Assignment 3: HyperLogLog

Gustav Gyrst and Katarzyna Toborek

2021-11-03

1 Introduction

The following report provides a description of an implementation of `HyperLogLog`, which is an efficient algorithm used for estimating a number of distinct elements in a given input stream. The implementation makes use of a few smaller functions, implemented and tested independently before assembling the algorithm. The functions, described in more detail in the next section, are `hash`, `rho` and `registers`. Based on the registers, `HyperLogLog` produces an estimate of the stream cardinality. The total space usage to produce the result is $m \log(\log n)$ bits, where m is the number of registers and n is the size of the input stream.

2 Implementation

All code parts for the project was implemented in Python 3.8.8.

2.1 Hash

The `hash` algorithm used for `hyperLogLog` is implemented using bitwise operations. A helper function is used that returns the number of set bits in a given input value. We use the given matrix A containing 32 $32-bit$ integers to calculate the hash. For each value n from the input stream, the helper function is applied to the result of multiplying n with a single element from A (bitwise $A[i]\&n$). Then the operation $\&1$ is applied to the result, what corresponds to modulo 2 operation - if it evaluates to 0, the integer is even, if it's 1, the integer is odd. Next the result is multiplied by ith power of 2, what is done by applying a left shift i to the result ($<< i$). These operations are done in a `for` loop, with i being in range $0-32$. With each iteration the results are accumulated to return a final hash value of n.

2.2 Rho

The `Rho` method returns the position of the first 1 in the binary representation. The bitwise right shift was used where we start with shifting $32 - i$ (where i iterates over range 33) and then checks whether the current bit in this position is a 1 by using the bitwise AND operator. It returns i when it finds the first bit, otherwise it returns `None`.

2.3 Registers

The `registers` function combines both the `hash` function, `rho` and also introduces another hash-function $f(x)$, which purpose is to calculate an index for each input value. This index is used to map a value to certain "bin" in the register M - $M[index]$. When m is of size 1024, we use a right shift of 21 to get indexes between $0 - 1023$. When we have the index for our input, we calculate the hash value of it using our `hash` function. Then we apply the `rho` function on this hash value and store the rho value for the hashed input under in the index given by $f(x)$ in our register. If the current value at this position is smaller than the new value found, the current value is then replaced. The function then returns M with its values.

2.4 HyperLogLog

At last, `hyperLogLog` algorithm uses the results provided by `registers` to produce the desired output. The implementation follows the pseudocode provided in the problem description. We use the provided "magic" constant $\alpha_m = 0.7213/(1 + 1.079/m)$ for the raw estimate and calculate the number of empty registers. Based on these values the final estimate is returned: for smaller cardinalities and at least one non-empty register linear counting is used; for larger estimates, large range correction is applied instead.

3 Testing for correctness

For the greater part our implementation was tested for correctness on Codejudge, where each function used for `HyperLogLog` was first tested independently before testing the algorithm performance. We assume that the quality of the Codejudge tests was much higher than anything we could come up with ourselves.

3.1 CodeJudge Tests

Each part of the implementation (`hash`, `rho`, `registers`, and `HyperLogLog`) was tested on different small and large inputs including some corner cases. For instance the `hash` algorithm was tested on small input the smallest and largest number (00000000 and ffffffff) and also a large test with an input of size 500+. Similarly for `rho` and `registers` the tests included both small and large inputs. In particular the `registers` implementation was tested on two large inputs.

`HyperLogLog` algorithm was tested with the *Threshold* exercise on Codejudge. There the success of the tests depended on the algorithm providing an estimation of the input cardinality that has an estimation error of less than 10%.

All of the tested functions performed as expected, thus ensuring a correct implementation.

3.2 Evaluating the quality of the hash function

To ensure the quality of the hash function we run an experiment where we determined the distribution of the hash values of ρ ($\rho(h(x))$) for one million hash values with $x \in \{1, ..., 10^6\}$. The results of this experiment support the assertion that the aforementioned distribution satisfies $Pr[\rho(y) = i] = 2i$ for all i from 1 to k for random $y \in \{0, 1\}$. To check the performance of the function, we stored the results in a list and visualized it on a plot. The following list stores a fraction of $x-$es that had a resulting ρ value - 0.5 at $\rho = 1$, 0.25 at $\rho = 2$ etc.:

$[0.5, 0.25, 0.125, 0.0625, 0.03125, 0.01562, 0.00781, 0.00391, 0.00195, 0.00098]$. It is easy to notice that the number of x decreases by half on each next ρ, what corresponds to the desired probability $Pr[\rho(y) = i] = 2^{-i}$.

