Applied Algorithms

Mandatory Assignment #2

November 5, 2019

IT UNIVERSITY OF COPENHAGEN

Distinct elements using hashing

Authors	
Hugo Brito	**René Haas**
hubr@itu.dk	renha@itu.dk

Introduction

In this report we show our findings with the HyperLogLog algorithm for estimating distinct elements. All our experiments are reproducible and can be rerun by using the `Makefile`. If the reader wishes to rerun our experiments simply `cd` to the `src` folder and type `make`.

Exercise 1.

In this first exercise we are given a matrix $A \in \{0,1\}^{b \times k}$ where $b = k = 32$. We are asked to implement the hash function

$$h(x)_i = \sum_{j=1}^{b} A_{i,j} x_j \mod 2$$

We implement the function in Java (see `Ex1.java`) with the following code:

```
1 public static int h(int x) {
2     int res = 0;
3     for (int i = 0; i < A.length; i++) {
4         res += (Integer.bitCount(A[i] & x) % 2) << (31-i);
5     }
6     return res;
7 }
```

We start by calculating the bit-wise & (and) operation between x and the rows of A. The result of this operation is a binary number which is equivalent to a vector consisting of the element-wise products of the components of x and the i^{th} row of A. We then count the sum with `Integer.bitCount` and then take the modulus 2 of that number. This gives us an integer which is either 0 or 1. We make this the integer the j bit in the result by using the shift operator «. This implementation passes the tests on CodeJudge.

Exercise 2.

We are asked to implement the function $\rho(y) = \min\{i \mid y_{k-i} = 1\}$. We do this the easy way in Java by simply using the `Integer.numberOfLeadingZeros` function from the Java standard library and adding 1 to the result. We add 1 because we are interested in the first position that is not 0. The implementation can be found on the `Ex2.java` file and looks like the following:

```
1 public static int rho(int x) {
2     if (x == 0) { throw new InputMismatchException("Zero is undefined"); }
3     return Integer.numberOfLeadingZeros(x);
4 }
```

It is claimed that the distribution of hash values of ρ satisfies $Pr(\rho(y) = i) = 2^{-i}$. In Figure 1 we show a histogram over the values of $\rho(x)$ for $x \in \{1, \cdots, 10^6\}$. We find that there is direct experimental support for this claim, although in reality $Pr(\rho(y) = i) = 2^{-i}/2$. This is verified experimentally in Fig 1 and the factor of $1/2$ is also theoretically necessary since $\sum_{i=1}^{\infty} = 2$ and a probability distributions must be normalised.

Exercise 3.

We have implemented the algorithm and this implementation for the values $m = 1\,024$ and $k = 32$ can be found in `Ex3.java` file. This file also passes all instances of CodeJudge.
For the given m, k and the given sequence of distinct integers $x \in \{10^6, \cdots, 2 \times 10^6 - 1\}$ the implementation of this algorithm estimates that there are $973\,089,272$ distinct values. The error[1] is then $|1\,000\,000 - 973\,089,272| = 26\,910,728$, which translates to $2,69\%$.

[1] Taken from thoughtco.com

Exercise 4.

Lastly, we wanted to find the co-relation between the space usage of the HyperLogLog counter influences its estimation error.

