COMS E6998: Fine-grained complexity (Spring'26) February 17, 2026

Lecture 5: The polynomial method

Instructor: *Josh Alman* Scribes: *Jonah Stockwell, Hongyi Li; (Joe Bartusek, Fall'22)*

1 Introduction

So far, we have been thinking about conjectures about OV and APSP, and we have observed that the best-known running times often look a bit "weird". For example, for APSP there is an algorithm due to [Wil14] running in $\frac{n^3}{2^{\Omega(\sqrt{\log n})}}$ time. Similarly, for OV, if the dimension is $d = c \cdot \log n$, the best known algorithm due to [AWY15] runs in time $n^{2-\frac{1}{O(\log c)}}$.[1] Both of these algorithms are designed using the *polynomial method*. In these notes, we will give an algorithm for OV, and the algorithm for APSP will be part of the homework.

2 The Polynomial Method

To discuss the polynomial method, we first need to learn about polynomials and Boolean functions.

2.1 Polynomials that Compute Boolean functions

Definition 1. *A Boolean function is a function $f : \{0,1\}^n \to \{0,1\}$* [2]

We will consider polynomials in n variables $p(x_1, x_2, \ldots, x_n)$ over $\mathbb{F}_2$, the finite field with two elements (i.e. integers mod 2).

Definition 2. *We say $p : \mathbb{F}_2^n \to \mathbb{F}_2$ computes the Boolean function f if for all $x \in \{0,1\}^n$, $p(x) = f(x)$* (mod 2)*.*

We start with polynomials that compute some of our favorite Boolean functions: NOT, AND, and OR. Formally, we define

$$\text{NOT}(x) = \begin{cases} 1, & \text{if } x = 0; \\ 0, & \text{if } x = 1. \end{cases}$$

Over $\mathbb{F}_2$, addition and subtraction give the same operation ($-x = x$), so we can take

$$p_{\text{NOT}}(x) = 1 - x = 1 + x.$$

[1]Notice here that we are saying that $\forall c$, $\exists \delta$ such that we can get time $O(n^{2-\delta})$. In constrast, SETH says that it is impossible to have $\exists \delta$, $\forall c$ time $O(n^{2-\delta})$. In some sense, this flip of quantifiers is what keeps us from refuting SETH.

[2]i.e. a function from binary strings to a single output, or equivalently from n variables that take the values True or False, and return a single True or False bit.

Now, we define

$$\text{AND}(x_1, \ldots, x_n) = \begin{cases} 1, & \text{if } x_1 = x_2 = \cdots = x_n = 1; \\ 0, & \text{otherwise.} \end{cases}$$

The polynomial computing AND is simply $p_{\text{AND}}(x_1, \ldots, x_n) = x_1 \cdot x_2 \cdot \ldots \cdot x_n$. Lastly we define

$$\text{OR}(x_1, \ldots, x_n) = \begin{cases} 0, & \text{if } x_1 = x_2 = \cdots = x_n = 0; \\ 1, & \text{otherwise.} \end{cases}$$

We notice that by DeMorgan's law, $\text{OR}(x_1, \ldots, x_n) = \text{NOT}(\text{AND}(\text{NOT}(x_1), \ldots, \text{NOT}(x_n)))$. Thus, we can simply substitute:

$$p_{\text{OR}}(x_1, \ldots, x_n) = p_{\text{NOT}}(p_{\text{AND}}(p_{\text{NOT}}(x_1), \ldots, p_{\text{NOT}}(x_n)) = 1 - (1 - x_1) \cdots (1 - x_n).$$

Now that we have some polynomials that compute Boolean functions we care about, it is natural to wonder how "good" they are. We have two main measures of complexity of a polynomial: degree and sparsity.

