Learning Objectives:

- See examples of randomized algorithms
- Get more comfort with analyzing probabilistic processes and tail bounds
- Success amplification using tail bounds

19
Probabilistic computation

> *"in 1946 .. (I asked myself) what are the chances that a Canfield solitaire laid out with 52 cards will come out successfully? After spending a lot of time trying to estimate them by pure combinatorial calculations, I wondered whether a more practical method ... might not be to lay it our say one hundred times and simple observe and count"*, Stanislaw Ulam, 1983

> *"The salient features of our method are that it is probabilistic ... and with a controllable miniscule probability of error."*, Michael Rabin, 1977

In early computer systems, much effort was taken to drive *out* randomness and noise. Hardware components were prone to non-deterministic behavior from a number of causes, whether it was vacuum tubes overheating or actual physical bugs causing short circuits (see Fig. 19.1). This motivated John von Neumann, one of the early computing pioneers, to write a paper on how to *error correct* computation, introducing the notion of *redundancy*.

So it is quite surprising that randomness turned out not just a hindrance but also a *resource* for computation, enabling us to achieve tasks much more efficiently than previously known. One of the first applications involved the very same John von Neumann. While he was sick in bed and playing cards, Stan Ulam came up with the observation that calculating statistics of a system could be done much faster by running several randomized simulations. He mentioned this idea to von Neumann, who became very excited about it; indeed, it turned out to be crucial for the neutron transport calculations that were needed for development of the Atom bomb and later on the hydrogen bomb. Because this project was highly classified, Ulam, von Neumann and their collaborators came up with the codeword "Monte Carlo" for this approach (based on the famous casinos where Ulam's uncle gambled). The name stuck, and randomized algorithms are known as Monte Carlo algorithms to this day.[1]

In this chapter, we will see some examples of randomized algorithms that use randomness to compute a quantity in a faster or simpler way than was known otherwise. We will describe the algorithms

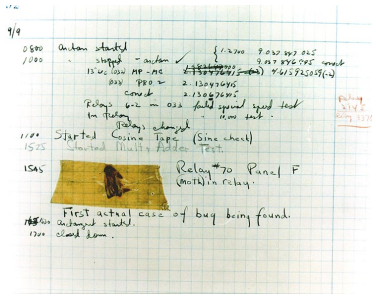

Figure 19.1: A 1947 entry in the log book of the Harvard MARK II computer containing an actual bug that caused a hardware malfunction. By Courtesy of the Naval Surface Warfare Center.

[1] Some texts also talk about "Las Vegas algorithms" that always return the right answer but whose running time is only polynomial on the average. Since this Monte Carlo vs Las Vegas terminology is confusing, we will not use these terms anymore, and simply talk about randomized algorithms.

in an informal / "pseudo-code" way, rather than as Turing macines or NAND-TM/NAND-RAM programs. In Chapter 20 we will discuss how to augment the computational models we say before to incorporate the ability to "toss coins".

This chapter: A non-mathy overview

This chapter gives some examples of randomized algorithms to get a sense of why probability can be useful for computation. We will also see the technique of *success amplification* which is key for many randomized algorithms.

19.1 FINDING APPROXIMATELY GOOD MAXIMUM CUTS

We start with the following example. Recall the *maximum cut problem* of finding, given a graph $G = (V, E)$, the cut that maximizes the number of edges. This problem is **NP**-hard, which means that we do not know of any efficient algorithm that can solve it, but randomization enables a simple algorithm that can cut at least half of the edges:

Theorem 19.1 — Approximating max cut. There is an efficient probabilistic algorithm that on input an n-vertex m-edge graph G, outputs a cut $(S, \overline{S})$ that cuts at least $m/2$ of the edges of G in expectation.

Proof Idea:

We simply choose a *random cut*: we choose a subset S of vertices by choosing every vertex v to be a member of S with probability $1/2$ independently. It's not hard to see that each edge is cut with probability $1/2$ and so the expected number of cut edges is $m/2$.

