README - Search

Craig Perkins
Alexander Tang

Our program assumes that the input format from the indexer is in the following format: (The same format specified in the indexer instruction page)

<list> i
testdir/file1 1 testdir/file2 1 testdir/subtestdir/file3 1
</list>
<list> inside
testdir/file1 1 testdir/file2 1 testdir/subtestdir/file3 1
</list>

To implement Search, we used a hash table called uthash. The hashtable is composed of a structure defined in our program called a Record. The Record struct has 4 fields, an integer used as an id (this is the output of the hash function and the key for the table), a char* for the word to be put in the table, a SortedList* to the list of filenames and frequencies of occurrence, and a UT_hash_handle struct which makes the item hashable. On the ilab machines, an integer takes up 4 Bytes, a char* takes up N bytes, where N is the length of the string, a SortedList* takes up 8 Bytes and a UT_hash_handle takes 52 Bytes on a 64-bit machine. The record struct takes up 64+N Bytes. Each record has a SortedList* to a SortedList of filenames. Each Node of a pointer has a void* pointer to a filename, a Node* to the next node of the list and an integer, refcount, to know how many iterators are pointing to a node. Both the void* and Node* take up 8 bytes of memory. In total a node takes up 20 Bytes of memory. We also need to allocate memory for the filenames to be stored in memory, which takes N bytes of memory if N is the size of the filename. Also each SortedList structure has a pointer to the first node, an unsigned_long for the size, and 2 pointers to functions, one to know how to compare the data items and one to know how to destroy the data items. In total a SortedList takes up 40 Bytes of memory.

If the program search is run with a file containing N unique words of average length M and L unique filenames with average length P, then in the worst case the program will take up (40+64+M)*N + (16+P)*L*N Bytes of memory.

In terms of runtime, the creation of the Hash Table to SortedList is the slowest operation of the program. In the best case the program will be run with the output of the indexing of 1 file. In this case if there are N words in the file that was indexed than the creation time of the table will be done in O(N) time (this is assuming that the resizing and rehashing is done in amortized time). In the worst case we assume the program indexes a total of N words from L files. In this case the creation of the table will take O(N*L) time.

For search or the program works by making a SortedList of all of the filenames found and ignores duplicates. Assuming search or is run with N words with an average of L filenames of each word, then search or is done in O(N^2*L^2) time. In the worst case all filenames are unique and the problem can be simplified to creating a SortedList of N*L unique elements. Since insertion is done linearly, the process will take at worst case O(N^2*L^2).

For search and the program works by taking the intersection of the SortedLists of each word of the input to search and. Assuming there are N words of input and each word has on average M filenames that the word is in, then search and will run in O(N*M) time. To and 2 sorted lists takes 2*M time and is done N times for a total of O(N*M) time.

The test cases for the function were done with the inverted index file “test_index”.