Definition 3. *For a polynomial p, the* <u>degree</u> $\deg(p)$ *is the maximum number of variables multiplied together in a single monomial.*

Definition 4. *For a polynomial p, the* <u>sparsity</u> *of p is the number of monomials in the fully expanded form of p.*

So, for p_{AND}, the degree is n and the sparsity is 1. For p_{OR}, the degree is n, and the sparsity is $2^n - 1$. So, naturally, we wonder if we can do better (i.e. come up with polynomials that compute the same functions but have a lower degree or sparsity)? To answer this question we have the following claims:

Claim 5 (Every Boolean function is computed by some polynomial)**.** *For every Boolean function $f : \{0,1\}^n \to \{0,1\}$, there exists a polynomial $p : \mathbb{F}_2^n \to \mathbb{F}_2$ such that $p(x) = f(x)$ for all $x \in \{0,1\}^n$.*

Proof. Define

$$p(x) \; = \sum_{a \in \{0,1\}^n} f(a) \left(\prod_{i:\, a_i = 1} x_i \right) \left(\prod_{i:\, a_i = 0} (1 - x_i) \right).$$

For any Boolean input $x \in \{0,1\}^n$, the product $\left(\prod_{i:\, a_i=1} x_i\right) \left(\prod_{i:\, a_i=0} (1 - x_i)\right)$ evaluates to 1 if and only if $x = a$, and to 0 otherwise. Hence exactly one term in the sum survives, and we get $p(x) = f(x)$. □

Definition 6. *A polynomial is* <u>multilinear</u> *if it has no variable raised to a power > 1*

Claim 7. *We may assume without loss of generality that polynomials for boolean functions are multilinear: since $x_i^2 = x_i$ for $x_i \in \{0,1\}$, we can replace any x_i^t (for $t \geq 2$) by x_i without changing the function on $\{0,1\}^n$.*

Now we introduce an important fact:

Fact 8. *Every Boolean function has a unique multilinear polynomial that computes it.*

Proof. We will show this claim by contradiction. Suppose we have two multilinear polynomials $p \neq q$ that both compute f on $\{0,1\}^n$. Let $r(x) = p(x) - q(x)$. Then r is a nonzero multilinear polynomial, but $r(x) = 0$ for every $x \in \{0,1\}^n$.

Pick a monomial of minimum degree that appears in r with nonzero coefficient; denote it $x_S = \prod_{i \in S} x_i$, where $S \subseteq [n]$ is minimal. Now define

$$(1_S)_i = \begin{cases} 1 & i \in S \\ 0 & i \notin S. \end{cases}$$

Any monomial involving a variable outside S becomes 0 on 1_S. Any monomial x_T with $T \subsetneq S$ does not appear in r, since we choose S to be the minimal degree monomial. Therefore,

$$r(1_S) = (\text{coefficient of } x_S \text{ in } r) \neq 0,$$

contradicting that $r(x) = 0$ for all boolean x. Hence $p = q$. $\square$

Thus, if we require *exact* computation of AND and OR, we cannot do better than the above degree and sparsity. So, we turn to randomness.

2.2 Probabilistic Polynomials

If we cannot hope to have one polynomial that exactly computes a function and has low degree/sparsity, then perhaps instead we can pick a distribution of polynomials that is accurate with high probability on any particular input. Thus, we have the following definition:

Definition 9. *A probabilistic polynomial for $f : \{0,1\}^n \to \{0,1\}$ with error $\epsilon \in (0, \frac{1}{2})$ is a distribution $\mathcal{P}$ on polynomials $p : \mathbb{F}_2^n \to \mathbb{F}_2$, such that, for all $x \in \{0,1\}^n$,*

$$\Pr_{p \sim \mathcal{P}} [p(x) = f(x)] \geq 1 - \epsilon.$$

Example 10. *Some "bad" examples of probabilistic polynomials:*

- *Here is a polynomial for AND: $p(x_1, \ldots, x_n) = 0$. This is NOT a probabilistic polynomial, because on the all-1s input, it is always wrong.*
- *Here is another for AND: pick a random $i \in \{1, \ldots, n\}$, and set $p(x_1, \ldots, x_n) = x_i$. This is also NOT a probabilistic polynomial as it has high error $(1 - \frac{1}{n})$ on inputs with almost all 1s, like $(0, 1, 1, \ldots, 1)$.*

So, how do we create probabilistic polynomials for these Boolean functions that we care about. Originally from a circuit complexity context, we have the following theorem due to Razborov and Smolensky.

