Addison Waller, 701

Assignment 3 Writeup

Base information

Overall, this project was written in Python using a variety of different imported libraries. As explained in the README.MD it is important to also “pip3 install” bloom-filter2 and bbhash libraries. I added some additional functions that allow for creating a list of K and a list of K’. The build_k function simply creates a list of random strings that are a set size. The build_kprime function creates a list of random strings that are a set size but also randomly adds, 1/10 times, in a string from a K list. For each task, there is one method that can be called to run all the listed functions and do the tests I created, minus building each of the K lists. These are called bloom_filter_func and testing_mphf.

Task 1

To implement the Bloom filter I decided to use the bloom-filter2 Python library. To use this Library I created a class called MyBloomFilter. This class contained four functions: init, build_bloom_filter, get_false_positives, and query_bloom. The init function created a Bloom filter using the library initialization function using both an error_rate and a max_num_elements. This would create an empty bloom filter. Then using the build_bloom_filter function I take a list of Ks (random strings of a set size) and add each one to the bloom filter. The get_false_positives calculates the false positives and false negatives using the two lists of K and K’ and the true Negative value. The final method of the class is the query_bloom function. This function takes the list created by build_kprime, K’, and goes over each key checking if it is in the bloom filter or not, adding None if not, and the key to a new list otherwise. Then I created a function where each function from the class can be called as well as the time of querying can be tested and the amount of memory the bloom filter takes up.

When starting this section the part that I found to be the most difficult was understanding and calculating the false positives. Once I re-understood the definition and reasoning behind the false positives I was able to start working on the logic. The logic was something that was tripping me up because of the difference between locating false positives and false negatives as well as which two lists to check against. I had to make sure to write out each detail so that I fully understood which piece of the Ks was used and which the nots and Nones. A final piece I had to figure out was the exact formula for the final false positive percentage, which I ended up looking up.

While the false positive calculation was the hardest for me to figure out the results were very interesting. Below you will see *Table 1*. This is a table showing the different false positive percentages for each K and K’ against the different error rates. As you can see from the chart, there was actually a very very small difference between each of the False Positives and they were

very small, all under 1%. Later, when discussing the Minimum Perfect Hash Function, it can be seen how much a false positive rate can differ. The theory is that the error rate and the resulting false positive rate will have a strong correlation. This leads to the relationship that when the error rate decreases the bloom filter will have to store more data to reach that false positive rate. As seen in *Table 1*, my results fit very well with the relationship between the error rate and false positive rate. But sadly my results do not match the other half of the theory when it comes to size and seem to be more random depending on the error rate.

Table 1: A table displaying the Bloom Filter False Positive Rates across all Ks and Error Rates

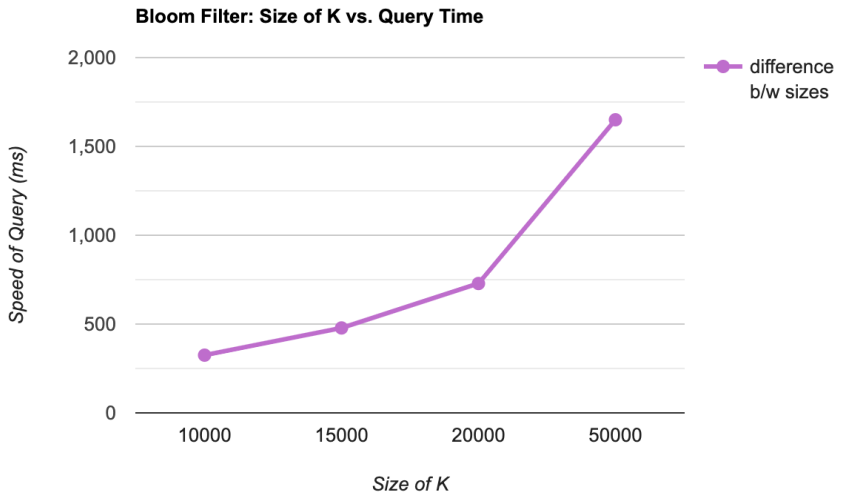
	False Positive Rate		
Error_rate/ Which K(size)	0.0078125	0.00390625	0.0009765625
K1(10000)	0.83%	0.40%	0.14%
K2(15000)	0.87%	0.34%	0.82%
K3(50000)	0.78%	0.37%	0.10%
K4(20000)	0.77%	0.44%	0.08%

Table 2: A table displaying the speed of querying on the bloom filter. The error rate does not seem to have much effect across the different runs but the size of the K does.

