Programming Assignment 4

Nathaniel Sigafoos

3/25/17

The problem addressed in this assignment was to use a custom LinkedList to implement a spelling checker on a text document. The program would read through a file and count the number of correctly spelled words, the number of incorrectly spelled words (both according to a given dictionary), as well as the number of comparisons it took to determine whether or not a word was spelled correctly. Because the dictionary structure used LinkedLists, the algorithm was forced to do a linear search for each word. What is meant was that you could learn more about the dictionary of words, by looking at the result. Because when a word is not in the dictionary, the algorithm is forced to search all the way to the end. This meant the average number of comparisons made to determine that a word was not in the dictionary was approximately the same as the average number of words under each letter in the dictionary. This was backed up by the average comparisons needed to find a word being about half that number. This makes sense, because, on average, in a linear search must look half way through the list to find a word.

While writing this algorithm, we came across two main problems: how to deal with hyphenated words, and how to deal with words with apostrophes. Dealing with hyphens was straightforward enough, we simply defined "words" as being separated by whitespace or hyphens. This meant that when there was a hyphenated word in the file, our algorithm would split it at the hyphen, and then feed us each side individually. This allowed us to easily check the spelling of both sides of the hyphen.

Dealing with apostrophes was a little trickier, because there were several cases where we might want to deal with the apostrophes differently. For example, if a word ends with an 's, making it possessive, we would want to just chop it off and check the spelling of the non-possessive word. However, if a word was a contraction (e.g. wasn't), it could be a correctly spelled word. But if we just cut off everything after the apostrophe, it might not be one anymore. Some would still be words, such as "can't" → "can", but others like "wasn't" → "wasn", would not be. The only way we could come up with to get around this issue was first look for the word with the apostrophe, and if it wasn't found, look for it again without it. However, we decided not bother with this because of the huge hit the algorithm's performance would take. In addition, we had no way of knowing if our dictionary was even complete, so we still might get false results. Because of this, we opted for the simple solution of always just chopping off everything after an apostrophe. This would still give us an accurate result most of the time, and it would do so without drastically hurting the performance of the algorithm.

As it is now, out algorithm 's biggest weakness is the fact that it relies heavily on regular expressions to parse words. Most of the parsing work is done by one complicated regular expression, which is used to remove invalid characters from every "word" in the file. The plus side of this is that the algorithm itself is relatively simple, allowing the regular expression to do most of the work. The issue is that regular expressions are often inefficient and resource consuming, thus hurting the algorithm's performance.