⋆

Proof of Theorem 19.1. The algorithm is extremely simple:

Algorithm Random Cut:

Input: Graph $G = (V, E)$ with n vertices and m edges. Denote $V = \{v_0, v_1, \ldots, v_{n-1}\}$.

Operation:

1. Pick x uniformly at random in $\{0,1\}^n$.
2. Let $S \subseteq V$ be the set $\{v_i : x_i = 1 , i \in [n]\}$ that includes all vertices corresponding to coordinates of x where $x_i = 1$.
3. Output the cut $(S, \overline{S})$.

We claim that the expected number of edges cut by the algorithm is $m/2$. Indeed, for every edge $e \in E$, let X_e be the random variable such that $X_e(x) = 1$ if the edge e is cut by x, and $X_e(x) = 0$ otherwise. For

every such edge $e = \{i, j\}$, $X_e(x) = 1$ if and only if $x_i \neq x_j$. Since the pair (x_i, x_j) obtains each of the values $00, 01, 10, 11$ with probability $1/4$, the probability that $x_i \neq x_j$ is $1/2$ and hence $\mathbb{E}[X_e] = 1/2$. If we let X be the random variable corresponding to the total number of edges cut by S, then $X = \sum_{e \in E} X_e$ and hence by linearity of expectation

$$\mathbb{E}[X] = \sum_{e \in E} \mathbb{E}[X_e] = m(1/2) = m/2 \,. \tag{19.1}$$

■

Randomized algorithms work in the worst case. It is tempting to think of a randomized algorithm such as the one of Theorem 19.1 as an algorithm that works for a "random input graph" but it is actually much better than that. The expectation in this theorem is *not* taken over the choice of the graph, but rather only over the *random choices of the algorithm*. In particular, *for every graph* G, the algorithm is guaranteed to cut half of the edges of the input graph in expectation. That is,

Big Idea 24 A randomized algorithm outputs the correct value with good probability on *every possible input*.

We will define more formally what "good probability" means in Chapter 20 but the crucial point is that this probability is always only taken over the random choices of the algorithm, while the input is *not* chosen at random.

19.1.1 Amplifying the success of randomized algorithms

Theorem 19.1 gives us an algorithm that cuts $m/2$ edges in *expectation*. But, as we saw before, expectation does not immediately imply concentration, and so a priori, it may be the case that when we run the algorithm, most of the time we don't get a cut matching the expectation. Luckily, we can *amplify* the probability of success by repeating the process several times and outputting the best cut we find. We start by arguing that the probability the algorithm above succeeds in cutting at least $m/2$ edges is not *too* tiny.

Lemma 19.2 The probability that a random cut in an m edge graph cuts at least $m/2$ edges is at least $1/(2m)$.

Proof Idea:

To see the idea behind the proof, think of the case that $m = 1000$. In this case one can show that we will cut at least 500 edges with probability at least 0.001 (and so in particular larger than $1/(2m) = 1/2000$). Specifically, if we assume otherwise, then this means that with probability more than 0.999 the algorithm cuts 499 or fewer edges. But since

we can never cut more than the total of 1000 edges, given this assumption, the highest value of the expected number of edges cut is if we cut exactly 499 edges with probability 0.999 and cut 1000 edges with probability 0.001. Yet even in this case the expected number of edges will be $0.999 \cdot 499 + 0.001 \cdot 1000 < 500$, which contradicts the fact that we've calculated the expectation to be at least 500 in Theorem 19.1.

