CS 388R: Randomized Algorithms Fall 2015

Lecture 13 — October 14, 2015

Prof. Eric Price *Scribe: John Kallaugher, Elliot Meyerson*

1 Overview

This lecture is about the use of probabilistic "fingerprints" to test equality for objects that are large or expensive to compute explicitly.

2 Matrix Multiplication

Given $A, B, C \in \mathbb{R}^{n\times n}$, we would like to know whether:

$$AB = C$$

The most obvious way to check this would be to directly compute AB, and then check whether it is equal to C. This takes $O(n^3)$ time using the naive method for matrix multiplication, which can be reduced to $O(n^{2.373})$ with horrible constants through more advanced methods.

Suppose, instead, we choose a random $r \in \mathbb{R}^n$, and check whether:

$$ABr = Cr$$

Then clearly this will hold if $AB = C$. But what if $AB \neq C$?

Then $AB - C$ has rows $(v_i)_{i\in[n]}$, at least one of which is non-zero, and $ABr = Cr$ will hold iff $v_i r = 0$ for all i.

So we would like to bound $\mathbb{P}[vr | v \neq 0]$, and thus the probability that $ABr = Cr$ when $AB \neq C$.

Let r be drawn uniformly from $\{0,1\}^n$, and let i be an index such that $v_i \neq 0$. Then let r^a_{-i} be the event that $(r_1, \dots, r_{i-1}, r_{i+1}, \dots, r_n) = a$ for any $a \in \{0,1\}^{n-1}$.

$$\begin{aligned}
\mathbb{P}[vr = 0] &= \sum_a \mathbb{P}\left[r_i v_i = -\sum_{j\neq i} r_j v_j \middle| r^a_{-i} \right] \mathbb{P}[r^a_{-i}] \\
&\leq \max_a \mathbb{P}\left[r_i v_i = -\sum_{j\neq i} r_j v_j \middle| r^a_{-i} \right] \\
&\leq \frac{1}{2}
\end{aligned}$$

As r_i is uniformly distributed among 2 values, at most one of which can satisfy the equation.

Similarly, by drawing r from $[k]^n$, we can achieve $\mathbb{P}[vr|v \neq 0] \leq \frac{1}{k}$.

If the entries of A, B, C are bounded by $n^{O(1)}$, then for any integral constant $c > 0$ we can choose $k = n^c$.

So $vr \leq n^c n \max_i v_i \leq n^{O(1)}$, so no calculation in creating the fingerprint requires more than a constant number of words, with the word length w is assumed to satisfy $2^w = \theta(n)$. As there are $O(n^2)$ such calculations, we can compute the fingerprint in time $O(n^2)$ for a failure probability of n^{-c}.

This method does not work for finite fields, as there may be as few as 2 distinct values that the fingerprint can take. However, in this case, we can repeat the fingerprinting $c \log_2 n$ times to get failure probability n^{-c} in $O(n^2 \log n)$ time.

3 Polynomial identity testing

We can also use fingerprinting to answer questions like "$P(x)Q(x) = R(x)$?", where P, Q, R are polynomials of degree $d, d, 2d$, resp. (or more generally check whether some factored form of a polynomial is equal to another).

Suppose

$$P(x) = a_0 x^d + a_1 x^{d-1} + ... + a_{d-1} x + a_d$$
$$Q(x) = b_0 x^d + b_1 x^{d-1} + ... + b_{d-1} x + b_d$$
$$R(x) = c_0 x^{2d} + c_1 x^{2d-1} + ... + c_{2d-1} x + c_{2d}$$

The naiive deterministic computation of $(P \cdot Q)(x)$ takes $O(d^2)$ time, which using convolution and the FFT can be reduced to $O(d \log d)$.

However, simply evaluating a degree $O(d)$ polynomial for any random input x takes only $O(d)$ time: first sequentially compute each power of x, then multiply each by its coefficient, and take the sum. So for any random x we can check $P(x)Q(x) = R(x)$, or equivalently $P(x)Q(x) - R(x) = 0$, in $O(d)$ time. Let $T = P \cdot Q - R$.

Suppose x is drawn uniformly at random from a set S. If $T = 0$, then $\mathbb{P}[T(x) = 0] = 1$. Suppose $T \neq 0$, then T has at most $2d$ roots in S, so $\mathbb{P}[T(x) = 0] \leq \frac{2d}{|S|}$.

To get an $O(\frac{1}{d})$ failure probability we might think to choose x from $[d^2]$. Unfortunately, if we assume our word size is $\approx \log(d)$, then the powers of d each take on average $O(d)$ space to represent, and thus evaluating $T(x)$ takes $O(d^2)$ time.

