Design Document - Author Identification

Lev Teytelman

March 4, 2022

1 Program Description

This program will read from a database of texts, as well as one from standard input. It will then calculate the "distance" between the texts, done by taking the normalized occurrences of each word and comparing the standard input's amount to each database text. It will then add these values to a priority queue to sort, and dequeue and print the k lowest-distance ones.

2 Included Files

- bf.h - The definition for the `BloomFilter` struct and its functions.
- bf.c - The implementation of the Bloom filter and its related functions.
- bv.h - The definition for the `BitVector` struct and its functions.
- bv.c - The implementation of the bit vector and its related functions.
- ht.h - The definition for the `HashTable` struct and its functions.
- ht.c - The implementation of the hash table and its related functions.
- identify.c - The file containing the main function and the core logic of the program.
- metric.h - Enum definitions for distance calculation types.
- node.h - The definition of the `Node` struct and its functions.
- node.c - The implementation of the node and its related functions.
- parser.h - The definition of the `next_word` function, which processes a text file with the given regular expression.
- parser.c - The implementation of the parser function.

- pq.h - The definition of the `PriorityQueue` struct and its functions.
- pq.c - The implementation of the priority queue, its related functions, and an internal `Entry` struct for keeping the author name and distance together.
- salts.h - The constant definition of the salts to use when hashing for the hash table and Bloom filter.
- speck.h - The definition of the `hash` function.
- speck.c - The implementation of the function, with helper functions to do so.
- text.h - The definition of the `Text` struct and its functions.
- text.c - The implementation of the text and its related functions.
- Makefile - The file used to easily compile and link files into an executable binary.
- README.md - A description of the program, along with steps to build and run it.
- DESIGN.pdf - This document, explaining the logic of the program and what's being done behind the scenes, along with pseudocode.
- WRITEUP.pdf - The document analyzing the findings of the project and the code used to make it work.

3 Structure

This program will, like previous assignments, take a set of startup flags to modify the program behavior and execution. It will perform the following steps:

1. Read the noise file and create a list of words to ignore.
2. Read the standard input into a text, ignoring the noise.
3. Attempt to open all files in the specified database and read them into texts, ignoring the words from the created noise text.
4. Calculate the distance between the word frequencies of the standard input and each database file, and add the distances along with the authors to a priority queue.
5. Dequeue the first few distances/authors and print them out.

4 Error Handling

We will need to ensure that our program can handle a varied range of inputs. Our structs will also need to play nicely with each other, and we need to make sure that we don't try to modify some external address or value and get inexplicable undefined behavior. When allocating memory for our hash tables and Bloom filters, we will need to allocate large amounts of memory (up to 2^20 bytes per Bloom filter!). Therefore, we may run into the issue (especially with larger databases) where we are unable to allocate memory. Instead of continuing on with a non-existent array or struct, we will need to clean up after ourselves (free anything we've already allocated up to this point) and return `NULL`:

```
def create_thing(size):
    *thing = (*thing) malloc(sizeof(thing))
    thing.array = (*int) malloc(size * sizeof(int))
    if (thing.array == None):
        free(thing)
        return None
    return thing
```

This way, the program will not have any memory leaks, and the caller can interpret the return value as the program running out of memory, and act accordingly.

Bit vectors are finite. When manipulating bits, we will need to pay attention to which ones we are accessing. Therefore, we will have a function that checks whether the bit is out of range of the vector, which other functions will call before doing anything else:

```
def out_of_range(vector, index):
    return index >= vector.size
```

This protects the user from segfaults during use and will instead fail safely and clearly.

If our hash table becomes completely full, our program will loop infinitely if it tries to insert a value that the table doesn't have room for. Therefore, we will prevent this by keeping track of the original index and breaking out of the loop if the index gets back to the original:

```
def hash_insert(table, value):
    index = hash(value)
    original = index
    while (table.values[index] != None):
        index = (index + 1) % table.size
        if (index == original):
            return None
    table.values[index] = value
    return index
```

This is obviously a simplified version compared to what will be implemented in the program, but the idea is the same.

When utilizing our Bloom filter, we may get some false positives. For this reason, we will use the filter as our first line of defense, and check the hash table to confirm when we do get a positive. Especially as the hash table grows in size, lookups will become less efficient as collisions increase and the amount of linear probing increases:

```
def text_contains(text, word):
    if (!bf_probe(text.bf, word)):
        return False
    return ht_lookup(text.ht, word) != None
```

This minimizes the number of hash table lookups and therefore the amount of linear probing by returning early if we know for sure that the text does not have the word.

5 Pseudocode

Let us consider the underlying data types used in our data structures. Our bit vector will simply consist of a byte (`uint8_t`) array and a `uint32_t` representing the size of the bit vector. To set a bit, we will need to get the respective byte and modify a specific bit within it. This will be directed by the index we pass into the function:

```
def set_bit(vector, index):
    if (out_of_range(vector, index)):
        return False
    vector.array[index / 8] |= 1 << index % 8
    return True
```

This will ensure proper error handling if the given index is out of range. Otherwise, it will get the byte containing the required bit by dividing the index by 8 (every byte corresponds to 8 bits) and using the bitwise OR operation to modify just the index we want to modify, which is now just the index modulo 8.

To clear a bit, we will do something similar, but ensuring that all the other bits in the byte are untouched:

```
def clr_bit(vector, index):
    if (out_of_range(vector, index)):
        return False
    vector.array[index / 8] &= ~(1 << index % 8)
    return True
```

This will leave all the bits but the required one alone and set the one in question to 0. It will, of course, use the previously mentioned `out_of_range` function to properly handle invalid inputs.