⋆

Proof of Lemma 19.2. Let p be the probability that we cut at least $m/2$ edges and suppose, towards a contradiction, that $p < 1/(2m)$. Since the number of edges cut is an integer, and $m/2$ is a multiple of 0.5, by definition of p, with probability $1-p$ we cut at most $m/2 - 0.5$ edges. Moreover, since we can never cut more than m edges, under our assumption that $p < 1/(2m)$, we can bound the expected number of edges cut by

$$pm + (1-p)(m/2 - 0.5) \leq pm + m/2 - 0.5 \tag{19.2}$$

But if $p < 1/(2m)$ then $pm < 0.5$ and so the right-hand side is smaller than $m/2$, which contradicts the fact that (as proven in Theorem 19.1) the expected number of edges cut is at least $m/2$.

■

19.1.2 Success amplification

Lemma 19.2 shows that our algorithm succeeds at least *some* of the time, but we'd like to succeed almost *all* of the time. The approach to do that is to simply *repeat* our algorithm many times, with fresh randomness each time, and output the best cut we get in one of these repetitions. It turns out that with extremely high probability we will get a cut of size at least $m/2$. For example, if we repeat this experiment $2000m$ times, then (using the inequality $(1-1/k)^k \leq 1/e \leq 1/2$) we can show that the probability that we will never cut at least $m/2$ edges is at most

$$(1 - 1/(2m))^{2000m} \leq 2^{-1000} \ . \tag{19.3}$$

More generally, the same calculations can be used to show the following lemma:

Lemma 19.3 There is an algorithm that on input a graph $G = (V, E)$ and a number k, runs in time polynomial in $|V|$ and k and outputs a cut $(S, \overline{S})$ such that

$$\Pr[\text{number of edges cut by } (S, \overline{S}) \ \geq |E|/2] \geq 1 - 2^{-k} \ . \tag{19.4}$$

Proof of Lemma 19.3. The algorithm will work as follows:

Algorithm AMPLIFY RANDOM CUT:

Input: Graph $G = (V, E)$ with n vertices and m edges. Denote $V = \{v_0, v_1, \ldots, v_{n-1}\}$. Number $k > 0$.

Operation:

1. Repeat the following $200km$ times:
   a. Pick x uniformly at random in $\{0,1\}^n$.
   b. Let $S \subseteq V$ be the set $\{v_i \ : \ x_i = 1 \ , \ i \in [n]\}$ that includes all vertices corresponding to coordinates of x where $x_i = 1$.
   c. If $(S, \overline{S})$ cuts at least $m/2$ then halt and output $(S, \overline{S})$.
2. Output "failed"

We leave completing the analysis as an exercise to the reader (see Exercise 19.1).

■

19.1.3 Two-sided amplification

The analysis above relied on the fact that the maximum cut has *one sided error*. By this we mean that if we get a cut of size at least $m/2$ then we know we have succeeded. This is common for randomized algorithms, but is not the only case. In particular, consider the task of computing some Boolean function $F : \{0,1\}^* \to \{0,1\}$. A randomized algorithm A for computing F, given input x, might toss coins and succeed in outputting $F(x)$ with probability, say, 0.9. We say that A has *two sided errors* if there is positive probability that $A(x)$ outputs 1 when $F(x) = 0$, and positive probability that $A(x)$ outputs 0 when $F(x) = 1$. In such a case, to simplify A's success, we cannot simply repeat it k times and output 1 if a single one of those repetitions resulted in 1, nor can we output 0 if a single one of the repetitions resulted in 0. But we can output the *majority value* of these repetitions. By the Chernoff bound (Theorem 18.12), with probability *exponentially close* to 1 (i.e., $1 - 2^{\Omega(k)}$), the fraction of the repetitions where A will output $F(x)$ will be at least, say 0.89, and in such cases we will of course output the correct answer.

The above translates into the following theorem

Theorem 19.4 — Two-sided amplification. If $F : \{0,1\}^* \to \{0,1\}$ is a function such that there is a polynomial-time algorithm A satisfying

$$\Pr[A(x) = F(x)] \geq 0.51 \tag{19.5}$$

for every $x \in \{0,1\}^*$, then there is a polynomial time algorithm B satisfying

$$\Pr[B(x) = F(x)] \geq 1 - 2^{-|x|} \tag{19.6}$$

for every $x \in \{0,1\}^*$.

