Chapter 5

Verifying Identities, and Some Complexity

By Sariel Har-Peled, April 26, 2022[1]

The events of September 8 prompted Foch to draft the later legendary signal: “My centre is giving way, my right is in retreat, situation excellent. I attack.” It was probably never sent.

John Keegan, The first world war

5.1. Verifying equality

5.1.1. Vectors

You are given two binary vectors $\mathbf{v} = (v_1, \ldots, v_n), \mathbf{u} = (u_1, \ldots, u_n) \in \{0,1\}^n$ and you would like to decide if they are equal or not. Unfortunately, the only access you have to the two vectors is via a black-box that enables you to compute the dot-product of two binary vectors over $\mathbb{Z}_2$. Formally, given two binary vectors as above, their dot-product is $\langle \mathbf{v}, \mathbf{u} \rangle = \sum_{i=1}^{n} v_i u_i$ (which is a non-negative integer number). Their dot product modulo 2, is $\langle \mathbf{v}, \mathbf{u} \rangle \bmod 2$ (i.e., it is 1 if $\langle \mathbf{v}, \mathbf{u} \rangle$ is odd and 0 otherwise).

Naturally, we could the use the black-box to read the vectors (using $2n$ calls), but since we are interested only in deciding if they are equal or not, this should require less calls to the black-box (which is expensive).

Lemma 5.1.1. *Given two binary vectors $\mathbf{v}, \mathbf{u} \in \{0,1\}^n$, a randomized algorithm can, using two computations of dot-product modulo 2, decide if $\mathbf{v}$ is equal to $\mathbf{u}$ or not. The algorithm may return one of the following two values:*

- $\neq$: *Then $\mathbf{v} \neq \mathbf{u}$.*
- $=$: *Then the probability that the algorithm made a mistake (i.e., the vectors are different) is at most* $1/2$.

The running time of the algorithm is $O(n + B(n))$, where $B(n)$ is the time to compute a single dot-product of vectors of length n.

Proof: Pick a random vector $\mathbf{r} = (r_1, \ldots, r_n) \in \{0,1\}^n$ by picking each coordinate independently with probability 1/2. Compute the two dot-products $\langle \mathbf{v}, \mathbf{r} \rangle$ and $\langle \mathbf{u}, \mathbf{r} \rangle$.

(A) If $\langle \mathbf{v}, \mathbf{r} \rangle \equiv \langle \mathbf{v}, \mathbf{r} \rangle \bmod 2 \quad \Rightarrow \quad$ the algorithm returns ‘$=$’.

(B) If $\langle \mathbf{v}, \mathbf{r} \rangle \not\equiv \langle \mathbf{v}, \mathbf{r} \rangle \bmod 2 \quad \Rightarrow \quad$ the algorithm returns ‘$\neq$’.

Clearly, if the ‘$\neq$’ is returned then $\mathbf{v} \neq \mathbf{u}$.

So, assume that the algorithm returned ‘$=$’ but $\mathbf{v} \neq \mathbf{u}$. For the sake of simplicity of exposition, assume that they differ on the nth bit: $u_n \neq v_n$. We then have that

$$\alpha = \langle \mathbf{v}, \mathbf{r} \rangle = \overbrace{\sum_{i=1}^{n-1} v_i r_i}^{=\alpha'} \quad + \quad v_n r_n \qquad \text{and} \qquad \beta = \langle \mathbf{u}, \mathbf{r} \rangle = \overbrace{\sum_{i=1}^{n-1} u_i r_i}^{=\beta'} \quad + \quad u_n r_n.$$

Now, there are two possibilities:

(A) If $\alpha' \not\equiv \beta' \bmod 2$, then, with probability half, we have $r_i = 0$, and as such $\alpha \not\equiv \beta \bmod 2$.

(B) If $\alpha' \equiv \beta' \bmod 2$, then, with probability half, we have $r_i = 1$, and as such $\alpha \not\equiv \beta \bmod 2$.

