CS 388R: Randomized Algorithms, Fall 2023 September 27th, 2023

Lecture 11: Fingerprinting

Prof. Eric Price *Scribe: Ronak Ramachandran, Marlan McInnes-Taylor*

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

1 Overview

In the last lecture we explored problems in routing. Today, we'll be covering a new topic: fingerprinting. Fingerprinting is useful when you want to check whether two values X and Y are equal, but it would take too long to compute and compare the values bit-by-bit. The idea is to instead compute and compare hashes of the values, $h(X)$ and $h(Y)$. Note that $h(X) \neq h(Y) \implies X \neq Y$ with certainty, and with the right choice of hash family, $h(X) = h(y) \implies X = Y$ with high probability.

2 Example 1: Checking Matrix Multiplication

Consider the following problem: given $A, B, C \in \mathbb{R}^n$, check whether $AB = C$. One way to solve this would be to calculate AB and compare it to C, but this would take $O(n^\omega)$ time (here, ω is the matrix multiplication exponent, currently known to be $\omega < 2.373$). Can we do better?

One thing we might try is to pick a random vector $r \in \{0, 1, \dots, 2^k - 1\}^n$ and see whether $A(Br) = Cr$. Since we only need to do 3 matrix-vector multiplications (we never actually calculate AB), this only takes $O(n^2)$ time.

Claim 1. *If $AB \neq C$,*

$$\mathbb{P}[ABr = Cr] \leq \frac{1}{2}.$$

Proof. $AB \neq C \implies AB - C \neq 0$, so there exists a non-zero row v of $AB - C$. Then

$$\mathbb{P}[v \cdot r = 0] \leq \max_{r_j \text{ s.t. } i \neq j} \mathbb{P}_{r_i}\left[v_i r_i + \sum_{j \neq i} v_j r_j = 0 \mid r_{\neq i} \right] \leq \frac{1}{2}.$$

where the last step follows because there is at most one assignment of r_i that results in the correct sum given some fixed values for the other elements of r.

$\square$

3 Example 2: Polynomial Identity Testing

In polynomial identity testing, we given a polynomial p of degree d and we need to check whether $p = 0$. If $p \neq 0$, then p has at most d roots, so we could evaluate $p(0), \dots, p(d)$ and then check whether all $d+1$ of these evaluations are 0. If all evaluations are 0, we can output "YES," and if not, then we can output "NO." It's hard to say how efficient this is but we know:

1. This needs $d+1$ evaluations.
2. $p(d)$ might be big ($\approx d^d$, which takes d bits to store).

We can do fewer evaluations by picking a random x from $\{0, \dots, m-1\}$. Then

$$\mathbb{P}[p(x) = 0 \mid p \neq 0] \leq \frac{d}{m} \leq \frac{1}{4},$$

when we choose m to be $4d$. We can reduce the storage size for evaluations of p by evaluating mod p.

Our new scheme then becomes: (1) Pick $x_1, \dots, x_k \in [p]$ uniformly at random, and (2) return whether $p(x_1, \dots, x_k) = 0$, where here $p(x_1, \dots, x_k) := p(x_1) + \dots + p(x_k)$. By the Schwartz-Zippel lemma,

$$\mathbb{P}[p(x_1, \dots, x_k) = 0 \mid p \neq 0] \leq \frac{d}{p}.$$

4 Example 3: String Matching

We shall now consider the problem of string searching. Ultimately, we desire to check whether string b is a substring of a larger string a. Imagine we have an n-bit string a and m-bit string b where $m \leq n$. A naive approach is to simply compare the m-bits of b to m-bit subpatterns of a at each of a's n possible positions, which takes $\mathrm{O}(nm)$ time. We shall examine the string searching algorithm proposed by Karp and Rabin [RK87] which utilizes hashing and randomness to offer a better bound on performing such a search.

4.1 String Hashing

First, lets consider how to compare two n-bit strings.

We desire a hash function $\mathrm{h}(\cdot)$ s.t. given arbitrary strings x, y if $x \neq y$ then $\mathrm{h}(x) \neq \mathrm{h}(y)$ with high probability.

Let:

$$\mathrm{h}(x) = \mathrm{h}_x(c) = \sum_{i=1}^{n} c^i \cdot x_i \pmod p$$
$$\mathrm{h}(y) = \mathrm{h}_y(c) = \sum_{i=1}^{n} c^i \cdot y_i \pmod p$$

Where:

$$x, y \in \{0,1\}^n$$
$$p \text{ is a fixed prime s.t. } p > 4n$$
$$c \text{ is a random value where } c \in [p-1]$$

We shall treat the x_i's and y_i's above as coefficients of a polynomial. Using this construction we simple check if $h(x) - h(y) = 0$ to determine their equivalence.

