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 is 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 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, as 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

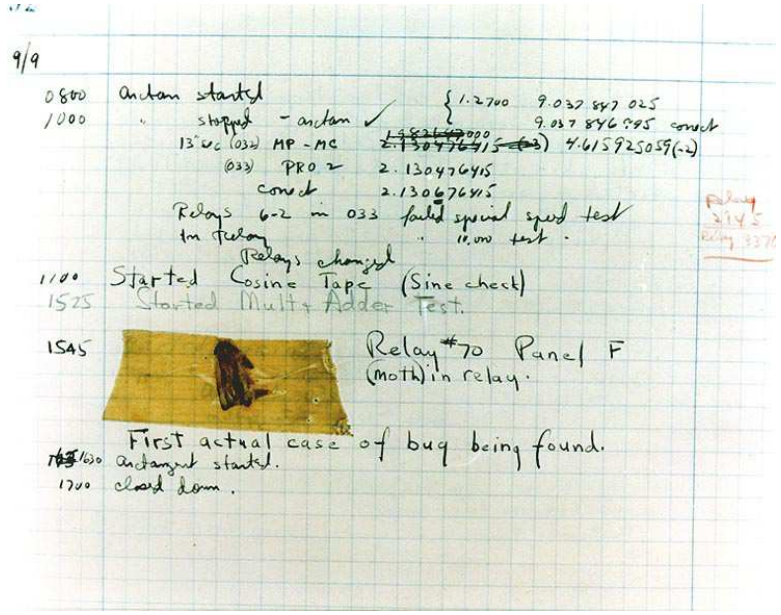

Figure 19.1: A 1947 entry in the log book of the Harvard MARK II computer containing an actual bug that caused a hardware malfunction. By Courtesy of the Naval Surface Warfare Center.

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 lecture, 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 in an informal / "pseudo-code" way, rather than as NAND or NAND++ programs. In the next lecture we will discuss how to augment the NAND and NAND++ models to incorporate the ability to "toss coins".

[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.

19.1 *Finding approximately good maximum cuts.*

Now that we have reviewed the basics of probability, let us see how we can use randomness to achieve algorithmic tasks. 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 set S such that the expected number of edges cut is at least $m/2$.

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: we choose x uniformly at random in $\{0,1\}^n$ and let S be the set corresponding to $\{i : x_i = 1\}$. For every edge e, we 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 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 $1/2$. Hence, $\mathbb{E}[X_e] = 1/2$ and if we let $X = \sum_e X_e$ over all the edges in the graph then $\mathbb{E}[X] = m(1/2) = m/2$. ■

19.1.1 *Amplification*

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 $m = 1000$, and suppose that the probability we cut at least 500 edges is only 0.001, or that in other words, with probability at least 0.999 the event A that we cut 499 or fewer edges holds. Then this is a contradiction for the fact we proved above that the expected number of cut edges is $m/2 = 500$. Indeed, even if we cut all the 1000 edges whenever A does not hold, the maximum value of the expectation will be smaller than $0.001 \cdot 1000 + 0.999 \cdot 499 < 1 + 499 = 500$.

Proof of Lemma 19.2. Let p be the probability that we cut at least $m/2$ edges. Suppose, towards the sake of contradiction, that $p < 1/m$, or, in other words that with probability more than $1-p$ we cut at most $m/2 - 0.5$ edges. (The latter holds since we can only cut an integer number of edges, and since $m/2$ is a multiple of 0.5, any integer smaller than it has at least 0.5 difference from it.) Since we can never cut more than m edges, under our assumption, we can bound the expected number of edges cut by

$$pm + (1-p)(m/2 - 0.5) \leq pm + m/2 - 0.5 \tag{19.1}$$

but if $p < 1/(2m)$ then $pm < 0.5$ and so the righthand side is smaller than $m/2$, contradicting our assumption. ■

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, for example, $2000m$ times, then the probability that we will never be able to cut at least $m/2$ edges is at most

$$(1 - 1/(2m))^{2000m} \leq 2^{-1000} \tag{19.2}$$

(using the inequality $(1-1/k)^k \leq 1/e \leq 1/2$).

19.1.2 *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 on 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 18.9), 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.

19.1.3 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 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 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 19.1 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 future lectures. For now, let us see a couple of examples where

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

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

19.1.4 *Solving SAT through randomization*

The 3SAT 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:

- On input an n variable 3CNF formula φ do the following for T steps:
  - 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, and choose a random literal in ℓ_i, ℓ_j, ℓ_k and modify x to satisfy this literal.
    3. Go back to step 1.

