Analysis of Bloom Filter Report

Owen Feng

1. Designing a Suitable Hash Function

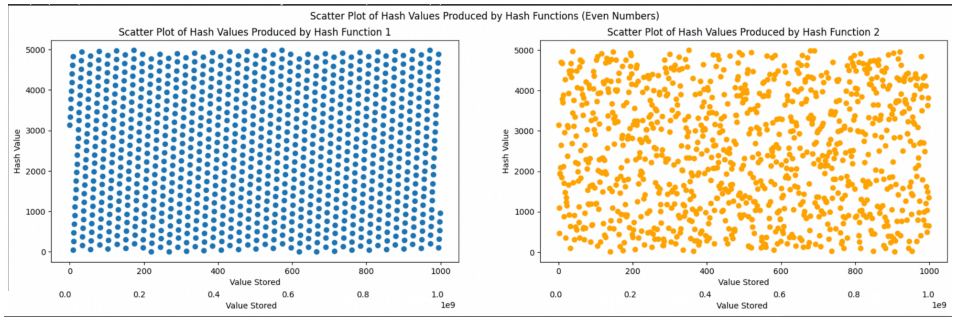
In my implementation, the first and second type of hash functions, called h1 and h2 respectively, are contained within create_hash1 and create_hash2, which are functions that generate and return hash functions based on the given parameters. create_hash1 picks a prime p greater than our universe's size U and picks two random values a_i, b_i within the ranges of {1, 2,... p−1} and {0, 1, ... , p − 1}. h1 then produces a hash value based on the equation:

$$h_i(x) = ((a_i * x + b_i) \bmod p) \bmod m$$

On the other hand, create_hash2 takes in a *seed* and the size of our hash table *m* as inputs and h2 then produces a random hash value within the range {0, 1, . . ., m} using a random generator function seeded at *seed* + *x*.

These hash functions are appropriate because they map any number from U = {0, 1, …, N-1} with uniform probability and are independent. With h1, the modulo *m* ensures that the final hash value falls within the range [0, m-1], effectively distributing values evenly across the Bloom filter's bit array. Furthermore, because each hash function has a different a_i and b_i, this ensures multiple hash functions return different results. For h2, because it uses the randint() function and gives a unique seed to every function, it has uniform distribution and is independent.

To test these hash functions, I generated thousands of data points using various data choices and kept track of the hash values returned by both functions. I then plotted the values of all the original values against the returned hash values, creating scatter plots for h1 and h2. As you can see from the figures below, both hash functions worked perfectly fine when working with random data. However, when we switched to sequential data (Taking the first thousand even numbers), we see that the scatter plot generated by h1 seems to have a highly structured grid-like pattern, a result that was likely attributed to the function's linear congruential design. This may be disadvantageous as non-randomized inputs can increase the risk of collisions in the Bloom filter, increasing our false positives rate. h2, on the other hand, seemed to handle the sequential data a lot better, with its scatter plot showing a much more randomized and well-distributed pattern. Therefore, because h2 seems to be less sensitive to sequential data, it appears to be the better function. Note that h2 is also not entirely random as it relies on the randint() function from the Python Random module which generates its numbers with Mersenne Twister which is deterministic. This means that, in theory, if values are chosen in a way that consistently results in the same seed, it could significantly increase collisions and reduce the function's randomness.



2. Implementing a Bloom Filter

I implemented my Bloom Filter by creating a BloomFilter Python class. A BloomFilter object would take in several inputs, *m* (which denotes the size of our hash table), *k* (our desired number of hash functions), and *hash_fn_generator* (specifies which hash function we want to use for our Bloom Filter– useful for testing). The BloomFilter constructor takes these inputs and constructs a hash table, *self.arrays = [0] * m*, and *k* hash functions, stored in *self.hash_functions = [hash_fn_generator(i,m) for i in range(k)]*. The BloomFilter class also defines the following functions:

`add(self, x):`
Adds an element into our hash table. For each hash function in *self.hash_functions*, we set the index of the value returned by the hash functions to 1 in *self.array*.

`contains(self, x):`
Checks to see if an element has been inserted into our hash table. For each hash function in self.hash we check the index returned to see if it's equal to 1. If it isn't, output **False**. If all the hash functions are marked 1, output **True**.

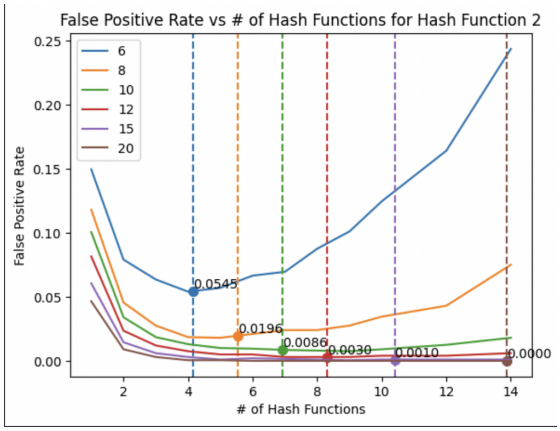
`get_false_positives_rate(self, n, queries):`
Calculate the false positives rate of a BloomFilter object. Generate *n* random inputs to insert into the filter and track them in a set. Then, run a specified number of *queries*. If a query returns **True** from the filter but the item is not in the set, increment a false positives count. Finally, compute the false positives rate as the false positives count divided by the total number of *queries*.