Error_rate/ Which K(size)	0.0078125	0.00390625	0.0009765625
K1(10000)	306.3089848	324.9533176	321.5219975
K2(15000)	459.4995975	478.1007767	467.7834511
K3(50000)	1697.401285	1649.989367	1668.51759
K4(20000)	689.756155	728.2831669	652.6560783

When it came to querying it was important to understand how the different sizes of the Ks affect the time to query the Bloom Filter. I ran 4 different iterations with the size of the K and K' being 10K, 15K, 50K, and 20K. Below in *Table 2,* you can see the specific data points for each K and each Error Rate that led to the query time but because the different error rates produced very similar results I only used the error rate of 0.00390625 or 1/(2^8) to chart the changes between the sizes of K. These changes can be clearly seen in *Graph 1*. As portrayed in the graph it is clear that as the size increases the speed of the query will also continue to increase at an exponential rate. The false positive rate does not appear to affect the lookup time.

Graph 1: A direct comparison between each Ks query speed when there is only one error rate being looked at (0.00390625)



An important aspect of running a program and trying to use effective data structures is understanding the memory they can take up. To understand this I gathered the memory space each of the bloom filters took up for each K and for each error rate. In *Graph 2* it is clear that the size of the K affects the space needed for the Bloom filter, which is expected. What shocked me was that the error rate had a clear correlation with the space needed for the bloom filter. In *Graph 2* it is clear that for the highest error rate, 1/(2^7) leads to smaller space usage and as the error rate increases the space usage also increases.

Graph 2: A bar graph showing the relationship between the size of K and space but also the error rate and space.

Task 2

In a very similar fashion to the Bloom Filter in Task 1, I imported a Python library called bbhash. To use this library I created a class called MyMPHF. This class contained three functions: init, query_MPHF, and get_false_positives. The Get_false_positives function is exactly like the one listed above in Task 1 where it produces the false positive and false negative values using the lists K and K' as well as the true negative value. The init function takes in a list of K, hashes using the Python hash function (ensuring each hash is positive with a modification), and appends it to a new list. Then this list is passed into the bbhash library function "PyMPHF " also with the length of this list, a set gamma, and the number of threads to produce a new minimum perfect hash function. The final function is the query_MPHF. This takes in the list K', hashes each of its values in the same way as init, and then uses the lookup function provided by the bbhash library to see if that hash is in the list of Ks. This will all be added to a list, with either None of the current K' being the value, and returned. Then I created a function where each function from the class can be called as well as how long it takes to query the minimum perfect hashing function, the number of bytes the MPFH takes up, and the percentage of the false positive.

One issue I ran into was downloading the minimum perfect hashing function library I wanted to use, bbhash. I was never actually able to pip3 install this library onto my computer. I tried installing Cython and Wheel (as well as uninstalling Wheel) and a million other fixes but in the end, I think something was off with my file path. What I ended up doing was downloading a Ubuntu docker image, running it, and downloading the newest Python and pip upgrades as well as all the pip installs I needed to do. This ended up working and I ran each of my tests through a Docker container. Another part I struggled with was selecting a good hash function. At first, I went with Sha256 but that does not output integers, which bbhash needs. I ended up switching over to the Python hash but bbhash does not take negative hash values so I had to do a modification that ensured the only hash values produced were positive.

