Asgn7 Design Doc

Elliott Jensen

November 2021

1 Introduction

This program searches a text document for words from a given list of words. The program receives a list of words and then searches the document to see if it includes those words. After the search has been performed a message is printed out along with the words that were found. To do this the program uses a bloom filter, hash tables and binary search trees.

2 Main: banhammer.c

The main function looks at three input files: newspeak.txt, badspeak.txt and an input file which is stdin by default. It stores the words in newspeak.txt and badspeak.txt in a bloom filter and then searches the input text for those words. First I parse through command line options in order to set the size of the bloom filter and hash table and to determine if statistics should be printed out. Then I read from badspeak.txt and newspeak.txt and populate the bloom filter and hashtable like so:

```
while (fscanf(badspeak, "%s", buf) != EOF):
    bf_insert(bf, buf)
    ht_insert(ht,buf, NULL)

while (fscanf(newspeak,"%s %s", buf1, buf2)) != EOF:
    bf_insert(bf, buf1)
    ht_insert(ht, buf1, buf2)
```

I then create the two binary search trees that will hold the oldspeak and badspeak words that the are found in the input text. I compile regex using the word ”[a-zA-Z0-9_'-]+” and then begin parsing through the input text. It looks like this:

```
while (word = next_word(stdin, &re) != NULL:
    word = lower(word)
    if bf_probe(bf, word):
        node = ht_lookup(ht,word)
```

```
            if node == NULL:
                continue
            if node->newspeak and node->oldspeak:
                counseling = true
                oldnew = bst_insert(oldnew, node->oldspeak, node->newspeak
            else if (node->oldspeak):
                thought_crime = true
                old = bst_insert(old, node->oldspeak, NULL)
```

If -s is specified I then print out the statistics of the program. Otherwise I print out the appropriate message which can be cound in messages.h. I print out mixspeak if the booleans thought_crime and counseling are true, the goodspeak message if only counseling is true and the badspeak message if only thought_crime is true.

3 Bloom Filter: bf.c

The primary data structure used to flag words is the bloom filter. The bloom filter is represented as a bit vector and uses hash functions in order to minimize false positives when adding words to the filter. The type definition (as show in asgn7 doc)looks like this:

```
struct BloomFilter {
    uint64_t primary [2]; // Primary hash function salt.
    uint64_t secondary [2]; // Secondary hash function salt.
    uint64_t tertiary [2]; // Tertiary hash function salt.
    BitVector *filter;
};
```

Instead of building our own hash function we are using the SPECK library. By using hash tables we are able to quickly find words in O(1) time and since we are using the SPECK implementation the program has been optimized for software.

bf_create(size): This function sets all of the salts of the bloom filter and creates the bit vector of size size that will be used.

bf_delete(bf): Deletes the bloom filter

bf_size(bf): Returns the size of the bloom filter

bf_insert(bf,oldspeak): This function hashes oldspeak with the three salts in order to calculate three indices which are then set in bloom filter. The index is found like this:

```
    index = hash bf->salt, oldspeak) % bf_size(bf)
```

bf_probe(bf,oldspeak): This function hashes oldspeak with the three salts in order to find the indices and then searches the bloom filter bit vector at those indeces and if all three return a 1 we return true to indicate that the word was found in the bloom filter.

bf_count(bf): Calculates the number of set bits within the bloom filter

bf_print(bf): Prints the bits within the bloom filter for the sake of debugging.

4 Hash Tables: ht.c

This file stores the type definition and functions for the Hash Table ADT. The hash table holds a series of root nodes of binary search trees that hold nodes that contain oldspeak and newspeak strings.

ht_create(size): Initializes the hash table salt and creates an array of nodes of size size.

ht_delete(ht): Iterates through the array of nodes and deletes each tree.

ht_lookup(ht, oldspeak): Calculates the index using the hash function and then searches the binary search tree at that index for the node containing oldspeak.

ht_insert(ht, oldspeak, newspeak): Similarly, this function finds the index of oldspeak using the hash function, then goes to the binary search tree at that index within the hash table and then inserts the node within that tree.

ht_count(ht): Iterates over the hash table and increments count only when the tree is not equal to NULL

ht_avg_bst_size(ht): This functions sums up all of the nodes within all of the trees and divides it by the number of trees.

ht_avg_bst_height(ht): This function iterates through the hash table and sums up the heights of every binary search tree within the hash table and then divides it by the number of trees.

ht_print(ht): Prints the hash table for the sake of debugging.

