Simon Lee
CSE 13S

WRITEUP

Introduction:

Alas, we have made it to the final write up of the quarter. And in this assignment we are the newly elected leader of the Glorious People's Republic of Santa Cruz (GPRSC), and we have to filter out the messages being sent out by our citizens. First we should define the three types of "speaks" those being:

Badspeak - forbidden words that will get you in trouble
***Oldspeak* -** words that are outdated and must be translated into its newspeak form
Newspeak - the translated oldspeak word which is now up to date

Why this is important is because we want to monitor and get rid of all the vile thoughts that our citizens are thinking and make sure they understand who's in charge now. And the way we filter out the badspeak and oldspeak is through building a firewall which will highlight and find these words using bloom filters, hash tables, and linked lists and send automated messages letting you know what's up.

Initial thoughts before Experiments:

The Primary tests we will run in this writeup all have to do with the relationship within linked lists and bloom filters. For example, does the false-positive rate in a bloom filter have the same or an increased/decreased rate as more elements are inserted and vice versa. The same can be said of the mtf or the move to front feature we have in our program and how that may affect the overall time complexities of a given test file. There are many things that we can test and how it can affect performance and I will be demonstrating and showing you some analysis and proof of how things may have been affected.

Time complexities of Linked Lists

One of the things we will pay close attention to while we experiment is the time complexity of a linked list and how it may change with the mtf (move to front) variable that we have previously talked about. The typical time complexity of a doubly linked list in terms of insertion and deletion is O(1). O(1) also known as constant time means that it is very quick and takes the same amount of time to execute every single time the program is executed. When it comes to accessing and searching for a specific element it takes O(n) time or linear time. Both these time complexities are very quick and it makes sense why we are using this ADT for this particular assignment. Overall the time complexity for a doubly linked list is pretty quick and once I begin actually talking about the experiments I performed hopefully it can put to perspective a few more things that were talked about in here.

Time Complexities of Bloom Filters

The other time complexity that we will be messing around with in experiments are bloom filters. Because bloom filters work with both hashing and bits it takes O(k) time where *k* represents the hash function of *m* amount of bits. Because we mentioned hashing, the SPECK hash function that we were given has a time complexity that varies based on the variant of SPECK we are using. Based on the

assignment document where it says "The hash function `hash()` takes two parameters: a 128-bit salt passed in the form of an array of two `uint64_t` s, and a key to hash." I am assuming that we are using a SPECK 64/128 which has a time complexity of $2^{125.56}$. But because I am not sure which version of speck we are using, I will just write down all the complexities of the different variants that involve a 64 bit or 128 bit SPECK. Refer to the table below to see all the different variants and how their time complexities differ.

SPECK CIPHER VARIANT	TIME COMPLEXITY
SPECK 128/256	$2^{253.35}$
SPECK 128/192	$2^{189.35}$
SPECK 128/128	$2^{125.35}$
SPECK 64/128	$2^{125.56}$
SPECK 64/32	$2^{63.39}$

SOURCE: https://en.wikipedia.org/wiki/Speck_(cipher)

Although I do not know exactly how they derive these numbers, I do see some patterns. If we were to split these SPECK variants into a group where each started with the 128 bits and one that started with the 64 bits, we can see that the second number of bits is what is affecting the overall time complexity of the group. For example the 128/256 variant has the longest time complexity of the bunch versus the 128/128 which is significantly faster because it has less bits to process.

Experiments:

MTF:

Move to the front as you can presume will have some difference when it comes to the average seek length. The average seek length is calculated by $\frac{links}{seeks}$ and we know that the number of seeks will not change. I ran a test using the test.txt which contains the following text:

"Dancing in the moonlight. Everybody's dancing in the moonlight. Zodiac Monkey, I want to party like it's 1969. Oh yeah this is the most beautiful time of the year. Boom it is the year 1992 and it is the day gentleman. It is the end of your misery. You will now be in a shrine, and no longer be on earth. Pack you bags because we are going to Mars and it is the end of the world. Snake monkey rat gerbil."

After running it through the tests I got the following statistics:

Without mtf:
Seeks: 14726
Average seek length: 0.734619
Hash table load: 76.390000%
Bloom filter load: 4.082680%
With mtf:
Seeks: 14726
Average seek length: 0.730477
Hash table load: 76.390000%
Bloom filter load: 4.082680%

As you can see nothing changes amongst three of the 4 statistics. The hash table load, the bloom filter load and the seeks all remain the same, However as previously predicted the average seek length is lower, not by a lot but lower. The reason being is the amount of links it has to travel is far different then the others. If we have the option to opt a node to the front of the linked list after adding something to our hash table we will have to travel less when actually searching through the hash table/ linked list. Refer to the following image below to understand what I am trying to say

Image 1:

Image 2:

In this example because the alphabetical order for the letters x and y are x->y we will consider that to not be the mtf option or image 1. So we can see from our first example that the move to front flag was not put in the command line argument and therefore has the data corresponding to without mtf or a longer average seek length.

