**Mathematical Toolkit** **Fall 2024**

# Lecture 10: November 4, 2024

Lecturer: Avrim Blum

Today we will discuss some applications of probabilistic reasoning.

## 1 The Probabilistic Method

The "probabilistic method" refers to an approach to showing something exists by setting up a probabilistic experiment and showing that the experiment has a nonzero chance of producing what you want. Or showing that something of size at least $k$ exists by setting up a probabilistic experiment and showing that the expected value under this experiment of the size of the thing you get is at least $k$.

Here is a simple example.

**Theorem 1.1** *Suppose $G$ is an undirected graph with $m$ edges. Then $G$ must contain a cut (a partitioning of the vertices into two sets $A$ and $B$) of size at least $m/2$ (at least $m/2$ edges cross between $A$ and $B$).*

**Proof:** Just put the vertices at random into $A$ or $B$. For any given edge $e$, the probability it crosses the cut is exactly $1/2$ (with probability $1/4$ both endpoints are in $A$ and with probability $1/4$ both endpoints are in $B$). Let $X_e$ be the indicator random variable for the event that $e$ crosses the cut, and let $X = \sum_e X_e$ be the total number of edges crossing the cut. By linearity of expectation, $\mathbb{E}[X] = m/2$. So, there must exist a cut of size at least $m/2$. ■

Here is another example. Recall that a CNF formula over variables $x_1, \ldots, x_n$ is a conjunction (AND) of clauses, where each clause is an OR of literals, where a literal is either a variable $x_i$ or its negation $\neg x_i$. A CNF formula is satisfiable if there is an assignment to the variables that makes the formula true (i.e., that satisfies at least one literal in each clause).

**Theorem 1.2** *Suppose $F$ is a $k$-CNF formula with less than $2^k$ clauses, in which every clause has size exactly $k$ (and you are not allowed to repeat variables inside a clause). Then $F$ must be satisfiable.*

**Proof:** Consider a random assignment $x$. The probability it fails to satisfy any given clause is $1/2^k$. So the expected number of clauses *not* satisfied is (# clauses)$/2^k < 1$. This means there must exist an assignment in which the number of clauses *not* satisfied is 0, i.e., a satisfying assignment. ■

What if a $k$-CNF has $m \geq 2^k$ clauses? A generalization of the above statement is that there must exist an assignment satisfying at least $\lceil m\left(1 - 1/2^k\right)\rceil$ clauses. For example, a 3-CNF formula must have an assignment satisfying at least a $7/8$ fraction of its clauses.

**Theorem 1.3** *Suppose $F$ is a $k$-CNF formula of $m$ clauses in which every clause has size exactly $k$ (and you are not allowed to repeat variables inside a clause). Then there must exist an assignment satisfying at least $\lceil m\left(1 - 1/2^k\right)\rceil$ clauses in $F$.*

**Proof:** (really the same as above). Consider again the uniform distribution over the sample space $\Omega = \{0,1\}^n$ of possible assignments to the variables $x_1, \ldots, x_n$. Let $X_i$ be the indicator RV for the event that clause $i$ is satisfied, and let $X = \sum_i X_i$ be the total number of clauses satisfied. We have $\mathbb{E}[X_i] = 1 - 1/2^k$ so $\mathbb{E}[X] = m(1 - 1/2^k)$. Since the number of clauses satisfied must be an integer, this means there must exist an assignment satisfying at least $\lceil m(1 - 1/2^k)\rceil$ clauses in $F$. ■

The above also gives an efficient *randomized* algorithm for finding an assignment satisfying least $\lceil m(1 - 1/2^k)\rceil$ clauses (since the random variable is bounded by $m$, so there has to be a reasonable chance that $X$ is at least its expectation). On your homework, you will give an efficient *deterministic* algorithm using a technique called the "conditional expectation method".

## 2 The Coupon Collector Problem

Consider the following problem: we have $n$ bins, and at each time step we place a ball into one of the bins independently at random. How long will it take until there are no empty bins? (This is often called the "coupon collector problem" because you think of each bin as a different kind of coupon, you get a random coupon at each time step, and you want to know how long until you get at least one of each kind of coupon.) Let $X$ be a random variable denoting the time it takes until there are no empty bins. Find $\mathbb{E}[X]$.

To solve this problem, let's write $X$ as a sum of simpler random variables:

$$X = \sum_{i=1}^{n} X_i,$$

where $X_i$ is the time to fill the ith new bin. (So, $X_1$ is equal to 1 with probability 1.) By linearity of expectation, we have

$$\mathbb{E}[X] = \sum_i \mathbb{E}[X_i]$$

Note that $X_i$ is a geometric random variable with parameter $\frac{n-i+1}{n}$, since if we have $i-1$ bins filled, $X_i$ represents the time until a ball is placed into any one of the remaining $n-i+1$ bins. Thus,

$$\mathbb{E}[X_i] = \frac{n}{n-i+1}.$$

Therefore,

$$\mathbb{E}[X] = \frac{n}{n} + \frac{n}{n-1} + \frac{n}{n-2} + \cdots + \frac{n}{1} = n \cdot H(n)$$

where $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$ is the $n^{th}$ *harmonic number*. It is known that $H_n = \ln n + \Theta(1)$ (which can be seen by upper and lower bounding the sum with an integral). Thus, we have that $\mathbb{E}[X] = n \ln n + \Theta(n)$.

