Fabrice Kurmann
fkurmann@ucsc.edu
CSE 13s Spring 2021

Writeup for Assignment 7, The Great Firewall of Santa Cruz

In this assignment, I learned about the use of Bloom filters to provide quick but somewhat incomplete answers to true or false questions, in this case regarding the membership of a certain word from an input string in a list of either badspeak - banned words, or oldspeak, words which have modern translations to be used instead.

The Bloom filter is able to assure that an input word is not on one of the lists, however, due to the nature of hash collisions when there are chances of false positives. To address this concern, a somewhat less compact but more secure checking system was implemented alongside the bloom filter, a hash table containing linked lists at each index.

The advantage of having an entire linked list at each index which you can hash to, as opposed to just a single bit in the Bloom filter is that in case of collisions, both input values can be stored, then simply accessed by parsing the linked list.

In this assignment, a few features of the Bloom filter and hash table/linked list were left to be set by the user, these features all have the potential of changing the speed and accuracy of the word checking. This writeup will therefore examine first, the importance of the size of a bloom filter, second, the importance of size of the hash table, and lastly, the potential benefits of moving commonly searched for linked list values to the front of the list, in order to make them quicker to find in future passes.

My Test File:
I created two test files, both containing around 5000 words. The first text was a regular article from the New York Times, to represent a sample of real world writing. For the second text, to try and give a helping hand to the move to front function I'll be explaining in section three, I created a text with far lower entropy than the NYT article, where moving an word to the front of the linked list would have a larger than usual benefit sice, due to the low entropy, that word is likely to show up again sooner.
Below is a table showing the entropy levels for these files in their encoded and natural states.

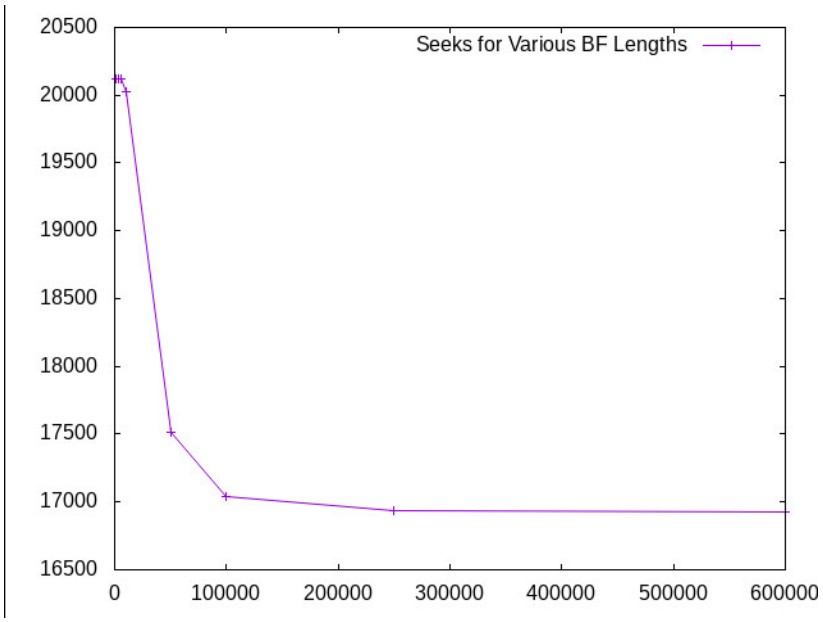
As it turned out, all graphs for both these files looked almost exactly the same and there was no meaningful difference between them, thus, I will focus on my test file from NYT articles for simplicity and ease of understanding.