Figure 1 depicts the described distribution.

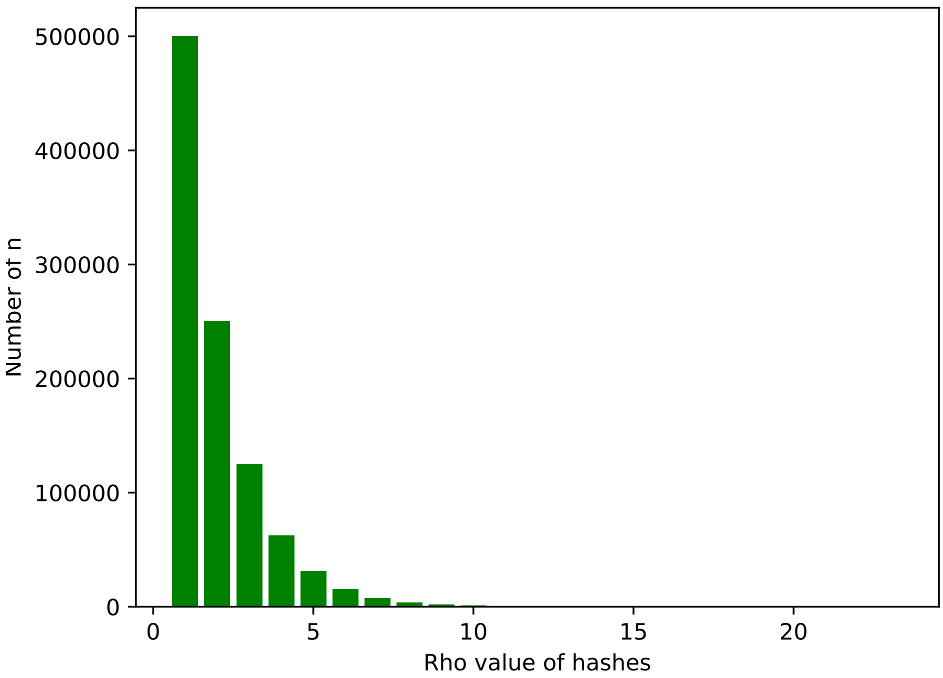

Figure 1: Distribution of $\rho(h(x))$ for one million hash values

4 Experiments

The experiments were made on a Surface book 3 with 1.30GHz 8 cores Intel i7 processor, Memory of 32GiB RAMS. Python 3.8.8 was used. We used python's standard library Random to create an input generator with a seed function to create lists of distinct 32-bit integers. The input generator method was tested for correctness by converting the list into a set to check that there where no duplicates in the list. Moreover, the seed functionality was tested by running the input generation multiple with a specific seed.

We ran two experiments with five different values for m (i.e., 256, 512, 1024, 2048, 4096). The first with 100 repetitions and a random input n distinct 32-bit integers of size 100,000 for each m test. The second with 1000 repetitions and a smaller random input of size 10,000. This was done by implementing a benchmark function that takes four inputs: the number of repetitions for the test, the size of n, the size of the register m, and m_fit value to make the $f(x)$ (the function that indexes the input to a register) fit the register size. The benchmark function returns a list with the resulting hyperloglog estimations for the specified m size. For each i in a repetition in a run of the benchmark function, the i value was parsed as a seed in the random input generator to get a new n-sized random list for each repetition. The seed would ensure that for the next experiment with a new m size, the test input stayed fixed.