Theorem 11 ([Raz87, Smo87]). *For $0 < \epsilon < 1$, there is a probabilistic polynomial $\mathcal{P}$ for OR with error ϵ and degree $O(\log(\frac{1}{\epsilon}))$.*

Proof. Let $k = \lceil \log_2(\frac{1}{\epsilon}) \rceil$. Our distribution on $\mathcal{P}$ will be given by the following randomized algorithm for creating a polynomial. First, pick k random subsets $S_1, \ldots, S_k \subseteq \{1, 2, \ldots, n\}$. Our polynomial is

$$p(x_1, \ldots, x_n) = 1 - \prod_{\ell=1}^{k} (1 - \sum_{i \in S_\ell} x_i) \pmod 2.$$

First, we check $(0, 0, \ldots, 0)$, which is always correct as $\forall S_\ell \ (1 - \sum_{i \in S_\ell} x_i) = 1$ and thus the output will be 0. For $x \neq 0$, we know there is some $x_i \neq 0$. We can reason with the principal of deferred decision. To construct some subset S_ℓ, we first decide whether for all elements $j \neq i$ are in the subset, and then we pick whether i is in the subset. We see that adding i to the subset exactly toggles whether the sum $\sum_{i \in S_\ell} x_i$ is even or odd, so there is $\frac{1}{2}$ probability that $\sum_{i \in S_\ell} x_i = 1$, and in that case we will be correct. So, for each S_ℓ we have $\frac{1}{2}$ probability of being correct and thus in total, we have $\frac{1}{2^k} = \epsilon$ error. Note that we can do a similar construction and create a probabilistic polynomial for AND (i.e. by DeMorgan's). $\square$

2.3 A Better OV Algorithm

Now armed with probabilistic polynomials for AND and OR, we can finally design our OV algorithm. First, we construct a polynomial for the OV function on 2 sets of s length-d vectors:

$$OV \begin{pmatrix} x_{11}, & \ldots, & x_{1d}, \\ x_{21}, & \ldots, & x_{2d}, \\ & \vdots & \\ x_{s1}, & \ldots, & x_{sd}, \\ y_{11}, & \ldots, & y_{1d}, \\ y_{21}, & \ldots, & y_{2d}, \\ & \vdots & \\ y_{s1}, & \ldots, & y_{sd} \end{pmatrix} = \bigvee_{i,j \in \{1,\ldots,s\}} \bigwedge_{\ell=1}^{d} (\overline{x_{i\ell}} \vee \overline{y_{j\ell}}) = \bigvee_{i,j \in \{1,\ldots,s\}} \bigwedge_{\ell=1}^{d} (1 - \underbrace{x_{i\ell} y_{j\ell}}_{\text{polynomial of degree 2}}).$$

In order to have a low degree OV polynomial, we replace the big OR and big ANDs with probabilistic polynomials. In particular, we want our big ANDs to have error $\frac{0.001}{s^2}$ so that over all s^2 pairs, we have $\geq .999$ probability of success. Then, we replace the big OR with a probabilistic polynomial with error $\frac{1}{1000}$. So, total error $\leq s^2 \cdot \frac{0.001}{s^2} + \frac{1}{1000} = \frac{2}{1000}$.

To compute the total degree, we refer to Theorem 11. With error rate around $O(\frac{1}{s^2})$, the degree of probabilistic polynomials for OR is $O(\log s)$ (notice AND/OR can convert to each other without changing degree). In the formula above, the base terms $x_{i\ell} y_{j\ell}$ have degree 2, and the probabilistic AND gives constant degree (since it depends on error rate). Thus the composition of polynomials gives a total degree of $O(\log s)$ (up to a constant factor), which is the key for making the resulting polynomial sparse enough for fast rectangular matrix multiplication later.

The final key idea is that we will combine our OV probabilistic polynomial (OVPP) with fast rectangular matrix multiplication. The naive way to solve OV with Matrix Multiplication is as follows.