As such, with probability at most half, the algorithm would fail to discover that the two vectors are different. ∎

5.1.1.1. Amplification

Of course, this is not a satisfying algorithm – it returns the correct answer only with probability half if the vectors are different. So, let us run the algorithm t times. Let $T_1, \ldots, T_t$ be the returned values from all these executions. If any of the t executions returns that the vectors are different, then we know that they are different.

$$\begin{aligned}\mathbb{P}\big[\text{Algorithm fails}\big] &= \mathbb{P}\big[\mathbf{v} \neq \mathbf{u}, \text{ but all } t \text{ executions return `}\mathbf{=}\text{'}\big]\\ &= \mathbb{P}\big[(T_1 = \text{`}\mathbf{=}\text{'}) \cap (T_2 = \text{`}\mathbf{=}\text{'}) \cap \cdots \cap (T_t = \text{`}\mathbf{=}\text{'})\big]\\ &= \mathbb{P}\big[T_1 = \text{`}\mathbf{=}\text{'}\big]\,\mathbb{P}\big[T_2 = \text{`}\mathbf{=}\text{'}\big] \cdots \mathbb{P}\big[T_t = \text{`}\mathbf{=}\text{'}\big] \leq \prod_{i=1}^{t} \frac{1}{2} = \frac{1}{2^t}.\end{aligned}$$

We thus get the following result.

Lemma 5.1.2. *Given two binary vectors $\mathbf{v}, \mathbf{u} \in \{0,1\}^n$ and a* **confidence** *parameter $\delta > 0$, a randomized algorithm can decide if $\mathbf{v}$ is equal to $\mathbf{u}$ or not. More precisely, the algorithm may return one of the two following results:*

≠: *Then $\mathbf{v} \neq \mathbf{u}$.*

=: *Then, with probability $\geq 1 - \delta$, we have $\mathbf{v} \neq \mathbf{u}$.*

The running time of the algorithm is $O\big((n + B(n)) \ln \delta^{-1}\big)$, where $B(n)$ is the time to compute a single dot-product of two vectors of length n.

Proof: Follows from the above by setting $t = \lceil \lg(1/\delta) \rceil$. ∎

5.1.2. Matrices

Given three binary matrices B, C, D of size $n \times n$, we are interested in deciding if $\mathsf{BC} = \mathsf{D}$. Computing BC is expensive – the fastest known (theoretical!) algorithm has running time (roughly) $O(n^{2.37})$. On the other hand, multiplying such a matrix with a vector $\mathbf{r}$ (modulo 2, as usual) takes only $O(n^2)$ time (and this algorithm is simple).

Lemma 5.1.3. *Given three binary matrices $\mathsf{B}, \mathsf{C}, \mathsf{D} \in \{0,1\}^{n \times n}$ and a confidence parameter $\delta > 0$, a randomized algorithm can decide if $\mathsf{BC} = \mathsf{D}$ or not. More precisely the algorithm can return one of the following two results:*

≠: *Then $\mathsf{BC} \neq \mathsf{D}$.*

=: *Then $\mathsf{BC} = \mathsf{D}$ with probability $\geq 1 - \delta$.*

The running time of the algorithm is $O\big(n^2 \log \delta^{-1}\big)$.

Proof: Compute a random vector $\mathbf{r} = (r_1, \ldots, r_n)$, and compute the quantity $\mathbf{x} = \mathsf{BC}\mathbf{r} = \mathsf{B}(\mathsf{C}\mathbf{r})$ in $O(n^2)$ time, using the associative property of matrix multiplication. Similarly, compute $\mathbf{y} = \mathsf{D}\mathbf{r}$. Now, if $\mathbf{x} \neq \mathbf{y}$ then return '**=**'.

Now, we execute this algorithm $t = \lceil \lg \delta^{-1} \rceil$ times. If all of these independent runs return that the matrices are equal then return '='.

The algorithm fails only if $\mathsf{BC} \neq \mathsf{D}$, but then, assume the ith row in two matrices BC and D are different. The probability that the algorithm would not detect that these rows are different is at most 1/2, by Lemma 5.1.1. As such, the probability that all t runs failed is at most $1/2^t \leq \delta$, as desired. ■

5.1.3. Checking identity for polynomials

5.1.3.1. The Schwartz–Zippel lemma

Let $\mathbb{F}$ be a field (i.e., real numbers). Let $\mathbb{F}[x_1, \ldots, X_n]$ denote the set of polynomials over the n variables $x_1, \ldots, x_n$ over $\mathbb{F}$. Such a polynomial is a sum of monomial, where a ***monomial*** has the form $c \cdot x_1^{i_1} \cdot x_2^{i_2} \cdots x_n^{i_n}$, where $c \in \mathbb{F}$. The ***degree*** of this monomial is $\mathrm{degree}\Big(c \cdot x_1^{i_1} \cdot x_2^{i_2} \cdots x_n^{i_n}\Big) = i_1 + i_2 + \cdots + i_n$. Thus, a polynomial of degree d over n variables has potentially up to dn^d monomials (the exact bound is messier, but an easy lower bound on this quantity is $\binom{n}{d}$).