Now refer to image 2. Where y is in front of x showcasing that the mtf command line option was indeed prompted. If we were to seek or search up the letter y now in both images, we can see that in image 1 it takes 2 seeks versus the 1 seek from image 2. Obviously the amount of words inside the linked list in this example are very small but imagine this linked list filled with multiple nodes (words) with hundreds of words in every bit. The search time would obviously vary at this point and the overall performance would be optimized if the mtf option was prompted.

So what we can learn from this experiment is that the amount of links traveled in the linked list will be less if the move to front is opted into and affect the performance of the search.

Do Linked Lists get longer

This question is sort of a yes and no question because they do get longer but not in a way that you'd typically think. The benefits to using a linked list is that you never have to define a contiguous size when creating one unlike an array. And because linked lists are simply just nodes in between sentinel nodes (head and tail) that is where they expand. You can have many nodes or in this case words all be in one of the lists. And as previously mentioned in the previous experiment the only effect this has is the average lookup time. Seeks never seem to change, but the links do change. So like the previous experiment the linked lists do get longer and this will affect the average lookup time.

What is the relationship between bloom filter size and lookups

Something interesting I've noticed is that the bloom filter load and the hash table load never change. They are essentially constants in the fact that they remain the same value no matter the size. And to be honest if we think about it, this makes sense because we are reading in the same badspeak.txt and newspeak.txt. Unless these texts are changed the load percentage should remain the same. But speaking about the actual prompt does bloom filter size affect the lookups? To actually test this I made a new badspeak2.txt which contains the following text:

Badspeak2.txt

DISCLAIMER: I VERY MUCH LIKE EVERYONE ON THIS LIST. DON'T GET YOUR MIND TWISTED

Banana
Apple
Monkey
Darrell
Eugene
Sabrina
Cse13
Technology
A
The
Harsh
Sahiti
Genius
Gobbledygook
Doodle
Knowledge
Computer-science
funny
Cryptography
Dancing

After generating a list of 20 random words, I ran my tests once again and tried to see the output of my data on two files. The test.txt file from the previous example and the corpora/canterbury/lcet10.txt and here were the results I got:

test.txt	**corpora/canterbury/lcet10.txt**
badspeak2.txt Seeks: 335 Average seek length: 0.053731 Hash Table load: 2.970000% Bloom filter load: 0.085831%	***badspeak2.txt*** Seeks: 18505 Average seek length: 14.381788 Hash Table load: 2.970000% Bloom filter load: 0.085831%
badspeak.txt Seeks: 14726 Average seek length: 0.734619 Hash Table load: 76.389999% Bloom filter load: 4.082680%	***badspeak.txt*** Seeks: 37694 Average seek length: 18.737226 Hash Table load: 76.389999% Bloom filter load: 4.082680%

So what can we learn from this experiment? First off my hypothesis about the hash table load and bloom filter load was correct in that they are constant depending on the badspeak.txt file that they have to read in. You can see that relationship easily in the table above. And because badspeak.txt affects the overall amount of insertions into the hash table and bloom filters we can see the average seek length and the amount of seeks are overall affected. But what's interesting about the average seek times is that length of the file being read in also has a major effect. In my test.txt file it has 82 letters in the file, meanwhile corpora/canterbury/lcet10.txt has 62,323 words in the file. That is the primary difference between why the average seek lengths differ from the two tests. If anything, I think this test proves to show why bloom filters are so efficient. Being able to execute a program of size 62,323 in a matter of seconds shows truly how fast a O(k) time complexity truly is.

Conclusion

If there is anything I learned from this lab, it would be that bloom filters are extremely powerful in showing whether an element is a member of a set. After performing some tests on the bloom filters, the move to front ruling in the linked lists, and whether linked lists actually grow in size, I am more well versed in how truly powerful some of these ADT time complexities are. If there is anything I've learned from this class it has to be about making clean and efficient code. The reason being is that having useless junk just lowers the overall efficiency and performance of the given code. I've also learned that C is just quite the quick language in comparison to the only other one I know in Python. This lab was truly fun and I am astonished to come to the realization that this is the end of my writeup which signals the end of the assignment.

Epilogue (reflection)

As always after every class, I like to reflect on the quarter. I think reflection helps me remember the time I spent here so I truly like writing these to show my struggle and growth. Enjoy!

Coming into this class, I thought it was very well structured and what not but the policies did scare me to death. Things like “if you fail an assignment, you fail the class”. Also though, I’ve taken a C++ course not too long ago at my local community college over the summer, we never really had to do assignments or labs and it was done solely on lectures. It was very hard to actually learn code this way so coming into the class I considered myself a novice at C. When we were first given our first assignment to set up a Virtual Machine, I nearly dropped the class because my computer would use up all the RAM, and always freeze then shut down. But I am so glad I stuck through and my computer finally let me set up a VM.

