Alexander Movsesyan

CMSC701 Project 3 Write Up

Task 1.

1. For implementing task 1, I opted to use python, as it was the easiest language to set up in. I created two lists of sets K and K'. One list hosted different length K' sets and the other of different length K sets. The K' sets were generated by getting a list of random integers, ranging from 0 to 100,000, and the K sets were generated by taking the first half of the K' sets. These same exact sets were used in all of the tasks in order to keep the data homogenous. I then iterated through the different sized sets and different expected false positive values to generate and query the bloom filters. The BloomFilters were created with a size twice as big as the set K.
2. For this task I found the most difficult part to be setting everything up. I initially was using the bloom-filter2 library, however this library proved to be extremely slow, thus I transitioned to using the RBloom library.
3. Here is the data that was collected:

expected error rate	set size	num false positives	time elapsed	size in bits of bf	realized error rate
0.0078125	500	0	0.000155210495	10104	0
0.0078125	5000	1	0.001391172409	100992	0.0002
0.0078125	50000	19	0.01416087151	1009888	0.00038
0.0078125	500000	124	0.1482591629	10098872	0.000248
0.0078125	5000000	3466	2.24319458	100988656	0.0006932
0.00390625	500	0	0.000147819519	11544	0
0.00390625	5000	0	0.001461029053	115416	0
0.00390625	50000	1	0.01462984085	1154160	0.00002
0.00390625	500000	45	0.1555309296	11541560	0.00009
0.00390625	5000000	2834	2.459370375	115415608	0.0005668

0.0009765625	500	0	0.0001502037048	14432	0
0.0009765625	5000	0	0.001549720764	144272	0
0.0009765625	50000	0	0.01550722122	1442696	0
0.0009765625	500000	28	0.1661024094	14426952	0.000056
0.0009765625	5000000	2581	2.540933609	144269504	0.0005162

We can see that the total query time increased with respect to the set's size (since we queried for every element in the set), but the query times remained relatively constant when averaging for only 1 query. Additionally we can see that the query time increased slightly when reducing the expected error rate, however this was a minimal amount. Also, as expected, we saw the realized error rates reduce as we reduced the expected error rates. Finally, we see the size of the bloom filter increase as we decrease the expected error rate.

Task 2.

1. To implement the second task, I again used python, leveraging the BBHash library. From this library, I used the BBHashTable. I created a new BBHash Table for every set K and did a query for every element in every set K'. I also used the same sets created in Task 1.
2. The most difficult part of this task, again, was setting it up. I had issues using the library on my Mac, so I opted to use another linux machine. Once everything was set up, it was pretty straightforward to use the library. However, it is important to note, that I couldn't find a size function to relay the size of the BBHashTable structure, so I opted to use the len function, which simply relayed the amount of hash values.
3. Here is the data that was collected:

set size	num false positives	time elapsed	length of mphf	realized false positive rate
500	0	0.003194570541	500	0
5000	0	0.02562212944	5000	0
50000	0	0.2317214012	50000	0
500000	26	2.367068768	500000	0.000052

5000000	2551	24.39286399	5000000	0.0005102

Again, we can see that the query time remained quite constant over varying sizes of sets. However, we can clearly see that the queries took much longer than the bloom filter counterpart. That being said, the realized false positive rates are less than even that of the bloom filter with the lowest expected false positive rate. Also, there seems to be a marginally increased query time when there are false positives, but this can also just be due to different factors (such as OS jitters).

Task 3.

1. To implement this task, I again used python and the BBHash and RBloom libraries, and the aforementioned list of sets K and K'. I used the PyBBHash class to generate the hash values that will be used to store each value in the bloom filter, and I used a separate array to store the fingerprint or the last n-bits of the value in K.
2. The most difficult part of this task was figuring out how to get the index from the MPHF library. I was initially using the BBHashTable structure, but realized I need to use the PyBBHash structure instead. Additionally similar to Task 2, there was no function to see the amount of space that the PyBBHash class requires.
3. The Collected Data:

set size	num false positives	time elapsed	length of fingerprint	bf size in bits	fingerprint num bits
500	4	0.0004572868347	500	14432	7
500	2	0.0004510879517	500	14432	8
500	1	0.0004596710205	500	14432	10
5000	24	0.005210161209	5000	144272	7
5000	11	0.004811763763	5000	144272	8
5000	1	0.005042076111	5000	144272	10
50000	218	0.05112552643	50000	1442696	7
50000	98	0.0514383316	50000	1442696	8
50000	25	0.0537033081	50000	1442696	10

		1			
500000	2128	0.6032590866	499989	14426952	7
500000	1091	0.6040787697	499989	14426952	8
500000	280	0.6141047478	499989	14426952	10
5000000	23434	7.910686016	4998815	144269504	7
5000000	13026	7.867715836	4998815	144269504	8
5000000	5192	8.042538166	4998815	144269504	10

As expected within the confines of this experiment, as the size of the fingerprint increases, the number of false positives decreases. Although we have no real data on the size of the MPHF, we know that it increases, as the sets increase, and the size of bloom filter increases with the size of the sets. Additionally, we can see that the time for a single query tends to increase as we increase the set size.

It is also important to note that with this approach, we see more false positives than the other approaches. A potential explanation for this can be that the values inside the sets K and K' are all unsigned integers (and are essentially hash values).