For a polynomial $f \in \mathbb{F}[x_1, \ldots, X_n]$, the ***zero set*** of f, is the set $Z_f = \{(x_1, \ldots, x_n) \mid f(x_1, \ldots, x_n) = 0\}$. Intuitively, the zero set $Z_f = \mathbb{F}^n$ only if $f(x_1, \ldots, x_n) = 0$ (and then it is the ***zero*** polynomial), but otherwise (i.e., $f \neq 0$) Z_f should be much "smaller".

Specifically, a polynomial in a single variable of degree d is either zero everywhere, or has at most d roots (i.e., $|Z_f| \leq d$). This is known as the *fundamental theorem of algebra*[2]. The picture gets much messier once one deals with multi-variate polynomials, but fortunately there is a simple and elegant lemma that bounds the number of zeros if we pick the values from the right set of values.

Lemma 5.1.4 (Schwartz–Zippel). *Let $f \in \mathbb{F}[x_1, \ldots, x_n]$ be a non-zero polynomial of total degree $d \geq 0$, over a field $\mathbb{F}$. Let $S \subseteq \mathbb{F}$ be finite. Let $\mathsf{r} = (r_1, \ldots, r_n)$ be randomly and uniformly chosen from S^n. Then*

$$\mathbb{P}[f(\mathsf{r}) = 0] \leq \frac{d}{|S|}.$$

Equivalently, we have $|Z_f \cap S^n| \leq d|S|^{n-1}$.

Proof: The proof is by induction on n. For $n = 1$, by the theorem, formally known as the fundamental theorem of algebra, $|Z_f \cap S| \leq |Z_f| \leq d$. So assume the theorem holds for $n - 1$. Since f is non-zero, it can be written as a sum of d polynomials in $n - 1$ variables. That is, f can be written as

$$f(x_1, \ldots, x_n) = \sum_{i=0}^{d} x_1^i f_i(x_2, \ldots, x_n),$$

where $\mathrm{degree}(f_i) \leq d - i$. Since f is not zero, one of the fs must be non-zero, and let i the maximum value such that $f_i \neq 0$.

Now, we randomly choose the values $r_2, \ldots, r_n \in S$ (independently and uniformly). And consider the polynomial in the single variable x, which is

$$g(x) = \sum_{j=0}^{d} f_j(r_2, \ldots, r_n) x^i.$$

[2]Wikipedia notes that the proof is not algebraic, and it is definitely not fundamental to modern algebra. So maybe it should be cited as "the theorem formerly known as the fundamental theorem of algebra".

Let $\mathcal{F}$ be the event that $f_i(r_2, \ldots, r_n) = 0$. Let $\mathcal{G}$ be the event that $g(x) = 0$. By induction, we have $\mathbb{P}[\mathcal{F}] \leq (d-i)/|S|$. More interestingly if $\mathcal{F}$ does not happen, then degree$(g) = i$. As such, by induction, we have that

$$\mathbb{P}[\mathcal{G} \mid \overline{\mathcal{F}}] = \mathbb{P}[g(x) = 0 \mid \overline{\mathcal{F}}] \leq \frac{i}{|S|}.$$

We conclude that

$$\mathbb{P}[f(\mathsf{r}) = 0] = \mathbb{P}[\mathcal{G} \cap \mathcal{F}] + \mathbb{P}[\mathcal{G} \cap \overline{\mathcal{F}}] \leq \mathbb{P}[\mathcal{F}] + \mathbb{P}[\mathcal{G} \mid \overline{\mathcal{F}}] \leq \frac{d-i}{|S|} + \frac{i}{|S|} \leq \frac{d}{|S|}. \qquad \blacksquare$$

Remark 5.1.5. Consider the polynomial $f(x, y) = (x-1)^2 + (y-1)^2 - 1$. The zero set of this polynomial is the unit circle. So the zero set Z_f is infinite in this case. However, note that for any choice of S, the set S^2 is a grid. The Schwartz-Zippel lemma, tells us that there relatively few grid points that are in the zero set.

