Algorithms and Probability (FS2025)
Week 9

Rui Zhang

April 16, 2025

Contents

1 Content

This week, we will take the probabilistic methods we've developed and start looking at algorithms that profit from all our theory.

1.1 Target Shooting

Target Shooting is something we have already looked at a few weeks ago. Back then we established the following:

Theorem: For the Target Shooting Algorithm which performs N iterations, the output Z is a random variable, which has expectation $|S|/|U|$ and variance

$$\frac{1}{N}\left(\frac{|S|}{|U|}-\left(\frac{|S|}{|U|}\right)^2\right)$$

And thus we established that for an increasing number of iterations N, we will get more and more accurate results. However, this is only a qualitative statement. We can derive the following quantitative statement about target shooting via Chernoff:

Theorem: Let $\delta, \varepsilon > 0$. If we let Target Shooting run with at least $N \geq 3\frac{|U|}{|S|}\varepsilon^{-2}\ln\left(\frac{2}{\delta}\right)$, iterations, then the result Z is within the interval

$$\left[(1-\varepsilon)\frac{|S|}{|U|}, (1+\varepsilon)\frac{|S|}{|U|}\right]$$

with probability at least $1-\delta$

1.2 Hashing

Hashing is a technique used often in programming to speed up datastructures and processes. We will look at one such example here: Assume you have an array of n elements $D = (s_1, s_2, \ldots, s_n)$. A pair is a duplicate (i, j) if $s_i = s_j$. Our job now is to find all duplicates in this array.

Naive Solution:

Take the array and sort it. Afterwards, we will scan the array once and for every sequence of consecutive elements, we will save all combinations of their indices as pairs / duplicates. The runtime for sorting is (?) and the runtime for the scan is $O(n + |Dupl(D)|)$, where $Dupl(D)$ are all possible duplicate-pairs in this array.

Note that this solution will not work if the elements are not easily comparable or are expensive to access.

Hashing Solution:

To solve the previous problem, we will introduce hash functions. The idea is that we map each element to a number using a hash function.

Theorem: Formally, a hash function is a function

$$h : U \to [m]$$

such that

$$\forall u \in U \ \forall i \in [m] \quad \Pr[h(u) = i] = \frac{1}{m}$$

where U is a so called "universe", which is honestly just a fancy term which says that the elements in the array D stems from this set (or universe or whatever). Practically, this means that the probability of the hash function assigning a value i to an element u is uniformly distributed. In particular, we have that

Theorem: Let $s, t \in U$, then $s = t \Rightarrow h(s) = h(t)$

Note that the other direction does not hold.

Now, our solution consists of calculating the hash for all elements in the array, sorting the hashes and finding the duplicates via a scan just like before, this time on the hashes. However, there is a problem: We might have collisions:

Definition: Collisions are the unwanted false positives we get from this approach: i.e. when we have two elements s, t such that $s \neq t$ but $h(s) = h(t)$ still holds because we got unlucky. The algorithm would then say "oh no there's a duplicate here!" But in reality there is only a duplicate on the level of the hashes, not the original elements.

As with all probabilistic algorithms, we deal with these sort of problems by adjusting parameters until the probability of this occurrence is very low. Therefore, we will now analyze how often we get collisions:

Let $K_{i,j}$ be the indicator variable for the event "(i, j) is a collision". We have

$$\Pr[K_{i,j} = 1] = \begin{cases} 1/m & \text{if } s_i \neq s_j \\ 0 & \text{else} \end{cases} \quad \Rightarrow \mathbb{E}[K_{i,j}] \leq 1/m$$

And with the number of collisions being $K = \sum_{1 \leq i < j \leq n} K_{i,j}$, we apply linearity of expectation and get

$$\mathbb{E}[K] \leq \binom{n}{2} \frac{1}{m}$$

And if we choose $m = n^2$, we get that the expected number of collisions is less than 1, and we can deal with that!

With this solution, we still keep a relatively good runtime and $O(n \log n)$ additional space needed for saving the hash values, which is doable.

1.3 Bloom Filters

Bloom Filters are a different way to solve the above problem, with an array D. This time, we will define a repeated entry instead of a duplicate:

Definition: A repeated entry in $D = (s_1, \dots, s_n)$ is an index $j \in [n]$ such that there exists a previous index $i \in [j-1]$ which has the same element $s_i = s_j$

Note that if we can find these repeated entries, then we can reconstruct all duplicates from them. This is efficient if we don't have that many duplicates.

The idea now is to let the bloom-filter work and give us as a result a list L which contains all repeated entries in D and false positives with a very small probability.

Definition: Let $m, k \in \mathbb{N}$ and $h_1, \dots, h_k : U \to [m]$ be k random hash functions and M a boolean array of length m initially set to all 0's.
Then, a bloom filter operates as follows:

1. Iterate through the array of elements D
2. For each element $s \in D$, calculate the hash vector $v = (h_1(s), \dots h_k(s))$
3. For each entry $h_i(s) \in v$ of the hash vector, check if $M[h_i(s)]$ is set to 1. If not, set to 1.
4. If all $h_i(s)$ was set to 1, then add s to the list of repeated entries L

Note that with this procedure, just like with hashing, we only have false positives yet not false negatives:

Theorem: If s_i is a repeated entry, then its index i will be in list L:

$$s_i \in \{s_1, \dots, s_{i-1}\} \Rightarrow i \in L$$

We will from now refer to the false positives (entries in L but which are not repeated entries) as "incorrect entries" and would like to calculate $\mathbb{E}$[number of incorrect entries in L]. To this end define the indicator variable

$$X_i = \text{"indicator variable for when } i \text{ is in incorrect entry"}$$

and note that

$$X_i = 1 \Leftrightarrow \begin{cases} s_i \notin (s_i, \dots s_i - 1) \text{ and} \\ M[h_1(s_i)] = M[h_2(s_i)] = \dots = M[h_k(s_i)] = 1 \end{cases}$$

And if we look at an individual entry $M[h_1(s_i)]$, we get:

$$\Pr\left[M[h_1(s_i)] = 0\right] = (1 - \frac{1}{m})^{kl} \geq (1 - \frac{1}{m})^{k(i-1)}$$

where l is the number of non-repeated entries in the $i-1$ previous elements the algorithm had to go through before reaching s_i. Note that this inequality is true as the base is less than 1 and the exponent is larger on the right hand side. It leaves us with the information that

$$\Pr\left[M[h_1(s_i)] = 1\right] = 1 - \Pr\left[M[h_1(s_i)] = 0\right] \leq 1 - (1 - \frac{1}{m})^{k(i-1)}$$

And since we want $M[h_j(s_i)] = 1$ to be true for all $j \in [i-1]$, we have that

$$\Pr[X_i = 1] = \Pr[\forall j \in [i-1]\ M[h_j(s_i]] = 1] \lessapprox (1 - (1 - \frac{1}{m})^{k(i-1)})^k$$

and with $\mathbb{E}[X_i] = \Pr[X_i = 1]$ and number of incorrect entries in $L = X_1 + X_2 + \cdots + X_n$, we get

$$\begin{aligned}\mathbb{E}[\text{number of incorrect entries}] &\leq n \cdot (1 - (1 - \frac{1}{m})^{k(i-1)})^k \\ &\leq n \cdot (1 - e^{-kn/m})^k\end{aligned}$$

And choosing $k = \ln n$ and $m = n \ln n$ gives us a constant number on the right hand side. This gives us a runtime of kn hashing operations and an extra required space of m for the array M

2 Kahoot!!

`https://quizizz.com/admin/quiz/67ffa780c785c72c8d5c6259`