

Matt Wilson & Chau Nguyen
CS315H
12/6/06

Note: The applet doesn't like us and I really don't know why. Results can be achieved through the command line, by running the main method in the WebQueryEngineApplet class.

Assignment 7 – “Google”

This assignment required the creation of a Web Crawler program to generate a database that could be searched with another program of our own, a Search Engine.

What we did:

We were charged with the creation of the underlying data structure of a search engine database, as well as operating the searches on it.

Prefix Tree/URL's

We chose to use a prefix Tree data structure to compress the words found on each page; for a lexicographical search, it's quite fast. To store the list of URL's we chose to use a simple linked list. Every URL on the list has a single prefix Tree associated with it storing the words on that page. However, the URL's and Trees are not strictly associated; they are simply two concurrent Lists. The List can be seen as the “main” structure, with the prefix trees as “sub-structures,” since the List is what is operated on at the highest level of the search. We discussed using a prefix tree as the main structure, and Lists as the sub-structure, but ultimately the overarching List won out.

Using the Trees

The Prefix Trees are simply constructed. Each complete word in the Tree contains a marker designating it as such. The *add* and *search* methods are recursive, but only because they perform identical operations on each level of the Tree. Iterative recursions over one level are not necessary.

Storage is the most interesting part of the Tree. The characters position in the Tree arrays is determined by their hash code. This is just their Unicode values. A back-hash returns a character given a certain number, in order to print the Tree. Punctuation is removed from the incoming strings for both searches and storage; also every string is stored and searched for in lowercase. This has resulted in some empty space in the array, but I'm ok with that; it *can* be expanded to handle more cases if needed.

Parsing

Parsing web pages is fairly easy, in respect to the code. The two biggest concerns are URL's in the document, and the text. Since both of these have specific tags that are already handled by an external HTML parser, only minimal operations need to be performed. Text is appended to a String Buffer, which holds a huge string of all the text

in a page. This is a useful system, as it is easy to get locations for the individual words. To parse the hyperlink tags required a little more thought; luckily there are a few important identifiers that designate a link, like “href=” and “file=”. The case of relative URL’s had to be caught as well; this used a helper method to “eat” the directories of a URL away, as designated by the “..”s present in the relative link. To do this, we used a really funky while loop that performs a naturally recursive function iteratively. When URL’s are found, they are added to the main List that functions as a work queue in order to be parsed later.

Parsing queries is surprisingly more difficult. We did use a parse tree to store and easily navigate the query, but it is not generated as suggested in the project guidelines. First, the string is “repaired”. That is, certain anomalies in the query are removed, like extra spaces, etc. This step is very important, due to the next step of parsing. The string is then pushed into a stack structure, which is then used to place the string elements into a tree in a postfix order. This replaces the recursive descent parser. Returning to the topic of string repairs, the characters order and presence in the string is vitally important to the stack operations; because our code lacks a recursive descent parser, without modification it is ill equipped to deal directly with malformed queries. The initial operations on the string insert an AND operator when it is implicit, proper parentheses where needed, and also proper negations including the use of DeMorgan’s principle. Though it is not a recursive descent parser, it is much more robust; it actually surprised us; inadvertently, after writing the code for implicit AND operators, we discovered that we could use operators without parentheses. This is because a repair operation necessary for the implicit AND adds them in whenever needed! We did test this of course, and it really is one of the rare cases where code does more than you intended, and correctly as well.

It is worth mentioning that the order of operator precedence is silly to establish when not using parentheses, but the order in a non parenthesized query results in the highest precedence at the rightmost operator, and decreases toward the left.

Why we did this:

URL Storage and the Prefix Tree

Using a Prefix Tree is a straightforward choice when dealing with words, however it has some limitations. First, we used a fixed array in each node, to prevent algorithm complications with attempting to access something that does not exist, and also for size reasons. Having an indefinitely sized array (i.e. an ArrayList) in each node could be a problem for generalization and program size in memory.

This limits what the tree can store, as characters must be in Unicode values. Currently, the array is set to a size of 256, though there is still much empty space due to not storing capital letters or punctuation. There is a better hash method that does not require a cast to lower case letters, and only stores letters and numbers, but this can be dangerous; if a non-word letter is given to it, it will give back -1, which is instant death if it ever is accidentally passed into an array. While checks still need to be done to be sure that 256

is never breached, it can handle a few cases that might crop up, like the Copyright symbol.

We did not check for non-English Unicode character sets either; while this is not *too* difficult to implement, it would require a good amount of work, and could result in unreasonably large tree nodes, as you would need a list (of indefinite size) of arrays for each character set. Dealing with foreign characters is not a problem; the code can handle bad characters by not storing or finding them, but this has the side effect of not being a truly complete search.

