

Assignment #7

Purpose

The purpose of this assignment is to construct a web crawler that indexes pages connected to each other from a starting universal resource locator (URL). Additionally, we needed to make a query engine that allows for the searching of the web index based on a predefined query language.

Implementation

WebCrawler

The basics of the WebCrawler are pretty simple. All that happens is the WebCrawler visits a page, the text in that page is split into words which are then indexed using the WebIndex, and any links not already visited that are encountered are properly formatted and added to a Queue. When the webpage is finished being indexed, the top of the Queue contains the next web page to crawl and this process is repeated. It finishes when the Queue is empty.

The one problem with the WebCrawler was properly formatting the URLs. Colin would love to tell you about it in depth some time, so I will summarize his comments here: everything has to be an absolute path, everything has to have the correct scheme, and things are checked as both a path and a file. Obviously, many headaches were caused as a result of the large variety of ways in which to link.

WebQueryEngine

WebQueryEngine uses a formatted text String to search the WebIndex for webpages that either do or do not (there is no try!) contain certain words. We have chosen to define a word to be a combination of anything that is not a space or punctuation (meaning that “!@#\$\$%^&*()_+” is not a word), which is in line with the minimum requirements of what a word is supposed to be. Since we do not define words to be composed with spaces, all spaces are first removed from the search String.

Before the searching occurs, the formatted text String is turned into an expression tree composed of QueryNodes (which either contain words or “&” or “|”). The process of creating the expression tree is handled recursively using the search String and dividing the String into two parts based on the lowest “&” or “|” (which is the “&” or “|” corresponding to the outermost parenthesis). Finally, the base case where there is either a word or phrase is handled.

The process of finding valid webpages is also handled recursively. Each expression tree and subtree fits one of two cases: the root’s value is either an “&” or an “|” and has left and right children or the root’s value is a word or phrase with no children. If it is either a “&” or a “|”, the list of webpages fulfilling the left

expression tree is generated. If the root's value is a "&", the list of webpages fulfilling the right expression and contained in the list generated from the left subtree is generated. If the root's value is a word or a phrase, a List is formed containing every webpage that contains the value and that List is returned. From those two cases, a correct list of webpages is generated.

WebIndex

The WebIndex class maps a webpage to its text. The text needs to be stored in such a way as to allow for the searching of individual words and the searching of consecutive words. This means that it is not enough to simply know whether a webpage contains a word; instead, the ordering of the words is a necessary property to maintain.

This first suggested to us that we should contain the ordered list of words from each webpage in separate ArrayLists, but this performed no compression on the text. For instance, the webpage with the text

```
The swift red ... lazy brown dog.
```

would be stored as

```
["The", "swift", "red", ... , "lazy", "brown", "dog"]
```

This maintains order, but takes up a lot of memory. In fact, it takes up so much memory that with a relatively small set of URLs (compared to the size of the internet), this approach led to an OutOfMemoryError ☹. Clearly, we were forced to implement an indexing scheme that utilized compression.

To do this, we borrowed our Dictionary implementation from the Boggle assignment (a node-based prefix tree), with a few modifications. The prefix tree is constructed so that a path from the root to any node forms a prefix. Therefore, every node in the tree corresponds to a unique path and every path corresponds to a unique prefix.

Now, instead of using an ArrayList for every webpage to index every word of text on the webpage, the ArrayLists index the nodes whose path to the root of the tree forms the word in that index. For example, if the String "swift" were to appear as the fifth word in a webpage's text, the ArrayList would have as its fifth index a reference to the node containing the character "t" and with a path to the root that forms "swift." Any other instance of the word "swift" in that webpage would also have a reference that node.

The most apparent advantage of this scheme comes from its reduction in the amount of memory the program takes. When indexing 10,000,000,000,000 webpages, there will be a large number of shared words among the webpages that would each be stored a large number of times by merely storing the Strings. Using the Dictionary implementation, however, means that identical words only have to be stored once. This means that the amount of memory consumed by adding a webpage to the index decreases as the size of the WebIndex increases because so many words are already contained.

This doesn't mean, however, that adding subsequent webpages does nothing. The URLs of the webpages are each stored individually, so there is a linear growth in memory proportional to the number of webpages. Additionally, the ArrayLists used to store the nodes take up memory themselves and each node reference takes up a minimal amount of space.

Since the word corresponding to a certain node's path is accessible only through a traversal of the tree (which means that word accesses have an $O(L)$ time, where L is the length of the word) and ArrayList access times are constant, it might seem as if the previous ArrayList version (storing the Strings) runs faster when querying.

However, the only time words are accessed is when the QueryEngine tries to find out whether the webpage contains a certain word. In the ArrayList version, the list is traversed and every word is tested for equality with the chosen word, a process which takes time proportional to the length of the String. In the node version, the list is still traversed and every node's character is compared to the corresponding index of the String. If they match, the node's parent is then compared to the previous index of the String, and so on. This process is still proportional to the length of the String and so the node-based index has an identical asymptotic runtime for querying.

As a final note, the OutOfMemoryError encountered by the first implementation was not encountered by the second. Using a small number of html files, the first implementation does actually take up less memory than the second, but this difference is minimal and completely erased when the number of html files increases.

Testing

We tested our WebCrawler on the two webs provided, both of which it indexed quite well. Additionally, Colin tested the WebCrawler on custom webpages that contained strange hyperlink formatting such as backslashes in path separators and relative paths.

Pair Programming Log

Wednesday, November Whatever, 2006

We worked on a general design of the classes in this project for about an hour. No code was written. We just planned.

Friday, December 1st, 2006

The WebQueryEngine and the first WebIndex were developed.

Saturday, December 2nd, 2006

Today we worked from 2:00 p.m. to 7:30 p.m. We worked on the WebCrawler and found the URI class after a lot of searching. Additionally, we wrote more of the WebEngine.

Tuesday, December 5th, 2006

Today we worked from 6:30 p.m. until 11:30 p.m. We made the new implementation of the WebIndex and tried using a similar method to store the website urls. For some reason, that took up more memory than keeping the urls as part of the keyset of a HashMap, so that time was lost. Then we worked on WebCrawler, trying to make it compatible with the many different ways a website could be linked to.

Wednesday, December 6th, 2006

Additionally, Colin Wragg stayed up until 5:00 a.m. today. After that, we worked from 12:30 pm to what is right now 3:25 p.m. This report was written and more annoying special cases of different url links were handled.