To address this space issue, let's instead check if $P(x)Q(x) \equiv R(x) \mod p$, where p is a prime larger than any coefficient of these polynomials, but still of size $O(1)$ words. Since p is larger than any of the coefficients, doing the $\mod p$ doesn't actually change the equality. There are still at most $2d$ roots of T in $\mathbb{Z}_p$, so the failure probability $\leq \frac{2d}{p}$ if x is drawn from $[p]$.

When T is multivariate in $x, y, ...$, if we choose $x, y, ...$ from $\mathbb{Z}_p$ i.i.d., then $\mathbb{P}[T(x, y, ...) = 0]$ is still at most $\frac{2d}{p}$, where $2d$ here is the total degree of T (by Schwartz-Zippel Lemma; try to prove at home if you'd like).

4 String matching

Suppose Alice has an n-bit string $a = a_1...a_n$, and Bob has an n-bit string $b = b_1...b_n$ (suppose for simplicity of argument that a and b are binary). Alice can send a single message M to Bob, and Bob would like to determine whether $a = b$. How large does M have to be?

One idea is for Alice to choose a random hash function h and send $h(a)$. If Alice and Bob have shared randomness (e.g., look at some opening stock price), they can compute h separately, M can consist only of $h(a)$. However, if they do not have shared randomness, Alice must send along h itself, which may make M large.

4.1 Rabin-Karp style hash

Another idea is to always use the polynomial $Q(x) = \sum a_i x^i$. That is, Bob checks if $\sum(a_i - b_i)x^i = 0$. Since Q has at most n roots, if we choose x uniformly at random from a set of size $\geq n^2$, we can ensure failure probability $\leq \frac{1}{n}$. As in polynomial identity checking we can consider instead $Q(x) \bmod p$, where p is some prime such that $p \geq n^2$ but p can still be represented in $O(\log n)$ bits. Then Alice sends x, p and $Q(x) \bmod p$ to Bob, each of which are of size $O(\log n)$ bits, so the total size of M is $O(\log n)$ bits.

4.1.1 The Rabin-Karp algorithm

The $h(x) = Q(x) \bmod p$ above is the standard hash function for the Rabin-Karp algorithm for finding occurrences of a *pattern* in a *text*.

Setup: Let T be an n-bit string (the *text*), and b be an m-bit string (the *pattern*), with $m < n$. Problem: Find all i s.t. $T_i...T_{i+m-1} = b$.

The naiive method of brute force checking each such substring takes $O(mn)$ time.

Instead let's take $h(b)$ and compare it to $h_j = h(T_j...T_{j+m-1})\ \forall j$. Although each h_j would take $O(m)$ time to compute from scratch, h_{j+1} can be computed in $O(1)$ time given h_j, since

$$h_j = \sum_{i=1}^{m} T_{j+i-1} x^{m-i},$$

and

$$h_{j+1} = \sum_{i=1}^{m} T_{j+i} x^{m-i} = h_j x - T_j x^m + T_{j+m}.$$

$\implies$ $O(m+n)$ time total to compute all h_j. There is a high probability of success, and only false positives, so we can use this to construct a corresponding Las Vegas algorithm by doing an exhaustive check on every positive in $O(m)$ time each. This Las Vegas algorithm has expected time $O(n + am)$, where a is the actual number of occurrences of b in T.

4.2 An Alternative Method: Treating a and b as Integers

We can instead treat our bitstrings a, b as integers $\sum_i a_i 2^i$, $\sum_i b_i 2^i$, and compare them modulo a random prime p with $p \leq k$, with k to be decided later. Clearly if $a = b$, $a - b \equiv 0 \mod p$, so what is the probability that if $a \neq b$, $a - b \equiv 0 \mod p$?

$a - b \equiv 0 \mod p$ iff p is a factor of $a - b$. These are n-bit integers, and therefore strictly smaller than 2^{n+1}, so $a - b$ has no more than n prime factors, as any prime factor must be at least 2.

Using the fact that the proportion of $[k]$ that is prime is $\theta\left(\frac{k}{\log k}\right)$, we may choose $k = O(n^2 \log n)$ to get n^2 primes $\leq k$. This will then give us a failure probability $\leq \frac{1}{n}$, and we will need $O(\log k) = O(\log n)$ communication to share p and our fingerprint.

4.2.1 Generating a Random Prime

We can repeatedly choose a random $x \in [k]$ and return the first one we find that is prime. Using the previously mentioned fact about the density of primes, we can expect to find a prime within $O(\log k)$ iterations. However, efficiently checking whether x is prime is non-trivial.

One randomised method for primality testing is to use the fact that the polynomials $P_1 = (X+1)^x$ and $P_2 = X^x + 1$ are equal iff x is prime. However, we still need to test whether these two polynomials are equal. A natural idea would be to evaluate them at a randomly chosen value of X, but unfortunately this does not work. Instead, we can choose a random Q of degree polylog in x, and check whether $P_1 \equiv P_2 \mod Q$. The details of this are outside the scope of this lecture, but it allows us to evaluate the primality of x in polylog time.