PA1: C++ Implementation of a Bloom Filter

Michael Glushchenko

February 9, 2022

1 Brief Implementation Explanation

1.1 Converting Strings to Integers

To convert a string to an integer, we process the string, character by character. First, the character's ASCII value is found; this value is then multiplied by $31^{\text{index of the character}}$. This ensures that both the character symbol and the symbol's position in the string affect that integer value of that string. Here's the function that implements this logic:

```cpp
unsigned int strToInt(std::string element) {
    unsigned int result = 0;
    for (unsigned int i = 0; i < element.length(); i++)
        result += element[i] * pow(31, i);
    return result;
}
```

The "testHashTable.cpp" file ensures that this function indeed gives different integer values to "ab" and "ba" and "aabb" and "bbaa".

1.2 Choosing the Size of the Hash Table

First, we implement the following two functions:

```cpp
int nextPrime(int n);
int isPrime(int n);
```

These functions help us find a prime number close to some integer n. In the beginning, we choose n to be the square root of the bloom filter size, and our hash table size to be nextPrime(n). The HashTable class keeps track of its size, and the number of elements that have been inserted thus far. Every time we insert a new element into the HashTable, we check the load factor; whenever it's more than 70%, we call the following function:

```cpp
int HashTable::resizeTable(std::string element, int index) const;
```

Here, we double the current size of the hash table, and find the closest prime that's greater than this new, doubled, size. We then rehash all elements into this new table, and delete the old one.

1.3 BloomFilter Hash Function Family

```
int BloomFilter::hash(std::string element, int index) const {
    unsigned int elementRep = strToInt(element);
    return ((elementRep ^ hashParameters[index])
                        % this->hashingPrimeNum) % this->size;
}
```

hashParameters is a vector of integers containing the random seeds of the m hash functions being used for the given bloom filter of size n. Each random seed is obtained via the following line of code:

```
int seed = std::rand() % n
```

Here, n is the size of the bloom filter. These seeds are generated when the executable PA1.out is created. Once they're generated, here's how we use a hash function with the random seed at index 2:

1. convert the element we want to insert to an integer.
2. obtain the random seed, located at index 2 of hashParameters.
3. perform a bitwise-XOR operation between the element and the random seed.
4. take the mod of the value obtained in (2) with respect to hashingPrimeNum
    - here, hashingPrimeNum is obtained by nextPrime(bloomFilterSize).
    - this ensures a uniform distribution among the buckets of the HashTable.
5. finally, take the mod of the value obtained in (3) with respect to the size of the bloom filter.
    - this ensures we have the amount of buckets that we want.

Note that each random seed in hashParameters is independently generated via a separate call to generateHashParameter().

1.4 BloomFilter Itself

For the actual bloom filter, we use a std::vector<bool>, since C++ optimizes this to behave more like an array of bits than a C++ container, with each bool element only taking a single bit of space (rather than the usual 1 byte a bool takes up).

2 Testing Different Values of c and d

To test how the scale factors c and d affect the bloom filter's false positive probability, we will create two charts:

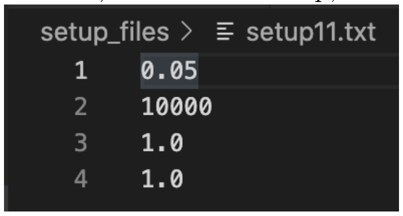
1. varying c values vs false positive probability, d remains constant.
2. varying d values vs false positive probability, c remains constant.

This way, we can see the effect of each scale factor individually.