We omit the proof of Theorem 19.4, since we will prove a more general result later on in Theorem 20.5.

19.1.4 What does this mean?

We have shown a probabilistic algorithm that on any m edge graph G, will output a cut of at least $m/2$ edges with probability at least $1 - 2^{-1000}$. Does it mean that we can consider this problem as "easy"? Should we be somewhat wary of using a probabilistic algorithm, since it can sometimes fail?

First of all, it is important to emphasize that this is still a *worst case* guarantee. That is, we are not assuming anything about the *input graph*: the probability is only due to the *internal randomness of the algorithm*. While a probabilistic algorithm might not seem as nice as a deterministic algorithm that is *guaranteed* to give an output, to get a sense of what a failure probability of 2^{-1000} means, note that:

- The chance of winning the Massachusetts Mega Million lottery is one over $(75)^5 \cdot 15$ which is roughly 2^{-35}. So 2^{-1000} corresponds to winning the lottery about 300 times in a row, at which point you might not care so much about your algorithm failing.
- The chance for a U.S. resident to be struck by lightning is about $1/700000$, which corresponds to about 2^{-45} chance that you'll be struck by lightning the very second that you're reading this sentence (after which again you might not care so much about the algorithm's performance).
- Since the earth is about 5 billion years old, we can estimate the chance that an asteroid of the magnitude that caused the dinosaurs' extinction will hit us this very second to be about 2^{-60}. It is quite likely that even a deterministic algorithm will fail if this happens.

So, in practical terms, a probabilistic algorithm is just as good as a deterministic one. But it is still a theoretically fascinating question whether randomized algorithms actually yield more power, or whether is it the case that for any computational problem that can be solved by probabilistic algorithm, there is a deterministic algorithm with nearly the same performance.[2] For example, we will see in Exercise 19.2 that there is in fact a deterministic algorithm that can cut at least $m/2$ edges in an m-edge graph. We will discuss this question in generality in Chapter 20. For now, let us see a couple of examples

[2] This question does have some significance to practice, since hardware that generates high quality randomness at speed is non-trivial to construct.

where randomization leads to algorithms that are better in some sense than the known deterministic algorithms.

19.1.5 Solving SAT through randomization

The 3SAT problem is **NP** hard, and so it is unlikely that it has a polynomial (or even subexponential) time algorithm. But this does not mean that we can't do at least somewhat better than the trivial 2^n algorithm for n-variable 3SAT. The best known worst-case algorithms for 3SAT are randomized, and are related to the following simple algorithm, variants of which are also used in practice:

Algorithm WalkSAT:

Input: An n variable 3CNF formula φ.

Parameters: $T, S \in \mathbb{N}$

Operation:

1. Repeat the following T steps:
   a. Choose a random assignment $x \in \{0,1\}^n$ and repeat the following for S steps:
      1. If x satisfies φ then output x.
      2. Otherwise, choose a random clause $(\ell_i \vee \ell_j \vee \ell_k)$ that x does not satisfy, choose a random literal in ℓ_i, ℓ_j, ℓ_k and modify x to satisfy this literal.
2. If all the $T \cdot S$ repetitions above did not result in a satisfying assignment then output `Unsatisfiable`

The running time of this algorithm is $S \cdot T \cdot poly(n)$, and so the key question is how small we can make S and T so that the probability that WalkSAT outputs `Unsatisfiable` on a satisfiable formula φ is small. It is known that we can do so with $ST = \tilde{O}((4/3)^n) = \tilde{O}(1.333\ldots^n)$ (see Exercise 19.4 for a weaker result), but we'll show below a simpler analysis yielding $ST = \tilde{O}(\sqrt{3}^n) = \tilde{O}(1.74^n)$, which is still much better than the trivial 2^n bound.[3]

[3] At the time of this writing, the best known randomized algorithms for 3SAT run in time roughly $O(1.308^n)$, and the best known deterministic algorithms run in time $O(1.3303^n)$ in the worst case.

