CMSC858D: Homework 2

Domenick J. Braccia

January 29, 2020

Task 1 – Basic Bloom Filter

Implementation

Determining 'M' and 'k'

Given a target False Discovery Rate p, and number of elements inserted n, the size of the bit vector can be found with:

$$m = -\frac{n \ln p}{(\ln 2)^2}$$

And the number of hash functions to use can be found with:

$$k = \frac{m}{n} \ln 2$$

I then implemented the functional script 'bf.py build' to build Bloom filters over N elements from randomly generated k-mers of length 100. Each nucleotide was sampled with uniform probability at each position in the k-mer. To build a Bloom filter over 1 000 000 100-mers, that meant that I needed to define a Bloom filter of size m = **4792529 bits** with k = **~ 3.32 (rounded to 4) hash functions.** To query the built Bloom filter, I implemented the 'bf.py query' function as instructed in the problem description.

Most difficult Part

The part of this assignment that I found to be most difficult was in actually setting up the Bloom filter data structure and constructor function. I initially started implementing it in C++, as I thought it would be easier to have control over the bitvector and the data members of the Bloom filter. Unfortunately, this turned out to be more challenging than I had thought; library dependencies, build errors and debugging build errors were a nightmare for me in C++, so, I switched to implementing in Python. At this point in the project, I had already spent 15 or so hours reacquainting myself with Bloom filter basics, so once I switched to Python, I was able to complete the implementation of a constructor, and a query method in about 2.5 hours.

Results

To test my implementation of the Bloom filter (BF), I generated BFs with varying number of input 100-mers, Ns = {1 000, 10 000, 100 000, 1 000 000, 10 000 000}, and a varying False Positive Rate (FPR), FPRs = {0.01, 0.05, 0.10, 0.20, 0.25}. For each combination of N and FPR, I issued three sets of queries: one set had 100 100-mers fully contained in the BF, the second had 50 100-mers present in the BF and 50 not present, and the third had zero 100-mers present in the BF and 100 not present. These correspond to the "100pct", "50pct", and "0pct" labels in figures 2 and 3 below, respectively.

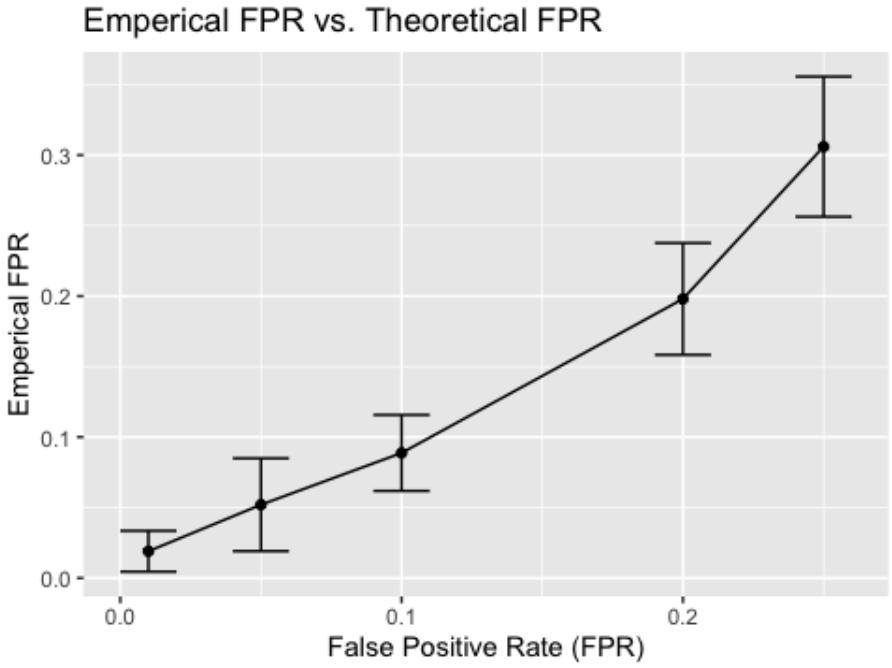

Figure 1: Measurment of emperical FPR vs. theoretical FPR of all query streams (100pct, 50pct and 0pct) and N values. The emperical FPR valeus at each point are averaged from varying N's and query streams. The error bars are Standard Error (SE).

