Analysis – The Great Firewall of Santa Cruz

Daniel Chang

Fall 2021

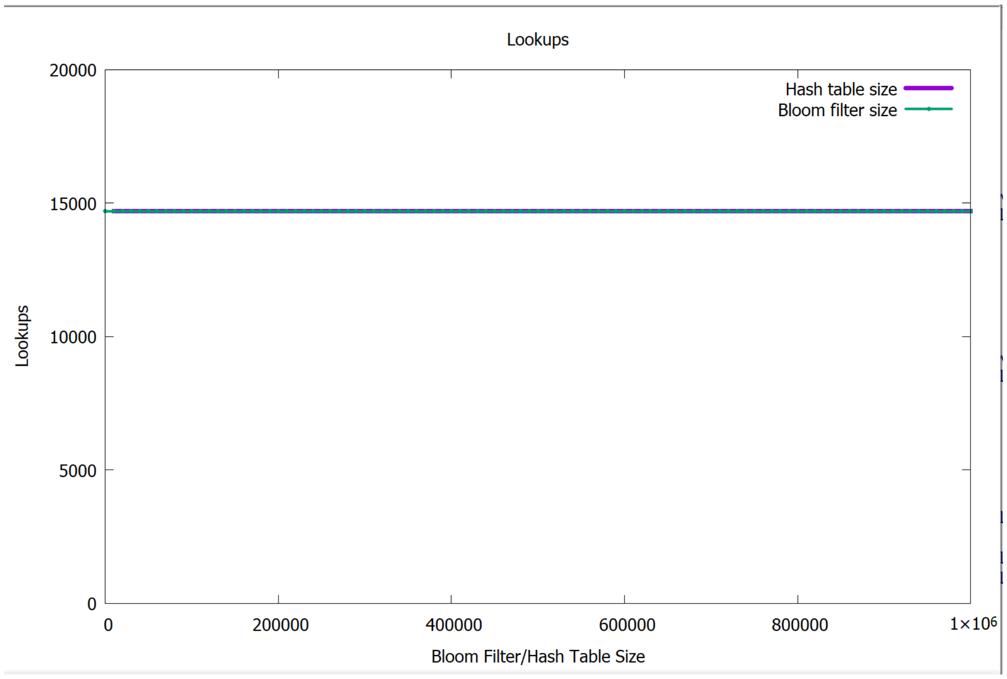
1 Introduction

In this assignment we code a censorship algorithm. We compare input to a list of banned words to determine if each word is legal or not. The two main data structures for this algorithm are a Hash Table and Bloom Filter. The goal of this paper is to determine how the size of the Hash Table and Bloom Filter affect the performance of the algorithm. We look at five metrics: number of Binary Search Tree branches traversed, average BST height, average BST size, number of Hash Table lookups, and percent load.

2 Analysis

In this section we will look at how Hash Table size and Bloom Filter size affect the efficiency of the GPRSC.

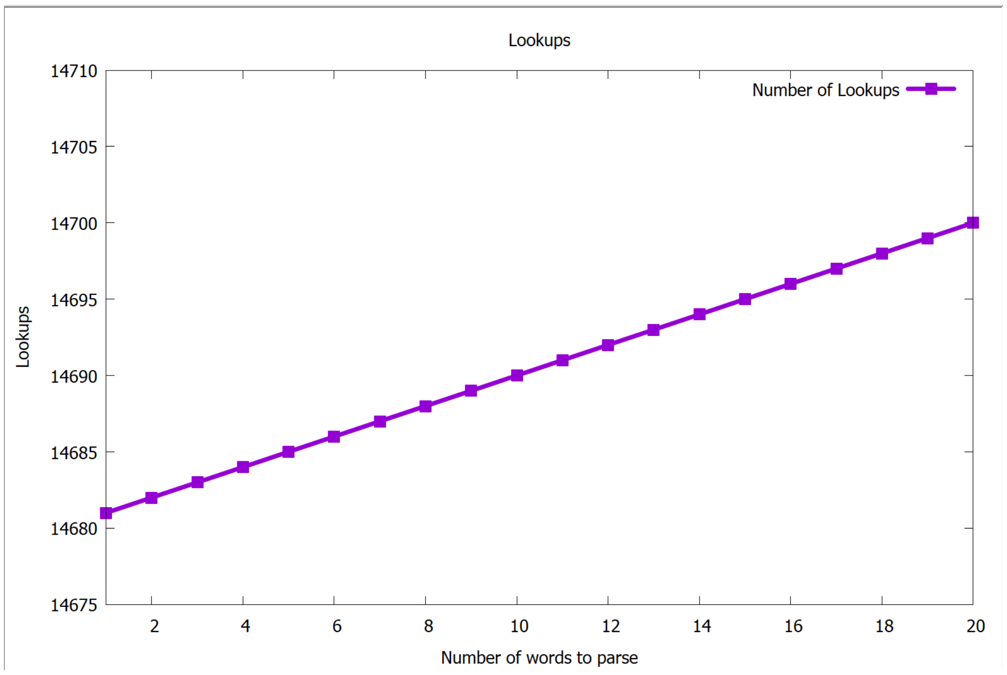
First, let's look at how the number of lookups in the Hash Table changes.



