Fingerprints, Polynomials, and Pattern Matching

Notes by Alistair Sinclair

1 Example 1: Checking Matrix Multiplication

Somebody gives you some code that purports to multiply matrices. You want to check it on some sample inputs. So you need an algorithm that solves the following problem:

Input : three $n \times n$ real matrices A, B, C.

Question : is $C = A \times B$?

Of course, you could just multiply A and B and compare the answer with C; this would take $O(n^{2.81})$ time in practice, and $O(n^{2.38})$ time even in theory. You'd like to do the checking in much less time than matrix multiplication itself.
Here's a very simple randomized algorithm, due to Freivalds, that runs in only $O(n^2)$ time:

> pick a vector $r \in \{0,1\}^n$ u.a.r.
> **if** $A(Br) = Cr$ **then** output "yes" **else** output "no"

The algorithm runs in $O(n^2)$ time since it does only 3 matrix-vector multiplications. (Check this.)
Why does it work? Well, if $AB = C$ then the output will always be "yes". (Why?) But if $AB \neq C$ then the algorithm will make a mistake if it happens to choose an r for which $ABr = Cr$. This, however, is unlikely:

Claim: If $AB \neq C$ then $\Pr[ABr = Cr] \leq \frac{1}{2}$.

We'll prove the claim in a moment. First, note that it implies:

- If $AB = C$ then the algorithm is always correct.
- If $AB \neq C$ then the algorithm makes an error with probability $\leq \frac{1}{2}$.

To make the error probability in the second case very small (say, less than ϵ, the probability that an earthquake will strike while we are running the computation), we just run $t = \log_2(1/\epsilon)$ trials and output "yes" unless some trial says "no". (Check this.)

Proof of Claim: Define $D = AB - C$. Our assumption is that $D \neq 0$. We want to show that $\Pr[Dr = 0] \leq \frac{1}{2}$ for randomly chosen r.
Since $D \neq 0$, it has a non-zero entry, say d_{ij}. Now the ith entry of Dr is $\sum_k d_{ik} r_k$. And if $Dr = 0$ then this entry must certainly be zero. But this happens iff $r_j = -(\sum_{k \neq j} d_{ik} r_k)/d_{ij}$; i.e., once the values r_k for $k \neq j$ have been chosen, there is only one value for r_j that would give us zero.
But r_j itself is being chosen from the two possible values $\{0, 1\}$, so at least one of these choices must fail to give us zero. (Note that the numerical values here are unimportant — it matters only that there are two different choices.)

Hence $\Pr[Dr = 0] \leq \frac{1}{2}$, as claimed. $\square$

Ex: In the above proof, where did we use the fact that $d_{ij} \neq 0$? $\square$

Ex: Suppose that, instead of picking the entries of r from the set $\{0, 1\}$, we picked them from a larger set S. Show that the failure probability of a single run of the algorithm would then be at most $\frac{1}{|S|}$. By making S larger, we can thus reduce the error probability. Comment on the relative merits in practice of this scheme and the method based on repeated trials mentioned above. $\square$

2 Example 2: Checking Polynomial Identities

A very frequently used operation in computer algebra systems is the following: given two polynomials, Q_1, Q_2, with multiple variables $x_1, \ldots, x_n$, are Q_1, Q_2 identical, i.e., is $Q_1(x_1, \ldots, x_n) = Q_2(x_1, \ldots, x_n)$ at all points $(x_1, \ldots, x_n)$? How does the system answer such a question?

Obviously, if Q_1, Q_2 are written out explicitly, the question is trivially answered in linear time just by comparing their coefficients. But in practice they are usually given in very compact form (e.g., as products of factors, or as determinants of matrices), so that we can evaluate them efficiently, but expanding them out and looking at their coefficients is out of the question.

Ex: Consider the polynomial

$$Q(x_1, x_2, \ldots, x_n) = \prod_{i<j: i,j \neq 1} (x_i - x_j) - \prod_{i<j: i,j \neq 2} (x_i - x_j) + \prod_{i<j: i,j \neq 3} (x_i - x_j) - \cdots \pm \prod_{i<j: i,j \neq n} (x_i - x_j).$$

Show that evaluating Q at any given point can be done efficiently, but that expanding out Q to find all its coefficients is computationally infeasible even for moderate values of n. $\square$

Here is a very simple randomized algorithm, due to Schwartz and Zippel. Testing $Q_1 \equiv Q_2$ is equivalent to testing $Q \equiv 0$, where $Q = Q_1 - Q_2$. So we focus on this problem.

Algorithm

> pick $r_1, \ldots, r_n$ independently and u.a.r. from a set S
> **if** $Q(r_1, \ldots, r_n) = 0$ **then** output "yes" **else** output "no"

This algorithm is clearly efficient, requiring only the evaluation of Q at a single point. Moreover, if $Q \equiv 0$ then it is always correct. (Why?)
In the Theorem below, we'll see that if $Q \not\equiv 0$ then the algorithm is incorrect with probability at most $\frac{1}{2}$, provided the set S is large enough. We can then reduce this error probability to ϵ by repeated trials as in the previous example.