Theorem 19.5 — WalkSAT simple analysis. If we set $T = 100 \cdot \sqrt{3}^n$ and $S = n/2$, then the probability we output `Unsatisfiable` for a satisfiable φ is at most $1/2$.

Proof. Suppose that φ is a satisfiable formula and let x^* be a satisfying assignment for it. For every $x \in \{0,1\}^n$, denote by $\Delta(x, x^*)$ the number of coordinates that differ between x and x^*. The heart of the proof is the following claim:

Claim I: For every x, x^* as above, in every local improvement step, the value $\Delta(x, x^*)$ is decreased by one with probability at least $1/3$.

Proof of Claim I: Since x^* is a *satisfying* assignment, if C is a clause that x does *not* satisfy, then at least one of the variables involve in C

must get different values in x and x^*. Thus when we change x by one of the three literals in the clause, we have probability at least $1/3$ of decreasing the distance.

The second claim is that our starting point is not that bad:

Claim 2: With probability at least $1/2$ over a random $x \in \{0,1\}^n$, $\Delta(x,x^*) \leq n/2$.

Proof of Claim II: Consider the map $FLIP : \{0,1\}^n \to \{0,1\}^n$ that simply "flips" all the bits of its input from 0 to 1 and vice versa. That is, $FLIP(x_0,\ldots,x_{n-1}) = (1-x_0,\ldots,1-x_{n-1})$. Clearly $FLIP$ is one to one. Moreover, if x is of distance k to x^*, then $FLIP(x)$ is distance $n-k$ to x^*. Now let B be the "bad event" in which x is of distance $> n/2$ from x^*. Then the set $A = FLIP(B) = \{FLIP(x) \ : \ x \in \{0,1\}^n\}$ satisfies $|A| = |B|$ and that if $x \in A$ then x is of distance $< n/2$ from x^*. Since A and B are disjoint events, $\Pr[A] + \Pr[B] \leq 1$. Since they have the same cardinality, they have the same probability and so we get that $2\Pr[B] \leq 1$ or $\Pr[B] \leq 1/2$. (See also Fig. 19.2).

Claims I and II imply that each of the T iterations of the outer loop succeeds with probability at least $1/2 \cdot \sqrt{3}^{-n}$. Indeed, by Claim II, the original guess x will satisfy $\Delta(x,x^*) \leq n/2$ with probability $\Pr[\Delta(x,x^*) \leq n/2] \geq 1/2$. By Claim I, even conditioned on all the history so far, for each of the $S = n/2$ steps of the inner loop we have probability at least $\geq 1/3$ of being "lucky" and decreasing the distance (i.e. the output of Δ) by one. The chance we will be lucky in all $n/2$ steps is hence at least $(1/3)^{n/2} = \sqrt{3}^{-n}$.

Since any single iteration of the outer loop succeeds with probability at least $\frac{1}{2} \cdot \sqrt{3}^{-n}$, the probability that we never do so in $T = 100\sqrt{3}^n$ repetitions is at most $(1 - \frac{1}{2\sqrt{3}^n})^{100\cdot\sqrt{3}^n} \leq (1/e)^{50}$.

■

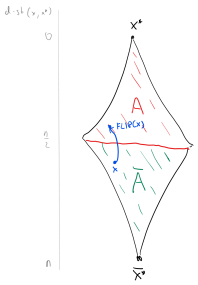

Figure 19.2: For every $x^* \in \{0,1\}^n$, we can sort all strings in $\{0,1\}^n$ according to their distance from x^* (top to bottom in the above figure), where we let $A = \{x \in \{0,1\}^n \mid dist(x,x^* \leq n/2\}$ be the "top half" of strings. If we define $FLIP : \{0,1\}^n \to \{0,1\}$ to be the map that "flips" the bits of a given string x then it maps every $x \in \overline{A}$ to an output $FLIP(x) \in A$ in a one-to-one way, and so it demonstrates that $|\overline{A}| \leq |A|$ which implies that $\Pr[A] \geq \Pr[\overline{A}]$ and hence $\Pr[A] \geq 1/2$.

