

TSoogle

John Sheu, Chris Agerton

December 7, 2006

In Soviet Russia, the applet searches you!

1 Overview

NOTE: It seems `javac` doesn't like `WebIndexPage.java`; Eclipse has no problems with it, however. As a workaround, we have provided the class files in `classes.jar`. You can either create an Eclipse project for this assignment or use our compiled version.

For the final assignment of the class, a fully functional search engine was constructed. This was the least restrictive project yet and required the use of many concepts from throughout the semester.

2 Design

For purposes of fast lookup and compression, a two-way associated map was used to associate each text token encountered in text parsing with an ID number. A `TreeMap` with a `String` key and an `Integer` served as the forward mapping of token to ID; this ID was in turn an index into an `ArrayList` of `WebToken` objects. Each of these objects encapsulated a `String` field with the original token and a list of all web pages in which this token appeared. As the assignment required also the implementation of a phrase query. This implied that the entire text source of each page would have to be somehow represented; to make this possible, some measure of compression was necessary. Firstly, because we had associated each text token with an ID number, the text of the page itself could be replaced with a list of ID numbers, affording a large savings, as each ID number is an `int`, being 4 bytes, whereas each character would have been series of chars, each 2 bytes. As each URL was also passed through the tokenization process, a `TreeMap` of `Integer` URL IDs against `WebIndexPage` page objects was used to store the mapping.

3 Implementation

3.1 WebCrawler

The purpose of the `WebCrawler` is to traverse a web of linked hypertext documents and store index information in the `WebIndex` class. This was made easy by Java's `HTMLToolkit.ParserCallback` class. When used, the Java class calls methods in the `WebCrawler` class when it hits certain elements in the page. URLs are noted and passed back for recursive descent; text tokens are tokenized and added to the page representation.

The token mappings are themselves relatively efficient. The mapping of token to ID is stored in a `TreeMap`, giving $O(\log(n))$ lookup time for each token (as the map is implemented as a red-black tree internally). The mapping of IDs to tokens is stored in an `ArrayList`, giving an $O(\log(1))$ lookup time for these. Most operations, especially in web crawling, are token-to-ID lookups. Thus, for further efficiencies, this is further accelerated by caching. A scheme of two chained least-recently-used caches is used, for resistance against scanning (having cache entries evicted when a large block of nonrepeating text is encountered.). A token is first looked for in a small primary cache; if exists, it is returned. If not, the second cache is searched. A token not found in either is then looked up in the usual manner. Then, the token and its associated ID is cached in the secondary cache. The token is only promoted to the primary cache if it is found again on

a subsequent lookup in the secondary cache; thus, entries that are seldom found, are unlikely to wind up in the primary cache, evicting a more-common entry. In the "scanning" scenario as outlined above, none of the entries encountered would be promoted, as should be the case as the primary cache is reserved for entries that are often encountered. The sizes of the caches themselves were roughly optimized empirically.

Kung-Fu Bonus Point:

The `main()` method of the class uses a `PageFetch` thread to download URLs before they are passed to the parser. Because this can take place while another page is being parsed, there is a significant improvement. If we were routinely indexing high latency pages, this could easily be modified to run several threads in the background.

3.2 PageFetch

`PageFetch` was an especially fun class to write because it was primarily developed by someone who had very limited with threads. After this is started, it pulls `Strings` out of the `urlQueue`. Normally, at this point, it calls `getData()` to retrieve a `char[]` containing the stream data. It then creates a `BufferedReader` of a `CharArrayReader` containing the result. The `getData()` method simply downloads 1KB buffers in a linked list until the page is complete. At this point, it creates a single buffer and copies all of the linked lists in. For a real search engine, this might have more interesting limits to size or perhaps restrictions to weed out non-text/html data. Also, if the `dataQueue` is almost empty the caching is skipped and it sends a simple `BufferedReader` connected to the source of the URL.

3.3 Query

The assignment sheet places great emphasis on the query language and parsing methods. However, it was a relatively minor proportion of the time spent on the assignment. To create the parse tree, we recognized that an intermediate representation would be most useful; this turned out to be RPN, or Reverse Polish Notation. To create the RPN-notated form of the text query, Dijkstra's "shunting algorithm" (an excellent description, which we heavily referenced at en.wikipedia.org/wiki/Shunting-yard_algorithm) was implemented. Once implemented, converting RPN to a parse tree was a trivial task. Thus, we are able to handle a great number of the "karma cases" regarding parsing essentially for free, including full negative queries and removal of restrictions on parentheses.