Given, $x_1, \ldots, x_n, y_1, \ldots, y_n \in \{0, 1\}^d$, we can simply compute

$$\begin{bmatrix} x_1^\top \\ x_2^\top \\ \vdots \\ x_n^\top \end{bmatrix} \begin{bmatrix} y_1 & y_2 & \ldots & y_n \end{bmatrix} = \begin{bmatrix} \langle x_1, y_1 \rangle & \langle x_1, y_2 \rangle & \ldots & \langle x_1, y_n \rangle \\ \langle x_2, y_1 \rangle & \langle x_2, y_2 \rangle & \ldots & \langle x_2, y_n \rangle \\ \vdots & \vdots & \ddots & \vdots \\ \langle x_n, y_1 \rangle & \langle x_n, y_2 \rangle & \ldots & \langle x_n, y_n \rangle \end{bmatrix}$$

in time $O(n^2 d)$ and check for a zero entry in $O(n^2)$. However, we have the following "super-fast" rectangular matrix multiplication algorithm due to Coppersmith:

Theorem 12 ([Cop82]). *There exists an algorithm to multiply an $n \times n^{0.1}$ matrix with a $n^{0.1} \times n$ matrix* *in $O(n^2 \log^2 n)$ time.*

Unfortunately, in our case for $d = O(\log n)$ this is worse than the naive algorithm, and even if we could multiply matrices quicker, we still have to make an $O(n^2)$ check... To solve this issue, we seek to reduce both the *number of vectors* and *dimension of vectors*. Let us partition $x_1, \dots, x_n$ into $\frac{n}{s}$ parts of size s and do the same for $y_1, \dots, y_n$. Now for each group of x's of size s, we want to create a vector g_i and for each group of y's of size s, we want to create a vector h_i such that when we compute the following matrix product we approximately compute the OV function for each pair of s-sized groups.

$$\underbrace{\begin{bmatrix} g_1^\top \\ g_2^\top \\ \vdots \\ g_{n/s}^\top \end{bmatrix}}_{\in \mathbb{R}^{(n/s)\times m}} \underbrace{\begin{bmatrix} h_1 & h_2 & \cdots & h_{n/s} \end{bmatrix}}_{\in \mathbb{R}^{m\times(n/s)}} = \begin{bmatrix} \langle g_1, h_1\rangle & \langle g_1, h_2\rangle & \cdots & \langle g_1, h_{n/s}\rangle \\ \langle g_2, h_1\rangle & \langle g_2, h_2\rangle & \cdots & \langle g_2, h_{n/s}\rangle \\ \vdots & \vdots & \ddots & \vdots \\ \langle g_{n/s}, h_1\rangle & \langle g_{n/s}, h_2\rangle & \cdots & \langle g_{n/s}, h_{n/s}\rangle \end{bmatrix} \text{ for } m = O((n/s)^{0.1})$$

Specifically, we want entry (i, j) to be the OV probabilistic polynomial where we plug in groups i and j, in order to take advantage of its property of degree reduction. So, given the $\frac{2}{1000}$ error we computed earlier, for most of the pairs of groups, we will have a correct answer (which we can then amplify to get correct answers on all inputs). Now, the question is, given $x_{11}, \dots, x_{sd}, y_{11}, \dots, y_{sd}$, how do we find g_i, h_j such that $\langle g_i, h_j\rangle = \text{OVPP}(x_{11} \dots, y_{11}, \dots)$. First, we want to compute how many monomials our OVPP will have (as this will determine the dimension of our vectors). We can upper bound the number of monomials in the OVPP as follows: for the big ANDs in the OV formula, we notice that as the maximum degree is $\leq a \cdot \log s$, then the total number of monomials is upper bounded by $\sum_{\ell=0}^{a \log s} \binom{2d}{\ell}$ because it only considers $2d$ variables at a time. Then, we know that by construction our big OR polynomial consists of a product of a constant number of sums of $O(s^2)$ OR polynomials. By expanding naively, we obtain an upper bound of

$$s^{2\cdot O(1)} \left(\sum_{\ell=0}^{a \log s} \binom{2d}{\ell} \right)^{O(1)}$$

to make this less than $n^{0.1}$, we will take $s = n^{1/(b \log(c))}$ and choose appropriate $b > 0$. We will make use of the following two properties of combinations.