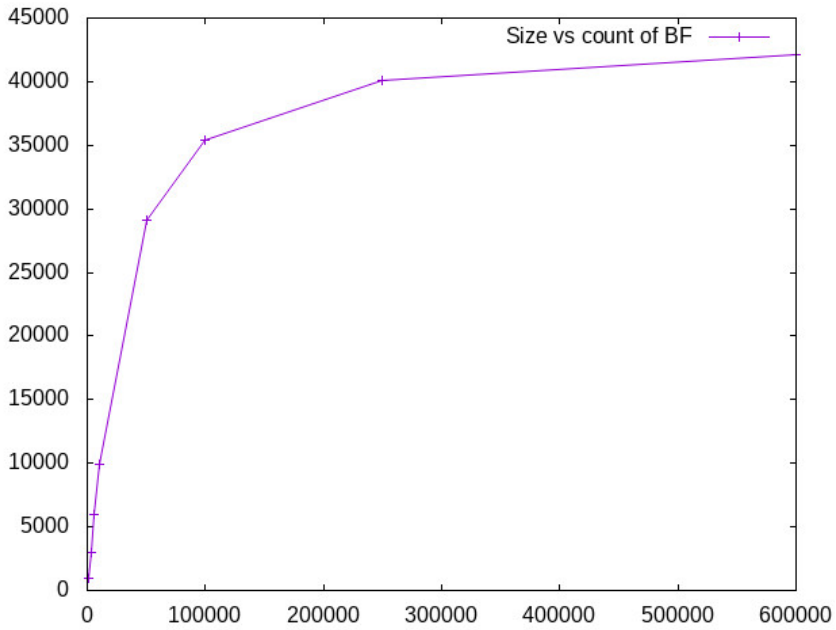
Altering the Size of the Bloom Filter

A larger Bloom filter will contain fewer hash collisions since there are more available hash values to map strings to. For example, a Bloom filter with only 10 values will almost certainly have collisions with even a short input text since each string will be hashed to a value between 0 and 9. In fact, if there are more than 10 different strings, it is certain that, even in the best case, there will be a hash collision.

Expanding the Bloom filter is also a tempting option because it is relatively cheap, you are simply making a longer bit vector, not something that is terribly expensive, especially compared to expanding the hash table, which may have to store an entire linked list at each index. Finally, getting a definitive answer (no) from the bit vector saves the program from having to check the hash table, which may involve the rather slow process of parsing a linked list.

To understand the benefits and find a potential sweet spot Bloom filter length for my particular test texts, I ran the banhammer program on my test texts with multiple different Bloom filter sizes:





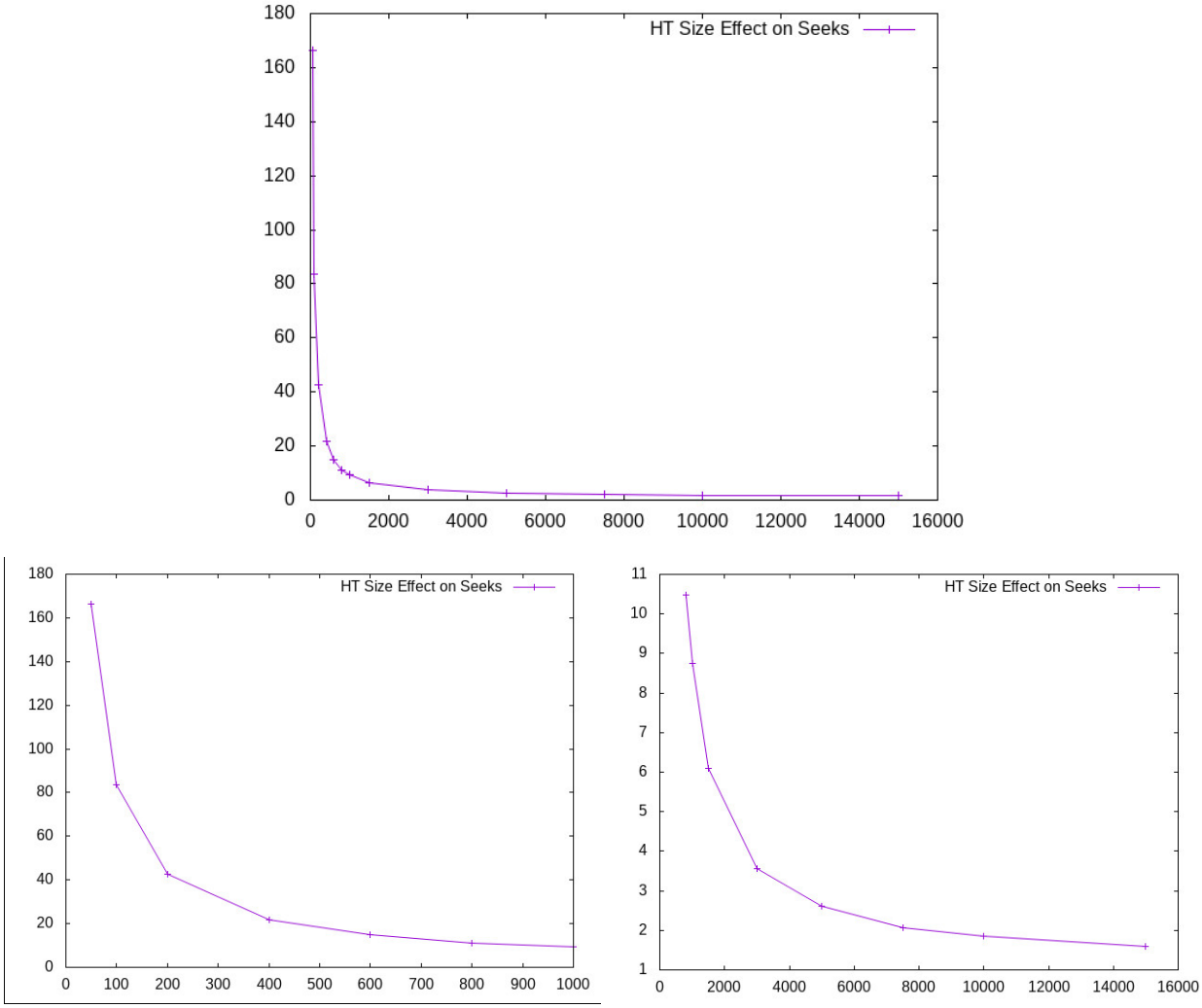
The above graph shows the number of seeks conducted based on Bloom Filter size. As expected, a larger bloom filter will trigger less positive or uncertain answers, causing less need for additional seeks caused by having to resort to checking the hash table. Also clear is that the number of seeks for this length of text file levels off at a Bloom filter size of around 200,000. Since seeks don't decrease further after this, there is little value in storing a larger Bloom Filter.