5 Binary Search Trees: bst.c

This file includes the library of functions that are used to manipulate the binary search tree ADT. This ADT revolves around using strcmp so that string of lower value are stored on the left and strings of higher value are stored on the right. Using this rule we can very quickly find nodes within the trees and insert nodes into the correct locations.

bst_create(): This creates an empty binary search tree so it simply returns NULL.

bst_delete(node): This function should receive the root of the binary search tree. It performs a post order traversal in order to delete all of the nodes

bst_height(root): Height is defined as max(height of left, height of right) plus 1. Using this we can use recursion in order to find the height of any node in a binary search tree.

bst_size(root): Size is defined as size of left plus size of right plus 1. Using this we can use recursion in order to find the size of any node in a binary search tree

bst_find(root, oldspeak): Use strcmp() to see if the node is to the left or the right and depending on which it is recursively call find on that node until either the node is reached or NULL is reached.

bst_insert(root, oldspeak, newspeak): First check that the node you are trying to insert doesn't already exist in the binary search tree by calling bst on the node. Then similarly to bst find, use strcmp() to see if the node should be inserted on the left or on the right. Recursively call bst_insert until NULL is reached indicating that the node should be inserted there.

The list of words that the input text will be searched for are organized in a binary search tree. To implement this the file node.c and bst.c are used. In node.c each node will have a word to look for and a word that it should be replaced with and also its left and write nodes. The implementation looks like this:

```
struct Node {
    char *oldspeak;
    char *newspeak;
    Node *left;
    Node *right;
};
```

This implementation is chosen because it allows us to quickly find words in time complexity O(log(n)).

6 Bit Vectors: bv.c

The bit vector stores a series of bits and in this program the bloom filter uses it in order to store the oldspeak words in it. Here is the type definition for the bit vector ADT:

```
    struct BitVector {
        uint32_t length;
        uint8_t *vector;
    };
```

bv_create(length): This function creates an array of bits of size length. Importantly it checks to see if length is divisible by 8 and if it is not it adds one extra uint8_t to the array like this:

```
    if length % 8 == 0:
        size = length / 8
    else:
        size = (length / 8) + 1
    bv->vector = (uint8_t *) calloc(size, sizeof(uint8_t))
```

bv_delete(bv): Deletes the array within the bit vector and then deletes the bit vector

bv_length(bv): Calculates the length of the bit vector

bv_set_bit(bv, i): If i is outside of the range the function returns false. Otherwise it returns true in order to indicate the bit was successfully set.

bv_clr_bit(bv, i): If i is outside of the range the function returns false. Otherwise it returns true in order to indicate that the bit was successfully cleared.

bv_get_bit(bv, i): Returns false if i is outside of the range. Otherwise it returns true in order to indicate that the bit at that index is 1 or false if that bit is 0.

bv_print(bf): iterates through the array of bits and prints 1 if bv_get_bit returns 1 and prints 0 otherwise. It looks like this:

```
for i in bit_vector:
    if bv_get_bit:
        print '1'
    else:
        print '0'
print '\n'
```

7 Nodes: node.c

This file includes the node ADT and the functions used to manipulate it.

node_create(oldspeak, newspeak): This function creates a node with oldspeak and newspeak stored in it. The function sets newspeak to NULL if a newspeak is not specifed in the argument. It looks like this:

```
    if newspeak:
        n->newspeak = strudup(newspeak)
    else:
        n->newspeak = NULL
```

node_delete(n): Makes sure that n, n-¿oldspeak and n-¿newspeak exists before deleting them.

node_print(n): Prints "oldspeak -¿ newspeak" if newspeak is not set to NULL in that node. Otherwise prints out just oldspeak.