Assignment 3: HyperLogLog

Laurenz Aisenpreis, Kristof Gasior and Lasse Bach Andersen

02-11-2021

1 Introduction

In this assignment we were asked to implement the HyperLogLog algorithm, for estimating the number of distinct elements in a stream. The implementation was based on the pseudocode found in the Introduction paper, and on the CodeJudge assignements, which we used to verify the very basics of our solution. The quality of the hashing function was tested by comparing different register-sizes and input-sizes with the estimation error, and the results of our experiment are visualized in tables and histogram-plots.

2 Implementation of hash function h

The hash function $h(x)$, takes an integer x which can be represented as a vector by transforming it to a binary number and treating each bit as an element of the vector. Calculating the cross product of A and $\vec{x}$ results in a vector $\vec{y}$. Performing modulo 2 on each element of $\vec{y}$ gives us a vector that can be interpreted as a binary number, i.e. the hashed value.

3 Implementation of function ρ

The function $\rho(\vec{y})$ takes the vector produced by $h(x)$ and returns the position of the first bit that is set to 1 starting from the leftmost bit as index 1. I.e. $\rho(1000_2) = 1$ and $\rho(0101_2) = 2$. We implemented this functionality using the built in function `Integer.numberOfLeadingZeros()`.

4 Implementation of *HyperLogLog*

The algorithm was implemented in Java 11.01, using the tests provided on *CodeJudge* as test environment. In order to conduct the experiments we needed to adapt the hash function f as described in section 6.2. This implementation was then used to perform the experiments that are described in section 6.

5 Evaluating the quality of the hash function h

The value of ρ reflects the position of the first bit set to 1 from the left side in the binary number representation of an integer. For the leftmost bit, the probability of the bit being 1 is 0.5. for the second bit also being 1 the probability will therefore be $0.5 \cdot 0.5$, and the i^{th} bit being 0.5^i, which is equivalent to $Pr[\rho(\text{y}) = \text{i}] = 2^{-i}$. The distribution of ρ-values is shown in figure 1. The results clearly reflect the quality of our hash function, since the distribution of ρ-values roughly halves for whenever ρ is incremented.

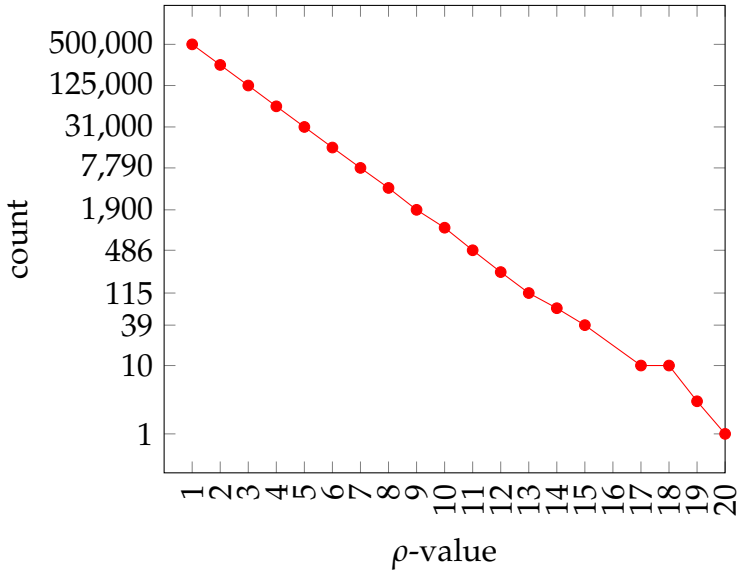


Figure 1: Distribution of ρ-values

6 Experiments

The following experiments were run in OpenJDK Runtime Environment AdoptOpenJDK-11.0.11+9 (build 11.0.11+9) on a MacBook Pro (Retina 15-inch, Mid 2015), 2.8 GHz Intel Core i7, 16 GB 1600 MHz DDR3.

6.1 Input function

Our input function takes as arguments an integer of size n and a seed value and generates a list of random distinct integers in the following way: first, we initialize a *BitSet*. Then, we add random elements to the set until the cardinality of the set equals n and return the *BitSet*.

6.2 Adaption of hash function f

In order to adapt the *HyperLogLog* to work with different sizes of m, we have adjusted the function f. Instead of having a fixed size bitshift of 21, we added a dependence on m, i.e. $31 - \lfloor \log_2(m) \rfloor$. Hence, we ensure that the range of hashed integers $0, 1, ..., j$ by function f will be in accordance to the register size m, i.e. $j < m$. It is crucial to note that the relation of the register size and the hashed values of f is best when m takes on a value of 2^k for any $k \in \mathbb{W}$. E.g. let $m = 1576$, then according to our implementation we would now have 1576 registers, but our hashing function f is only capable of hashing to $2^{\lfloor \log_2(1576) \rfloor} = 2^{10} = 1024$ registers.

6.3 Relation of m and the estimation error

Given the estimation error of $\sigma = 1.04/\sqrt{m}$ it is clear that an increasing m should result in a smaller error, i.e. a more precise estimation of n:

$$\hat{n} = n\,(1 + \sigma) = n\left(1 + 1.04/\sqrt{m}\right)$$

Generating 100 different sets of size $n = 1.000.000$ of distinct integers allows us to compute $\hat{n}$ a 100 times and record the frequency of $\hat{n}$. The frequency should follow the normal distribution, i.e. around 65% of the estimations should be within $n \pm n\sigma$ and around 95% of the estimations should be within $n \pm n2\sigma$. As observed in the histograms presented in figure **??** (a), (b), (c) and (d), increasing m results in neater and neater normal distributions.

