CS 388R: Randomized Algorithms Fall 2017

Lecture 10 — Oct. 3, 2017

Prof. Eric Price *Scribe: Jianwei Chen, Josh Vekhter*

NOTE: THESE NOTES HAVE NOT BEEN EDITED OR CHECKED FOR CORRECTNESS

1 Overview

In the last lecture we talked about routing algorithms.

In this lecture we study fingerprinting. It is used when we want to check if two large files/strings/ matrices/polynomials are the same or not. One application of these techniques is verifying that a file you downloaded hasn't been tampered with.

2 Matrix Multiplication

To begin, let's consider the following problem: Given matrices $A, B, C \in \mathbb{R}^{n\times n}$, how do we know if $AB = C$?

The simplest approach is to compute the value of AB using a deterministic algorithm and check if it is equal to C. There have been numerous algorithmic attempts to improve the efficiency of this computation, bounding it as $O(n^w)$ for various exponents:

- For naive approach, $w = 3$
- For Strassen's Algorithm (1969), $w = 2.8074$
- For Coppersmith–Winograd's algorithm (1990), $w = 2.3755$
- For Andrew Strothers' (2010), $w = 2.374$
- For Virginia Williams' (2011), $w = 2.37286...$

It is a long standing open conjecture that the $O(n^2)$ computations are possible. It is also known that this lower bound is sharp, because it is necessary to read all of the entries of both matrices to be confident that they are the same.

However, there also exists a randomized algorithm that can easily solve this problem in $O(n^2)$ time. Simply choose a random vector $r \in \{0, 1\}^n$, and check whether $ABr = Cr$.

Now we need the analyze the possibility of that $AB \neq C, ABr = Cr$. We are going to show that $\mathbb{P}[ABr \neq Cr] \leq \frac{1}{2}$.

As $AB \neq C \iff AB - C \neq 0$, there exists a nonzero row $v = \{v_1, v_2, \ldots, v_n\}$ of $AB - C$. Suppose $v_i \neq 0$. To do the analysis we assume that we first picked all $r_j (j \neq i)$, then

$$\begin{aligned}
\mathbb{P}\left[\sum v_k r_k = 0\right] &= \mathbb{P}\left[v_i r_i + \sum_{j \neq i} v_j r_j = 0\right] \\
&= \mathbb{P}\left[r_i = \frac{\sum_{j \neq i} v_j r_j}{v_i}\right] \\
&\leq \frac{1}{2}
\end{aligned}$$

Here we use the fact that $\sum_{j \neq i} v_j r_j / v_i$ can only be 0 or 1 (or neither) and r_i is chosen randomly among 0 and 1, this gives us the probability of at most half.

3 Polynomial Matching

Now we consider another class of finger printing problems also in the spirit of partial evaluation: Checking if $P(x) * Q(x) = R(x)$.[1]

To be a bit more concrete, let's suppose we know (or can quickly estimate) the degree of each polynomial. For convenience, let's say that both P and Q are of degree d and are reduced (i.e. they have the form $a_1 x^d + a_2 x^{d-1} + \cdots + a_d x + a_{d_1}$).

In this case, the naïve approach yields a $O(d^2)$ algorithm to reduce this product (wuddup FOIL method).

A more sophisticated deterministic approach can reduce this product in $O(dlogd)$ time (looking at you FFT algorithms)!

Another natural question to ask about polynomials is if they are equal to the zero polynomial (i.e. $p(x) = 0$ for all x). Note that this is a more general problem than polynomial comparison (because given polynomials $p(x)$ and $q(x)$, one natural way to test equality is to check if $p(x) - q(x) = 0$ for all x). Here the naïve (deterministic) approach is to apply the test the polynomial at any $d+1$ points. If they are all zero then the polynomial must be zero everywhere by the fundamental theorem of algebra (recall that this theorem says a mono-variate polynomial over $\mathbb{C}$ has exactly d roots, and note that it is possible to achieve an analogue on finite fields via the euclidean algorithm).

But there's a practical issue here because often polynomials come in unreduced forms. As a back of the envelope argument, note that a d degree polynomial can be represented as the product of $O(2^{\sqrt{d}})$ strings with distinct terms (because each power of x^k may or may not appear in each string). It may take up to $O(d)$ time to evaluate each pairwise product, and there are potentially exponentially many of these. As a more concrete example, note that the naïve algorithm for computing the determinant of a matrix generates a multi-variate polynomial with $O(d!)$ terms to reduce. More generally, it can be shown that it's possible to construct a correspondence between evaluating zeros in multi-variate polynomials over $\mathbb{F}^2$ and 3SAT (see [1]).

Thus a natural question to ask is: can we determine if a polynomial $p(x)$ is the zero polynomial with less than $d + 1$ evaluations w.h.p? Suppose we restrict ourselves to the case where $p(x)$ is

[1] A more general version of this formulation is verifying if $\prod P_i(x) = Q(x)$

evaluated over the finite field of size B. In this case, our chance of selecting a root of the polynomial is $\leq \frac{d}{B}$. Thus if we set $B = 2d$ then at each trial, we have a $\frac{1}{2}$ chance of selecting a root which implies that after k trials, if each time the polynomial evaluated to zero, then the probability that it is not the zero polynomial is $\frac{1}{2^k}$, which allows us to determine if $p(x)$ is the zero polynomial in $O(1)$ evaluations w.h.p.

