Assignment 2: HyperLogLog

Martin Aumüller and Holger Dell

Hand-in date: 2025-10-08 11:59

The assignment is to be solved in groups of 1–3 people.

1. Introduction

Your task is to implement a variant of the HyperLogLog algorithm [FFGM07] for estimating the number of distinct elements in a stream of integers. The variant you will be implementing differs slightly from the description of [FFGM07]. You will also need to implement the auxiliary algorithms that your implementation depends on, particularly the hash function and the function ρ. You will need to test your implementations for correctness, perform experiments where you determine empirically certain statistical properties of some of your subroutines, and the estimation error of your implementation. You can use any programming language that you like, but the description in the assignment will assume that it is Java.

In this assignment, it will be important to remember that every `int` x can be seen both as an integer between -2^{31} and $2^{31}-1$ and as a sequence $x \in \{0,1\}^{32}$ of 32 bits. For example, recall from the lecture that `0b00111101` is just another way to write the integer 61 in Java because we have $0 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 61$. Also, `0b11111111111111111111111111111111` (all 32 bits are set to `1`) represents the integer -1 due to the two's complement representation of negative numbers. In the HyperLogLog algorithm, it is useful to ignore the actual integer that is being represented (and ignore whether it is "positive" or "negative"), and only focus on the fact that an `int` stores 32 bits.

2. Implementing the hash function

Your task. Since HyperLogLog requires access to a high-quality hash function h as a subroutine, your first task is to implement a suitable function `int h(int x)`. The purpose of the hash function is to assign a random-looking sequence of 32 bits to each integer that is given in the input. You can implement any hash function that you wish; however, the hash function needs to be of high quality since HyperLogLog may not work well otherwise.

A suitable hash function. We now mathematically describe a hash function that satisfies the requirements of the assignment. The function $h_A \colon \{0,1\}^b \to \{0,1\}^k$ we define maps b-bit strings to k-bit strings; to implement `h` in this way, you need to set $b = k = 32$, and remember that 32 bits correspond to one `int`. Let A be a binary $(k \times b)$-matrix, that is, a matrix with k rows and b columns, and all entries are either 0 or 1. For all $x \in \{0,1\}^b$, we define $h_A(x)$ using the matrix-vector product via $h_A(x) = Ax \bmod 2$. In other words, for each $i \in \{1,\dots,k\}$, the i-th bit $(h_A(x))_i$ of the output can be written as a sum modulo two:

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

In yet other words, the i-th bit of the output is the *parity* of the inner product $\langle A_i, x\rangle$ of the i-th row A_i of the matrix A with the input vector x. The parity of an integer is simply 1 if and only if the integer is odd and 0 otherwise; thus this is equal to taking the integer modulo two. Taken together, we can write $(h_A(x))_i = (\langle A_i, x\rangle \bmod 2)$.

Small example. Let $b = k = 4$, and define A and x via the following equations:

$$A = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{bmatrix}, \qquad x = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \end{bmatrix}.$$

Then different ways of writing and then evaluating $h_A(x)$ are as follows:

$$\begin{aligned} h_A(x) = Ax &= \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \end{bmatrix} \bmod 2 = \begin{bmatrix} \langle \texttt{1111}, \texttt{1011}\rangle \bmod 2 \\ \langle \texttt{0010}, \texttt{1011}\rangle \bmod 2 \\ \langle \texttt{0111}, \texttt{1011}\rangle \bmod 2 \\ \langle \texttt{1101}, \texttt{1011}\rangle \bmod 2 \end{bmatrix} \\ &= \begin{bmatrix} (1\cdot 1 + 1 \cdot 0 + 1 \cdot 1 + 1 \cdot 1) \bmod 2 \\ (0\cdot 1 + 0 \cdot 0 + 1 \cdot 1 + 0 \cdot 1) \bmod 2 \\ (0\cdot 1 + 1 \cdot 0 + 1 \cdot 1 + 1 \cdot 1) \bmod 2 \\ (1\cdot 1 + 1 \cdot 0 + 0 \cdot 1 + 1 \cdot 1) \bmod 2 \end{bmatrix} = \begin{bmatrix} 3 \\ 1 \\ 2 \\ 2 \end{bmatrix} \bmod 2 = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}. \end{aligned}$$