  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 can we make S and T so that the probability that WalkSAT outputs `Unsatisfiable` on a satisfiable formula φ will be small. It is known that we can do so with $ST = \tilde{O}((4/3)^n)$ (see Exercise 19.3 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]

Theorem 19.3 — 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 $1/2$.

[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. As mentioned above, the simple WalkSAT algorithm takes $\tilde{O}((4/3)^n) = \tilde{O}(1.333..^n)$ time.

- *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^*. We claim that

$(*)$: in every local improvement step, with probability at least $1/3$ we will reduce $\Delta(x, x^*)$ by one. Hence, if the original guess x satisfied $\Delta(x, x^*) \leq n/2$ (an event that, as we will show, happens with probability at least $1/2$) then with probability at least $(1/3)^{n/2} = \sqrt{3}^{-n/2}$ after $n/2$ steps we will reach a satisfying assignment. This is a pretty lousy probability of success, but if we repeat this $100\sqrt{3}^n$ times then it is likely that it that it will happen once.

To prove the claim $(*)$ note that any clause that x does not satisfy, it differs from x^* by at least one literal. So when we change x by one of the three literals in the clause, we have probability at least $1/3$ of decreasing the distance.

We now prove our earlier claim that with probability $1/2$ over $x \in \{0,1\}^n$, $\Delta(x, x^*) \leq n/2$. Indeed, consider the map $FLIP : \{0,1\}^n \to \{0,1\}^n$ where $FLIP(x_0, \ldots, x_{n-1}) = (1 - x_0, \ldots, 1 - x_{n-1})$. We leave it to the reader to verify that **(1)** $FLIP$ is one to one, and **(2)** $\Delta(FLIP(x), x^*) = n - \Delta(x, x^*)$ (and so in particular if $x \in \overline{A}$ then $FLIP(x) \in A$). Thus, if $A = \{x \in \{0,1\}^n : \Delta(x, x^*) \leq n/2\}$ then $FLIP$ is a one-to-one map from $\overline{A}$ to A, implying that $|A| \geq |\overline{A}|$ and hence $\mathbb{P}[A] \geq 1/2$ (see Fig. 19.2).

The above means that in any single repetition of the outer loop, we will end up with a satisfying assignment with probability $\frac{1}{2} \cdot \sqrt{3}^{-n}$. Hence the probability that we never do so in $100\sqrt{3}^n$ repetitions is at most $(1 - \frac{1}{2\sqrt{3}^n})^{100 \cdot \sqrt{3}^n} \leq (1/e)^{50}$. ■

19.1.5 Bipartite matching.

The *matching* problem is one of the canonical optimization problems, arising in all kinds of applications, including matching residents and hospitals, kidney donors and patients, or flights and 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 determined whether there is a *perfect matching* which is 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

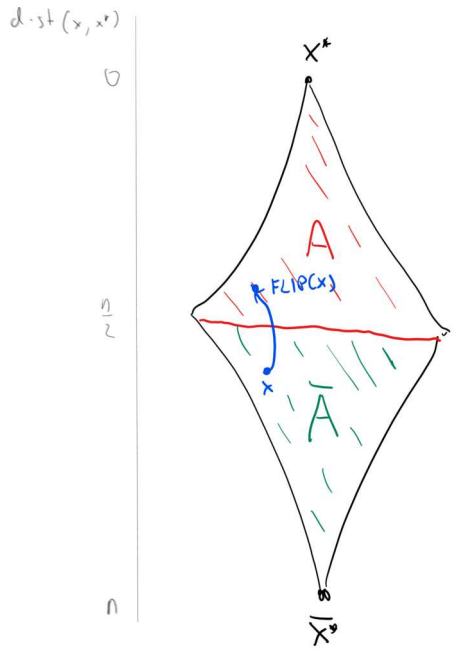

Figure 19.2: For every $x^* \in \{0,1\}^n$, we can sort all string in $\{0,1\}^n$ according to their distance from x^* (top to bottom in the above figure), where we let $A = \{x \in \{0,1\}^n \mid dist(x,x^* \leq n/2\}$ be the "top half" of strings. If we define $FLIP : \{0,1\}^n \to \{0,1\}$ to be the 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 $\mathbb{P}[A] \geq \mathbb{P}[\overline{A}]$ and hence $\mathbb{P}[A] \geq 1/2$.