19.1.6 Bipartite matching

The *matching* problem is one of the canonical optimization problems, arising in all kinds of applications: matching residents with hospitals, kidney donors with patients, flights with crews, and many others. One prototypical variant is *bipartite perfect matching*. In this problem, we are given a bipartite graph $G = (L \cup R, E)$ which has $2n$ vertices partitioned into n-sized sets L and R, where all edges have one endpoint in L and the other in R. The goal is to determine whether there is a *perfect matching*, a subset $M \subseteq E$ of n disjoint edges. That is, M matches every vertex in L to a unique vertex in R.

The bipartite matching problem turns out to have a polynomial-time algorithm, since we can reduce finding a matching in G to finding a maximum flow (or equivalently, minimum cut) in a related

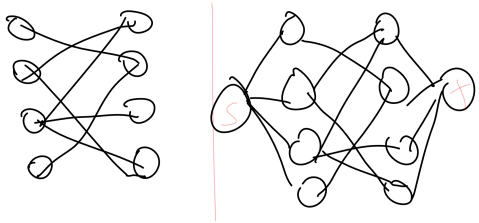

Figure 19.3: The bipartite matching problem in the graph $G = (L \cup R, E)$ can be reduced to the minimum s,t cut problem in the graph G' obtained by adding vertices s,t to G, connecting s with L and connecting t with R.

graph G' (see Fig. 19.3). However, we will see a different probabilistic algorithm to determine whether a graph contains such a matching.

Let us label G's vertices as $L = \{\ell_0, \ldots, \ell_{n-1}\}$ and $R = \{r_0, \ldots, r_{n-1}\}$. A matching M corresponds to a *permutation* $\pi \in S_n$ (i.e., one-to-one and onto function $\pi : [n] \to [n]$) where for every $i \in [n]$, we define $\pi(i)$ to be the unique j such that M contains the edge $\{\ell_i, r_j\}$. Define an $n \times n$ matrix $A = A(G)$ where $A_{i,j} = 1$ if and only if the edge $\{\ell_i, r_j\}$ is present and $A_{i,j} = 0$ otherwise. The correspondence between matchings and permutations implies the following claim:

Lemma 19.6 — Matching polynomial. Define $P = P(G)$ to be the polynomial mapping $\mathbb{R}^{n^2}$ to $\mathbb{R}$ where

$$P(x_{0,0}, \ldots, x_{n-1,n-1}) = \sum_{\pi \in S_n} \left(\prod_{i=0}^{n-1} sign(\pi) A_{i,\pi(i)} \right) \prod_{i=0}^{n-1} x_{i,\pi(i)} \qquad (19.7)$$

Then G has a perfect matching if and only if P is not identically zero. That is, G has a perfect matching if and only if there exists some assignment $x = (x_{i,j})_{i,j \in [n]} \in \mathbb{R}^{n^2}$ such that $P(x) \neq 0$.[4]

Proof. If G has a perfect matching M^*, then let π^* be the permutation corresponding to M and let $x^* \in \mathbb{Z}^{n^2}$ defined as follows: $x_{i,j} = 1$ if $j = \pi(i)$ and $x_{i,j} = 0$ otherwise. Note that for every $\pi \neq \pi^*$, $\prod_{i=0}^{n-1} x_{i,\pi(i)} = 0$ but $\prod_{i=0}^{n-1} x^*_{i,\pi^*(i)} = 1$. Hence $P(x^*)$ will equal $\prod_{i=0}^{n-1} A_{i,\pi^*(i)}$. But since M^* is a perfect matching in G, $\prod_{i=0}^{n-1} A_{i,\pi^*(i)} = 1$.