5.1.3.2. Applications

5.1.3.2.1. Checking if a polynomial is the zero polynomial. Let f be a polynomial of degree d, with n variables, over the reals that can be evaluated in $O(T)$ time. One can check if f zero, by picking randomly a n numbers from $S = [\![d^3]\!]$. By Lemma 5.1.4, we have that the probability of f to be zero over the chosen values is $\leq d/d^3$, which is a small number. As above, we can do amplification to get a high confidence answer.

5.1.3.2.2. Checking if two polynomials are equal. Given two polynomials f and g, one can now check if they are equal by checking if $f(r) = g(r)$, for some random input. The proof of correctness follows from the above, as one interpret the algorithm as checking if $f - g$ is the zero polynomial.

5.1.3.2.3. Verifying polynomials product. Given three polynomials f, g, and h, one can now check if $fg = h$. Again, one randomly pick a value r, and check if $f(r)g(r) = h(r)$. The proof of correctness follows from the above, as one interprets the algorithm as checking if $fg - h$ is the zero polynomial.

5.1.4. Checking if a bipartite graph has a perfect matching

Let $\mathsf{G} = (L \cup R, \mathsf{E})$ be a bipartite graph. Let $L = \{u_1, \ldots, u_n\}$ and $R = \{v_1, \ldots, v_n\}$. Consider the set of variables

$$\mathcal{V} = \left\{ x_{i,j} \;\middle|\; u_i v_j \in \mathsf{E} \right\}.$$

Let M be an $n \times n$ matrix, where $M[i,j] = 0$ if $u_i v_j \notin \mathsf{E}$, and $M[i,j] = x_{i,j}$ otherwise. Let Π be the set of all permutations of $[\![n]\!]$.

A ***perfect matching*** is a permutation $\pi : [\![n]\!] \to [\![n]\!]$, such that for all i, we have $u_i v_{\pi(i)} \in \mathsf{E}$. For such a permutation π, consider the monomial

$$f_\pi = \mathrm{sign}(\pi) \prod_{i=1}^{n} M[i,j],$$

where sign is the ***sign*** of the permutation (it is either -1 or $+1$ – for our purpose here we do not care about the exact definition of this quantity). It is either a polynomial of degree exactly n, or it is zero.

Furthermore, observe that for any two different permutation $\pi, \sigma \in \Pi$, we have that if f_π and f_σ are both non-zero, then $f_\pi \neq f_\sigma$ and $f_\pi \neq -f_\sigma$.

Consider the following "crazy" polynomial over the set of variables $\mathcal{V}$:

$$\psi = \psi() = \det(M) = \sum_{\pi \in \Pi} \mathrm{sign}(\pi) f_\pi.$$

If there is perfect matching in G, then there is a permutation π such that $f_\pi \neq 0$. But this implies that $\psi \neq 0$ (since it has a non-zero monomials, and the monomials can not cancel each other).

In the other direction, if there is no perfect matching in G, then $f_\pi = 0$ for all permutation π. This implies that $\psi = 0$. Thus, deciding if G has a perfect matching is equivalent to deciding if $\psi \neq 0$. The polynomial ψ is defined via a determinant of a matrix that variables as some of the entries (and zeros otherwise). By the above, all we need to do is to evaluate ψ over some random values. If we use exact arithmetic, we would just pick a random number in $[0, 1]$ for each variable, and evaluate ψ for these values of the variable. Namely, we filled the matrix M with values (so it is all numbers now), and we need to computes its determinant. Via Gaussian elimination, the determinant can be computed in cubic time. Thus, we can evaluate ψ in cubic time, which implies that with high probability we can check if G has a perfect matching.

If we do not want to be prisoners of the impreciseness of floating point arithmetic, then one can perform the above calculations over some finite field (usually,the field is simply working modulo a prime number).

5.2. Las Vegas and Monte Carlo algorithms

Definition 5.2.1. A ***Las Vegas algorithm*** is a randomized algorithms that *always* return the correct result. The only variant is that it's running time might change between executions.

An example for a Las Vegas algorithm is the **QuickSort** algorithm.

Definition 5.2.2. A ***Monte Carlo algorithm*** is a randomized algorithm that might output an incorrect result. However, the probability of error can be diminished by repeated executions of the algorithm.

The matrix multiplication algorithm is an example of a Monte Carlo algorithm.

5.2.1. Complexity Classes

I assume people know what are Turing machines, **NP**, **NPC**, RAM machines, uniform model, logarithmic model. **PSPACE**, and **EXP**. If you do now know what are those things, you should read about them. Some of that is covered in the randomized algorithms book, and some other stuff is covered in any basic text on complexity theory[3].