From this graph, it appears neither the Hash Table size nor the Bloom Filter size seem to affect the number of lookups. Why is this? The lookups is defined as the number of calls to `ht_lookup()` and `ht_insert()`. The number of calls to `ht_lookup()` is defined by the number of words we parse from stdin, while the number of calls to `ht_insert()` is defined as the number of badspeak and oldspeak words, as we insert each of these once into the Hash Table, regardless of its size.

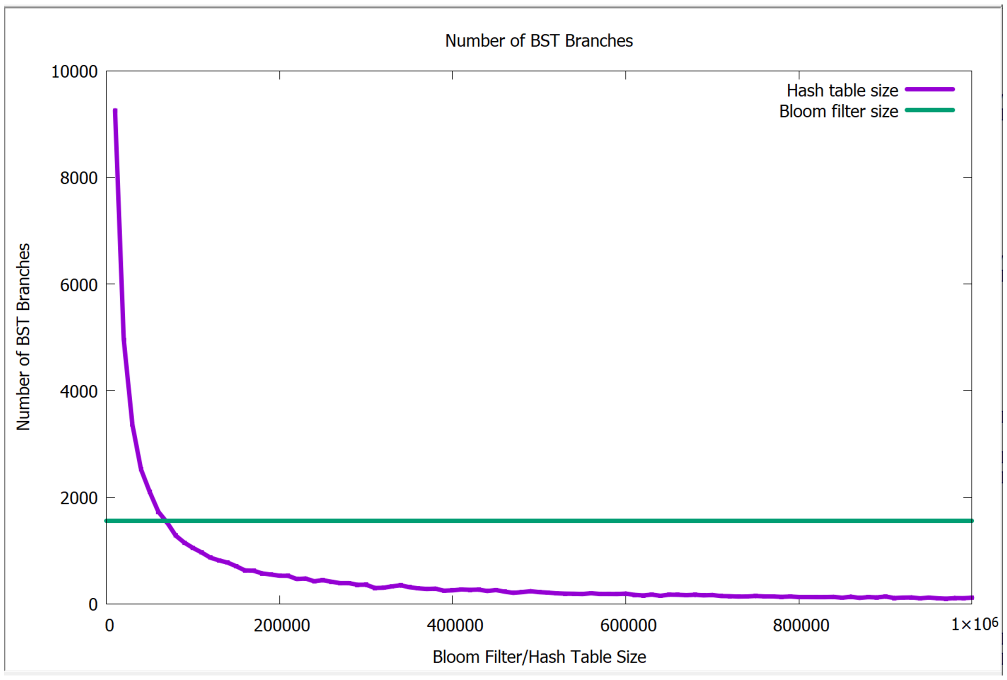
There is something else to consider, however. When the Bloom Filter encounters a false positive, then we get an extra call to `ht_lookup()`, because we have to check a legal word in the hash table. Therefore, when the Bloom Filter size decreases, we should see an increase in false positives, which in turn increases the number of lookups. In the graph above, too few words were parsed in `stdin` to see a significant difference.

So, what else does affect the number of lookups? As we saw earlier, it seems to depend on the number of words in `badspeak.txt`, `oldspeak.txt`, and the number of words parsed from `stdin`. To test this, let's vary the number of words in `stdin`:



Since each extra word in `stdin` is one extra call to `ht_insert()`, it makes sense that this graph is a line.