In order to evaluate different combinations of m and n we have conducted an experiment where we have computed the fraction of $\hat{n}$ that

resulted within one and two standard deviations of the true value. We decided to use values of n such that $n = \{10.000, 100.000, 10.000.000\}$ and m such that $m = \{64, 256, 1024, 4096\}$. The results of our experiment reveal interesting insights about the effect of different sizes of m and n on the implementation of *HyperLogLog*.

We note that an increase of the input size does not necessarily increase the prediction accuracy. Additionally, an increase of the size of m does not generate a higher accuracy when n is set equal in every case. Especially for small and large values of n, 10.000 and 10.000.000, our experiment did not suggest a clear linear relationship between a larger m and a higher accuracy.

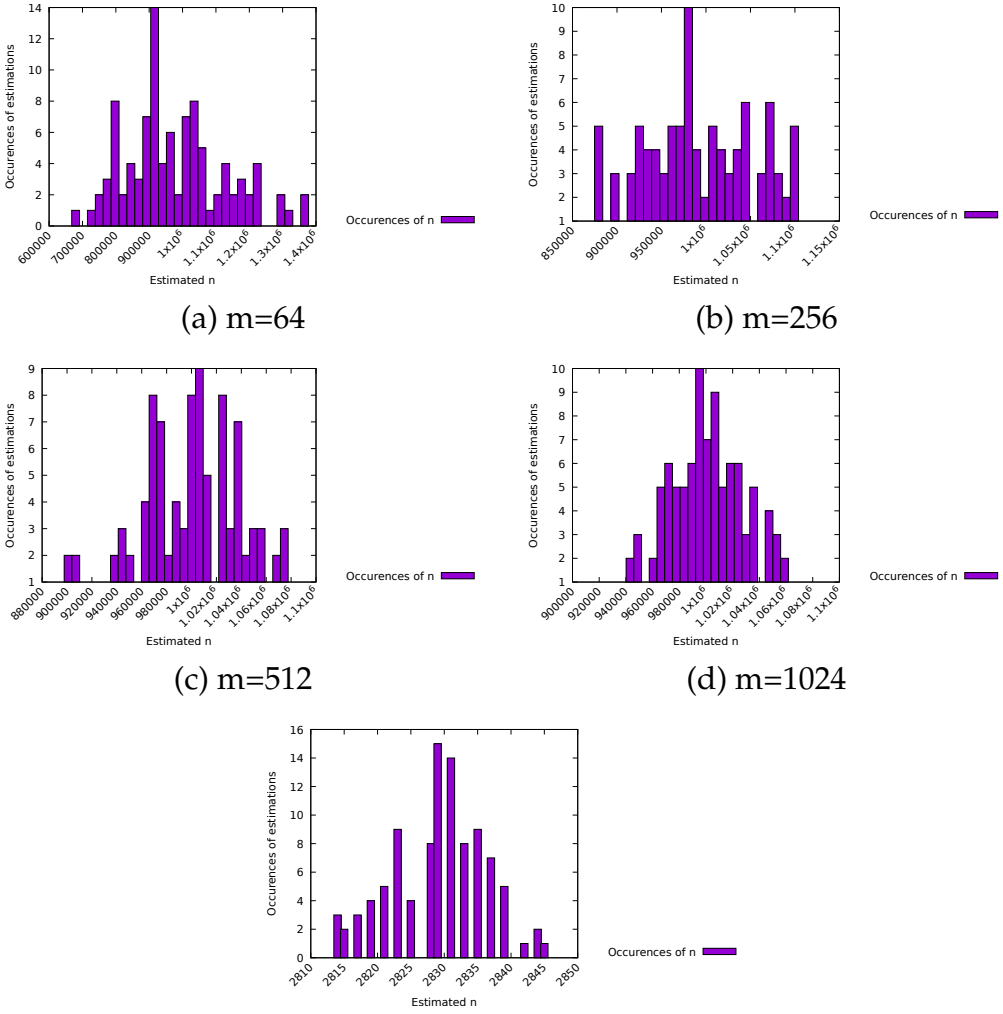
Whereas most of the observations fall within the expected range of the *empirical rule*, i.e. that 68% of the observed data will occur within the first standard deviation, and 95% will take place in the second deviation, we do still observe outliers in our experiments.

First, the estimation is very accurate for $n = 10.000$ with $m = 64$, but drops significantly when increasing n by a factor of 10.

Secondly, we observe a substantially low accuracy for $n = 10.000$ and $m = 4096$. We hypothesize that the latter is due to the fact that the register size is too large for the relatively small input size. This would lead to few empty registers and a misleading estimation of *HyperLogLog*, because a high ρ-value in a register would not explain a lot of exchanges for this register anymore. To be more precise, a good hash function should distribute the hashed values uniformly, which is why we would expect to have at least one element in each register, but around two for $n = 10.000$ and $m = 4096$. However, *HyperLogLog* makes an important assumption: if there is a high ρ-value in a register, several other elements should have been mapped to the same register before, with lower ρ-value. This is clearly not the case anymore with a ratio of m and n as given in this example. This is how we explain the high deviation in this case.

Table 1: Results, 100 runs

n	m	$n \pm 1\sigma$	$n \pm 2\sigma$
10000	64	79%	99%
100000	64	68%	93%
1000000	64	59%	94%
10000000	64	73%	95%
10000	256	66%	96%
100000	256	70%	93%
1000000	256	61%	98%
10000000	256	69%	95%
10000	1024	76%	97%
100000	1024	69%	97%
1000000	1024	69%	96%
10000000	1024	68%	98%
10000	4096	26%	64%
100000	4096	71%	97%
1000000	4096	69%	93%
10000000	4096	70%	98%



(a) m=64

(b) m=256

(c) m=512

(d) m=1024

(e) m=4028

Figure 2: Relation of m and the estimation error