Big List vs. Big Prefix Tree

As said above, we found two different ways to store the Web Index

We thought it might more efficient to use a single tree, and store the URL's corresponding to the words in the tree's nodes. As the URL's respect a file structure, it is reasonable to store these as a tree themselves, but somewhat pointless. Because the URL's involved must be iterated through, no matter what structure is used, there is very little that can be done to improve their efficiency. A List is really the best choice for storing URL's. Searching for the words first and then getting the resulting URL's should be faster than searching through all the URL's and finding if they have the words we need:

Algorithmically, if you have N URL's, and search for a k length string, you have a time of $O(Nk)$, when using a main List. This results in many URLs that are checked that do not contain the word. Considering the possibility of thousands, millions, or billions of indexed pages, and the need for a lightning fast search, this clearly isn't the best form for the problem. In contrast, using a Tree at the top results in a time of just $O(k)$, the length of the word.

Using operators, specifically AND, results in a good comparison as well. Using a main List, you would perform the AND on every URL, a possibly time consuming process. If instead a Tree is used as the main structure, URL's containing the word can be filtered down through the AND (as suggested in the handout), so URL's are not checked needlessly.

To analyze this AND operator algorithmically, if we take n to be the number of URL's returned from finding the first word of length k , and m to be the URL's from finding a word of length l , this results in a time of $O(k+l+(mn))$ with a Tree structure. $O(mn)$ is the time it takes to filter out the intersection between the two URL sets.

This is compared to the List structure, which is $O(N(k+l))$. N is almost certainly (possibly very much) larger than n or m .

If the AND operator is operating on more complex queries as it's words, then k and l take on the time it takes to find those, however m and n are still the resulting URL lists. This is in the case of an AND with two other AND's as its respective arguments, and so on.

Using the OR operator is really a much simpler analysis. A tree structure will find both the lists of words, and union them together. Allowing duplicates is faster, but less accurate of course. Not allowing duplicates is simple, $O(k+l+m+mn)$, which really turns out to be very fast (here, $n > m$). When dealing with small values of m and n , it is valuable to not allow duplicates, while large values may benefit from allowing dupes.

A List as a main structure is slower for this operation, as the OR calculation is performed at every step. It becomes $O(N(k+l))$; just like the AND operator. Assuming small values for m and n , and a large one for N , the Tree clearly gets a big boost using OR rather than AND, where a List does not.

The only difficulty using an overarching Tree instead of a list is when searching for complete phrases. The time constraints are very similar to using a List, but the code is more complex, putting a Tree structure at a disadvantage. To check for a phrase, word locations respective to the URL's they were found in are checked. This means that each URL stored in each word, must also contain specific locations of that word in the document.

Checking for a phrase is by far the most expensive operation of any of the queries. It entails the scanning of an ArrayList of locations for a word, and then the locations of the subsequent word to see if they match... and so on. Overall, this takes on the order of $O(N*(n^w))$ time, where here N is the total number of URL's, w is the number of words in the phrase, and n is the number of locations for each word in the phrase (which can vary). More plainly, it is $O(N*m*p*q*r*...)$ where each word has a variable amount of locations.

Using a Tree requires the same amount of time, but deals with smaller data sets. We will take R to be the resulting list of URL's that contain all the phrases words (which is the same result as if the phrase had AND operators). Then, each of these URL's personal ArrayLists must be checked for the words in the proper locations. Presumably, the number of URL's containing all the phrases words is smaller than the total number of URL's, so while you still must iterate over $R*n*m*p*... locations$, you *should* have fewer almost complete phrases than you would get reading every single URL.

Due to the similarity of the time constraints when searching for a phrase, but the increased complexity of the Tree structure, the List actually wins out here.

To conclude this surprise algorithm analysis section:

Prefix Tree as the main Structure:

Searching for a word: $O(k)$

Using AND: $O(k+l+(mn))$

Using OR: $O(k+l+m+mn)$

Phrases: $O(R*n*m*p*...)$ where $R < N$ (probably $R \ll N$)

List as a main Structure (current implementation)

Searching for a word: $O(Nk)$

Using AND: $O(N(k+1))$
Using OR: $O(K(k+1))$
Phrases: $O(N*n*m*p*...)$

All of these assumptions take much smaller values for m and n , the resulting URL's from a tree search, than the total number N URL's in the big List. This is feasible, as clearly every webpage will not contain the word "apple". When searching for more common words though, like "the," without restricting it from the search, it is possible that m and n can approach N .

Another issue to consider is the length of words in the Prefix Tree. Generally, it is assumed that the tree will take less time to search than the List, but considering size, the tree can become arbitrarily big. Since we are not reading from a defined dictionary, words can become extremely long, in fact, not even be words at all. Regardless of this, the tree is almost guaranteed to have k much, much smaller than N . A search database may contain millions of URL's if not more, and it would surprise me to find even a thousand letter long word on any page, much less have someone search for it.

Though the tree seems and indisputably faster construction, we decided on the List for simplicity of code.