Gaining efficiency for $b = k = 32$. While one could implement the computation of $h_A(x)$ by explicitly storing the matrix as an `int[][]` two-dimensional array, and each vector as an `int[]` array, doing so would be very wasteful since each entry of these arrays would be using 32 bits, even though we only need to store 1 bit for each entry.[1] Moreover, there would be many multiplication, addition, and modulo operations involved to compute the matrix-vector product. A much more efficient solution is to pack each *row* A_i of A into an `int` and represent x as an `int` as well. Then the inner product $\langle A_i, x\rangle$ can be computed very efficiently by taking the bitwise AND of the two `int`s and then applying an intrinsic operation called `Integer.bitCount`, which for a given `int` returns the number of bits set to `1`. For example, $\langle \texttt{0111}, \texttt{1011}\rangle \bmod 2$ can be computed by taking the bitwise AND, which yields `0011`. Then applying `Integer.bitCount` yields the integer 2. Finally, taking the bitwise AND of 2 with the integer 1 yields the parity of the inner product, which in this case is 0.

[1] Unfortunately, each `boolean` typically uses up at least 8 bits due to memory-alignment requirements of the Java Virtual Machine (JVM), and so this would not help much either.

Further specifications for your implementation. Your implementation for `h` should never explicitly unpack `int` values into an array of bits or vice-versa. Instead, use efficient operations directly on the `int`s themselves, such as arithmetic operations, bit operations, and `Integer.bitCount`. If you choose to implement the hash function h_A, use the integers in Appendix A as the rows of matrix A.

3. Implementing the function ρ

HyperLogLog makes heavy use of the function $\rho\colon \{0,1\}^k \to \mathbb{N}$ defined via

$$\rho(x) = \min\{i \mid x_i = 1\}.$$

That is, $\rho(x)$ is the position of the first `1` in the binary representation of the binary string x read from left to right. For example, for $k = 8$, we have $\rho(\texttt{11010000}) = 1$, $\rho(\texttt{00010000}) = 4$, and $\rho(\texttt{00000001}) = 8$. Note that $\rho(\texttt{00000000})$ is undefined.

Your task. Implement the function `int rho(int x)` for $k = 32$. It is acceptable to implement the function using basic bit operations, even though this might be a bit slow in practice. A faster and arguably simpler implementation uses the intrinsic function `Integer.numberOfLeadingZeros` in Java, please look up on your own what it does.

Note. There are correct implementations that are one line short.

4. Evaluating the quality of the hash function

Using the hash function h from Section 2 and the function ρ from Section 3, design and perform an experiment where you determine the distribution of $\rho(h(x))$ for one million hash values with $x \in \{1, \dots, 10^6\}$. Create a plot of the result and add it to your report. Discuss whether the results of your experiment support the claim that the distribution of the hash values of ρ satisfy $\Pr[\rho(y) = i] = 2^{-i}$ for all i from 1 to k for random $y \in \{0,1\}^k$.

5. Implementing HyperLogLog

Implement the HyperLogLog algorithm as presented in Algorithm 1. The algorithm corresponds to Figure 3 in [FFGM07], but has been slightly modified. The algorithm takes as input a (potentially very large) *stream*[2], and computes an estimate on the *cardinality* of the input stream, that is, the number of distinct elements in the stream. The original algorithm has the number of registers m as the parameter, but you can start with a fixed value of 1024 in this section; later, you will need to evaluate other values of m as well.

[2] A read-only sequence.

Algorithm 1 Pseudocode for HyperLogLog. Use $m = 1024$ and the following “magic” constant $\alpha_m = 0.7213/(1 + 1.079/m)$.