Theorem: Suppose $Q(x_1, \ldots, x_n)$ has degree at most d.[1] Then if $Q \not\equiv 0$ we have

$$\Pr[Q(r_1, \ldots, r_n) = 0] \leq \frac{d}{|S|}.$$

[1] The degree of a polynomial is the maximum degree of any of its terms. The degree of a term is the sum of the exponents of its variables. E.g., the degree of $Q(x_1, x_2) = 2x_1^3 x_2 + 17 x_1 x_2^2 - 7x_1^2 + x_1 x_2 - 1$ is 4.

With this theorem, to get the error probability $\le \frac{1}{2}$ we just take $|S| = 2d$. Thus we could take, e.g., $S = \{1, 2, \ldots, 2d\}$.

Proof of Theorem: Note that the theorem is immediate for polynomials in just one variable (i.e., $n = 1$); this follows from the well-known fact that such a polynomial of degree d can have at most d zeros.
To prove the theorem for n variables, we use induction on n. We have just proved the base case, $n = 1$, so we now assume $n > 1$.
We can write Q as

$$Q(x_1, \ldots, x_n) = \sum_{i=0}^{k} x_1^i Q_i(x_2, \ldots, x_n),$$

where k is the largest exponent of x_1 in Q.
So $Q_k(x_2, \ldots, x_n) \not\equiv 0$ by our definition of k, and its degree is at most $d - k$. (Why?)
Thus by the induction hypothesis we have that $\Pr[Q_k(r_2, \ldots, r_n) = 0] \le \frac{d-k}{|S|}$.
We now consider two cases, depending on the random values $r_2, \ldots, r_n$ chosen by the algorithm.

Case 1: $Q_k(r_2, \ldots, r_n) \ne 0$. In this case the polynomial $Q'(x_1) = \sum_{i=0}^{k} Q_i(r_2, \ldots, r_n) x_1^i$ in the single variable x_1 has degree k and is not identically zero, so it has at most k zeros. I.e., $Q'(r_1) = 0$ for at most k values r_1. But this means that $\Pr[Q(r_1, \ldots, r_n) = 0] \le \frac{k}{|S|}$. (Why?)
Case 2: $Q_k(r_2, \ldots, r_n) = 0$. In this case we can't say anything useful about $\Pr[Q(r_1, r_2, \ldots, r_n) = 0]$. However, we know from the above discussion that this case only happens with probability $\le \frac{d-k}{|S|}$.

Now let $\mathcal{E}$ denote the event that $Q_k(r_2, \ldots, r_n) \ne 0$. Putting together cases 1 and 2 we get:

$$\begin{aligned}
\Pr[Q(r_1, \ldots, r_n) = 0] &= \Pr[Q(r_1, \ldots, r_n) = 0 \,|\, \mathcal{E}] \Pr[\mathcal{E}] + \Pr[Q(r_1, \ldots, r_n) = 0 \,|\, \overline{\mathcal{E}}] \Pr[\overline{\mathcal{E}}] \\
&\le \Pr[Q(r_1, \ldots, r_n) = 0 \,|\, \mathcal{E}] + \Pr[\overline{\mathcal{E}}] \\
&\le \tfrac{k}{|S|} + \tfrac{d-k}{|S|} \;=\; \tfrac{d}{|S|}.
\end{aligned}$$

This completes the proof by induction. □

3 Fingerprinting

Both of the above are examples of fingerprinting. Suppose we want to compare two items, Z_1 and Z_2. Instead of comparing them directly, we compute random fingerprints $\textsc{fing}(Z_1), \textsc{fing}(Z_2)$ and compare these. The fingerprint function FING has the following properties:

- If $Z_1 \ne Z_2$ then $\Pr[\textsc{fing}(Z_1) = \textsc{fing}(Z_2)]$ is small.
- It is much more efficient to compute and compare fingerprints than to compare Z_1, Z_2 directly.

For Freivalds' algorithm, if A is a $n \times n$ matrix then $\textsc{fing}(A) = Ar$, for r a random vector in $\{0, 1\}^n$.
For the Schwartz-Zippel algorithm, if Q is a polynomial in n variables then $\textsc{fing}(Q) = Q(r_1, \ldots, r_n)$, for r_i chosen randomly from a set S of size $2d$.
We give two further applications of this idea.

4 Example 3: Comparing Databases

Alice and Bob are far apart. Each has a copy of a database (of n bits), a and b respectively. They want to check consistency of their copies.

Obviously Alice could send a to Bob, and he could compare it to b. But this requires transmission of n bits, which for realistic values of n is costly and error-prone. Instead, suppose Alice first computes a much smaller fingerprint $\textsc{fing}(a)$ and sends this to Bob. He then computes $\textsc{fing}(b)$ and compares it with $\textsc{fing}(a)$. If the fingerprints are equal, he announces that the copies are identical.

What kind of fingerprint function should we use here?

Let's view a copy of the database as an n-bit binary number, i.e., $a = \sum_{i=0}^{n-1} a_i 2^i$ and $b = \sum_{i=0}^{n-1} b_i 2^i$.