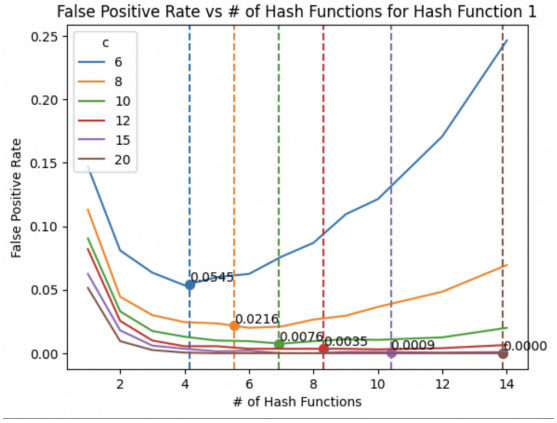
3. Analyze False Positive Rate for different values of *k*

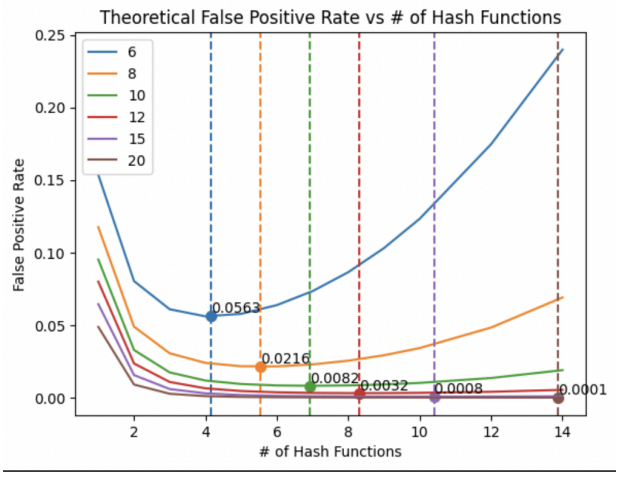
In order to better understand the effectiveness of our filters, I also need to check each function's false positives rate and check to see if they're performing as expected. The function **calculate_false_positives_rate** evaluates the false positives rates across different values of *c* and *k*. For each combination of c and k, the function initializes a Bloom filter using each hash function. To ensure reliable results, it runs multiple trials (rebuilding the Bloom filter each time), runs **get_false_positives_rate** for each trial, and takes the median value of all false positives rates. It then stores these false positives rates in dictionaries that map *c* values to lists of *(k, median false positives rate)* for each hash function.

After calculating the false positive rates for various c and k combinations, I compared them to the theoretical rates using the formula: $(1 - e^{-k/c})^k$. The results showed that the false positive rates of both hash functions closely matched the theoretical predictions. This confirms their effectiveness, as they maintain uniform and random hash distribution, minimizing false positives. The close alignment with theoretical values also suggests that our Bloom filter will perform predictably under random data conditions, as expected. However, if we were to use sequential data as in Task 1, h1 would likely produce a dramatically higher false positives rate due to its sensitivity to patterns.

I also visualized this by plotting the relationship between *k* and the median false positive rates for both the hash functions and theoretical calculations. From these graphs, we can see that false positive rates decrease as *c* (the size of the filter relative to the number of elements) increases, confirming that a larger filter reduces collisions and improves accuracy. Furthermore, the vertical lines marking the optimal *k* for each *c* for each of these curves demonstrate that the theoretical equation $k = c * ln(2)$ accurately predicts the ideal number of hash functions to minimize false positives. In addition, the similarity between the behavior of both hash functions and their alignment with the theoretical results shows that the chosen hash functions were effective and closely replicate the expected behavior of a Bloom filter.


False Positive Rate vs # of Hash Functions for Hash Function 2
False Positive Rate
# of Hash Functions
6
8
10
12
15
20
0.0545
0.0196
0.0086
0.0030
0.0010
0.0000


False Positive Rate vs # of Hash Functions for Hash Function 1
False Positive Rate
# of Hash Functions
c
6
8
10
12
15
20
0.0545
0.0216
0.0076
0.0035
0.0009
0.0000


Theoretical False Positive Rate vs # of Hash Functions
False Positive Rate
# of Hash Functions
6
8
10
12
15
20
0.0563
0.0216
0.0082
0.0032
0.0008
0.0001