The Minimum Perfect Hashing Function (MPHF) results when it comes to the *observed* false positives were very similar in comparison to each other but when compared to the bloom filter they were very different. As seen in *Table 3* the MPHF had a False Positive rate of almost 50% for each K, meaning that again size does not have an effect, but when looking at *Table 4* it is clear that MPHF and the Bloom filter have a very different effect on the false positive rates. One part of this is because we were able to adjust the error rate for the bloom filter but this is not possible for the MPHF.

Table 3: This displays the False Positive Rates for each K when using MPHF

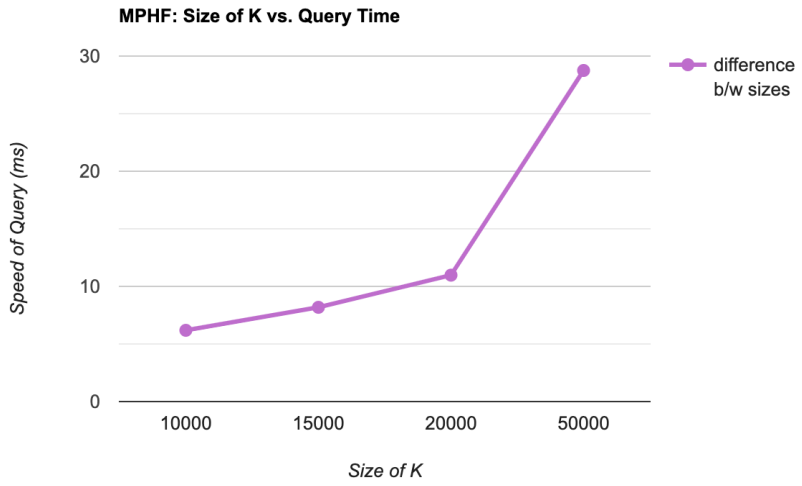
	MPHF False Positive Rates
K1(10K)	49.94%
K2(15K)	49.77%
K3(50K)	49.77%
K4(20K)	49.88%

Table 4: This displays the comparison between false positive rates for MPHF and the bloom filter on the same Ks

	MPHF False Positive Rates	Averages of Bloom Filter Positive Rates
K1(10K)	49.94%	0.46%
K2(15K)	49.77%	0.68%
K3(50K)	49.77%	0.42%
K4(20K)	49.88%	0.43%

In the same way that the total time to query k' was calculated for the bloom filter, it felt important to do the same for the MPHF. As you can see in *Graph 3* there is a very similar growth due to size as we see in the bloom filter. As the size of the K or K' gets larger so does the time to query it. But unlike the bloom filter, the MPHF is super fast. I tried to make a bar graph to show this relationship, which can be seen in *Graph 4*, but after creating it the difference between the two's average time was so big you can not even see the MPHF results. So that these can actually be compared I took one of the error rates for each K query time results for the Bloom Filter and compared it to the query time for the MPHF, which can be seen in *Table 5*. These false positive differences actually fit with what I believed before starting this experiment because I expected that MPHF would have a much worse false positive rate.

Graph 3: The time it takes to run a query on each K of the minimum perfect hash function.



Graph 4: This is the "failed" comparison graph for the difference between the query time of the bloom filter versus the MPHF.

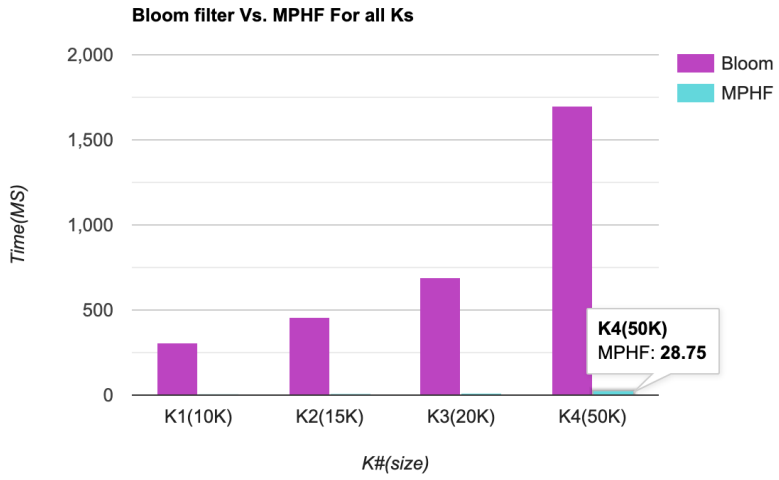


Table 5: This displays the difference between the bloom filter versus the MPHF when it comes to Query time.

