Bloom Filter Project

CMPSC 130A, Sawyer Rice

1. Hash Functions

Prior to looking into the properties of Bloom Filters, I first examined the characteristics of two hashing schemes. Hashing Scheme 1 entailed randomly selecting a seed and summing it with the value you are looking to hash, and seeding a random number generator at this sum. Hashing Scheme 2 took a different approach and randomly selected two values, a and b, and then took the product of the value being hashed with a, and added on b. This final sum was then taken mod p, where p is a sizable prime number. For all of my experiments, I chose p to be $2^{31} - 1$, and N, the size of the universe, to also be $2^{31} - 1$.

As seen in Figure 1, for uncorrelated, randomly selected input, both hashing schemes seemed to exhibit similar behavior. Both hashing methods randomly, and relatively evenly distributed inputs across indices. In these plots, m, the size of the hash table, was fixed at 5000, and n, the number of insertions, was fixed at 833 such that c, or $m/n \approx 6$.

Figure 2 further indicates similarity between the two hashing schemes. Using a hash table with m fixed at 5000, and n varying between 0 and 5000 in increments of 50, the maximum loads, or largest amount of elements at any one index, followed extremely similar trajectories. The insertions in Figure 2 were also random and did not include repeated insertions.

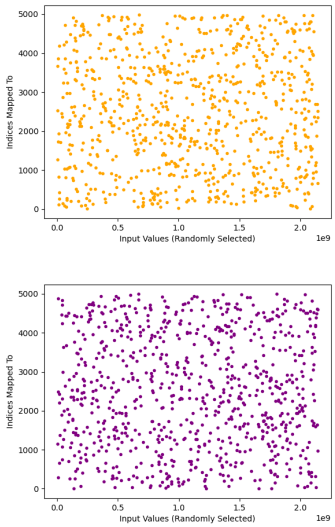

Figure 1: *Hashing Scheme 1, 2*

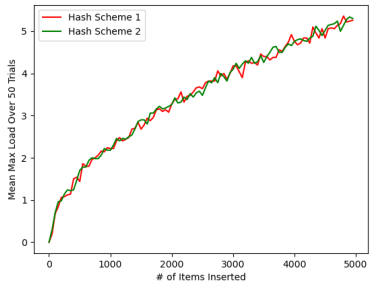

Figure 2: *Average Max Load*

Figure 3 illustrates a pitfall of Hashing Scheme 2, as clearly recognizable patterns can be observed in the output of Scheme 2 when correlated input is utilized. In the first graph in Figure 3, the numbers 0 to 832 were inserted into a table of size 5000, and the resulting linear patterns occurred. Such patterns did not arise when hashing with correlated inputs in Hashing Scheme 1. The second graph in the figure compares the mean number of collisions for each hashing scheme for correlated inputs from 0 to n. On the x-axis, n varies from 0 to 2500 in a table of size $m = 5000$, and on the y-axis changes in the mean number of collisions are observed. Despite distributing the inputs into linear patterns, Scheme 2 maintains a lower average number of collisions for almost all n values.

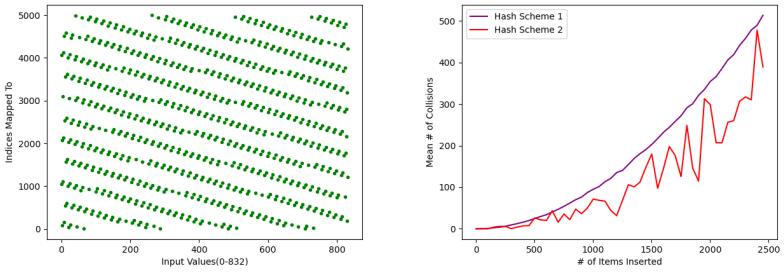

Figure 3: *Hashing Scheme 2 Using Correlated Insertions (1 to n)*

2. Bloom Filter

My Bloom Filter utilizes an integer array initialized with all 0's. Both insert() and contains() operate in a standard manner. Insert() flips 0's to 1's to indicate the presence of a value in the table, and contains() checks to be sure all necessary 1's are flipped in order to assert with high confidence that a value is present in the table.