One detail worth noting here is that in order for the above argument to work, B actually needs to be relatively prime to all factors of $p(x)$, i.e. we need to find a prime L such that L is prime and $L > 2d$. By [Bertrand]'s Postulate, such a prime must exist between $n < p < 2n$, and so we know that $L < 4d$. It's also worth noting that a very similar argument gives rise to the Schwartz–Zippel lemma for testing zeros in multi-variate polynomials (which is more surprising because non-zero multivariate polynomials can have infinitely many roots)[2][3].

4 String Fingerprinting

Let's consider an application of the probabilistic polynomial matching algorithm described above. Suppose Alice has a string $a = a_1 a_2 \dots a_n \in \{0,1\}^n$, Bob has has a string $b = b_1 b_2 \dots b_n \in \{0,1\}^n$, and we want to compute randomized fingerprints to check if $a = b$.

A natural idea is that if Alice and Bob have some shared seed of randomness, they can both compute a hash of their strings and send the hashed value as the fingerprint. However, if they don't share such randomness, we might need to send the hash function along with the fingerprints.

4.1 Rabin-Karp Hashing

We can use the idea in fingerprinting polynomials and treat the strings as polynomials. This is Rabin-Karp Hashing Algorithm. Namely, Alice use a random x computes $\sum a_i x^i$ and send x and the $\sum a_i x^i$, Bob computes $\sum b_i x^i$ and check if they are the same. This can be done using $O(\log n)$ bits using the previous method learned in polynomial matching.

4.2 Another Method

When we express those strings as binary vectors, we can consider another equivalent setting of the problem: We have $a \in [2^n]$ and $b \in [2^n]$, we want to know if $a = b$.

One way is to check if $(a - b) \equiv 0 \pmod{p_i}$, p_i is a chosen prime in some way. Then we need to ask how many primes can divide a n-bit number $a - b$? We know that $\pi(n)$ (the number of primes below n) has such relationship: $\pi(n) \sim \frac{n}{\log n} < n$. Suppose we choose p_i uniformly at random from primes that is less and equal to B, our intuition is to make B reasonably small compared to the n-bit a and b and guarantee that our checking don't have a high failing rate. If we set this failing

rate to $1/n$, we have

$$\begin{aligned}\text{chance of failure } &= \frac{\#\text{primes can divide (a-b) and below B}}{\#\text{ primes} \le B} \\ &\le \frac{\#\text{primes that can divide (a-b)}}{\theta\left(B/\log B\right)} \\ &\le \frac{n}{\theta\left(B/\log B\right)} \\ &\le \frac{1}{n}\end{aligned}$$

Therefore, when we set $B = n^2 \log n$, we satisfy the above inequality. In this algorithm, all we need to transfer is a random prime p below B, and $a \bmod b$ to the other end, all of them take $O(\log n)$ bits.

Still, one question remains: how to pick a prime uniformly at random? Here we propose a simple method: just pick a number randomly and check if it's random, if the number is random, return this number otherwise repeat the process. There are certain deterministic algorithm to check if n is a prime, we would like to mention a randomized version.

We can use the fact that

$$n \text{ is a prime } \iff x^n + 1 = (x+1)^n$$

. To test if those two expression are equal, we can use $x^n + 1$ and $(x+1)^n$ both mod a random degree($\approx \log n$) polynomial Q over $\mathbb{Z}_n$. The details of this are outside the scope of this lecture, but it allows us to evaluate the primality of x in polylog time.

5 Pattern Matching

The next problem we are going to study is somewhat relevant to the previous section: Given string $a = a_1 a_2 \dots a_n$ and string $b = b_1 b_2 \dots b_m (m < n)$, we want to know if b is a substring of a.

This problem is well studied and its time complexity has been bounded to $O(n+m)$ (By the famous KMP algorithm). Today we're going to show an randomized algorithm with similar bound but easier to implement.

We can see that we can break this problem to $n - m + 1$ subproblems that involves checking if $a_i a_{i+1} \dots a_{i+m}$ equals $b_1 b_2 \dots b_m$, this is where we can use the Rabin-Karp method we previously mentioned. However, if we use this hashing algorithm in a naive way which is to compute the hash fresh from start, we are going to get a really bad $O(mn)$ (same as deterministic brute force!) time complexity. Actually, after some investigation we can find that the following problem we encounter in computing hash continuously can be solved in $O(1)$:

$$\text{Given } x = f\left(a_{k+1} a_{k+2} \dots a_{k+m}\right) = \sum_{i=k+1}^{k+m} a_i x^{i-k-1} \text{ mod p}$$

$$\text{Find } y = f\left(a_{k+2} a_{k+3} \dots a_{k+m+1}\right) = \sum_{i=k+2}^{k+m+1} a_i x^{i-k-2} \text{ mod p}$$

We shall use the following equation:$y = 2x + a_{m+1} - a_i \cdot 2^m$ to iteratively compute the hash value of the substring of a after the first one.

If we use the Monte Carlo version of this algorithm, we have a straight $O(m+n)$ time complexity, and Las Vegas version time complexity will be $O\left(m + n + |L| \cdot m\right)$, in which $|L|$ is the occurrence of hash matching among the $m - n + 1$ substrings of a, and the term of $|L| \cdot m$ is used to check the actual equality of the substring and pattern.

References

[Bertrand] https://en.wikipedia.org/wiki/Proof_of_Bertrand%27s_postulate

[1] https://en.wikipedia.org/wiki/Arithmetic_circuit_complexity

[2] Jacob T Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM (JACM)* , 27(4):701717, 1980.

[3] Richard Zippel. *Probabilistic algorithms for sparse polynomials Springer, 1979.*