Type Function/ K#(size)	Bloom Filter	MPHF
K1(10000)	306.3089848	6.185770035
K2(15000)	459.4995975	8.182764053
K3(50000)	1697.401285	28.7501812
K4(20000)	689.756155	10.97559929

When I was trying to calculate the size of the varying MPHF I used the same approach I did for the bloom filter, which was asizeof from the pympler python library. Even though I used the same approach I did not receive a similar structure of results. The bloom filter was seen to

vary based on the size as well as the error rate. When I ran this function on the MPHF I got a consistent answer of 24 bytes. My only thought behind this is that the program is setting aside the same amount of space for all of the MPHF.

Task 3

The goal of Task 3 was to implement the Fingerpaint array. Because the goal of this was to build on top of the Minimum Perfect Hashing function I added it to the MyMPHF class. This way each of the functions could use the self-command to assess the MPH. The two functions I added to this class are create_fingerprint and false_positive_fingerprint. Create_fingerprint makes an empty vector, using the numpy library, and then looks through K adding the first b bits of the hashed value from K to the fingerprint vector. This returns the fingerprint vector once it has been completed. The second function false_positive_fingerprint calculates the false positive rate of each iteration as well as queries on the fingerprint array. This function enumerates over K' calculating the false positive values using some of the information found in the given fingerprint array. The calls to run these functions are located in the testing_mphf function.

The part I found the most difficult about this task was gaining a complete understanding of the vector library I was using, numpy. While numpy does have good documentation it can sometimes get a little confusing because there is so much information for the library. I at first was creating a 2-dimensional array when I only really needed a one-dimensional array. In a similar way, I had a harder time coming up with a function that would select the correct amount of bits from the hashed number. The first way I attempted this was to convert the integer into a string, take out the correct number of characters needed, and convert the string back into an integer. This did not feel like the most effective way so I decided to try to switch over to bit shifting and masking, which I ended up using. This allowed me to compute the correct amount of bits I needed without doing too many conversions.

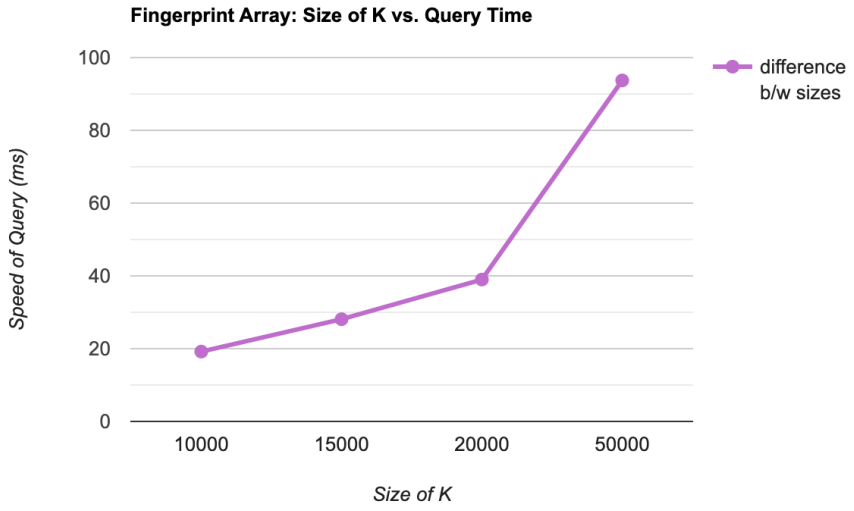
When it comes to some evaluations of the fingerprinting method one important one is to calculate the **observed** false positive rates for each of the Ks on each of the given Bit sizes. The outputs of these tests can be seen in *Table 6.* When looking at these outputs it is clear that all of them are under 1%. This is very reminiscent of *Table 1* which displays the False Positive Rates for the Bloom filter. This was what we wanted to see by adding the fingerpaint array because when we ran the MPHF on its own the False Positive Rates were almost all around 50%, which can be seen in *Table 3*.

