6.S897 Algebra and Computation April 29, 2015

Lecture 22

Lecturer: Madhu Sudan *Scribe: Michael Cohen*

We talked about the history of the use of algebra in broader parts of theoretical computer science, in particular complexity theory and coding theory. The first part of the class covered complexity (and cryptography) and the rest was about coding theory (Reed-Solomon codes).

1 Algebra in complexity and cryptography

We discussed several different results in complexity and cryptography:

1.1 Shamir's Secret Sharing Scheme

One of the earliest "algebraic" results in general computer science, Shamir introduced his "secret sharing" scheme in [Sha79]. The problem here was simple: one wishes to share some secret among n people such that any subset of k of them can recover the secret, but any subset of fewer than k of them has no information.

The solution was to first embed the secret as an element s of a finite field F, then generate a random degree-k polynomial P having s as its constant term. One then gives person i $P(x_i)$, where the x_i are distinct, nonzero elements of F. The polynomial is uniquely determined by its values on k points, but its distribution of values on any $k-1$ points are random regardless of the constant term.

1.2 Testing equivalence of read-once branching programs

Testing whether two algebraic formulae are equivalent is often easy (if they compute low-degree polynomials): this is polynomial identity testing, and essentially follows from the fact that degree d polynomials have at most d roots. Testing this for boolean functions, on the other hand, is generally quite hard (for instance, SAT corresponds to checking whether a boolean formula is equivalent to 0).

[BCW80], however, introduced a natural and fairly powerful description of boolean functions, sometimes called *read-once branching programs*. These are decision graphs–each node corresponds to reading an input bit, then branching to one of two other nodes depending on its value. The "read-once" constraint means that no possible path through the graph reads the same variable more than once.

The authors were able to find a low-degree polynomial associated with each boolean function (independent of any branching programs that compute it) and

showed that this polynomial may be evaluated efficiently using any read-once branching program computing that function. Polynomial identity testing on the polynomials from two read-once branching program thus tests the equality of the functions they compute.

This argument is also described in Sipser's undergraduate theory textbook!

1.3 #P-completeness of 0-1 Permanent

In [Val79], Valiant proved that computing permanent is #P-hard. In fact, he was able to show that this hardness applies to the permanent even if the entries of the matrix are restricted to being from $\{0, 1\}$.

The argument for this is interesting to us. First, one may easily reduce computing a permanent modulo a prime p to computing a permanent of integers all nonnegative and bounded by $p - 1$; this in turn may be reduced to a $\{0, 1\}$ permanent around p times larger. But one may compute the permanent of any matrix, if bounded by B, by computing it mod primes of size $O(\log B)$ and then applying the Chinese Remainder Theorem to get it modulo the product of all these primes (which will be larger than B).

1.4 Secure multi-party computation

This was similar type of idea to Shamir's Secret Sharing Scheme: computing a function when the inputs are spread among several actors, without leaking extra information. This kind of scenario was analyzed in [BOGW88].

1.5 Constant-depth circuit lower bounds

A famous early circuit lower bound was that PARITY is not in AC_0. Smolensky extended this in [Smo87] to showing that even AC_0 augmented "mod p gates" cannot compute PARITY (or indeed test for multiples of any prime $q \neq p$). He was also able to show that MAJORITY could not be computed in these settings.

These results worked by associating the circuit with polynomials over $\mathbb{F}_p$, and showing that certain polynomials must have high "complexity." This also worked as a reasonable alternative proof even of the plain PARITY not in AC_0 result.

1.6 Toda's Theorem

Toda proved, in [Tod91], that $\mathrm{P}^{\#\mathrm{P}}$ contains the whole polynomial hierarchy ("Toda's theorem"). His proof logically split into two steps. First, he proved that any language in the polynomial hierarchy could be expressed as the set of all z satisfying

$$\Pr_x \left[\bigoplus_y \phi(x, y, z) = 1 \right] \geq 2/3$$

with x and y ranging over bitstrings of some polynomial size, and ϕ a polynomial-time computable function.

Then, he proved that any such language can be recognized by a machine in $\mathrm{P}^{\#\mathrm{P}}$. The mechanism for this was interesting and algebraic.

He provided a low-degree ($\mathrm{poly}(k)$) polynomial mapping anything with parity 0 to something 0 mod 2^k, and anything with parity 1 something -1 mod 2^k.

The argument was essentially Hensel-lifting: values 0 and 1 could be lifted from 2^k to 2^{2k} by applying a constant-degree polynomial. ϕ could be transformed, with polynomial blowup, to apply such a polynomial to its total number of solutions. If this was set so that k is at least the number of bits in x, then simply taking the total number of solutions in x and y (after the transformation), and taking the negation of the result mod 2^k, will give the number of x for which ϕ has a number of solutions with parity 1. A threshold on this then solves the problem.

1.7 PP closed under intersection

[BRS91] proved that PP is closed under intersection. Similar to Toda's theorem, it involved applying transformations to the number of inputs making a Turing machine accept. However, instead of simply applying polynomials, the authors used rational functions. They used pre-existing constructions of certain rational function approximations.

1.8 Later results

By the 1990s, arithmetization and algebraic techniques in general became common in complexity theory. They figured prominently in famous results like IP = PSPACE and the PCP theorem.