This has a false positive rate of $\frac{n}{p}$ which is the greatest chance that a random choice for $c \in [p-1]$ is chosen as one of the n roots of $\sum c^i \cdot x_i$ or $\sum c^i \cdot y_i$.

4.2 Rabin-Karp Algorithm [RK87]

We shall now extend the approach above to perform pattern matching where strings are of different lengths. We will first partition a into length m substrings, and hash these substrings along with string b using the the approach outlined in Section 4.1. We then compare the hashed string b to the hashed substrings of a as above. If we find a match, output `YES`, else output `NO`.

Let:

$$\mathrm{h}_{a_1}(c) = \sum_{i=1}^{m} c^{m-i} \cdot a_i \pmod p$$
$$\mathrm{h}_{b}(c) = \sum_{i=1}^{m} c^{i} \cdot b_i \pmod p$$

Where:

$$a \in \{0,1\}^n$$
$$b \in \{0,1\}^m$$
$$m \leq n$$

Then, to quickly compute the next substring hash:

$$\mathrm{h}_{a_2}(c) = \sum_{i=2}^{m+1} c^{m-i+1} \cdot a_i = a_{m+1} + c \cdot h_{a_1}(c) - a_1 c^{m-1}$$

For each partition of a we check if $h_{a_i} - h_b = 0$. Computing the next hash takes constant time, thus the overall algorithm takes $O(n+m)$ time. Using the union bound over all length m substrings of a, the expected number of false positives is at most $n \cdot \frac{m}{p}$.

5 Primality Testing

How do we find a prime within some range of integers? One option is to repeatedly pick a random number and then test whether it's prime - in expectation, this will require $O(\log p)$ tries before success. How do we quickly check whether an integer is prime? Naively, we might check every possible factor of p - this will take $O(\sqrt{p})$ time, which is $O(2^{n/2})$ for an n-bit prime. We will now examine two more sophisticated techniques.

5.1 Fermat

A better option is Fermat's Primality Test, which uses Fermat's Little Theorem.

Theorem 2. *(Fermat's Little Theorem) If p is prime, then $a^{p-1} \equiv 1 (\text{mod } p)$ for all $a \neq 0 (\text{mod } p)$.*

The idea is given p, we pick a random $a \in \{1, \ldots, p-1\}$, and if $a^{p-1} \not\equiv 1 (\text{mod } p)$, we output NO, otherwise, "probably YES." We say probably YES, because $\exists a$ s.t. $a^{p-1} \equiv 1 (\text{mod } p)$ even if p is not prime. In fact for a special set of composite numbers called the Carmichael numbers, $a^{p-1} \equiv 1 (\text{mod } p)$ for all $a \neq p$. The number of Carmichael numbers less than x is known to be $\geq x^{0.33} = o(x)$.

5.2 Miller-Rabin [M75] [R80]

Similar to Fermat's Primality Test, the Miller-Rabin Primality Test checks if a given number n demonstrates particular properties which hold for prime numbers. A deterministic version of this test was first proposed by Miller [M75], with a probabilistic version later introduced by Rabin [R80]. We will focus on the probabilistic version.

Let:

$$n = 2k + 1;\ k \in \mathbb{Z}^+ \text{ (if } n \text{ were even we'd need only check if } n = 2\text{)}$$
$$a \in \mathbb{Z}^+;\ \text{`a base' coprime to } n$$

Construct:

$$n - 1 = 2^q m;\ \text{where } q, m \in \mathbb{Z}^+ \text{ and } m \text{ is odd}$$

Consider the sequence $a^{n-1} = a^{2^q m}, a^{2^{q-1} m}, \ldots, a^{2^m} \pmod{n}$. If n is prime then the sequence begins with 1 and every subsequent member is 1, or the first member of the sequence $\neq 1$ is instead $= -1$. If a sequence for some n fails both of these conditions, then n is not prime.

The probability for any composite number (including Carmichael numbers) to pass this test is $\leq \frac{1}{4}$. Therefore repeating the test multiple times using different values for a can reduce the probability of a false positive, where for k repetitions the resulting time complexity is $O(k \log n)$.

References

[RK87] Karp, Richard M. and Rabin, Michael O. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249-260, 1987.

[M75] Miller, Gary L. Riemann's Hypothesis and Tests for Primality. *Proceedings of the Seventh Annual ACM Symposium on Theory of Computing*, pp.234-239, 1975.

[R80] Rabin, Michael O. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1):128-138, 1980.