Once a parse tree was constructed, the query was executed. Essentially, this is as outlined in the hand-out. Each leaf node represents a text (or phrase) query; then these individual queries are combined by boolean operators moving upward toward the tree, until a resulting set is output once the root node has been processed. Thus, the time taken for a search (especially for slow searches, such as phrase queries or when ranking is turned on) is proportional to the number of search terms, not the number of operators, as the searches are slow. (All operators implemented operate on integers, as merging operations, and are thus empirically negligible in searching.)

3.4 WebIndex

Optimizing `WebIndex` for memory usage proved an interesting challenge. One potential option was to load the token-index tree only on-demand, but this was eventually regarded as unworkable as we preferred to rely on the robust `TreeMap` classes. However, the web page representations themselves can be retrieved on demand, as for most purposes they are not necessary; for example, simple searching only requires that the pages containing each token be found, not the content of the pages themselves. For phrase queries, the page text is necessarily retrieved, so phrase queries turn out to be a relatively slow operation. For purposes of ranking, also, counts of each token's appearance in the page must be retrieved, making it slower as well.

For the purposes of conserving disk space, all streams are GZIP compressed. As disk IO and decompression are slow, we chose to offload these operations onto a separate thread. This has the advantage of utilizing the processor for text searching even as it blocks on relatively slow disk IO operations. In essence, a `WebPageRetriever` object is tasked with retrieving pages, to be returned through a blocking queue.

In the end, our implementation of `WebIndex` is quite capable of handling large text corpuses. For testing, we used a downloaded copy of the Java 1.5.0 API, weighing in at 300MB. This was downloaded in the form of a 45MB zip file; for comparison, our index, saved on disk, is only 30MB. Creation of this index requires a maximum of 300MB of memory, as measured by the Unix utility `top`. Incidentally, the memory usage leveled off once approximately 75% of the corpus was parsed; thus we have reason to believe that for relatively homogenous corpuses, such as the Java API documentation, the search engine can efficiently “learn” a vocabulary, as words encountered are likely to be already in the dictionary.

4 Testing

Each logical chunk was first lightly tested on smallish hand-constructed webs to ensure basic functionality, then basted with a merlot marinade and baked at 350 Fahrenheit for a half-hour. This eventually merged with white box testing and debugging methods and frequently used the small `www.superspoof.com` mirror.

White box: (We used all the red on the marinade!) During the development stage, we used the Eclipse debugger to follow the execution of code during several key algorithms. This was useful in finding bugs before they became a problem.

Black box: Searches were performed on various data sets to ensure quotes would be found, the cardinality of the union of two searches is the sum of the cardinalities less the intersection.

Our main testing corpus was the 17,000 document behemoth known as the Java 1.5.0 API documentation. Results from running our engine on this corpus was compared to results given by various magical incantations of `grep` until unholy agreement was reached. To test web crawling, a sandboxed set of pages on a privately-owned web server was crawled as well.

5 Notes

5.1 Ranking

We tried two different ranking methods. The first attempted entailed tallying links and returning the pages in order of how many pages linked to them. Given our data set, this tended to always return index pages at the top with very low overhead and was otherwise generally uninteresting. As an afterthought, we added a simple method to tally the number of occurrences of each word in the search. This requires scanning each document returned and thus requires a relatively long time. (Searches without ranking, as the codebase stands now, are practically instantaneous; searches with ranking return in on the order of a few seconds). To turn on ranking, set `index.doRanking = true;` in the main method of `WebQueryEngine.java`. We feel you will be generally unimpressed.

5.2 Real World

URL traversal is currently limited to the “file:” protocol. If any testing takes place in an online sandbox, you will need to change the `handleStartTag` method of `WebCrawler`. Only one line will need to be commented: `if (file.equals(tempURL.getProtocol()))`. For high latency indexing, it may also be helpful to change the `PREFETCH_COUNT` constant at the beginning of the `WebCrawler` class. To limit the number of pages, change the `var MAXPAGES` in the main method of that file. There are also two token caching values (`CACHE_SIZE_L1` and `CACHE_SIZE_L2`) in the `WebIndex` class.

Note: the `serialVersionUID` variable of `WebIndex` fixed to allow minor changes to the serialized class without recrawling the web. This should be changed if data integrity is important.

5.3 Applet

We didn’t bother tinkering with such toys.

5.4 PeerProgramming

We collaborated in the Painter basement for many hours. For major design decisions, we worked closely together; for smaller tasks, we worked on separate workstations.

5.5 Attributions

The codebase supplied by our TA Mr. Awesomesauce was used, and the Java 1.5.0 API documentation thoroughly consulted. In particular, the Wikipedia entry at

http://en.wikipedia.org/wiki/Shunting_yard_algorithm
proved helpful in providing the algorithm to convert infix to reverse Polish notation for search queries.