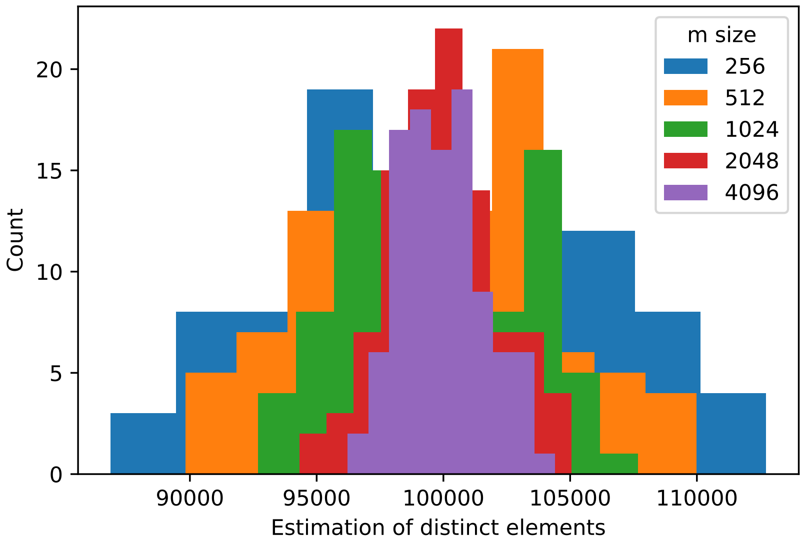

Figure 2: Experiment 1 with 100 repetitions, n of size 10000. Here all five tests are represented (m values 256, 512, 1024, 2048, 4096). Starting where the back-most histogram represents m of 256 all the way through to the front-most histogram representing the m of 4096.

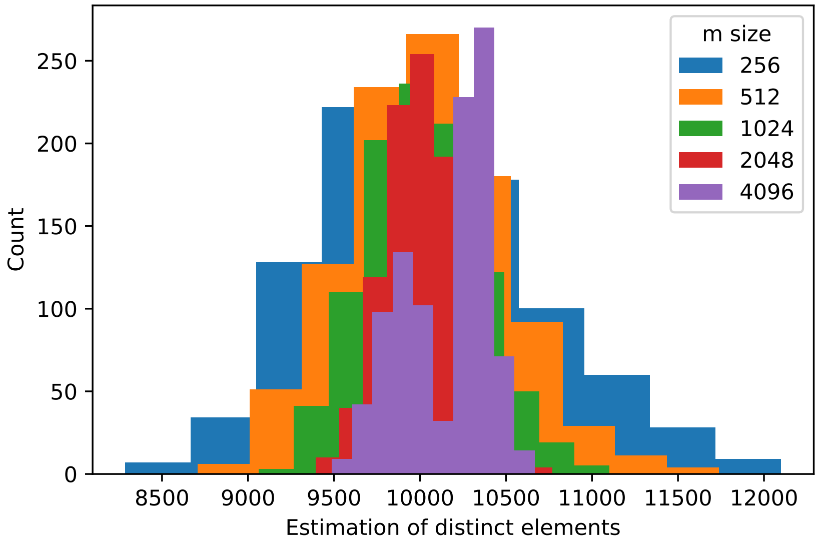

Figure 3: Experiment 2 with 1000 repetitions, n of size 10000. Here all five tests are represented (m values 256, 512, 1024, 2048, 4096). Starting where the back-most histogram represents m of 256 all the way through to the front-most histogram representing the m of 4096.

Table 1: Experiment 1: 100 repetitions, n of size 100,000. Fraction of hyperloglog estimates that fall within 1 and 2 standard deviations from n (the number of distinct elements). Standard deviation is calculated as $1.04/\sqrt{m}$.

m	$\pm 1\sigma$	$\pm 2\sigma$
256	0.660000	0.980000
512	0.650000	0.940000
1024	0.580000	0.960000
2048	0.740000	0.960000
4096	0.660000	0.960000

Table 2: Experiment 2: 1000 repetitions, n of size 10,000. Fraction of hyperloglog estimates that fall within 1 and 2 standard deviations from n (the number of distinct elements). Standard deviation is calculated as $1,04/\sqrt{m}$.

m	$\pm 1\sigma$	$\pm 2\sigma$
256	0.682000	0.950000
512	0.683000	0.955000
1024	0.678000	0.953000
2048	0.716000	0.974000
4096	0.260000	0.645000