Note: For each 'by.py build' step, I specified the number of distinct keys in the keyfile to be exactly N every time, without actually calcuating that vale. Since I randomly genererated nucleotides of 100-mers from a uniform distribution, the maximum probability of randomly generating a repeat 100-mer was $Pr(repeat) \sum_n^N \frac{n}{4^k}$, where k = 100 and N can vary from 1E+03 to 1E+07. Even when generating 1E+07 100-mers, $Pr(repeat) \approx 10^{-47}$.

Figure 1 shows that the emperical FPR matches that of the theoretical FPR from the BF build step. All emperical FPRs besides 0.25 seemed to be very close to their theoretical underpinning. The SE bar on the point corresponding to a theoretical FPR of 0.25 does not seem to cross over the expected value of 0.25 on the y-axis. With more samples and measurments, this would likely return to a more reasonable value in the emperical ~0.25 range.

In Figure 2, average query time across all three query streams (100pct, 50pct and 0pct) are show. There is a general decrease in the query time as percent present in the BF decreases. This is because the algorithm I implemented for checking membership of a key in a BF breaks after the first hash value checked is not present in the corresponding BF. For queries that are present in the BF, each hash value from each independant hash function must be calculated and checked in order to report membership.

There seems to be very little to no relationship between query time, query type and N value, as evidenced by Figure 3. The query time seemed to randomly spike for different query types at different values of N, not following any particular trend. For example, the query time for the 0pct type spikes for N = 10 000, but then returns to moderately low values for increasing N.

Figure 4, much like Figure 3, seem to suggest that there is very little interplay between query time, N, and FPR. The query time, again, randomly spikes for different FPRs at different values of N, not following one well defined pattern. The spikes seem to be due to one or a few outliers, since the error bars for those averages are quite large.

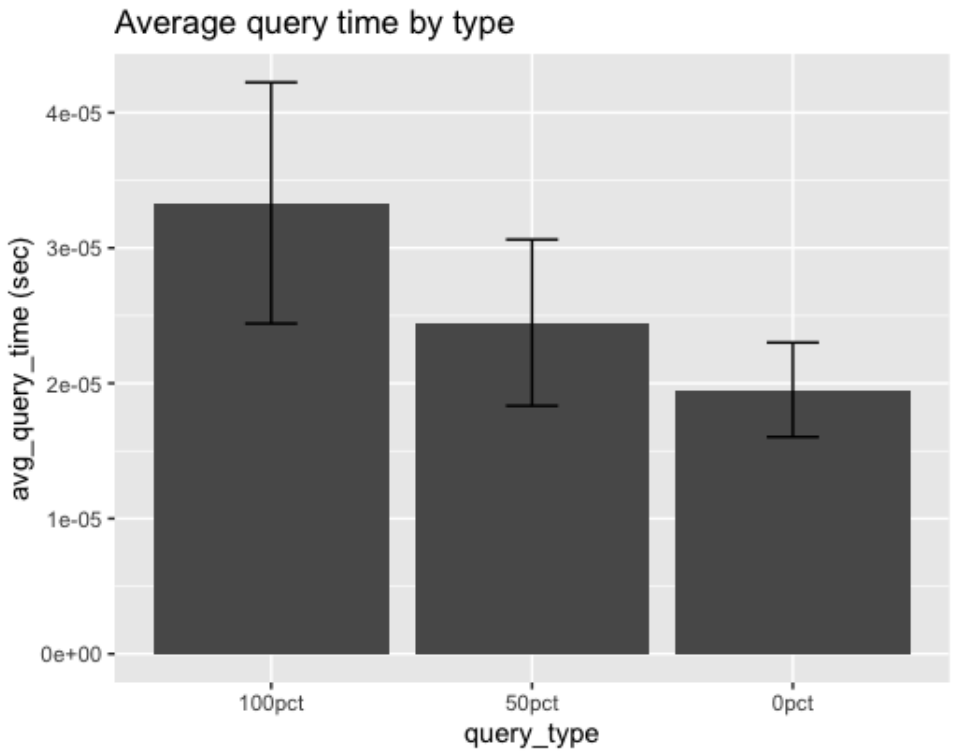

Figure 2: **Average query time by query type.** These query time averages are across all values of N and FPR. Error bars are SE.