To get a specific bit, we will perform a right shift on the byte and isolate it from all the other remaining bits with a bitwise AND:

```
def get_bit(vector, index):
    if (out_of_range(vector, index)):
        return False
    return (vector.array[index / 8] >> i % 8) & 1
```

This bit vector will be used in our Bloom filter to insert and probe for the hashes of words.

Our Bloom filter will contain the salts needed to hash words, as well as the above-mentioned bit vector. We will need to hash the words with the three different salts, which we can streamline with the following function:

```
def calc_hashes(filter, word):
    hash1 = hash(filter.salt1, word)
    hash2 = hash(filter.salt2, word)
    hash3 = hash(filter.salt3, word)
    return hash1, hash2, hash3
```

This will simplify our other Bloom filter functions by centralizing this logic to the function call `calc_hashes` instead of having to insert the same few lines over and over again. We will use this function within our insertion and probing functions (which are similar to a bit vector's setter and getter):

```
def filter_insert(filter, word):
    hashes = calc_hashes(filter, word)
    set_bit(filter.vector, hashes[0])
    set_bit(filter.vector, hashes[1])
    set_bit(filter.vector, hashes[2])
```

We will do a similar procedure for checking for the word's existence:

```
def filter_probe(filter, word):
    hashes = calc_hashes(filter, word)
    b1 = get_bit(filter.vector, hashes[0])
    b2 = get_bit(filter.vector, hashes[1])
    b3 = get_bit(filter.vector, hashes[2])
    return b1 && b2 && b3
```

Our hash table is a more complex structure. It will have a single hash salt, an array of Node pointers, and the size of that array. Since hashing this time is a single line, we won't need a dedicated function for it. However, what happens when we have collisions? We could have false positives like the Bloom filter, but that's not ideal. Therefore, we will iterate over the hash table until we find an empty spot (or come back to the start of the array, meaning we didn't find anything). For our Node insertion, we will need to either find an existing Node or create a new one, depending on what we find first in the table:

```
def table_insert(table, word):
    i = hash(table.salt, word) % table.size
    original = i
    while (table.array[i] != None && table.array[i].word != word):
        i = (i + 1) % table.size
        if (i == original):
            return None
    if (table.array[i] == None):
        table.array[i] = node_create(word)
    table.array[i].count += 1
    return table.array[i]
```

We take the modulus of the table size to ensure the index is never out of range of the table. The same is done for the lookup function:

```
def table_lookup(table, word):
    i = hash(table.salt, word) % table.size
    original = i
    while (table.array[i] != None && table.array[i].word != word):
        i = (i + 1) % table.size
        if (i == original):
            return None
    return table.array[i]
```

Both of these functions return the Node pointer that they either found or created, or null if there was no room in the table/the node was not in the table. This linear search in our hash table can get quite inefficient, especially as the hash table approaches full capacity. Therefore, we will use them both in our text object.

Since the Bloom filter lookup is guaranteed to be linear time, we will use that for our first check in the `text_contains` function. Along with the Bloom filter and hash table, the text struct also keeps a count of total number of words added. If we do see that the filter contains the word, we will not take its word for it, since the filter can have false positives, especially as it fills. Therefore, we will also check that the word is in the hash table to make sure:

```
def text_contains(text, word):
    if (!filter_probe(text.filter, word)):
        return False
    return table_lookup(text.table, word) != None
```

This way, we avoid linear probing if possible by returning early if we know for sure that the word is not in the text.

Of course, since we are utilizing both structures for our lookups, our insertion function must also use them both:

```
def text_insert(text, word):
    filter_insert(text.filter, word)
    table_insert(text.table, word)
    text.words++
```

Now that we have the protocols for our text insertion and checking, we will need to calculate the distance between two texts with the `text_dist` function. We will need to account for all three `Metric` types, which will be done with a switch case:

```
def text_dist(text1, text2, metric):
    switch (metric):
        case EUCLIDEAN:
            return euclidean(text1, text2)
        case MANHATTAN:
            return manhattan(text1, text2)
        case COSINE:
            return cosine(text1, text2)
```

For finding the actual distances between two texts, we will need to normalize the frequencies of the words, which we will do with the `text_frequency` function:

```
def text_frequency(text, word):
    if (!text_contains(text, word)):
        return 0
    node = ht_lookup(text.table, word)
    return node == None ? 0 : (node.count / text.words)
```

When calculating the distances between the texts, we will need to make sure we consider the words in both texts, but not overcount the words that appear in both:

```
def euclidean(text1, text2):
    total = 0
    for word in text1:
        f1 = text_frequency(text1, word)
        f2 = text_frequency(text2, word)
        total += (f1 - f2) ** 2
    for word in text2:
        if (text_contains(text1, word)):
            continue
        total += text_frequency(text2, word) ** 2
    return sqrt(total)
```

In this code, `f2` may be 0 in the case that `text2` does not contain the given word. By using `continue` in the second loop, we skip over words that we have already considered in the first loop. The other distance-calculating functions will have similar setups, just different operations.

Once we calculate the distances of the texts, we will need to sort them using a priority queue. Our priority queue will be similar to other assignments, except that instead of receiving a struct, we will receive information separately, package it into a struct, and then add it to the queue. We will have an `Entry` struct that serves this purpose, storing the author and distance. When we enqueue, we will simply create a new `Entry` with the specified parameters and add it to the priority queue, and do the opposite when dequeueing:

```
def enqueue(queue, author, distance):
    queue[size] = new Entry(author, distance)
    size++

def dequeue(queue):
    return queue[0].author, queue[0].distance
```

This allows us to keep the code much more organized and streamlined, simply comparing the distance of the when sorting instead of having two separate arrays for each half of the pair.