Now define $\textsc{fing}(a) = a \bmod p$, where p is a prime number chosen at random from the range $\{1, \ldots, k\}$, for some suitable k. We want $\Pr[\textsc{fing}(a) = \textsc{fing}(b)]$ to be small if $a \neq b$.

Suppose $a \neq b$. When is $\textsc{fing}(a) = \textsc{fing}(b)$? Well, for this to happen, we need that $a \bmod p = b \bmod p$, i.e., that p divides $d = a - b \neq 0$. But d is an (at most) n-bit number, so the size of d is less than 2^n. This means that at most n different primes can divide d. (Why?)

So: as long as we make k large enough so that the number of primes in the range $\{1, \ldots, k\}$ is much larger than n we will be in good shape. To ensure this, we need a standard fact from Number Theory:

Prime Number Theorem: Let $\pi(k)$ denote the number of primes less than k. Then $\pi(k) \sim \frac{k}{\ln k}$ as $k \to \infty$. $\square$

Now all we need to do is set $k = cn \ln(cn)$ for any c we like. By the Prime Number Theorem, with this choice of k,

$$\Pr[\textsc{fing}(a) = \textsc{fing}(b) | a \neq b] \leq \tfrac{n}{\pi(k)} \sim \tfrac{1}{c}.$$

So, if we take $c = \frac{1}{\epsilon}$ we will achieve an error probability less than ϵ.

Finally, note that Alice only needs to send to Bob the numbers $a \bmod p$ and p (so that Bob knows which fingerprint to compute), both of which are at most k. So the number of bits sent by Alice is at most $2 \log_2 k = O(\log n)$.

Ex: We did not explain how Alice selects a random prime $p \in \{1, \ldots, k\}$. What she does, in fact, is to generate a random number in $\{1, \ldots, k\}$, test if it is prime, and if not throw it away and try again. (The test for primality can be done efficiently in time polynomial in $\log n$ using a randomized algorithm which you may have seen in CS170.) Use the Prime Number Theorem to show that the expected number of trials Alice has to perform before she hits a prime is $\sim \ln k = O(\log n)$. $\square$

5 Example 4: Randomized Pattern Matching

Consider the classical problem of searching for a pattern Y in a string X. I.e., we want to know whether the string $Y = y_1 y_2 \ldots y_m$ occurs as a contiguous substring of $X = x_1 x_2 \ldots x_n$. The naïve approach of trying every possible match takes $O(nm)$ time. (Why?) There is a rather complicated deterministic algorithm that runs in $O(n + m)$ time (which is clearly best possible). A beautifully simple randomized algorithm, due to Karp and Rabin, also runs in $O(n+m)$ time and is based on the same idea as in the above example.

Assume for simplicity that the alphabet is binary. Let $X(j) = x_j x_{j+1} \ldots x_{j+m-1}$ denote the substring of X of length m starting at position j.

Algorithm

> pick a random prime p in the range $\{1, \ldots, k\}$
> **for** $j = 1$ **to** $n - m + 1$ **do**
>   **if** $X(j) = Y \bmod p$ **then** report match and stop

The test in the **if**-statement here is just $\text{FING}(X(j)) = \text{FING}(Y)$ for the same fingerprint function FING as in the Alice and Bob problem.

If the algorithm runs to completion without reporting a match, then Y definitely does not occur in X. (Why?) So the only error the algorithm can make is to report a false match. What is the probability that this happens? By the same analysis as above, for each j if $X(j) \neq Y$ then $\Pr[\text{FING}(X(j)) = \text{FING}(Y)] \leq \frac{m}{\pi(k)}$. Therefore, if $Y \neq X(j)$ for all j, we have

$$\Pr[\text{algorithm reports a match}] \leq \tfrac{nm}{\pi(k)} \sim \tfrac{1}{c}$$

if we choose $k = cnm \ln(cnm)$ (exactly as before).

What about the running time? Well, in a naïve implementation each iteration of the loop requires the computation of a fingerprint of an m-bit number, which takes $O(m)$ time, giving a total running time of $O(nm)$. However, this can be improved by noticing that

$$\text{FING}(X(j+1)) = 2(\text{FING}(X(j)) - 2^{m-1}x_j) + x_{j+m} \bmod p.$$

(Check this.) Hence, under the realistic assumption that arithmetic operations on fingerprints — which are small — can be done in constant time, each iteration actually takes only constant time (except the first, which takes $O(m)$ time to compute $\text{FING}(X(1))$ and $\text{FING}(Y)$).
The overall running time of the algorithm is therefore $O(n+m)$ as claimed earlier.

Ex: In practice, we would want to eliminate the possibility of false matches entirely. To do this, we could make the algorithm test any match before reporting it. If it is found to be a false match, the algorithm could simply restart with a new random prime p. The resulting algorithm never makes an error. Show that its expected running time is at most $\frac{c}{c-1}T \approx T$, where T is the running time of the original algorithm, and that the probability it runs for at least $(\ell+1)T$ time is at most $c^{-\ell}$. □