2 Algebra in coding theory: the Reed-Solomon code

2.1 Basics of error-correcting codes

We consider sending a message $m \in \Sigma^k$ (k symbols from alphabet Σ) over a noisy channel. Instead of sending it directly, we will have *encoding* and *decoding* functions E and D, so that E maps Σ^k to Σ^n and D maps Σ^n to Σ^k, for some $n > k$. That is, we blow up the length of a message to make it more robust to noise.

We specifically define a t-error-correcting code as one such that if $x = E(m)$ and $\tilde{x}$ has Hamming distance at most t from x, then $D(\tilde{x}) = x$–corrupting any encoding by up to t symbols does not prevent its valid decoding.

If we do not care about computational issues, the validity of such a code is determined solely by the image of E (assuming E is injective). A necessary and sufficient condition is that the minimum Hamming distance, Δ, between two distinct elements of the image of E is at least $2t + 1$ (if this is the case, there is always at most one m such that $\tilde{x}$ has Hamming distance at most t from $E(m)$).

There is a universal lower bound that

$$\frac{k}{n} + \frac{\Delta}{n} \leq 1 + \frac{1}{n}$$

2.2 The Reed-Solomon code

The Reed-Solomon code is a specific example of an error correcting code. Here, the alphabet Σ is some finite field $\mathbb{F}$. The message m is interpreted as the

coefficients of a polynomial P_m of degree at most $k-1$ over $\mathbb{F}$, and the encoding function simply returns $(P_m(x_1), P_m(x_2), ..., P_m(x_n))$ for n distinct field elements x_i. Thus, this is only valid for $|\Sigma| \geq n$.

Given any two distinct valid codewords m and m', their encoding is different everywhere except at x_i where $P_m - P_{m'} = 0$. As a degree $k-1$ nonzero polynomial, this has at most $k-1$ zeroes, so $E(m)$ and $E(m')$ have Hamming distance at least $n-k+1$. This matches the universal bound above, so the Reed-Solomon code is in some sense optimal (though it requires a somewhat large alphabet).

The alphabet size issue is not as fatal as it may sound. If one needs to send the message as bits, one need not naively encode the field elements as binary numbers from 1 to $|\mathbb{F}|$. Rather, one can encode these themselves with an error correcting code. Furthermore, since these strings are now only $O(\log n)$ bits, one has much more flexibility in choosing codes–strategies like random codes now have only polynomial complexity.

2.3 Encoding and decoding Reed-Solomon

Encoding is trivially polynomial time for the Reed-Solomon code: just evaluate the polynomial. In fact, using fast multipoint evaluation it can be nearly linear time.

Decoding, however, is trickier. There are many different algorithms which decode them in different regimes. We will examine one by Madhu Sudan, and will focus on the case $n >> k$ (this corresponds to a very high error rate). This algorithm can perform "list decoding", returning the nearest codewords even if they are further than $\frac{\Delta-1}{2}$ away.

The input to this algorithm are values x_i and y_i, $1 \leq i \leq n$, such that there exists a degree k polynomial $M(x)$ such that $M(x_i) = y_i$ for at least a values of i. We will require that $a > 2k\sqrt{n}$.

The idea is to find a bivariate polynomial Q with x and y degrees at most $\lfloor\sqrt{n}\rfloor$ such that

$$Q(x_i, y_i) = 0$$

for all i.

Such a polynomial is guaranteed to exist: this is a system of n linear equations in $(\lfloor\sqrt{n}\rfloor+1)^2 > n$ variables, and all right hand sides are 0, so its solution space must be a linear subspace of $(\lfloor\sqrt{n}\rfloor+1)^2 - n \geq 1$ dimensions. In particular, it has nonzero solutions, which can be found with standard linear algebra routines.

But the polynomial $y - M(x)$ is 0 for a (x_i, y_i) points, all of which are also zeroes of $Q(x_i, y_i)$. Then Bezout's theorem implies that if $a > 2k\sqrt{n}$ (the product of the degree of Q and the degree of $y - M(x)$), Q and $y - M(x)$ must have a common factor; since $y - M(x)$ is irreducible, it must be a factor of Q. Thus we can find M by simply factorizing Q and looking for factors of that form.

References

[BCW80] Manuel Blum, Ashok K. Chandra, and Mark N. Wegman. Equivalence of free boolean graphs can be decided probabilistically in

polynomial time. *Information Processing Letters*, 10(2):80 – 82, 1980.

[BOGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, pages 1–10, New York, NY, USA, 1988. ACM.

[BRS91] Richard Beigel, Nick Reingold, and Daniel Spielman. Pp is closed under intersection. In *Proceedings of the Twenty-third Annual ACM Symposium on Theory of Computing*, STOC '91, pages 1–9, New York, NY, USA, 1991. ACM.

[Sha79] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, November 1979.

[Smo87] R. Smolensky. Algebraic methods in the theory of lower bounds for boolean circuit complexity. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 77–82, New York, NY, USA, 1987. ACM.

[Tod91] Seinosuke Toda. Pp is as hard as the polynomial-time hierarchy. *SIAM J. Comput.*, 20(5):865–877, October 1991.

[Val79] L.G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8(2):189 – 201, 1979.