On the other hand, suppose that P is not identically zero. By (19.7), this means that at least one of the terms $\prod_{i=0}^{n-1} A_{i,\pi(i)}$ is not equal to zero. But then this permutation π must be a perfect matching in G. ■

As we've seen before, for every $x \in \mathbb{R}^{n^2}$, we can compute $P(x)$ by simply computing the *determinant* of the matrix $A(x)$, which is obtained by replacing $A_{i,j}$ with $A_{i,j} x_{i,j}$. This reduces testing perfect matching to the *zero testing* problem for polynomials: given some polynomial $P(\cdot)$, test whether P is identically zero or not. The intuition behind our randomized algorithm for zero testing is the following:

If a polynomial is not identically zero, then it can't have "too many" roots.

This intuition sort of makes sense. For one variable polynomials, we know that a non-zero linear function has at most one root, a quadratic function (e.g., a parabola) has at most two roots, and generally a degree d equation has at most d roots. While in more than one variable there can be an infinite number of roots (e.g., the polynomial $x_0 + y_0$ vanishes on the line $y = -x$) it is still the case that the set of roots is very "small" compared to the set of all inputs. For example,

[4] The sign of a permutation $\pi : [n] \to [n]$, denoted by $sign(\pi)$, can be defined in several equivalent ways, one of which is that $sign(\pi) = (-1)^{INV(\pi)}$ where $INV(pi) = |\{(x, y) \in [n] \mid x < y \wedge \pi(x) > \pi(y)\}$ (i.e., $INV(\pi)$ is the number of pairs of elements that are *inverted* by π). The importance of the term $sign(\pi)$ is that it makes P equal to the *determinant* of the matrix $(x_{i,j})$ and hence efficiently computable.

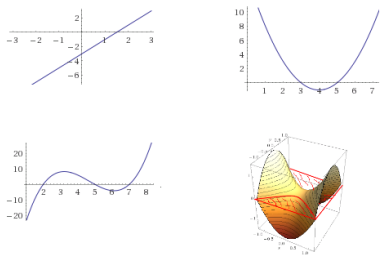

Figure 19.4: A degree d curve in one variable can have at most d roots. In higher dimensions, a n-variate degree-d polynomial can have an infinite number roots though the set of roots will be an $n-1$ dimensional surface. Over a finite field $\mathbb{F}$, an n-variate degree d polynomial has at most $d|\mathbb{F}|^{n-1}$ roots.

the root of a bivariate polynomial form a curve, the roots of a three-variable polynomial form a surface, and more generally the roots of an n-variable polynomial are a space of dimension $n-1$.

This intuition leads to the following simple randomized algorithm:

To decide if P is identically zero, choose a "random" input x and check if $P(x) \neq 0$.

This makes sense: if there are only "few" roots, then we expect that with high probability the random input x is not going to be one of those roots. However, to transform this into an actual algorithm, we need to make both the intuition and the notion of a "random" input precise. Choosing a random real number is quite problematic, especially when you have only a finite number of coins at your disposal, and so we start by reducing the task to a finite setting. We will use the following result:

Theorem 19.7 — Schwartz–Zippel lemma. For every integer q, and polynomial $P : \mathbb{R}^n \to \mathbb{R}$ with integer coefficients. If P has degree at most d and is not identically zero, then it has at most dq^{n-1} roots in the set $[q]^n = \{(x_0, \ldots, x_{n-1}) : x_i \in \{0, \ldots, q-1\}\}$.

We omit the (not too complicated) proof of Theorem 19.7. We remark that it holds not just over the real numbers but over any field as well. Since the matching polynomial P of Lemma 19.6 has degree at most n, Theorem 19.7 leads directly to a simple algorithm for testing if it is non-zero:

Algorithm Perfect-Matching:

Input: Bipartite graph G on $2n$ vertices $\{\ell_0, \ldots, \ell_{n-1}, r_0, \ldots, r_{n-1}\}$.