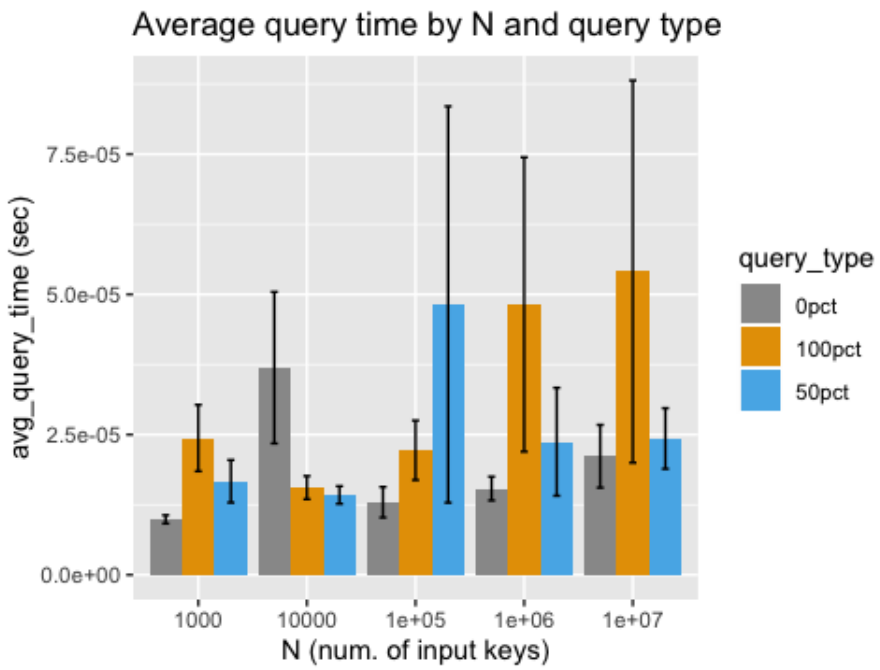

Figure 3: Average query time with varying N and across query types. The averaged values are across FPR, and error bars are SE.

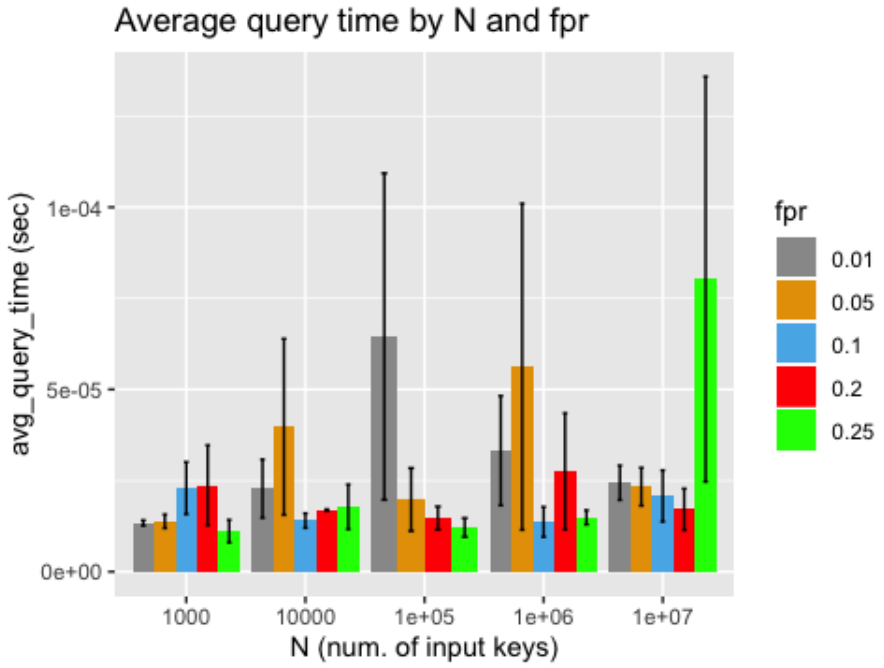

Figure 4: **Average query time with varying N and across FPRs**. The averaged values are across query types, and error bars are SE.

Task 2 – Blocked Bloom Filter

Implementation

The implementation for the Blocked Bloom filter was fairly straight forward once I had the data structure for the regular Bloom filter in place. Instead of calculating a hash value for each hash function for a given kmer and then flipping those bits in the Bloom filter, I let the first hash value determine which block to flip all the rest of the bits for that kmer.

Most difficult Part

The most difficult part of this task was just figuring out how to initially modify my existing Bloom filter to become a blocked one. Once I realized I can keep a similar bit vector (but extending it to fill out the final block should it not match up to the requested block size) and just change the first hash value calculated to be the one that determines which block I should insert 1's into, the rest was easy.

Plots

All of the plots for this section were made in the same way as the previous section, on regular Bloom filters. The only difference here is that I used less query sets and FPRs to benchmark the results. I tested average query time and emperical vs. theroetical FPR for N = {1000, 10 000, 100 000, 1 000 000} and FPR = {0.01, 0.1, 0.2} and combinations therein.