The graph below shows the ratio of Bloom filter size (x axis) and Bloom filter count (y axis). This tells a similar story to the above graph, up to a size of 200,000 the count of the bloom filter, or the number of bits that are set goes up quite rapidly. However, that number seems to slow down in it's increase as it approaches an asymptote of around 45,000. A Bloom filter of 200,000 roughly 4 times the counted values seems to be enough to assure that most of the time, hashed values don't collide onto the same index, causing additional seeks.

Altering the Size of the Hash Table

A larger hash table provides a utility as does a larger Bloom filter. Since for every hash collision, the length of the linked list containing equally hashed strings will grow linearly, so will the lookup times in case the Bloom filter also gives an inconclusive result. The benefit of a larger hash table is therefore shorter linked lists at each index. This means, less links to traverse and shorter lookup times. In terms of a trade off, a larger hash table must contain more linked lists, however, they will be shorter each. I am guessing that the prior of these two facts is not completely outweighed by the latter since expanding a linked list is less memory than simply adding a single link, but it is worth mentioning regardless.

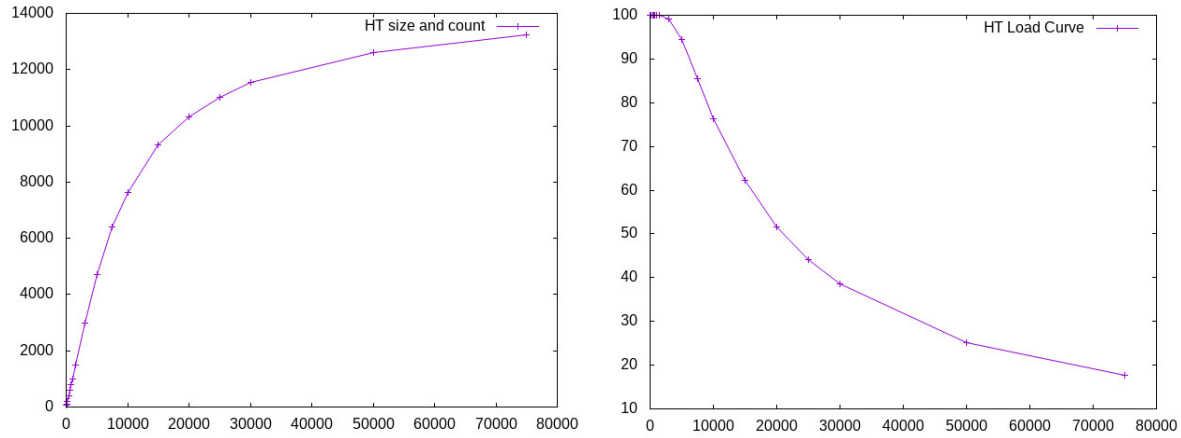
To test the results of a larger hash table on search times, I tested by input text with three different hash table sizes. To assure that I would see a considerable difference, I also opted to lowball my Bloom filter size to allow for more opportunities to parse the linked lists.



Above, the number of links per seek (average seek length) for hash tables of different sizes. Lower left, is the above graph zoomed in on lower X values, the lower right is the above graph zoomed in on higher x values.

As expected, a larger hash table will result in shorter average seek lengths since linked lists will be shorter. How quickly links go down was surprising to me as the decline seems to be exponential, indicated by the identical shapes of the lower right and left graphs, both segments of the upper graph. Clearly, even a small increase in hash table size when the table is very small has a very beneficial effect on the average seek length. Importantly, this benefit could become even more important with a yet lower Bloom filter size.

In the next section, I will compare these results with those where move to front is toggled for linked lists to see if that functionality, which does have a considerable cost due to the moving of linked list elements, is considerably quicker in average seek length.



