Kyven Wang

kwang126@ucsc.edu

4 June 2021

CSE13S Spring 2021

Assignment 7: The Great Firewall of Santa Cruz: Bloom Filters, Linked Lists and Hash Tables

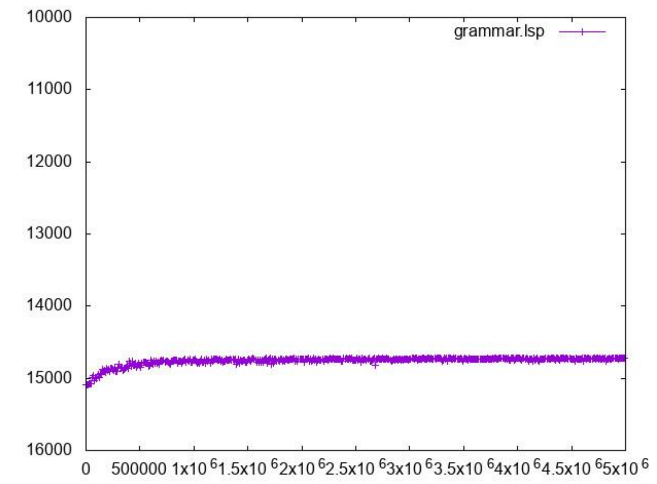
Writeup

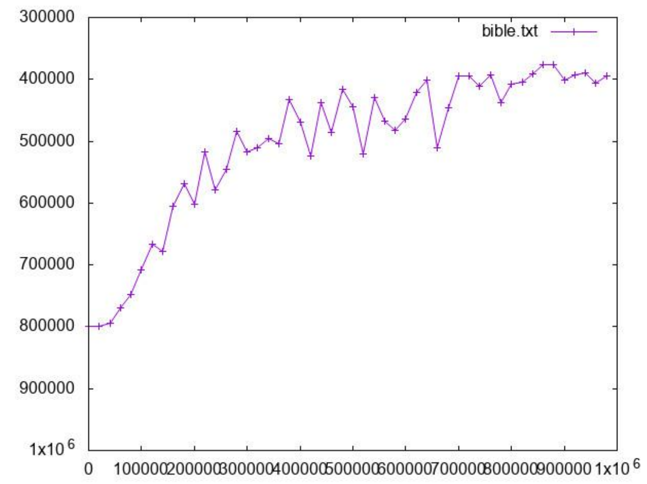
In this assignment, we had to create a language filter that takes in input and detects any badspeak (strictly forbidden word) and oldspeak (has a replacement newspeak word). Depending on what is found, a different notification message will be printed and the transgressions printed underneath. If nothing is found, nothing is printed. While the things we had to implement for this assignment felt pretty familiar (bit vectors, nodes, and implementing things using those), there were still things that I gained valuable experience and knowledge about. The most significant of these things was the Bloom filter, hash table, and regular expressions. Implementing the linked list was also a valuable experience, but it somehow didn't feel as new to me. This is because I recalled Professor Long's lecture where he discussed how moving things around in linked lists works. Understanding that, the linked list was a very easy thing to grasp and implement. On the other hand, Bloom filters and hash tables were things I had less of an easy time grasping before this lab. What these two things have in common are they utilize a cipher to encode a word and use that encoding to retain information about that word.

In the case of the Bloom filter, it uses the encoding to set certain bits, which can tell you one of two things: if a word has definitely not been added (all corresponding bits are 0) or if a word has possibly been added (any of the corresponding bits are 1). In the case of a hash table, it uses the encoding to determine an index of a linked list to insert the word into. Using these two things, you can make a pretty fast language filter. A lookup is not performed on the hash table at all if the Bloom filter probe returns a false. Even if a lookup is performed, it is very fast because the hash table is able to jump to a list containing the word, shortening the lookup process. Looking back on it, hash tables shouldn't be alien to me at all considering my experience in a previous lab using a look-up table, which, conceptually, is very similar to a hash table (at least in my eyes).

After everything was implemented, I still had one more thing to do: the regular expression. I have actually had some experience with regular expressions in the past from a high school computer science class, so this was pretty familiar to me. However, back then I understood a lot less than I do now, which means that this time I was able to look deeper into regular expressions and understand them better. I have heard many moans and groans about regular expressions, but really all you need to do is read and understand the rules and it becomes very simple to understand, yet powerful. Most of my trouble with this actually came from trying to use the regex library in C. Neglecting to go back and read the assignment document, I simply plugged my regular expression into the regex compiler function thinking it would work. What I didn't realize is that, for our purposes, I needed to set the third arg as REG_EXTENDED. Again, another lesson learned about reading.