The Parser

While the query parser does seem to perform some "funky" steps to get to the final goal, it ended up being quite robust. For a strict search, all that is needed is a small amount of code worked into a recursive descent parser; however, this presents its own problems when attempting to add robustness to the search. Our method, which is undoubtedly longer, and possibly slower, is certainly much more robust than an unmodified R.D. parser. Considering the data size given as a query, since it is very small, the time difference is really negligible. One change we would have liked to make is to have the methods use String Buffers, as opposed to Strings for efficiency, and a few more automated methods to 'eat' various things might make it somewhat more compact.

The HTML parsing we wrote is fairly simple, and really does nothing too special. Its main goal is to discover and add URLs to a list to be parsed. The most prominent key identifiers of what are URL's are easy to find; a much more advanced search might include image links and other types, which ours does not. It was interesting investigating the structure of the WebCrawler and HTML Kit parser method calling order; static and dynamic type becomes very important and rather cool.

Overall Design:

I have spent a great amount of space discussing the general design of the project, and also snuck in a few algorithm analysis points during it, but I haven't yet gone into the small details of the project. This is because aside from the general design, it is nothing too special. There are very few truly confusing methods that we wrote, and most perform

basic operations. Parsing was quite difficult to work out in both cases, but it boils down to looking for specific patterns in the text, and handling a few special cases like relative URL's and malformed search queries. As said above, the query parser is much more extensive than the HTML parser; this illustrates an important principle in Computer Science; it is much easier for computers to understand computers, than to have them try and understand people. Because the query parser is a direct input, it has many, many more opportunities to break, and many more bizarre cases to enforce. In an HTML document, if it doesn't work, it just doesn't work; it's code.

There were some important decisions made aside from the general structure; these include how to make sure the search works exactly as planned; since we did not program a "soft" search, it will only find exact matches of complete tokens. This necessitates that punctuation be at least removed from the end of words, if not all together. In the end, our design removes all punctuation from a token; the question remains if this is a good or bad design issue. The punctuation could be vitally important, but on the other hand, removal is critical to getting good results.

Another design issue was how to incorporate the halfway completed classes already given to us. Some of these functioned quite oddly, like the super call of parser *always* returning an empty list. In theory, we should probably have used this list to then store the URL's we got, but instead we used a class variable. This methodology resulted in using many, many lists as class variables to store the various lists being processed. A slightly more function oriented approach might be better programming, but the current solution certainly works.

The Karma:

Even though I've said many things our search doesn't do, it still accomplishes the goal of the design well. On top of this, we have included some of the extra karma stuff. Our search supports full negative queries, DeMorgan-ing its way across the search. It also does not require parenthesis to be placed in the query, as they are put in when the string is "repaired" before processing. An interesting side note, we did not know it would support this, since it wasn't explicitly programmed, but tests have confirmed that it somehow does!

Testing:

Testing was not exhaustive, but it was certainly thorough. Testing everything but the parser was fairly easy; knowing what specific functions and prefix trees should return gave us confidence that our program's foundations were solid. More difficult to test were the parsers, and both for different reasons. Testing the HTML parser was an issue because of the difficulty in knowing if it was truly working or not. Since the inputs are uniform (though diverse), there was not much to worry about on the parsing side, but considering the nature of the program, finding proper test cases with known results was

rather difficult. We did discover ways to test the HTML parser with debugging and various printouts of new URL's found and list sizes.

Conversely, the parser was difficult to test because of the many, many inputs it can be given. The correct output of the parser is known, and is quite easy to get back. But accounting for all the absurdities a human might put into a query is an incredibly daunting task.

In the end, we resorted to simply testing inputs targeted to a specific case (and multiple at the same time) for the query parser. It was shown to be rather robust in its handling; though I can't imagine we truly tested for every input we can be given. I am confident it can handle most of anything though.

Conclusion:

To repeat what is said above, this program illustrates an important fact of Computer Science; computers understand computers much better than they understand people. This was a fact learned along the way, as well as the importance of design decisions; sometimes the smallest change can mean big differences in speed, or the length of code needed, or special cases generated, the list goes on.

Also of interest is the ways that real search engines must work. Our engine, while functional and quite cool, is little more than a toy compared to the real Google algorithms. We did not include special characters, nor did we even bother parsing more complex links than regular Hyperlinks. To coerce computers to go through the complexities they must generate so quickly really exemplifies the achievements of Google and other engines.

It was both surprisingly difficult at times, and also surprisingly easy at others to do the operations we needed. Often finding a bug in the project came down to some detail or complexity, and often slight design changes (mostly adding in protection for bad inputs), but the program itself remains quite sound.

So, as a truly final conclusion, we have this; Google really **is** awesome.

Programming Hours/Credits:

Matt-
LTree, HTML parsing (WebCrawler)

Chau-
Query Parsing (WebQueryEngine), WebIndex

About 14~20 hours spent on this assignment in lab. Chau put in a lot of work on her own time for the Query parsing. Matt spent some time researching ways to parse the HTML, and some minor adjustments to the code.