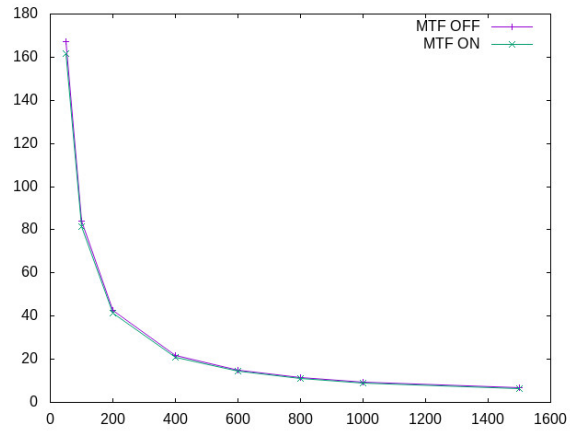
The left graph showing hash table size (x axis) and count (y axis) tells a similar story to that for Bloom Table size and count: once size is at around 4 times the value of count, there is enough space to mitigate most hash collisions (in this case a size of 45,000 for a count with an asymptote near 13,000) and additional size does little to increase count.

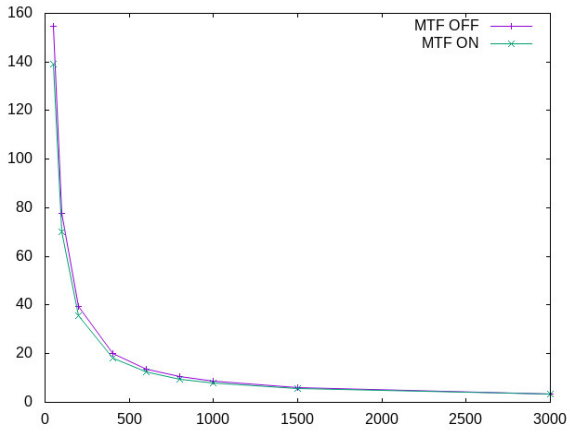
I decided to also graph hash table load (right), to back up this trend. Hash table load simply tracks the percentage of hash table indexes used given the hash table size. Unsurprisingly, this starts at 100% when a hash table contains fewer indexes than there are different words in the sets badspeak and goodspeak. This load begins to drop as the hash table grows. This metric isn't as informative as the count vs size graphed above since load will continue to drop as size increases, even if it is dropping with no benefit to the average seek length.

Effects of Toggling Move to Front

Following the principle of whatever just happened is likely to happen again, there may be benefit in always moving a searched for linked list value to the front of the list. This is in anticipation of that particular link, corresponding to a string in the input, to be searched for again soon, since that exact word occurs only a few spaces later again in the test text. Thanks to being close to the front of the linked list, the word will be quicker to find the next time around.
Moving words to the front is not free, it involves removing a link and inserting it again at the front of the list. However, this one time cost may be worth it if over the course of 1000 searches, a word is found according to the best case scenario of linear search instead of the worst case scenario.

To test, I tested my input texts each with MTF on and off. Additionally, to force the program to have to search more often, I lowballed the Bloom filter size as in the tests above. Finally, to attempt to expand the benefit of move to front, I also tested with a lowballed hash table size, where linked lists would be longer than each index.





For this third section I felt it was beneficial to include results from both of my test texts, the natural NYT article and the low entropy text I created using lots of words from both badspeak and oldspeak, often in close proximity to each other. When creating the second text file, I had the move to front algorithm in mind and wanted to build a case where it could really shine.

Given this context, I was rather surprised to see that the effects of move to front on the average seek length was very insignificant. In my normal text, where terms were often not repeated soon after each other, move to front provided practically no benefit to seek length, it did however also not make it worse. Given the added complexity of moving items in a linked list however, this program likely ran slower with MTF on.

With test text 2, a general trend seems to be that move to front provided an decrease in average seek length of around 10%. This number stayed relatively constant across multiple manipulations of input and seek lengths from 800 to only 1.5. 10% is significant, but not really that important when the seek length is only 1.5 to begin with. For this reason, I consider move to front to be helpful only when the Bloom filter and hash table size are strictly limited and linked lists will be very long.

Conclusion:

I really enjoyed learning about this simple yet effective way of checking for set membership using the combination of both the fast and often definitive Bloom filter backed up by a hash table of linked lists to fill in inconclusive Bloom filter results.

For my example texts, both around 5000 words and both very different in terms of entropy and variance of words and appearance and variance of badspeak and oldspeak words, a good

Bloom filter size would be around 200,000 bits. Meanwhile, a good hash table size is around 8000. Exceeding either of these values brought little extra efficiency and, especially in the case of the Bloom filter, only added storage for no benefit. Interestingly, both these values are lower than the default for the program as specified by the assignment handout. This makes sense as these ideal numbers do seem to be highly dependent on the oldspeak and badspeak lists, the longer these lists, the more Bloom filter storage is likely needed, same principle for hash filter storage.