Implementing an Efficient Search Engine
Moogle: Report

Gabrielle Ehrlich && Steve Tricanowicz
CS 51
3/6/2010
Moogle

Plan:

We planned to implement optional case sensitivity so that the user could choose whether or not they wanted to use it. This is a common feature in many search features in word processing applications and wanted to extend it to web searches. We also planned that if the word they searched came back with no results, to suggest a list of possible words that were in the dictionary that they might have meant. Searching each of these words would ensure at least one result.

Actual Implementation:

We managed to implement both of these features. When you go to the Moogle search page, there is a check box which if checked makes the search case sensitive, but otherwise the search is not. The default is no case sensitivity for two reasons: it broadens the search to increase chances of getting relevant results, and it is the standard of almost every other search engine.

We also implemented a feature that acts if the word or words searched returns no results. It will then generate a list or words of Hamming distance (number of letters changed in a string of equal length) one that are in the dictionary, and ask if that was the intended search. We had wanted to keep the searches that we did for the suggestions, but we found that due to how the loops ran, that was not so simple, since any LinkSet that we would wish to cache is automatically searched upon searching a query.

For spelling suggestions we knew our implementation wasn't going to be very efficient, so we tried (no pun intended) to look up other options. We came across a really cool structure called a triangulation trie that seemed to do exactly what we wanted. These would have used a normal trie implementation with the Hamming Distance for a metric. Unfortunately, they were kind of complicated and we weren't entirely sure how to implement them functionally (imperative tries can just be arrays of pointers), but they would have been much more efficient. Our spelling suggestions runs in roughly O(*mn*) time where *m* is the length of the word, and *n* is the size of the dict. Tries, on the other hand, would have been in time linear of the length of the word. Even more interesting, triangulation tries, we found are in O($\sqrt{m}$). Nevertheless, we were able to improve this run time by short circuiting the Hamming Distance loop if the two strings are not the same size; plus, we save on space by not duplicating entries in the dictionary and the trie.

Changes:

We had to add our own module, queue to implement the breadth first search which the crawler executes. We had to change a lot of how query worked. For example, we needed to add the ability to parse case sensitivity. We did this by adding more regular expression checks to see if the option was added, and to strip it off of the query if it is. We also had to change how the

server loop executed the query to choose a different dictionary for if it was case sensitive or not.

This is a very important design choice of ours. Instead of increasing the size of the dictionary to include both case sensitive or insensitive options (we need to store both to provide the option of both), we decided to increase the amount of up front work and store them in two different dictionaries of approximately the same size (the case sensitive dictionary is, of course, slightly larger). Thus, a different dictionary is passed to the searcher method depending on the type of search. This provides vast search time improvements, especially considering our not optimal spell checking algorithm.

A more minor change was necessary to implement spelling suggestions we just had to change how the response was set up if the list of links generated was empty. We simply changed the body of the result html which is returned to the browser. We also decided to improve usability by moving the search bar to the results page if there was no results.

Design-to-Implementation:

We learned a lot about how modules work and interact. We found that it was harder to implement our ideas in the specific instance, even if we knew how they should work in theory. It wasn't too bad once we figured out all the details of modules but that definitely took some getting used to. The biggest thing to get used to was the interaction between the Caml code and the web browser via the Unix library. This was the biggest challenge in being able to pass the case sensitivity option to the back end. However, we expected the spell checker to be much harder to implement then it actually was which was presently surprising. The implementation was facilitated by the data structures we already had, namely the dictionary.

Partner Breakdown:

Most of the work was done together for this project. In some instances, work was delegated to each partner; for example, Part I involved Steve doing the insert function, and Gabrielle doing remove. However, we would always review the other's code, swapping projects to get it actually working. Steve did the helper methods to dictionary and I implemented sets. For Part II, we worked together on pretty much everything, usually working on the same computer together. Gabrielle wrote the pseudo code for crawler, and Steve helped get it hooked up to all the modules. Then Steve figured out how to do all the regular expressions we needed for case sensitivity, and how to get the user input on the matter, while I set up the hamming function for spell checker. Most of the work, though was tied up in debugging, on which we both worked together in order to actually work.