I further added makeHash() and clearHash() functions to offer fluid destruction and recreation of my hash functions and integer arrays. This allowed me to create loops that had the ability to test the Bloom Filters and hash functions many different times with new randomly selected values. For each data value in the graphs shown in Part 3, I wiped the hash functions and integer array 50 times and found the median of these false positive rates to ascertain more accurate results.

For all of my Bloom Filter data collection, I fixed m at 10,000, N at $2^{31} - 1$, and p at $2^{31} - 1$. All of these variables were declared globally in order to allow for easy and universal adjustment. In order to vary c, I simply adjusted n in order to achieve the desired c value.

Additionally, when utilizing the C++ random number generator, rand(), I first seeded the random number generator with the current time in nanoseconds in order to ensure I was selecting my random numbers in a less predictable manner.

3. False Positive Rate

To measure the false positive rate in a Bloom Filter utilizing Hashing Scheme 1 or 2, I first fixed m at $10,000$, and n at $1,666$, such that $c \approx 6$. To increase the accuracy of results, for each value of k, I ran 50 trials, each time with k new hash functions, and n randomly selected values to insert. Once the first n terms were inserted into the table, I then randomly selected n new numbers that I was certain to not exist in the table, and then subsequently checked for their presence in the table.

Once I had my total number of false positives for each set of trials, I divided it by n, the number of non-present elements that I checked the presence of, to find my false positive rate. I added all 50 of these trials into a C++ vector, sorted them in ascending order, and took the middle element as the median false positive rate for the particular value of k.

As seen in Figure 4, the false positive rate for Hashing Scheme 1, Hashing Scheme 2, and the theoretical false positive rate are all extremely similar, almost too similar to see any difference with the naked eye. The second graph zooms in on the same data to k values only between 1 and 10 in order to better see the slight nuances in the three curves. My theoretical curve follows the equation $y = (1 - e^{-k/c})^k$, where $c = 10,000/1,666$. When plotting the theoretical curve, k was the dependent variable and c was fixed. The curve $y = (1 - e^{-k/c})^k$ achieves an absolute minimum at $k = c\ln(2)$, as shown by the first and second derivative test. Both the curves of Hashing Scheme 1 and Hashing Scheme 2 indicate that the false positive rate dips lowest around $k = 4$, a fact that is in accordance with the notion that the optimal k value is $c\ln(2)$. Given that $c \approx 6$, $c\ln(2) \approx 4.158$, or 4, as 4 is the closest whole number.

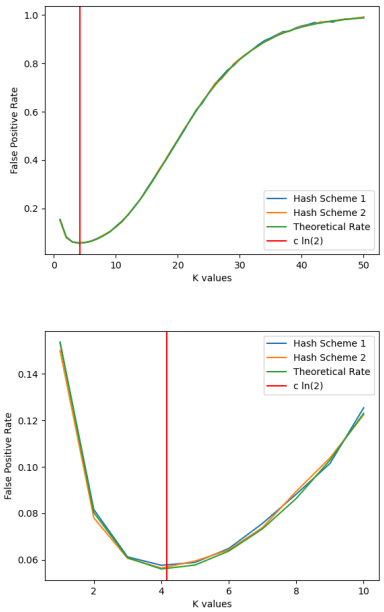

Figure 4: *False Positive Rates,* $c \approx 6$

For $c \approx 6$, Hashing Scheme 1 and 2 performed near identically. In order to measure the relative accuracy of each Hashing Scheme in comparison with the theoretical curve, I found the norm of the difference of each scheme with the theoretical curve and divided that value by the norm of the theoretical curve in order to normalize the error. For Hashing Scheme 1, this normalized error was 0.016968, and for Hashing Scheme 2, the normalized error was 0.0194275. Thus, Hashing Scheme 1 performed slightly better than Hashing Scheme 2 for $c \approx 6$.