The difference in average query time between the 100pct query stream and the 50pct query stream are nearly the same, and there is a significant drop in time from the 50pct to the 0pct stream, as we saw with the regular Bloom filter (Figure 5).

The measurments of the emperical FPR seemed to be close to the theoretical values, though for the FPR 0.20 filter, the emperical FPR was a bit lower than expected. By increasing the query size

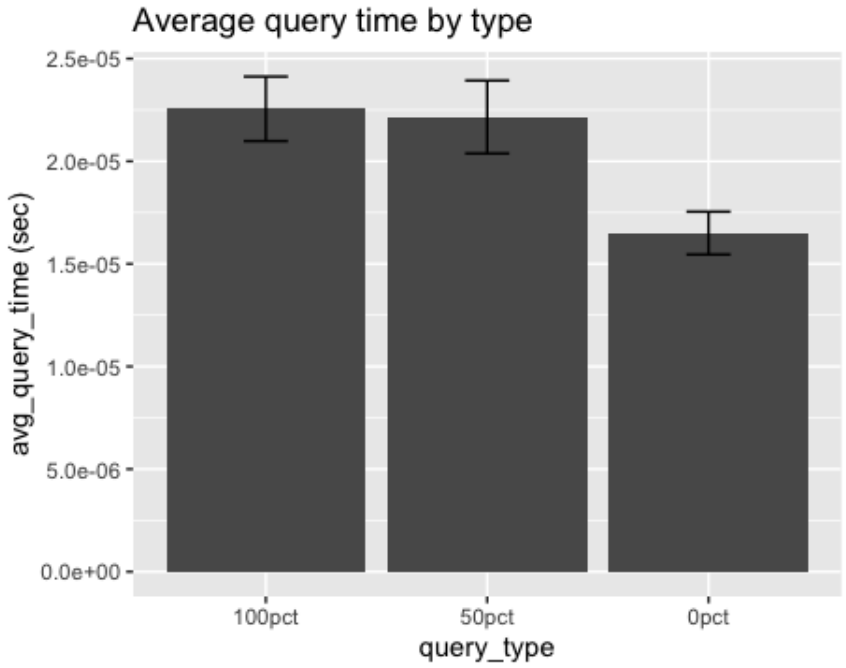

Figure 5: **Average query time by query type.** These query time averages are across all values of N and FPR. Error bars are SE.

and stratifying by query stream, it is likely that the error bars would fall within the expected FPR or 0.20 (Figure 6).

The main consistency with average query time by N and query type is that the 0pct stream has the lowest average query time across all values of N. This is to be expected for the same reasons as the regular Bloom filter (Figure 7).

With a lower FPR, the average query time is consistently high accross all values of N. This makes sense since the algorithm for checking membership in the blocked Bloom filter has to go through more iterations before it can return a "Y" or a "N" to the k-mer query. As FPR increases, the query time decreases, on average (Figure 8).

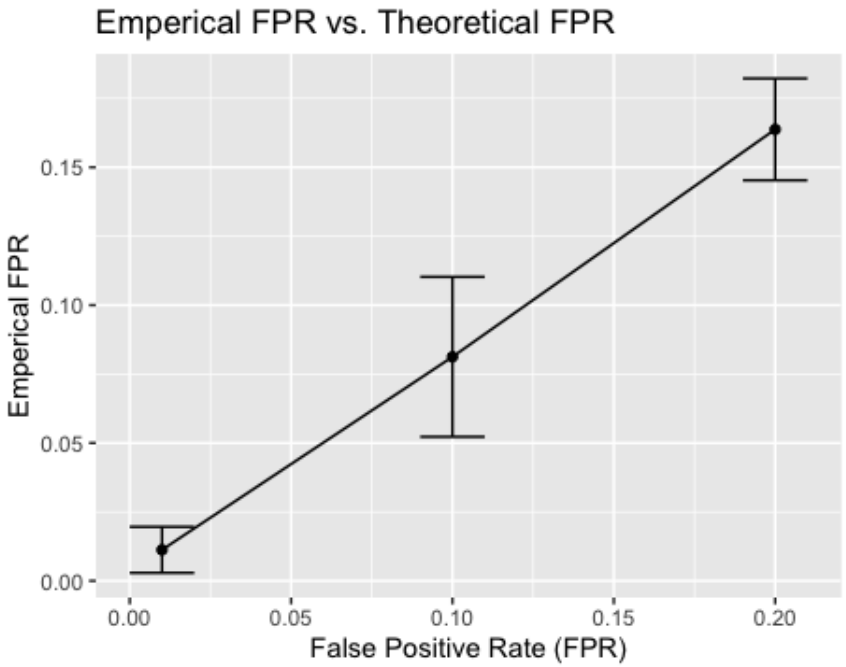

Figure 6: Measurment of emperical FPR vs. theoretical FPR of all query streams (100pct, 50pct and 0pct) and N values. The emperical FPR values at each point are averaged from varying N's and query streams. The error bars are Standard Error (SE).