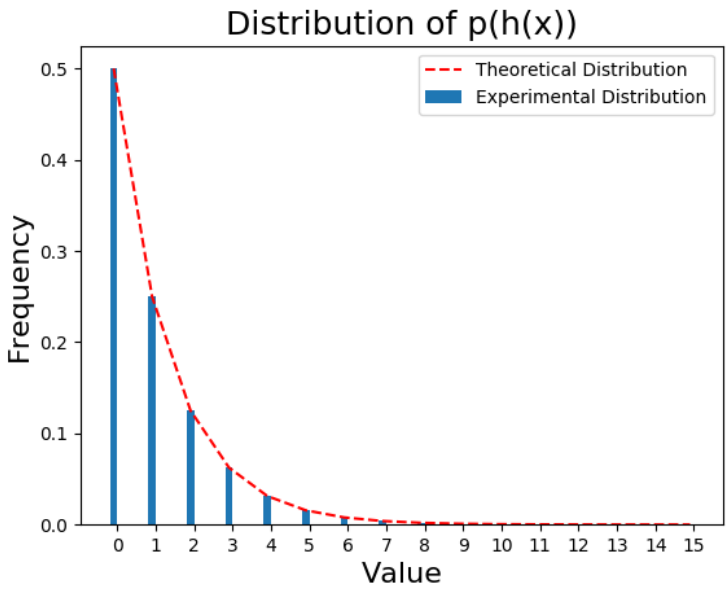

Figure 1: Histogram of $\rho(x)$ for $x \in \{1, \cdots, 10^6\}$

The file `Ex4.java` implements the input generator to be used by the HyperLogLog algorithm. We wanted to be certain that the algorithm would always estimate the number of distinct integers and compare that estimation to the real number of distinct integers. In order to achieve this machinery, the above-mentioned generator would add the generated integers to set. Below the code snippet that implements this description:

```
1 do {
2   int i = a + random.nextInt(-a + b + 1);
3   if (i != 0) { // zero cannot be used because it has no leading zeros and
        cannot be hashed
4     integers.add(i);
5     // omitted for brevity
6   }
7 } while (integers.size() < n);
```

Variable description:

- `a`: The lower bound of the random generator
- `b`: The upper bound of the random generator
- `integers`: The set of distinct integers
- `n`: The real number of distinct integers

The file `Experiment.java` implements experiment we designed to assess the relation between the m and the estimation. It takes a set of

distinct integers and m as constructor parameters and outputs all the information needed to report on this relation, namely:

- The estimation
- A `boolean` value that would evaluate to `true` should the estimation fall within one σ range
- A `boolean` value that would evaluate to `true` should the estimation fall within 2σ range
- σ
- The lower and upper bounds of the σ range
- The lower and upper bounds of 2σ range

Finally, the file `main.java` combines all the above and outputs the results to a file. In itself, it contains a `long` generator that was initialised with the `seed` 42. The output of this `long` generator is then used as `seed` in the `Ex4.java` file to generate sets of distinct integers. Once the set is created, 3 `Experiment` are instantiated with different values of $m \in \{16, 256, 1024\}$. In each iteration a new set is generated with an unused `seed` (we garantee this by keeping a set `usedSeeds`) this is passed to the 3 instances. In order to achieve significant data, this process is repeated 100 000 times.

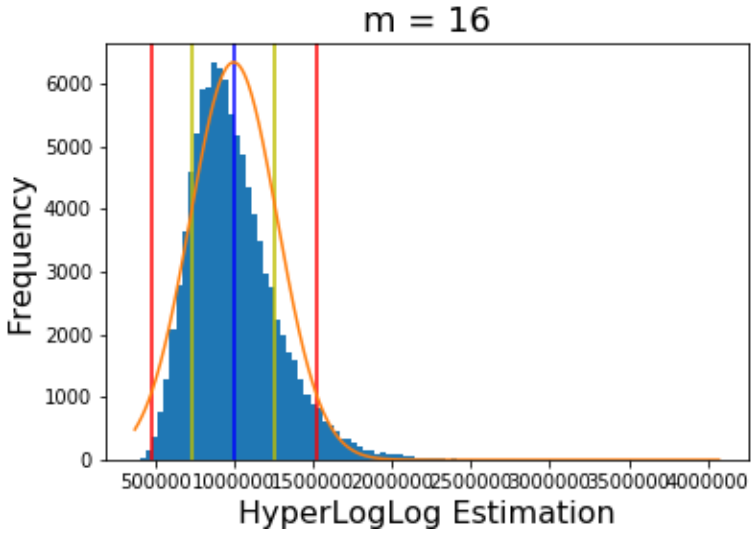

Figure 2: HiperLogLog estimation distribution for $m = 16$ over 100 000 distinct sets of 1 000 000 distinct integers

In figures 2, 3 and 4 we see histograms that resulted from these experiments. In the figures the yellow vertical lines represent $n(1 \pm \sigma)$ range and the red lines represent $n(1 \pm 2\sigma)$ range. We have also fitted

a Gaussian distribution and plotted that on top of the histogram in orange.