Fact 13 (Lower bound on factorial)**.** *For every integer $k \geq 1$,*

$$k! \;\geq\; \left(\frac{k}{e}\right)^k$$

Based on this we can deduce that

$$\binom{N}{k} = \frac{N(N-1)\cdots(N-k+1)}{k!} \leq \frac{N^k}{k!}.$$

Using the fact above, we obtain

$$\binom{N}{k} \leq \frac{N^k}{(k/e)^k} = \left(\frac{eN}{k}\right)^k.$$

To bound our sum of binomial coefficients, we can thus take

$$\sum_{k=0}^{K} \binom{N}{k} \leq \sum_{k=0}^{K} \binom{N}{K} = (K+1)\binom{N}{K} = (K+1)\left(\frac{eN}{k}\right)^{k}.$$

So, applying this to our monomial upperbound yields:

$$\begin{aligned}
s^{2\cdot O(1)}\left(\sum_{\ell=0}^{a\log s}\binom{2d}{\ell}\right)^{O(1)} &= n^{O(1)/(b\log(c))}\left(\sum_{\ell=0}^{\frac{a\log n}{b\log c}}\binom{2c\cdot\log n}{\ell}\right)^{O(1)} && \text{(substitute } s = n^{1/(b\log(c)}) \text{)}\\
&\leq n^{O(1)/(b\log(c))}\cdot\left(\frac{a\log n}{b\log c}+1\right)\cdot\left(\frac{2ec\cdot\log n}{\frac{a\log n}{b\log c}}\right)^{O(1)\frac{a\log n}{b\log c}} && \text{(use fact from above)}\\
&\leq n^{O(1)/(b\log(c))}\cdot\Theta(\log n)\cdot(n)^{O(1)\frac{a\log\left(\frac{2ecb\log c}{a}\right)}{b\log c}} \leq O(n^{0.1})
\end{aligned}$$

where we can make the final expression $\leq O(n^{0.1})$ by choosing b large enough.

Thus, if we take our OVPP and expand it as a sum of $O(n^{0.1})$ monomials, then we can express it as an inner product of a vector of the x part of each monomial and a vector of the y part of each monomial. For example if we have

$$\text{OVPP} = \underbrace{x_1y_1y_3 + x_2x_4y_7y_9 + \cdots}_{\leq n^{0.1}\text{ monomials}}$$

from which we construct $g = (x_1, x_2x_4, \dots) \in \mathbb{F}_2^{n^{0.1}}$ and $h = (y_1y_3, y_7y_9, \dots) \in \mathbb{F}_2^{n^{0.1}}$. So, now we can evaluate the OVPP for s sized subsets by taking inner products. Moroever, the length of these vectors is the number of monomials ($\leq O(n^{0.1})$ by the previous calculation), which allows us to apply the Coppersmith algorithm and compute OV with high accuracy in time $O(\frac{n^2}{s^2}\log^2 n) = O(n^{2-1/O(\log(c))}\text{poly}(d))$.

References

[AWY15] Amir Abboud, Ryan Williams, and Huacheng Yu. More applications of the polynomial method to algorithm design. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '15, page 218–230, USA, 2015. Society for Industrial and Applied Mathematics.

[Cop82] Don Coppersmith. Rapid multiplication of rectangular matrices. *SIAM Journal on Computing*, 11(3):467–471, 1982.

[Raz87] Alexander A. Razborov. Lower bounds on the size of bounded depth circuits over a complete basis with logical addition. *Mathematical Notes of the Academy of Sciences of the USSR*, 41(4):333–338, 1987.

[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, page 77–82, New York, NY, USA, 1987. Association for Computing Machinery.

[Wil14] Ryan Williams. Faster all-pairs shortest paths via circuit complexity, 2014.