# 3 DeMillo-Lipton-Schwartz-Zippel

Most problems that have polynomial-time randomized algorithms also have known polynomial time deterministic algorithms, but there are a few hold-outs.

Here is one: it is like an algebraic version of the SAT problem.

Say $p$ is an $n$-variable polynomial over some field $\mathbb{F}$ with $|\mathbb{F}| > 2d$, where $d$ is the degree of the polynomial. Assume $p$ is given in a form that can be evaluated on any given input in polynomial time. We want to know if $p$ is identically 0 or if on the other hand there exists an assignment $x$ (where each $x_i \in F$) for which $p(x) \neq 0$.

There is no known polynomial-time deterministic algorithm for this problem. But, one thing we *can* do is plug in random values and see if we get 0. The Demillo-Lipton-Schwartz-Zippel theorem says that if $p$ is not identically 0, then this is unlikely to happen just by chance. Note that this also gives us a way to test if two polynomials $p$ and $q$ are identical. We can plug in a random $x$ and see if $p(x) - q(x) = 0$. Here is the theorem we will prove:

**Theorem 3.1 (DeMillo-Lipton-Schwartz-Zippel)** *Say $p(x_1, ..., x_n)$ is a degree-d polynomial over some field $\mathbb{F}$, and not identically 0. For any finite $S \subseteq \mathbb{F}$, if we pick $r_1, ..., r_n$ at random in $S$, $\mathbb{P}[p(r_1, ..., r_n) = 0] \leq d/|S|$.*

**Proof:** We'll prove this by induction on $n$ (the number of variables). For the case $n = 1$, we have at most $d$ roots, so we satisfy the condition.

For the general case, there could be a large number of roots (e.g., over the reals there are an infinite number of roots of $x_1 x_2$). However, we can prove the theorem as follows. Say the maximum degree of $x_n$ is $i$. So, we have

$$p(x) = x_n^i p_i(x_1, ..., x_{n-1}) + x_n^{i-1} p_{i-1}(x_1, ..., x_{n-1}) + ...,$$

where $p_i$ is not identically 0 and has degree at most $d - i$.

Now, pick $x_1, ..., x_{n-1}$ at random. The probability we've zeroed out $p_i$ is some value $\alpha \leq (d-i)/|S|$ by induction. Assuming this doesn't happen, we have a degree-$i$ polynomial in 1 variable, so when we pick $x_n$ randomly, the chance we get 0 is at most $i/|S|$. Overall, the failure probability is at most $\alpha + (1-\alpha)i/|S| \leq d/|S| - i/|S| + (1-\alpha)i/|S| < d/|S|$. ■

## 4 Perfect Matchings in General Graphs

Now, here is a nice application of the above result. Say we have an undirected graph and want to find a perfect matching (a collection of edges such that every vertex is an endpoint of exactly one edge). You may have seen how to do this for *bipartite* graphs using network flow. But what about for general graphs? Here is a neat randomized algorithm that works for general graphs. This will solve the decision problem (does $G$ have a perfect matching) and then one can use that to actually find the matching. Note: there exist alternative deterministic algorithms, but this is the simplest polynomial-time algorithm I know of.

First, given a graph $G$, think of it as a directed graph $G'$ where each (undirected) edge in $G$ is replaced by two directed edges, one in each direction. $G$ has a perfect matching iff $G'$ has a cycle cover (a collection of disjoint cycles that cover all the vertices) in which all cycles have even length.

Now, given $G$, we create what's called the "Tutte matrix" $M$. For every edge $(i, j)$, $i < j$, we put variable $x_{ij}$ in entry $ij$ (above the diagonal) and $-x_{ij}$ in entry $ji$ (below the diagonal). The rest are 0. We can think of this like $G'$, but when we replace an undirected edge by two directed ones, we are giving one a positive sign and one a negative sign.

Now, consider the determinant of this matrix $det(M)$. This is a polynomial of degree at most $n$, and the claim is it is identically 0 iff $G$ has no perfect matchings. In particular, (1) every permutation that gives us a term of this polynomial is a directed cycle cover (every vertex has exactly one out-edge since there is one variable per row, and every vertex has exactly one in-edge since there is one variable per column), (2) two directed cycle covers correspond to the same term if and only if they have the same edges (but perhaps going around some cycles in opposite directions), and (3) if you sum over all cycle covers corresponding to the same term you will get zero if and only if there is an odd-length cycle in the cover. [Go through an example explaining why – this also uses the fact that the inverse of a permutation has the same sign]. Given this fact, we can tell if $G$ has a perfect

matching by checking if this polynomial is identically 0. We do this by just plugging in random values for the variables and then calculating the determinant (which we can do efficiently).

So, this gives us an efficient randomized algorithm for determining *if* a given graph has a perfect matching. If the answer is yes, we can then efficiently *find* a perfect matching by deleting any edge for which the answer on the graph remaining is still yes, until only the perfect matching itself is left.