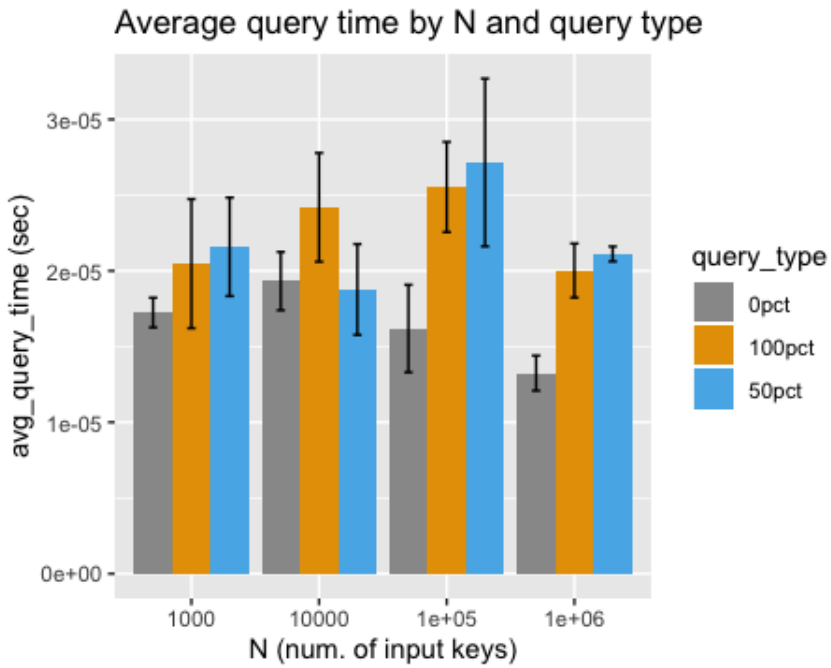

Figure 7: **Average query time with varying N and across query types**. The averaged values are across FPR, and error bars are SE.

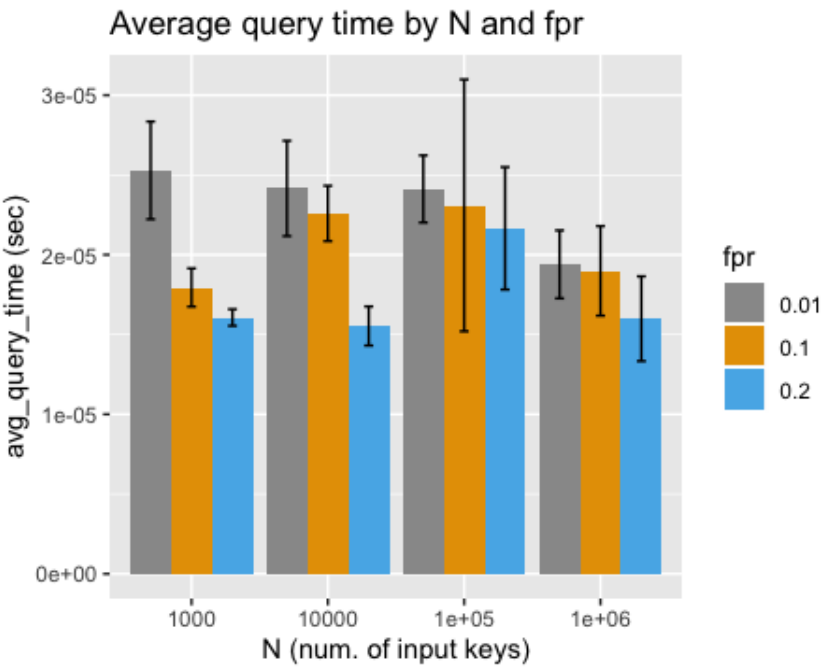

Figure 8: **Average query time with varying N and across FPRs**. The averaged values are across query types, and error bars are SE.