Definition 5.2.3. The class **P** consists of all languages L that have a polynomial time algorithm **Alg**, such that for any input Σ^*, we have

(A) $x \in L \Rightarrow$ **Alg**(x) accepts,

(B) $x \notin L \Rightarrow$ **Alg**(x) rejects.

[3]There is also the internet.

Definition 5.2.4. The class **NP** consists of all languages L that have a polynomial time algorithm **Alg**, such that for any input Σ^*, we have:

(i) If $x \in L \Rightarrow$ then $\exists y \in \Sigma^*$, **Alg**(x, y) accepts, where $|y|$ (i.e. the length of y) is bounded by a polynomial in $|x|$.
(ii) If $x \notin L \Rightarrow$ then $\forall y \in \Sigma^*$**Alg**(x, y) rejects.

Definition 5.2.5. For a complexity class $\mathcal{C}$, we define the complementary class co-$\mathcal{C}$ as the set of languages whose complement is in the class $\mathcal{C}$. That is

$$\text{co}-\mathcal{C} = \left\{ L \;\middle|\; \overline{L} \in \mathcal{C} \right\},$$

4 where $\overline{L} = \Sigma^* \setminus L$.

It is obvious that $\mathbf{P} = \text{co}-\mathbf{P}$ and $\mathbf{P} \subseteq \mathbf{NP} \cap \text{co}-\mathbf{NP}$. (It is currently unknown if $\mathbf{P} = \mathbf{NP} \cap \text{co}-\mathbf{NP}$ or whether $\mathbf{NP} = \text{co}-\mathbf{NP}$, although both statements are believed to be false.)

Definition 5.2.6. The class **RP** (for Randomized Polynomial time) consists of all languages L that have a randomized algorithm **Alg** with worst case polynomial running time such that for any input $x \in \Sigma^*$, we have

(i) If $x \in L$ then $\mathbb{P}[\textbf{Alg}(x) \text{ accepts}] \geq 1/2$.
(ii) $x \notin L$ then $\mathbb{P}[\textbf{Alg}(x) \text{ accepts}] = 0$.

An **RP** algorithm is a Monte Carlo algorithm, but this algorithm can make a mistake only if $x \in L$. As such, co$-$**RP** is all the languages that have a Monte Carlo algorithm that make a mistake only if $x \notin L$. A problem which is in $\mathbf{RP} \cap \text{co}-\mathbf{RP}$ has an algorithm that does not make a mistake, namely a Las Vegas algorithm.

Definition 5.2.7. The class **ZPP** (for Zero-error Probabilistic Polynomial time) is the class of languages that have a Las Vegas algorithm that runs in expected polynomial time.

Definition 5.2.8. The class **PP** (for Probabilistic Polynomial time) is the class of languages that have a randomized algorithm **Alg**, with worst case polynomial running time, such that for any input $x \in \Sigma^*$, we have

(i) If $x \in L$ then $\mathbb{P}[\textbf{Alg}(x) \text{ accepts}] > 1/2$.
(ii) If $x \notin L$ then $\mathbb{P}[\textbf{Alg}(x) \text{ accepts}] < 1/2$.

The class **PP** is not very useful. Why?

Exercise 5.2.9. Provide a **PP** algorithm for 3SAT.

Consider the mind-boggling stupid randomized algorithm that returns either yes or no with probability half. This algorithm is almost in **PP**, as it return the correct answer with probability half. An algorithm is in **PP** needs to be *slightly* better, and be correct with probability better than half. However, how much better can be made to be arbitrarily close to 1/2. In particular, there is no way to do effective amplification with such an algorithm.

Definition 5.2.10. The class **BPP** (for Bounded-error Probabilistic Polynomial time) is the class of languages that have a randomized algorithm **Alg** with worst case polynomial running time such that for any input $x \in \Sigma^*$, we have

(i) If $x \in L$ then $\mathbb{P}[\textbf{Alg}(x) \text{ accepts}] \geq 3/4$.
(ii) If $x \notin L$ then $\mathbb{P}[\textbf{Alg}(x) \text{ accepts}] \leq 1/4$.

5.3. Bibliographical notes

Section 5.2 follows [MR95, Section 1.5].

References

[MR95] R. Motwani and P. Raghavan. *Randomized algorithms*. Cambridge, UK: Cambridge University Press, 1995.