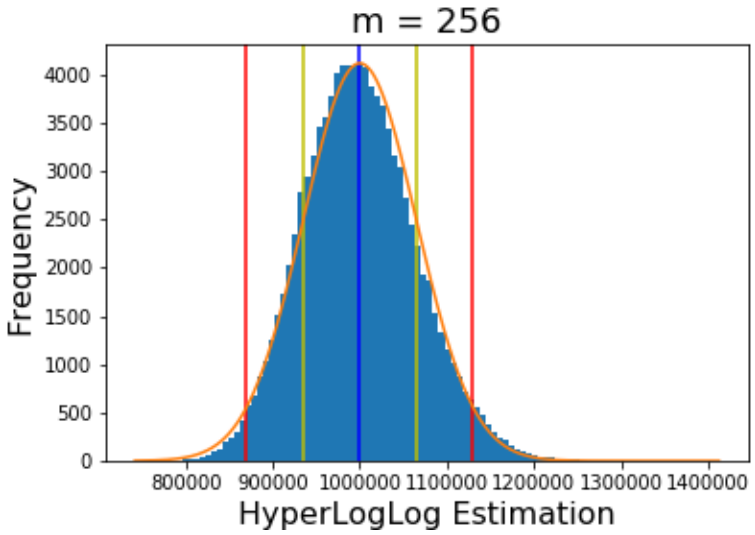

Figure 3: HiperLogLog estimation distribution for $m = 256$ over 100 000 distinct sets of 1 000 000 distinct integers

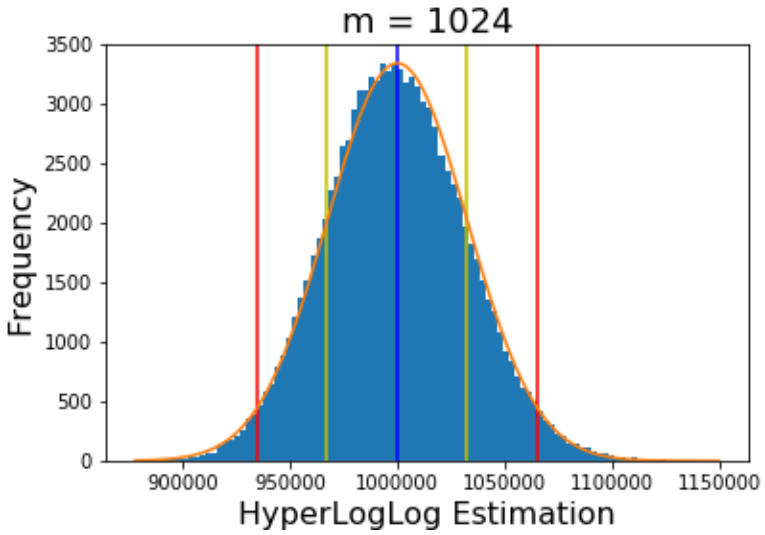

Figure 4: HiperLogLog estimation distribution for $m = 1\,024$ over 100 000 distinct sets of 1 000 000 distinct integers

We can see that the error distribution follows a normal Gaussian distribution. In Table 1 we see the fraction of runs that fall inside $n(1 \pm \sigma)$ and $n(1 \pm 2\sigma)$ for the three different values of m respectively.

	$m = 16$	$m = 256$	$m = 1024$
$n(1 \pm \sigma)$	69,453%	68,596%	68,521%
$n(1 \pm 2\sigma)$	95,042%	95,476%	95,523%

Table 1: Fractions of runs that fall inside $n(1 \pm \sigma)$ and $n(1 \pm 2\sigma)$ for different ms

Conclusion

The HyperLogLog algorithm is an effective strategy to estimate the cardinality of a set of integers as it does not require to keep track of the integers, saving memory space. We could also see that allocating to space (buckets) by increasing the size of m does not influence the distribution of the error, which always follows the Gaussian distribution. It is important to mention that being σ a function of m, decreasing m will also increase the error range, resulting in estimations that are further away from the real value in absolute terms. Se there is some gain in for instance increasing m from 16 to 256, but this gain is less expressive from 256 to 1024.