CS 214 Assignment 1: Tokenizer

Kevin Sung and Joyce Wang

October 18, 2013

1 Implementation

Our indexer uses a tokenizer and two main data structures, a sorted list and a hash table. We modified the tokenizer given in the assignment so that it takes a file as an input and returns the words in the file as tokens. Our hash table uses linked lists for its buckets. When our program is run, it creates a sorted list of word-list pairs that is used to store the index. Each word is associated with its list of file-count pairs. Our program then recursively scans the input directory for files. For each file it encounters, it creates a tokenizer and a hash table for that file. Then we get the words from the file one by one with the tokenizer. Every time a new word is encountered, it is placed into the hash table and associated with a count of 1, and it is placed into a sorted list that allows us to iterate through the entries of the hashtable after all the words have been put in it. If a word is already in the hash table, then we just increment its count. After a file has been processed, we get all the word-count pairs from the hash table and put the results into the sorted list of word-list pairs that represents the index. After all the files have been processed, the index is finished, so we write it to the output file.

2 Analysis of running time and memory usage

The bulk of the work is performed in the processing of the files. Each word in the file is tokenized once and checked against the hash table once. Therefore, the running time is $O(n)$, where n is the total number of words in all of the files that need to be processed. Entries in the hash table are created only for each distinct word, and in the index list, there is an entry for each distinct word, each of which could contain a list of every file in the worst case. This occurs if every distinct word occurs in every file. Therefore, the space usage is $O(pq)$, where p is the number of distinct words in the files, and q is the number of files.