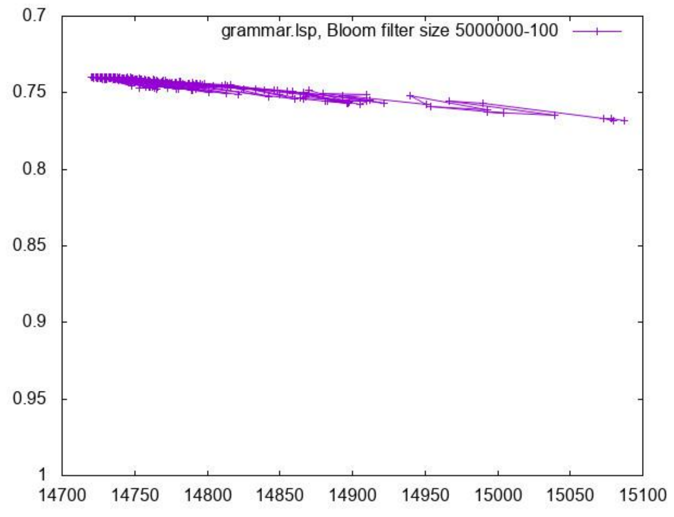
Now onto the statistics, starting with Bloom filter size vs number of seeks. x-axis is Bloom filter size, y-axis is number of seeks.

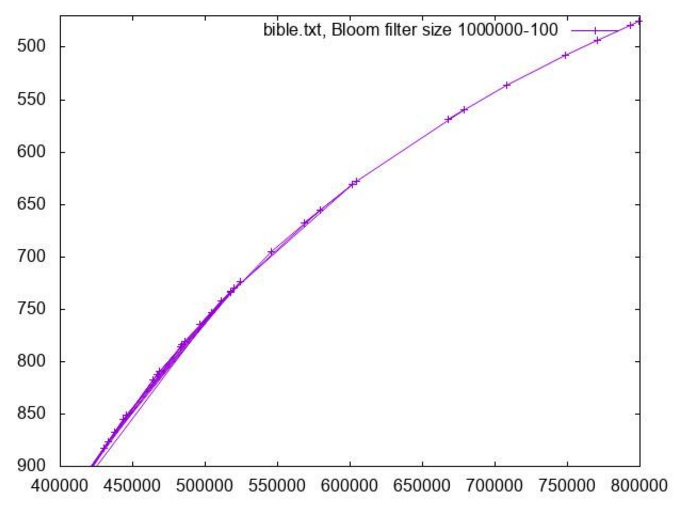




The file on the left is a relatively small one, while the one on the right is a relatively large one. We can see from this that our results can vary a lot depending on how much is actually put into the bloom filter. While the graph on the left is relatively uninteresting, the one on the right gives us some insight. As the Bloom filter size is increased, we can see that the amount of seeks decreases pretty drastically (the y-axis is decreasing from bottom to top, sorry). This is probably due to the fact that, as the Bloom filter increases in size, the chance of a false-positive decreases more and more. As the width increases, there is more and more space for bits such that a given word will be more and more likely to have its own bits without colliding with another word's bits.

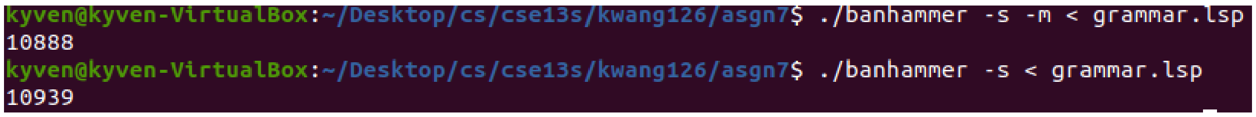
Next, we look at seek vs. average seek length. X-axis is seeks, y-axis is avg seek length.



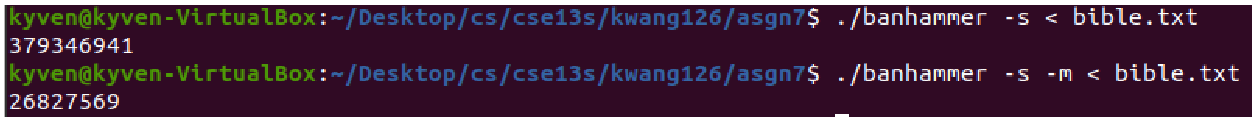


As we go from left to right in the left, the Bloom filter size is being decreased from 5000000 to 100, while in the graph on the right it is going from 1000000 to 100. Again, the file on the left is small, the file on the right is large. We can see from this that, again, the small file does not provide as much info as the large one. Looking at the large file, we can see a pretty smooth curve that indicates that, as the number of seeks increases, the average seek length decreases. I am actually unsure of why this is, but I can make a guess. It may be because, in a large Bloom filter, there are a lot less false-positives. Therefore, there are less unnecessary seeks, and so linked lists are not fully traversed unnecessarily as much. In a small Bloom filter, there are likely many false-positives, resulting in a lot more unnecessary list traversals and therefore inflated seek lengths.