Table 6: The data outputted by the False Positive rate calculations from the fingerpaint array

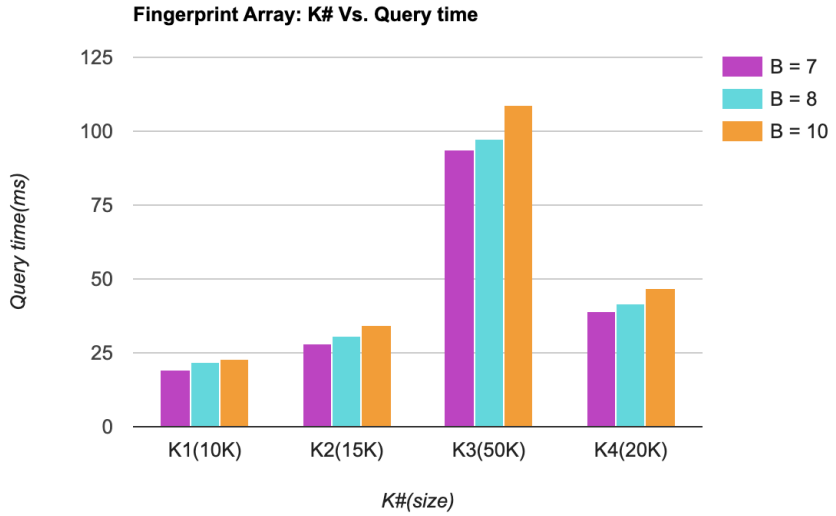
	False Positive Rate		
Bits/ Which K(size)	7	8	10
K1(10000)	0.71%	0.37%	0.08%
K2(15000)	0.75%	0.43%	0.11%
K3(50000)	0.78%	0.40%	0.08%
K4(20000)	0.79%	0.44%	0.08%

Another critical data point we need to collect is the amount of time it takes to query the fingerpaint array. As seen in *Graph 5* it is clear that the amount of time to query has a strong relationship with the size of the K to start with. The bigger the size, the longer to query. This was also a result that we saw in the Bloom filter, which can be examined in *Graph 1.* Another interesting relationship that appeared was the relationship between the number of Bits B inputted and the query rate. It appears as the size grows the query rate does as well. This relationship can be seen in *Graph 6*. This was not a relationship that appeared in the bloom filter between the query time and the error rate. While the pattern between the bloom filter and the fingerpaint array are similar in structure the fingerpaint array is actually much faster. For example, for K1 of the Fingerpaint array it only took 19.19 ms to query compared to the bloom filters 306 ms. This was one of our main goals when using the MPHF, the speed, and by adding the fingerpaint array we are able to lower the false positive rate to an ideal size.

Graph 5: The amount of time to query the fingerpaint array based on one B = 7



Graph 6: This graph shows the relationship between the time to query the fingerpaint array and the different Bit sizes that were inputted.



When it came to the size of the MPHF + the fingerprint array the difference between the bit values was non-existent and they all produced the same size for the same K. There was a variation in memory space between the different sizes of K. As seen in *Graph 7,* as the size of the K grew the memory space grew with it. This is similar to the trend we save in the bloom filter. Overall, the MPHF + fingerprint array and the bloom filter have much more similar patterns than the bloom filter and the MPHF alone, which was the overall goal of the paper.

Table 7: The size of an MPHF + fingerprint array across the different Ks with a consistent B = 7