2.1 The Setup

2.1.1 Setup Files

To start, we create 13 setup, each looking similar to this:

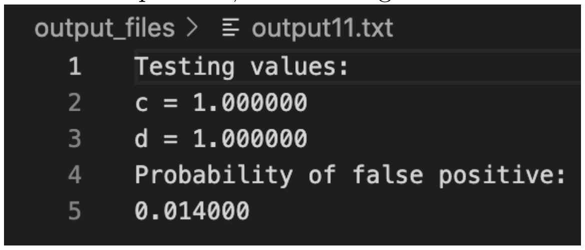


```
0.05
10000
1.0
1.0
```

The values tested for c and d are 1.0, 1.25, 1.5, 2.0, 2.5, 5.0, and 10.0. The value 10.0 doesn't make much sense to use as a test for d, since our setup results in a bloom filter that uses about 5-7 hash functions (so any factor over 7 would have similar effects on the false positive probability). Once we have the setup files, we run the makeTestFile() function, located in the createTestFileFramework.cpp, on every setup file we just created.

2.1.2 Output Files

As a result of running the above function on the setup files, we now get a folder with 13 output files, each looking similar to this:

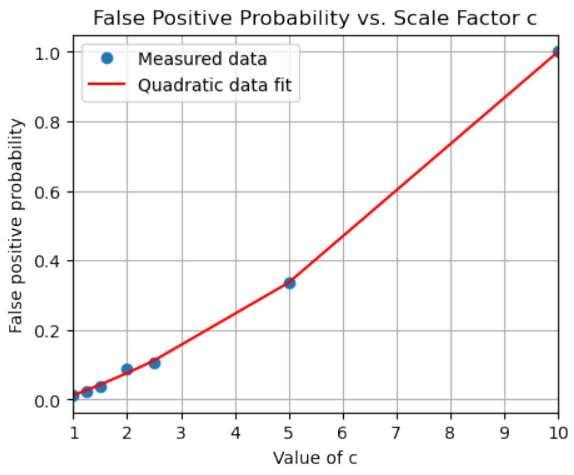


```
Testing values:
c = 1.000000
d = 1.000000
Probability of false positive:
0.014000
```

2.2 Charting and Regression Fit of the Data

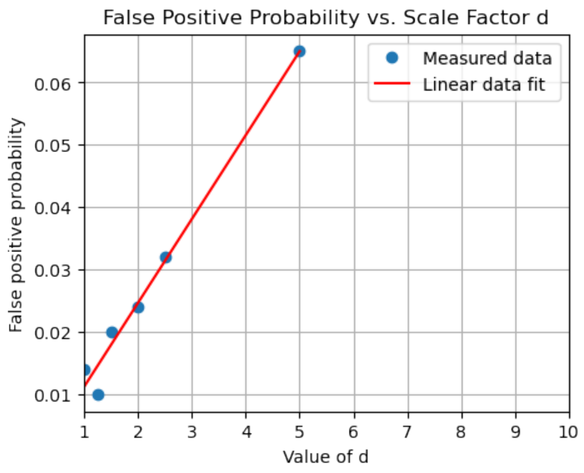
After the 13 output files were generated, we use the Python matplotlib library to create a graphical representation of the data we just collected, and plot a regression fit over that data using the lstsq function of the numpy.linalg library.

2.2.1 Constant d, Varying c



Granted we didn't collect a large amount of data points (mainly because this wasn't the main point of the assignment), we can see clear correlation between increasing the scale factor for the size of the bloom filter (i.e., decreasing the size of the bloom filter), and having a higher false positive probability for that bloom filter. Since all other factors were kept constant, the chart is enough to show a general relation between the scale factor c and the false positive probability of a search in that bloom filter.

2.2.2 Constant c, Varying d



Similar to the part above, we keep all factors constant, and change the scale factor of the number of hash functions, to obtain the positive relation we see here. The relation between the scale factor d and false positive probability seems to be linear, rather than quadratic.

2.3 Conclusion

Overall, we see that as our bloom filter size decreases (the scale factor c increases), the false positive probability for a search in the bloom filter goes up in a quadratic fashion. We also see that as the number of hash functions we use decreases, the false positive probability seems to increase in a linear fashion. One of the more major results we can see from the two charts above, is that c is much more significant than d when it comes to their effects on the false positive probability of a search.