Next, we can take a look at number of links with mtf on vs off.



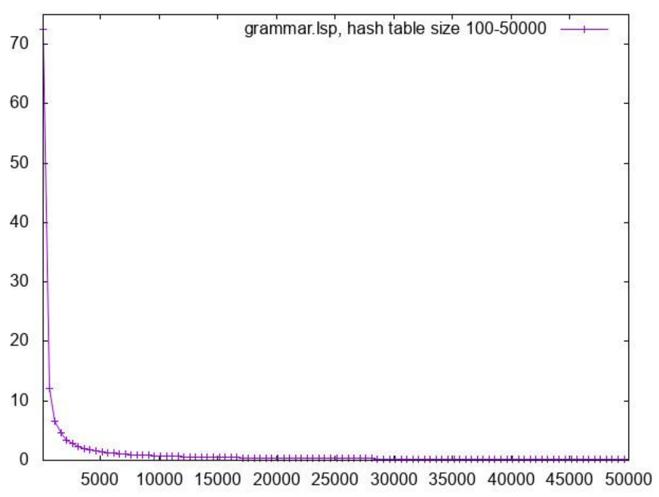
```
kyven@kyven-VirtualBox:~/Desktop/cs/cse13s/kwang126/asgn7$ ./banhammer -s -m < grammar.lsp
10888
kyven@kyven-VirtualBox:~/Desktop/cs/cse13s/kwang126/asgn7$ ./banhammer -s < grammar.lsp
10939
```

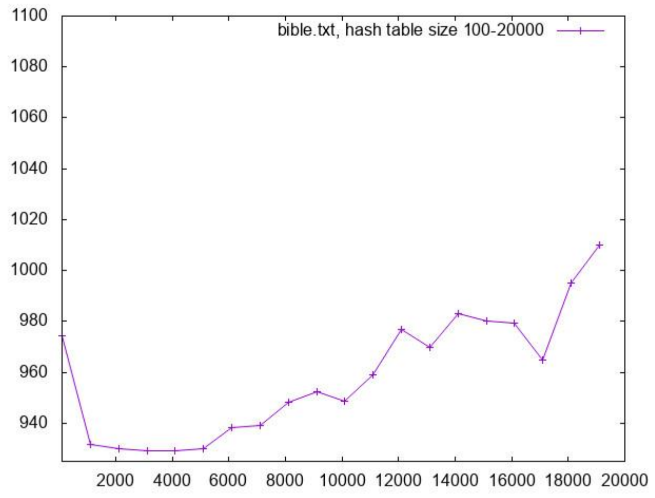


```
kyven@kyven-VirtualBox:~/Desktop/cs/cse13s/kwang126/asgn7$ ./banhammer -s < bible.txt
379346941
kyven@kyven-VirtualBox:~/Desktop/cs/cse13s/kwang126/asgn7$ ./banhammer -s -m < bible.txt
26827569
```

The commands with “-m” in them are running the program with mtf enabled. The program output is the total amount of links performed. We can see that, with the move-to-front rule enabled, the amount of links is decreased. This especially makes a difference in bible.txt, a relatively large file. We can see that, without mtf, there are ~380 million links, while in the one with mtf there are ~27 million links. I believe this huge difference is caused by the fact that words that are more commonly looked up will naturally “float” up to the top of the linked list, while more uncommonly looked up words naturally settle down to the bottom. Therefore, the more commonly looked up words will take a lot less links to arrive at and result in a lower total number of links.

Lastly, we look at hash table size vs. average seek length. X-axis is hash table size, y-axis is average seek length.





A relatively small file is on the left, a relatively large file is on the right. We can actually derive a pattern from both of these this time, with the graph of the smaller file making this pattern more obvious. With a very small hash table, we can see that the average seek length is larger than if the hash table were a little bit bigger. Then, after a certain size has been reached, we can see that in the case of the larger file the average seek length begins coming back up. I think this indicates a kind of "sweet spot" for hash table size. When it's just right, the seek lengths can be very small. However, too small or too large and the seek lengths begin rising. I think the patterns we see in the two graphs are also affected by the amount of words we insert into a hash table, and also the amount of unique words. When there are little duplicates, or little words in general, in the case of the graph on the left, the hash table can have a unique linked list for most words, making a larger hash table more beneficial to seek length, since in most cases the seek length will be very short. This seemingly only applies to small files with unique words, so in general I think this “sweet spot” that can be found should be the one that is used for the most optimal seek lengths.