5 Results

For the tests with small m sizes and $i = 100$ and $n = 100,000$, the `HyperLogLog` estimations are slightly skewed (skewed right for $m = 256$ and skewed left for $m = 512$). This imprecision could be explained by the large n/m. We do not see the same when the repetitions are raised from 100 to 1000 in the second test and the n-size was lowered to 10,000 in experiment 2. In general, the results of experiment 2 come out with a much more even normal distribution bell curve around the expected n-distinct elements. Except when m gets very large in relation to n (i.e., $m = 4096$). Here the distribution comes out bi-nominal. This is somewhat understandable as we have the small n/m value of 2,4, which can result in more empty registers. There is a pattern throughout both experiments (seen both on figure 2 and 3, that shows that the variance in the results becomes smaller as the size of the register m increases. In other words when we increase m on a fixed size of n, we see a gain in accuracy as m increases. Moreover, common for both experiments (with the exception of experiment 2, $m = 4096$ with $n/m = 2,4$), we have a normal distribution of the fraction of estimates that lie within $\pm 1\sigma$ (i.e., 68%) and $\pm 2\sigma$ (i.e., 95%) as seen in table 1 and 2 when σ is derived in relation to the size of m ($\sigma = 1,04/\sqrt{m}$).

Appendices

Results from Experiment 1

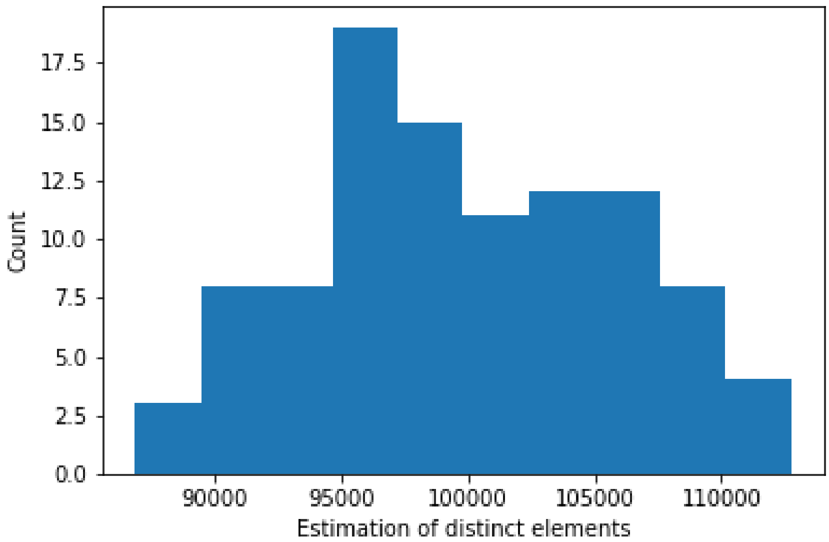

Figure 4: Experiment 1 with 100 repetitions, n of size 100000. m=256

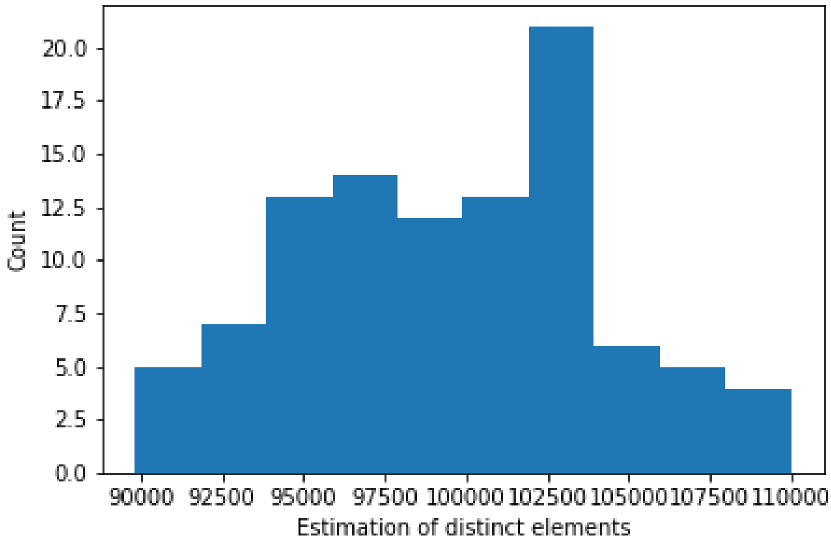

Figure 5: Experiment 1 with 100 repetitions, n of size 100000. m=512

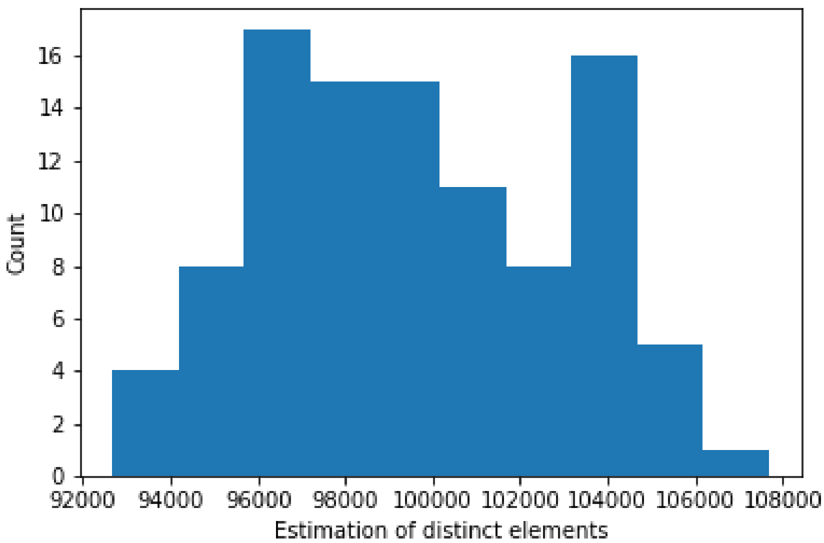

Figure 6: Experiment 1 with 100 repetitions, n of size 100000. m=1024

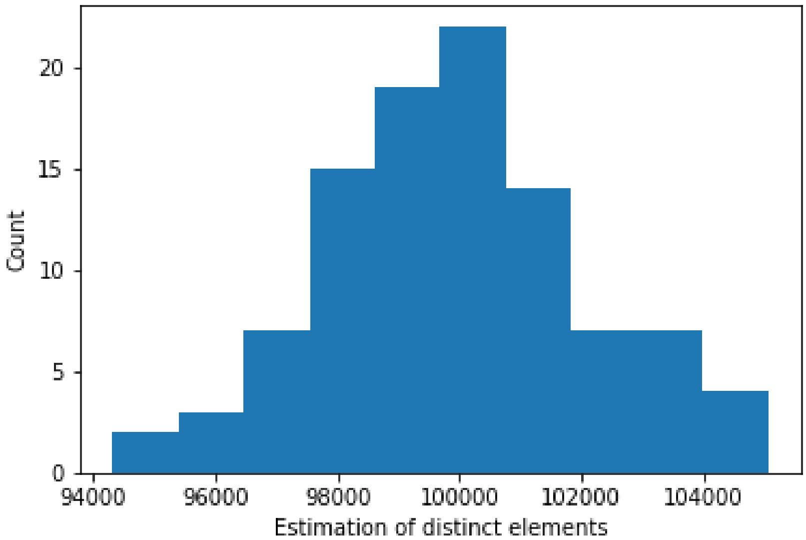

Figure 7: Experiment 1 with 100 repetitions, n of size 100000. m=2048

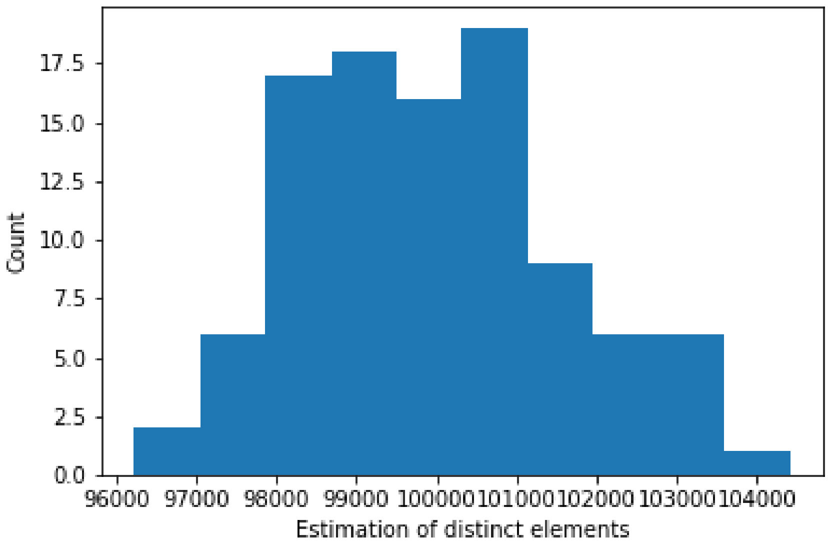