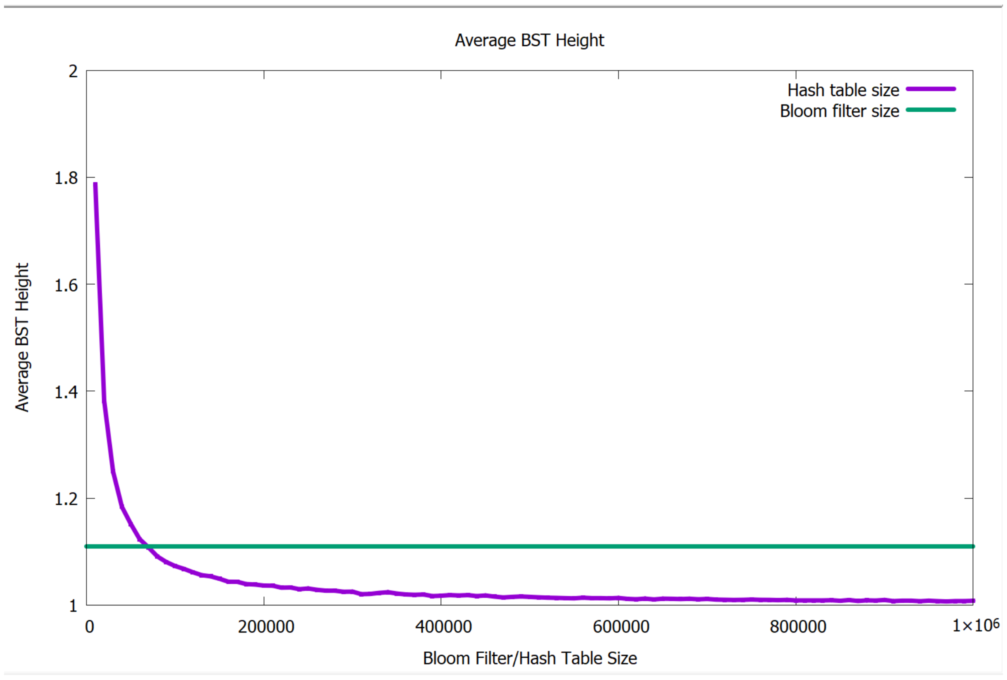
Next, let's look at how Hash Table size and Bloom Filter size affect the number of branches traversed in the Hash Table.

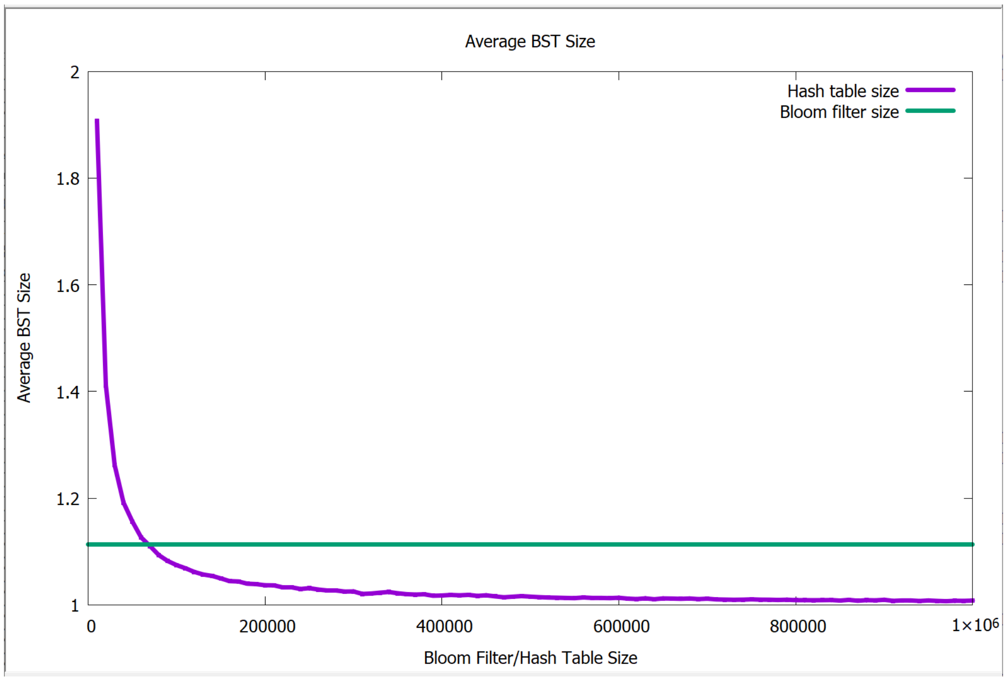


The size of the Bloom Filter does not affect the number of traversed branches in the graph above. In the case we had a small Bloom Filter and a large number of words to parse, we might see the number of traversed branches decrease as Bloom Filter size increases because false positives have an extra call to `bst_find()`.

