CMSC 701 HW3 Writeup

Task 1 — Empirical evaluation of the bloom filter

Description

For task 1, I used the library "bloom-filter2" for constructing bloom filter. To generate unique keys in set K, I used Python's random.choices() function so that each key is fixed length. To generate the keys in set K′, I first used the same function to generate some random and unique keys; then I shuffled all the keys in sets K and K′ and appended the first few keys from the shuffled set K so that the amount of the keys in K′ is the same as K. The ratio of "positive" and "negative" keys is determined by a variable "pos_key_percentage". Finally, to test the results, I iterated through 3 different sizes, 3 different expected false positive rates, and 3 different positive key percentage. Within each iteration, I ran the tests 5 times (except for when size is 100,000, which would take too long) and got the average false positive rates, time, and bloom filter size given the condition in that iteration.

Difficulties

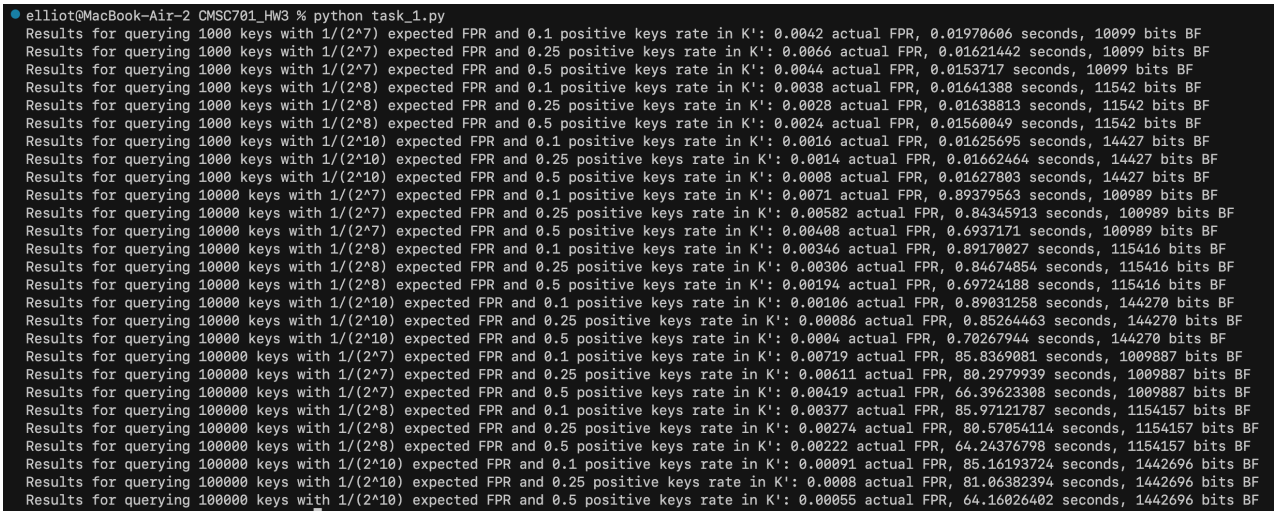
The most difficult part for me was to figure out how to present the results. I originally thought of printing out the results for each test run, but that would be over 100 lines of output if I decided to run each test 5 times; besides, I would need to manually calculate the average results for each 5 runs. The way I finally went with was to let the test_run() method to return the values I want for each run in an array. Then I put the results for each 5 runs under the same condition in an array so that I had an array of 5 arrays for each condition. Finally, I used the zip() function to get the average values of each "column".

Observations

By comparing different sizes of K and K′, we can see that the size of bloom filter increases proportionally with the increase of key size. The query time increase drastically when the key size increases. The actual FPR seems to be independent from the key size.

By comparing different expected false positive rates, we can see that the actual FPRs do not match the expected FPRs perfectly; however, they are similar to the expected FPRs. With the decrease of expected FPR, both the query time and the size of bloom filter increase, although not by much.

By comparing different mixtures of "positive" and "negative" keys in K′, we can see that the size of bloom filter is only affected by key size and the preset expected false positive rate. When there are more positive keys in K′, the observed false positive rates is lower, and it takes less time to query.



Command line output

Key size	Observed FPR	Time (seconds)	BF Size (bits)
1,000	0.0024	0.01560049	11,542
10,000	0.00194	0.69724188	115,416
100,000	0.00222	64.24376798	1,154,157

Comparing difference key sizes (expected FPR is 1/(2^8), positive keys rate is 0.5 for each entry)

Expected FPR	Observed FPR	Time (seconds)	BF Size (bits)
1/(2^7)	0.00408	0.6937171	100,989
1/(2^8)	0.00194	0.69724188	115,416
1/(2^10)	0.0004	0.70267944	144,270

Comparing different expected FPRs (key size is 10,000, positive keys rate is 0.5 for each entry)

Positive keys rate	Observed FPR	Time (seconds)	BF Size (bits)
0.1	0.00346	0.89170027	115,416
0.25	0.00306	0.84674854	115,416
0.5	0.00194	0.69724188	115,416

Comparing different positive keys rates (key size is 10,000, expected FPR is 1/(2^8) for each entry)

Task 2 — Empirical evaluation of a minimal perfect hash

Description

In task 2, I used BBHash for constructing MPHFs. A lot of code was reused from task 1. For instance, I used the same way to generate random and unique keys in set K and K′, except that the keys are integers instead of strings due to the limitations of BBHash library. Similarly, during query, instead of checking if key is in bloom filter, I checked if MPHF.lookup(key) would return “None”, and the rest is the same as task 1.