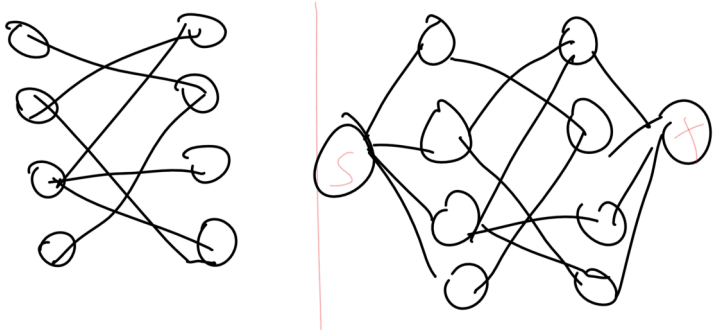

Figure 19.3: The bipartite matching problem in the graph $G = (L \cup R, E)$ can be reduced to the minimum s,t cut problem in the graph G' obtained by adding vertices s,t to G, connecting s with L and connecting t with R.

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

$$P(x_{0,0}, \ldots, x_{n-1,n-1}) = \sum_{\pi \in S_n} \left(\prod_{i=0}^{n-1} sign(\pi) A_{i,\pi(i)} \right) \prod_{i=0}^{n-1} x_{i,\pi(i)} \qquad (19.3)$$

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 $x_{i,j} = 0$. Note that for every $\pi \neq \pi^*$, $\prod_{i=0}^{n-1} x_{i,\pi(i)} = 0$ but $\prod_{i=0}^{n-1} x^*_{i,\pi^*(i)} = 1$ and 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 Eq. (19.3), 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 $A_{i,j}x_{i,j}$. So, 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 nonzero linear function has at most one root, a

[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.

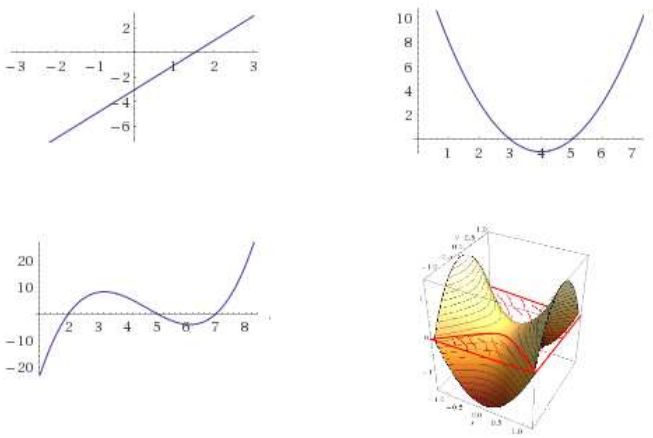

Figure 19.4: A degree d curve in one variable can have at most d roots. In higher dimensions, a n-variate degree-d polynomial can have an infinite number roots though the set of roots will be an $n-1$ dimensional surface. Over a finite field $\mathbb{F}$, an n-variate degree d polynomial has at most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

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, 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 as 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 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 19.5 — 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 19.5. We remark that it holds not just over the real numbers but over any field as well. Since the matching polynomial P of Lemma 19.4 has degree at most n, Theorem 19.5 leads directly to a simple algorithm for testing if it is nonzero:

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 corresponds equals $x_{i,j}$ if the edge $\{\ell_i, r_j\}$ is present and is equal to 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 19.4). While it is not necessarily faster than the cut-based algorithms for perfect matching, it does have some advantages and in particular it turns out to be more amenable for parallelization. (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.

19.2 *Lecture summary*

- 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.

19.3 Exercises

Exercise 19.1 — Deterministic max cut algorithm. [5] ■

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

Exercise 19.2 — 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 $F : \{0,1\}^n \rightarrow \{0,\ldots,M-1\} \cup \{\perp\}$ 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] ■

[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 $\{0,\ldots,M-1\}$.

Exercise 19.3 — Better walksat analysis.

1. Prove that for every $\epsilon > 0$, if n is large enough then for every $x^* \in \{0,1\}^n$ $\mathbb{P}_{x\sim\{0,1\}^n}[\Delta(x,x^*) \leq n/3] \leq 2^{-(1-H(1/3)-\epsilon)n}$ where $H(p) = p\log(1/p) + (1-p)\log(1/(1-p))$ is the same function as in Exercise 18.7.
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 19.4 — Faster bipartite matching (challenge). [7] ■

[7] TODO: add exercise to improve the matching algorithm by working modulo a prime

19.4 Bibliographical notes

monte carlo history: `http://permalink.lanl.gov/object/tr?what=info:lanl-repo/lareport/LA-UR-88-9068`

19.5 Further explorations

Some topics related to this lecture that might be accessible to advanced students include: (to be completed)

19.6 Acknowledgements