Figure 8: Experiment 1 with 100 repetitions, n of size 100000. m=4096

Results from Experiment 2

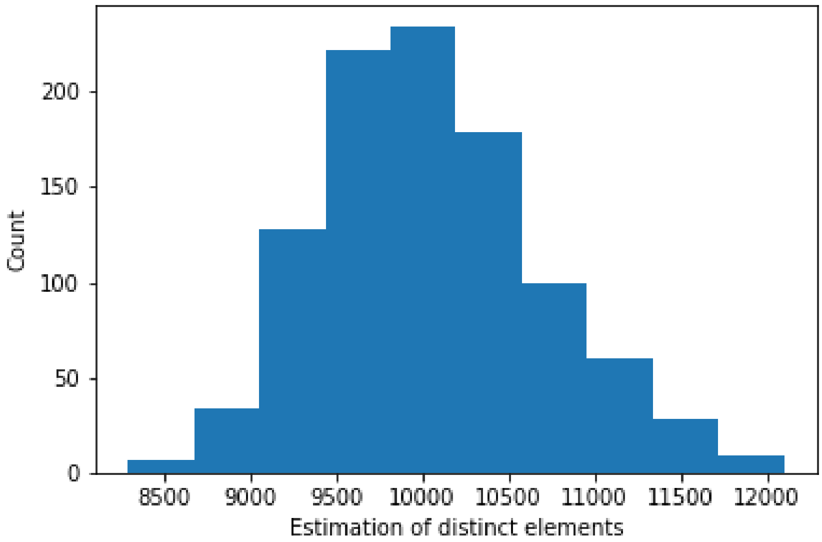

Figure 9: Experiment 2 with 1000 repetitions, n of size 10000. m=256

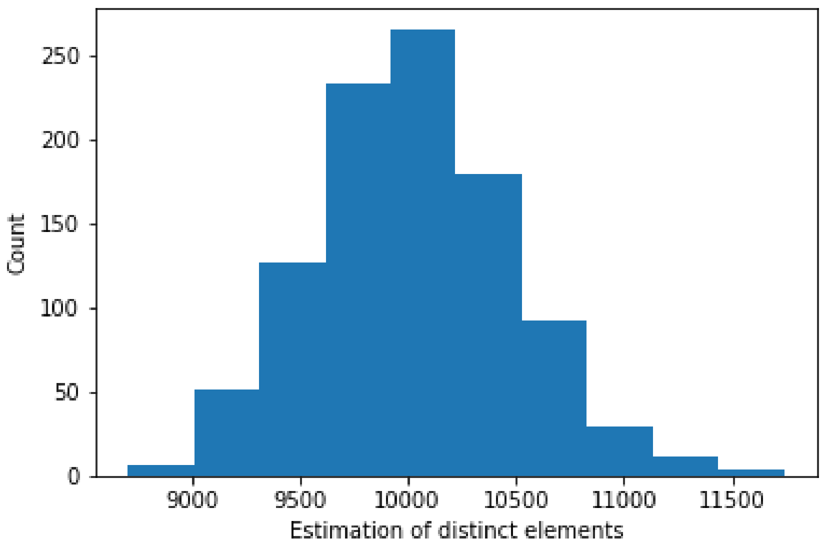

Figure 10: Experiment 2 with 1000 repetitions, n of size 10000. m=512

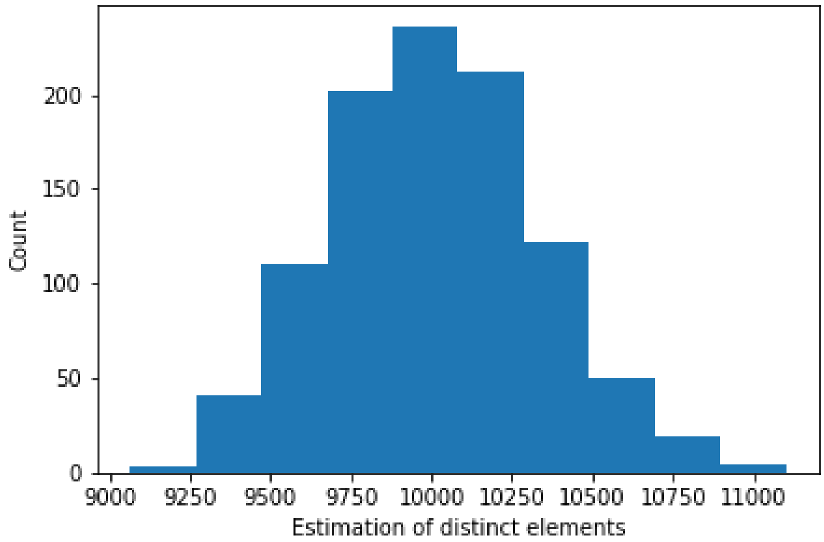

Figure 11: Experiment 2 with 1000 repetitions, n of size 10000. m=1024

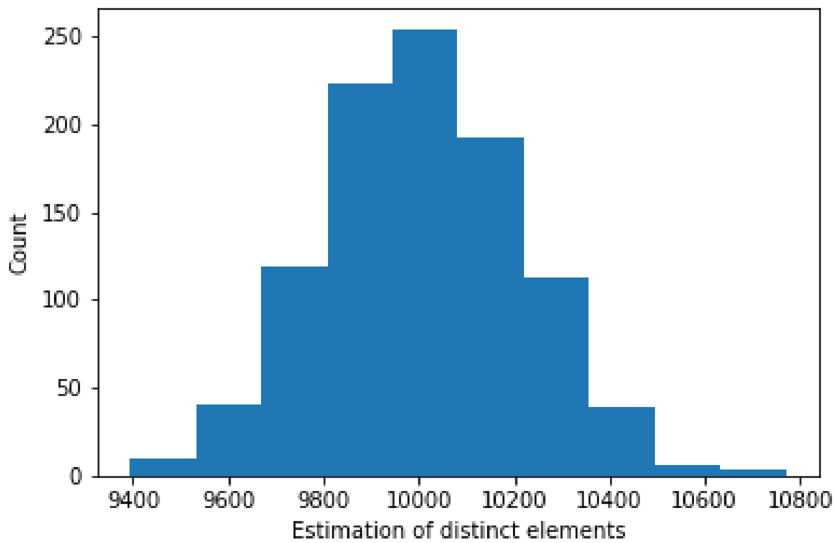

Figure 12: Experiment 2 with 1000 repetitions, n of size 10000. m=2048

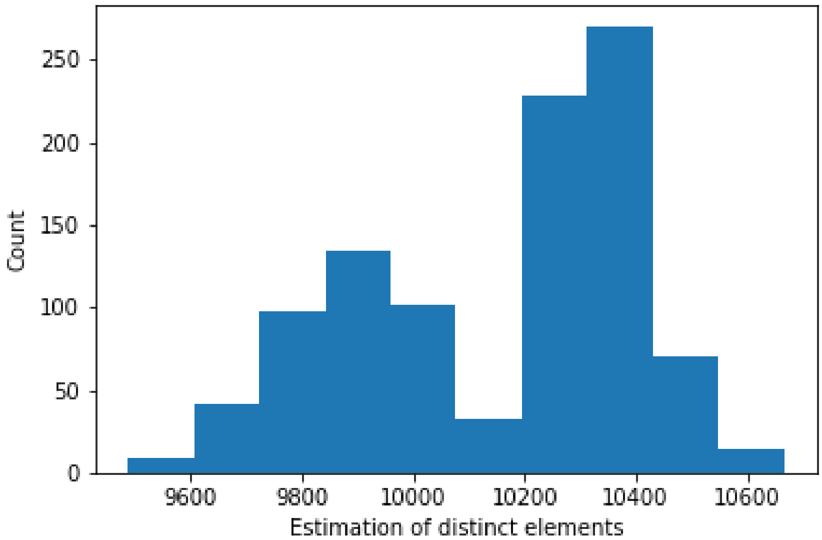

Figure 13: Experiment 2 with 1000 repetitions, n of size 10000. m=4096