As Hash Table size increases, the number of traversed branches decreases. This is because by increasing the number of binary search trees, the average size of the binary search trees will decrease, meaning there will be fewer branches to traverse. Also, the logarithmic nature of the graph can easily be explained as well. Since the number of traversed branches is heavily correlated with the average binary search tree height, and average binary search tree height is logarithmic, then it makes sense that the number of traversed branches would be logarithmic as hash table size increases.

We also look at average Binary Search Tree height and average Binary Search Tree size below:

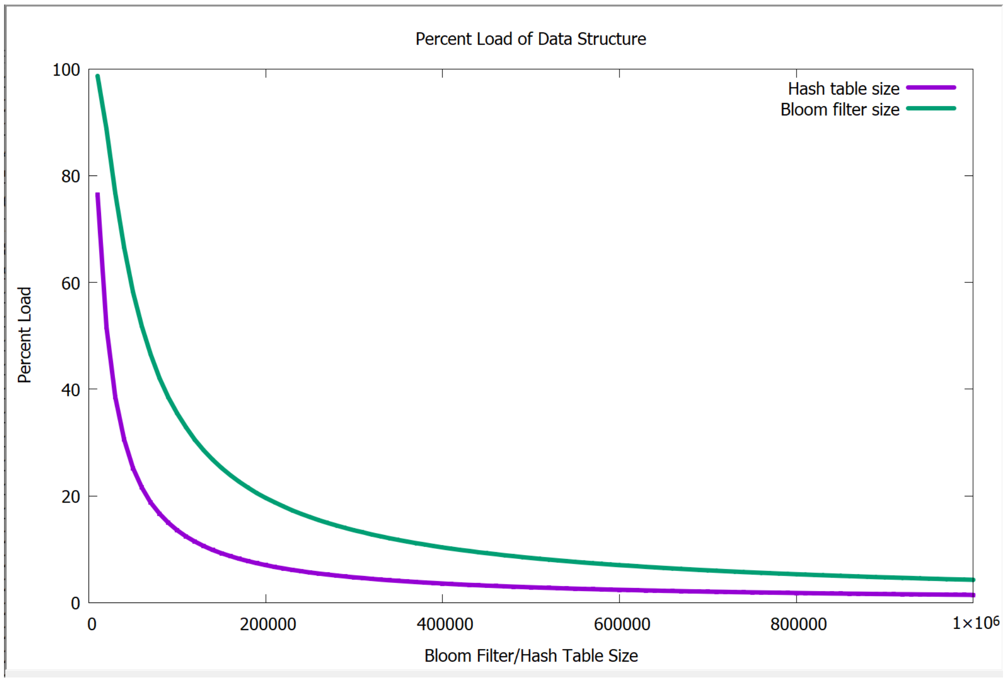




In this case, the Bloom Filter size actually DOES NOT affect average BST height and average BST height, unlike the previous case. This is because false positives from the Bloom Filter only create extra calls to `bst_find()`, but they don't actually insert extra nodes into the binary trees.

The Hash Table graph for Binary Search Tree size and height is very similar to the number of traversed branches. This initially seems wrong because Binary Search Tree height should increase logarithmically while size should increase linearly. However, we see that the average Binary Search Tree size is 2. For such small Binary Search Trees, this difference is barely noticeable. However, it can be noticed in the above graphs where the height graph's maximum is around 1.8 and the size graph's maximum is around 1.9.

Finally, we look at percent load of the Hash Table and Bloom Filter at various sizes.



It makes sense that percent load would decrease logarithmically for both the Hash Table and Bloom Filter because as the size increases, any additional node is less and less likely to either find an empty BST in the Hash Table or find a 0 bit in the Bloom Filter. This graph also gives us some insight into how the Bloom Filter might affect the Binary Search Tree metrics because false positive rate is proportional to percent load.

3 Conclusion

In this paper we analyzed how varying sizes of Hash Tables and Bloom Filters affected the efficiency of the Firewall algorithm. We measure the performance of the algorithm based on how many Hash Table lookups and Binary Search Tree branch traversals are needed to check a set of words. In each scenario, the performance of the algorithm improves as the Hash Table and Bloom Filter sizes increase. Ultimately, we want to find the right balance between minimizing the memory used by the two data structures and the performance. Finally, something not covered in this paper is how the algorithm performs on lists of up to hundreds of thousands of words. In such a scenario, Bloom Filter size would have a much greater impact on the performance of the algorithm due to false positives.