Operation:

1. For every $i, j \in [n]$, choose $x_{i,j}$ independently at random from $[2n] = \{0, \ldots 2n-1\}$.
2. Compute the determinant of the matrix $A(x)$ whose $(i,j)^{th}$ entry equals $x_{i,j}$ if the edge $\{\ell_i, r_j\}$ is present and 0 otherwise.
3. Output `no perfect matching` if this determinant is zero, and output `perfect matching` otherwise.

This algorithm can be improved further (e.g., see Exercise 19.5). While it is not necessarily faster than the cut-based algorithms for perfect matching, it does have some advantages. In particular, it is more amenable for parallelization. (However, it also has the significant disadvantage that it does not produce a matching but only states that one exists.) The Schwartz–Zippel Lemma, and the associated zero testing algorithm for polynomials, is widely used across computer science, including in several settings where we have no known deterministic algorithm matching their performance.

Chapter Recap

- Using concentration results, we can *amplify* in polynomial time the success probability of a probabilistic algorithm from a mere $1/p(n)$ to $1 - 2^{-q(n)}$ for every polynomials p and q.
- There are several randomized algorithms that are better in various senses (e.g., simpler, faster, or other advantages) than the best known deterministic algorithm for the same problem.

19.2 EXERCISES

Exercise 19.1 — Amplification for max cut. Prove Lemma 19.3 ■

Exercise 19.2 — Deterministic max cut algorithm. [5] ■

Exercise 19.3 — Simulating distributions using coins. Our model for probability involves tossing n coins, but sometimes algorithm require sampling from other distributions, such as selecting a uniform number in $\{0, \ldots, M-1\}$ for some M. Fortunately, we can simulate this with an exponentially small probability of error: prove that for every M, if $n > k\lceil \log M \rceil$, then there is a function $F : \{0,1\}^n \to \{0, \ldots, M-1\} \cup \{\perp\}$ such that **(1)** The probability that $F(x) = \perp$ is at most 2^{-k} and **(2)** the distribution of $F(x)$ conditioned on $F(x) \neq \perp$ is equal to the uniform distribution over $\{0, \ldots, M-1\}$.[6] ■

Exercise 19.4 — Better walksat analysis.

1. Prove that for every $\epsilon > 0$, if n is large enough then for every $x^* \in \{0,1\}^n$ $\Pr_{x \sim \{0,1\}^n}[\Delta(x, x^*) \leq n/3] \leq 2^{-(1-H(1/3)-\epsilon)n}$ where $H(p) = p\log(1/p) + (1-p)\log(1/(1-p))$ is the same function as in Exercise 18.10.
2. Prove that $2^{1-H(1/4)+(1/4)\log 3} = (3/2)$.
3. Use the above to prove that for every $\delta > 0$ and large enough n, if we set $T = 1000 \cdot (3/2+\delta)^n$ and $S = n/4$ in the WalkSAT algorithm then for every satisfiable 3CNF φ, the probability that we output `unsatisfiable` is at most $1/2$.

■

Exercise 19.5 — Faster bipartite matching (challenge). (to be completed: improve the matching algorithm by working modulo a prime) ■

[5] TODO: add exercise to give a deterministic max cut algorithm that gives $m/2$ edges. Talk about greedy approach.

[6] **Hint:** Think of $x \in \{0,1\}^n$ as choosing k numbers $y_1, \ldots, y_k \in \{0, \ldots, 2^{\lceil \log M \rceil} - 1\}$. Output the first such number that is in $\{0, \ldots, M-1\}$.

19.3 BIBLIOGRAPHICAL NOTES

The books of Motwani and Raghavan [MR95] and Mitzenmacher and Upfal [MU17] are two excellent resources for randomized algorithms. Some of the history of the discovery of Monte Carlo algorithm is covered here.

19.4 ACKNOWLEDGEMENTS