Difficulties

This part should not have been too difficult since the tests we need to perform are the same, but with a different AMQ. However, I spent hours trying to figure out how to correctly import bbhash to my project. I kept getting “ImportError: dynamic module does not define module export function” on “import bbhash” when I tried to run the python file for this task. I tried different python versions, different environments, even different architectures, and I got no luck. Eventually, I just switched to a different computer.

Observations

From the tables, we can see that the observed FPR seems to be associated with the positive keys rate and independent from the key size. Both the query time and the size of the MPHF increase with the increase of key size. Comparing different positive keys rate, we can see that the observed FPR is almost the same as negative keys rate (i.e., the percentage of keys in K′ that are absent from K). Additionally, the query time decreases when we increase the positive keys rate. What I did not expect was how high the false positive rates were. For instance, when the positive keys rate is 10%, the FPR is almost 90%.

Despite its drastically higher FPR, MPHF does seem to have some benefits in time and size when compared to the bloom filter. For the same key size and positive keys rate, the query time of MPHF is only about 50-60% of the bloom filter, and the size is only about 30-40%.

```
(my_env) elliot@MacBook-Air-2 CMSC701_HW3 % python task_2.py
Results for querying 1000 keys with 0.1 positive keys rate in K': 0.8636 observed FPR, 0.00399613 seconds, file size 12384 bits
Results for querying 1000 keys with 0.25 positive keys rate in K': 0.7196 observed FPR, 0.00374975 seconds, file size 12384 bits
Results for querying 1000 keys with 0.5 positive keys rate in K': 0.484 observed FPR, 0.00296054 seconds, file size 12384 bits
Results for querying 1000 keys with 0.75 positive keys rate in K': 0.2388 observed FPR, 0.00179935 seconds, file size 12384 bits
Results for querying 10000 keys with 0.1 positive keys rate in K': 0.89224 observed FPR, 0.49178424 seconds, file size 39584 bits
Results for querying 10000 keys with 0.25 positive keys rate in K': 0.74344 observed FPR, 0.46982923 seconds, file size 39584 bits
Results for querying 10000 keys with 0.5 positive keys rate in K': 0.49692 observed FPR, 0.36964846 seconds, file size 39584 bits
Results for querying 10000 keys with 0.75 positive keys rate in K': 0.24746 observed FPR, 0.20198269 seconds, file size 39584 bits
Results for querying 100000 keys with 0.1 positive keys rate in K': 0.89792 observed FPR, 50.02312303 seconds, file size 388704 bits
Results for querying 100000 keys with 0.25 positive keys rate in K': 0.74713 observed FPR, 47.41474724 seconds, file size 314592 bits
Results for querying 100000 keys with 0.5 positive keys rate in K': 0.49936 observed FPR, 37.93331599 seconds, file size 355296 bits
Results for querying 100000 keys with 0.75 positive keys rate in K': 0.24923 observed FPR, 22.0536139 seconds, file size 314592 bits
```

Command line output

Key size	Observed FPR	Time (seconds)	MPHF Size (bits)
1,000	0.484	0.00296054	12,384
10,000	0.49692	0.36964846	39,584
100,000	0.49936	37.93331599	314,592

Comparing difference key sizes (positive keys rate is 0.5 for each entry)

Positive keys rate	Observed FPR	Time (seconds)	MPHF Size (bits)
0.1	0.89224	0.49178424	39,584
0.25	0.74344	0.46982923	39,584
0.5	0.49692	0.36964846	39,584
0.75	0.24746	0.20198269	39,584

Comparing different positive keys rates (key size is 10,000 for each entry)

Task 3 — Augment the MPHF with a "fingerprint array"

Description

Task 3 builds on task 2, with an added fingerprint array to reject false positives and lower false positive rates. When building the fingerprint array, I first got the index in the fingerprint array, which is the value of `MPHF(key)`, using `MPHF.lookup(key)`. To get the value that needs to be stored at index in the fingerprint array, I used Python's built in hash function, `hash()`, and then I used "`bin(hash(key))[-int(b):]`" to get the last b bits. For the size of the fingerprint array, I just used a regular Python array and manually calculated it.

Difficulties

I did not have too many difficulties in this part. The main thing that bothered me was that on my computer, it would take a while to query on sets with size 100,000. It took my computer almost an hour to run every test in this part.

Observations

With the increase of key size, both the query time and size increase, and the observed FPR decreases marginally. With the increase of b's value, observed FPR decreases, query time stays roughly the same, and the size increases. With the increase of positive keys rate in K', both FPR and query time got improved at some level. It is interesting to note that the three tables here look very similar to the

three tables in task 1. The observed FPR here roughly matches 1/(2^b). Even the query time and size are not too far from the results with corresponding key size and positive keys rate when using the bloom filter.

Key size	Observed FPR	Time (seconds)	BF Size (bits)
1,000	0.0024	0.00507674	20,384
10,000	0.0021	0.68733058	119,584
100,000	0.00197	74.56077695	1,114,578

Comparing difference key sizes (b = 8, positive keys rate = 0.5 for each entry)

b	Observed FPR	Time (seconds)	BF Size (bits)
7	0.0037	0.69212928	116,445
8	0.0021	0.68733058	119,584
10	0.00052	0.68868084	139,584

Comparing different expected FPRs (key size = 10,000, positive keys rate = 0.5 for each entry)

Positive keys rate	Observed FPR	Time (seconds)	BF Size (bits)
0.1	0.00346	1.00557446	119,584
0.25	0.00298	0.85573339	119,584
0.5	0.0021	0.68733058	119,584
0.75	0.00066	0.58022437	119,584

Comparing different positive keys rates (key size = 10,000, b = 8 for each entry)