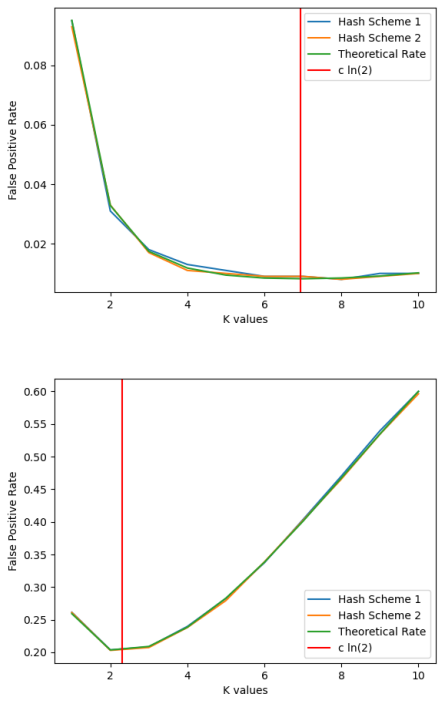

Figure 5: *False Positive Rates for* $c = 10$ *and* $c \approx 3.333$

Again, for $c = 10$, and $c \approx 3.333$, Hashing Scheme 1 and 2 performed within near perfection of the theoretical curve. In the first graph of Figure 5, n was set to 1,000 in order to achieve a c value of 10. For $c = 10$, both Hashing Scheme 1 and 2 incorrectly predicted the optimal k value. For $c = 10$, $c\ln(2) \approx 7$, but both Hashing Scheme 1 and 2 predicted the false positive rate to be slightly lower when $k = 8$. However, the differences between the false positive rates at $k = 7$, and at $k = 8$ were extremely minor. When c was equal to 10, the false positive rate dipped lower than for $c = 6$ and $c \approx 3.33$, as expected. The larger c is, the more empty space occupies the hash table, and thus the lower the chance of a false positive. In the second graph in Figure 5, n was set to 3,000 such that $c = 10,000/3,000$, or approximately 3.333. When $c \approx 3.333$, Hashing Schemes 1 and 2 correctly predicted that the false positive rate would dip lowest at $k = 2$ as $3.333 * \ln(2) \approx 2.3$.

However, upon examining the false positive rates of each hashing scheme on correlated inputs, interesting behavior occurred. Hashing Scheme 1 continued to perform within great precision of the theoretical rate. Hashing Scheme 1 had a relative residual norm with respect to the theoretical curve of 0.0030519. Contrastingly, Hashing Scheme 2 struggled to match the false positive rates of the theoretical curve. This is to be expected, however, as Hashing Scheme 2 was shown to have patterns in its output for correlated input in Figure 3.

Further, Hashing Scheme 2 experienced fewer collisions on average than Hashing Scheme 1 when $k = 1$ in the context of correlated insertions as shown in Figure 3. This fact translates to the graphs in Figure 6 as Hashing Scheme 2 averages a lower false positive rate than Hashing Scheme 1 when $k = 1$.

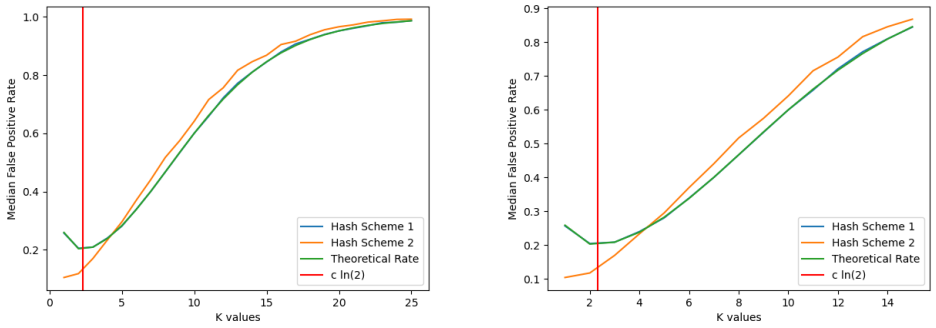

Figure 6: *False Positive Rates for* $c \approx 3.333$ *with Correlated Inserts*

This lowered false positive rate did not last long as around $k = 5$, Hashing Scheme 2 rose above the theoretical and Hashing Scheme 1 curves and remained above indefinitely. Overall, Hashing Scheme 2 achieved a relative residual norm with respect to the theoretical curve of 0.06268, a value over 6 times as large as the error of Hashing Scheme 1.

Overall, when compared in the context of both correlated and uncorrelated inputs, Hashing Scheme 1 clearly performs better than Hashing Scheme 2. Thus, Hashing Scheme 1 should be the preferred hashing scheme for Bloom Filter implementation.