Then when it came to lrc, I was so confused about having to define variable types again. Though I do remember doing so in C++, I was so not used to programming like this because I was only proficient in Python beforehand. What was also funny, was me not understanding how a void function worked and trying to find the purpose of why anyone would ever use a void function if it returned nothing. After barely finishing lrc on time, I knew this class was no joke and started attending literally every section I could.

Then we had the math library assignment and even as a Applied Mathematics Major, I didn’t even know about computing trigonometric functions using Taylor Series/Newton’s Method. I was so confused but realized that our TA’s were literally the best resources we had in the class. I also ended up struggling on sin and cos at x = 1.00 and why the trigonometric identity I put in would never trigger. But what I learned from these first two assignments was that I was learning a lot about C programming and that I was slowly getting the hang of it. That was until assignment 3 where we were first introduced to our first ADT’s in the stack and the queue, and where the nightmares began in segmentation faults. Segmentation Faults are simply when a program accesses memory that it doesn’t have access to because it wasn’t allocated memory for them to use. And for the next few assignments, I could be done with the assignment and just sit there for 3 days running `valgrind` or `gdb --args` and try to debug in misery. But to this day sorting has been a top 3 in terms of my favorite assignments.

Then we had the Depth-First-Search Traveling Salesman problem where we had to find a hamiltonian cycle in a graph ADT. Although I was studying graphs in my Math 100 class where we were diving pretty deep into graph theory, the concept of a graph did not stick to me as it was not our typical cartesian coordinate graph that we have been accustomed to our whole lives. But the introduction to graph theory after finally understanding how it worked was pretty cool in that it showed that math and computer science were great compliments to each other. Also learning my first shortest path finder algorithm introduced me to algorithms which I thought were so cool. Since then I have even taken the time to learn Breadth-First Search, Djikstra’s algorithm, and even Bellman Ford’s algorithm. All these shortest path algorithms are so unique and have truly opened my eyes on how cool computer science is and can be.

And at last we have made it to assignment 5 or the Hamming codes which was our last 1 week assignment. Knowing that in my head was a relief but it also showed me how burned out I was after this long journey. However Hamming codes introduced me to one of my first assignments that I was truly proud about. Using linear algebra, matrix multiplication and transpose matrices, I was able to construct an error correction algorithm which I thought was so freaking cool. I was also able to study about the entropy of information theory and work on my first encoder/decoder project where I could transform some file into mumbo jumbo and turn it back into its original form. After assignment 6 this was easily the coolest assignment in my ranking of the assignments.

Since I alluded to assignment 6, it seems like a great time to talk about Huffman coding which was a lossless data compression algorithm. Out of all things in statistics I never thought I'd use a histogram and somehow make that into its code form to perform data compression. Also this may be biased but what made this my favorite assignment was because I watched the TV show *Silicon Valley* and those characters that started "Pied Piper" also developed a lossless data compression algorithm, and I thought this assignment was so cool because of that. Easily the biggest pain in the a** though. When I say I worked every damn day for those 2 weeks, that was no understatement. It was truly the hardest but most rewarding feeling to actually figure out that I had made the Huffman coding assignment.

And at last we have made it to week 10 and I can say I am done with the Great Firewall of Santa Cruz, where we were instructed to build a firewall and filter out bad words and old words. It could have easily been an assignment where it only required a week but I am glad I have some time to actually begin studying for finals on week 10. But yeah it was truly a great experience and I would not exchange it for the world.

Looking forward now, I feel like I've gotten so much knowledge from this course. Alot of these assignments had their boundaries but I sort of want to continue some of these assignments and expand on them. Computer Science was not my first choice coming into University but I am so glad to be here and be learning such cool things in general. 11/10 class. Professor Long is a great professor and even if you don't think so, he is backed up by the best staff possible. This class is truly great

Which leads me to shoutout once more the TA's and Professor who helped me along the way, who taught me everything I know. They are Darrell Long (Prof), Eugene Chou (TA), Sahiti Vallamhr (TA), Sabrina Au (TA), Brian Zhao (TA), Eric Hernandez (Tutor), Miles Galpa Grossklag (Tutor) and Jloritz (discord student) who helped me every week. Also shoutout to the friends I was able to make in this class through attending sections week by week. With the Sars-Cov-2 pandemic that hit us, college was not what I really thought it would be. But with technology being just as immersive, I still was able to meet great people and just feel like I was not going through this class alone. I think you are what makes your own experiences and it's ironic that even through Zoom University, I was able to make a lot of connections with both students and faculty. But here's a shout to those students: Mhia Grace Mojica, Audrey Ostrom, Harlene Virk, Sean Fujiwara, and Sam Garcia. Week after week I saw the same students over and over in section and in due time I just reached out to a few and struggled with them. Well thank

you CSE 13 for everything. It was truly my favorite class and I can’t wait for what's ahead in my computer science journey