to a minimum so that it can be evaluated as a constant. Additionally, n's contribution to Big O is much smaller than that of p for the values chosen to help nullify n's effect on Big O.

Since the R^2 value is above .8, it proves that $p\log(p) = number\ of\ operations$ as was stated in the Big O equation.

Testing Note

The results of query results as well as the number of URLs found was compared with other students to check if my results were equal to that of others.

Black Box Testing

Black Box Testing helped to specifically check certain aspects of the parser behaved as intended. Edge cases for implicit ands as well as operator presidencies were checked this way.

Shunting-Yard Test

The below tests are intended to verify that the Shunting-Yard algorithm is correctly implemented. Each test was to convert a query to reverse-Poles notation and check it against what is expected.

andTest

Intended to check simple & operations work.

orTest

Intended to check simple | operations work.

notTest

Intended to check simple ! operations work.

parenthasesTest

Intended to check simple (and) operations work.

phraseTest

Intended to check simple phrase queries are parsed correctly.

implicitAndTest1

Checks that implicit and works when two words are only separated by a space.

implicitAndTest2

Checks that implicit and works between parentheses.

complexParseTest1

Checks that multiple ! operators together are parsed correctly. Parentheses and & operators are also present in this query to test how operators work together.

complexParseTest2

Checks that ! operator works with phrases. Parentheses, |, and & operators are also present in this query to test how operators work together.

complexParseTest3

Checks that implicit and works between two phrases as well as a closing parenthesis and a phrase. Parentheses, |, and & operators are also present in this query to test how operators work together.

complexParseTest4

Checks that implicit and works between a closing and opening parentheses. Parentheses, |, &, and implicit and operators are also present in this query to test how operators work together.

Single Thread Test

Checks that the results returned by a single threaded web crawler on superspoof are the same as the multi-threaded version. This is to make certain that data is not accidentally lost due to corruption or incorrect write permissions with threads.

White Box Testing

For this assignment, white box testing was very useful to generate randomized webpages to crawl. The below tests were used to check the crawler correctly assembles the index and finds all desired URLs for queries.

Test Div

Creates 100 random webpages to crawl that are all linked to each other by hrefs and iframes. The webpages contain randomized text and also have images with randomized titles. The web crawler is then run and the saved file is loaded into memory. The web pages generated have the following properties so that queries can have expected outputs:

If the name of the page is divisible by 3: “divthree” is on the page.

If the name of the page is not divisible by 3: “notdiv” is on the page.

If the name of the page is divisible by 2: “divtwo” is on the page.

If the name of the page is divisible by 7: “le phrase query” is on the page.

If the name of the page is not divisible by 7: “nope query please” is on the page.

allPagesTest

Search the query “!Disney” which should return all pages on the website.

div3Test

Search the query “divthree” which should return all pages divisible by 3.

notDivTest

Search the query “notdiv” which should return all pages not divisible by 3.

notNotDivTest

Search the query”!notdiv” which should return all pages divisible by 3.

notDiv3Test

Search the query “!divthree” which should return all pages not divisible by 3.

div2And3Test

Search the query “divthree & divtwo” which should return all pages divisible by 3 and 2.

div2Or3Test

Search the query “divthree | divtwo” which should return all pages divisible by 3 or 2.

notDiv2And3Test

Search the query “!divthree & divtwo” which should return all pages not divisible by 3 and divisible by 2.

div2AndNot3Test

Search the query “divthree & !divtwo” which should return all pages divisible by 3 and not divisible by 2.

notDiv2Or3Test

Search the query “!divthree | divtwo” which should return all pages not divisible by 3 or divisible by 2.

div2OrNot3Test

Search the query “divthree | !divtwo” which should return all pages divisible by 3 and not divisible by 2.

notDiv2AndNot3Test

Search the query “!divthree & !divtwo” which should return all pages not divisible by 3 and 2.

notDiv2OrNot3Test

Search the query “!divthree | !divtwo” which should return all pages not divisible by 3 or 2.

phraseTest

Search the query “\”le phrase query\”” which should return all pages divisible by 7.

complexQuery1

Search the query "\"nope query please\" | ((!divthree | divtwo) & notdiv2)" to test the program works for parentheses with multiple elements and operators.

complexQuery2

Search the query "!(\"nope query please\" & !Josh notdiv | divtwo \"nope query please\")" to test the program works for implicit and with multiple elements and operators.

TestURLGen

Creates 10000 random webpages to crawl that are all linked to each other by hrefs and iframes. The webpages contain randomized text and also have images with randomized titles. The words are stored so they can be compared against the Index created by the web crawler. The test also checks that the web crawler is able to handle very large inputs with large blocks of text.