Input: a stream of integers Y
Output: An estimate of the cardinality of the stream $\hat{n}$
function HyperLogLog(Y)
  for $j \leftarrow 1, 2, \ldots, m$ **do**
    $M[j] \leftarrow 0$ ▷ Initialize registers
  end for
  for $y \in Y$ **do**
    $j \leftarrow f(y)$ ▷ Hash function f selects the register
    $x \leftarrow h(y)$ ▷ Hash function h is used for ρ computations
    $M[j] \leftarrow \max\{M[j], \rho(x)\}$ ▷ Update the register
  end for
  $\hat{n} \leftarrow \alpha_m m^2 \cdot \left(\sum_{j=1}^{m} 2^{-M[j]}\right)^{-1}$ ▷ The raw estimate
  $V = |\{j | M[j] = 0\}|$ ▷ The number of empty registers
  if $\hat{n} \le \frac{5}{2}m$ and $V > 0$ **then**
    return $m \ln(m/V)$ ▷ Apply linear counting
  end if
  if $\hat{n} > \frac{1}{30}2^{32}$ **then**
    $\hat{n} \leftarrow -2^{32} \ln\left(1 - \frac{\hat{n}}{2^{32}}\right)$ ▷ Large range correction
  end if
  return $\hat{n}$
end function

The algorithm works by initializing a set of m registers denoted by $M[j]$ for $j = 0, 1, \ldots, m-1$, iterating over all elements in the input stream, using a hash function f to select a register, hashing each element with the hash function h, and updating the number of the corresponding register to be the maximum ρ value encountered thus far. Finally, the estimate is counted using the harmonic mean.

Note that there is a correction coefficient denoted by α_m, depending on the number of registers. Since we fix $m = 1024$, use $\alpha_m = 0.7213/(1 + 1.079/m) \approx 0.7205$.

The estimate is adjusted for small and large cardinalities. In the case the estimate is very small (with respect to m), linear counting is used instead. In the case the estimate is large, a different correction is used.

Use the hash function from Section 2 as h, and the ρ function from Section 3. For the hash function $f\colon \mathbb{Z} \to \{0, 1, \ldots, m-1\}$, you can use the following function: `f(x) = ((x*0xbc164501) & 0x7fffffff) >> 21`.

Test your implementation on the input sequence $10^6, 10^6 + 1, \ldots, 2 \cdot 10^6 - 1$ with 1 million distinct items. Include the estimate reported by your implementation in your report.

6. Estimation error

Write an input generator that takes as input an integer n and a random seed, and outputs a list of n *distinct* random 32-bit integers. Describe and run an experiment that gives a graphical representation of the connection of m and the estimation error. A recommended representation is a histogram plot over the distinct element count reported by the algorithm. Try at least 3 different values of m. Report in a table for each m and n the fraction of runs that reported a value in $n(1 \pm \sigma)$ and $n(1 \pm 2\sigma)$ for $\sigma = 1.04/\sqrt{m}$.

Note that the formula $\alpha_m = 0.7213/(1 + 1.079/m)$ only holds for $m \geq 128$. If you want to use smaller m, check [FFGM07] for the corresponding values of the correction coefficient. Also note that the hash function f from Section 5 for choosing the register index j maps to the integer range $\{0, 1, \ldots, 1023\}$; you will need to adapt your hash function to work with different values of m.

7. Report

Write a report in LaTeX, following the advice given in the lectures and in the feedback you got on your report for the first assignment. The report must not exceed **4 pages**, **excluding possible tables, figures, and references**. Hand in the report and the code you produced as a **single zip file** through LearnIT. Your group only needs to hand in one report.

References

[FFGM07] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. "HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm". In: *Discrete Mathematics & Theoretical Computer Science* DMTCS Proceedings vol. AH, 2007 Conference on Analysis of Algorithms (AofA 07) (Jan. 2007). DOI: `10.46298/dmtcs.3545`. URL: `https://dmtcs.episciences.org/3545`.

Appendix

A. The integers for the matrix A

Below you find 32 different 32-bit integers, each of which corresponds to one row of the matrix A in the suggested hash function from Section 2.

```
0x21ae4036
0x32435171
0xac3338cf
0xea97b40c
0x0e504b22
0x9ff9a4ef
0x111d014d
0x934f3787
0x6cd079bf
0x69db5c31
0xdf3c28ed
0x40daf2ad
0x82a5891c
0x4659c7b0
0x73dc0ca8
0xdad3aca2
0x00c74c7e
0x9a2521e2
0xf38eb6aa
0x64711ab6
0x5823150a
0xd13a3a9a
0x30a5aa04
0x0fb9a1da
0xef785119
0xc9f0b067
0x1e7dde42
0xdda4a7b2
0x1a1c2640
0x297c0633
0x744edb48
0x19adce93
```