CS221: Computational Complexity Prof. Salil Vadhan

Lecture 24: Counting Problems

11/20 *Scribe: Saurabh Sanghvi*

Contents

1 What is counting? Who cares?

We have already discussed several types of "problems"–whether languages, functions, search problems, or optimization problems. One theoretically and practically interesting type of problem since unmentioned is that of *counting*. For instance, we might ask:

- #SAT — Given φ, how many assignments satisfy φ?
- #Cycles — Given G, how many cycles are in G?

Why might we care about these sorts of problems?

- **Enumerative combinatorics**: This is a field that tries to find different ways to count things, but often runs into difficult walls. If we can say certain counting problems are hard, then we will provide a complexity-theoretic explanation for the lack of nice formulas for these problems.
- **Statistical physics**: Lots of counting problems make their way into this field. A physical system is described, which could be in various configurations. One often asks: how many (possibly weighted) configurations are there? Several results in complexity theory counting problems have actually come from work in the statistical physics community—the reverse occurs as well.
- **Network reliability**: Often counting problems arise when trying to compute the probability of a network's failure. Specifically, given a graph $G = (V, E)$ and $s, t \in V$, suppose each edge of a graph fails with probability 1/2. In this case, we have that:

  $$\Pr[s \text{ and } t \text{ remain connected}] = \frac{\#\text{ subgraphs of } G \text{ with path fom } s \text{ to } t}{2^{|E|}}$$

  (A *subgraph* of G is the graph defined by a subset of the graph's edges (and same vertex set)). Thus, counting can help determine whether a given network is reliable.

- **Artificial Intelligence** In AI an agent may have uncertainty about the value of various variables. For instance, it might know that x_1 is true with probability p_1, x_2 is true with probability p_2, etc. Calculating the probability of some formula being true is essentially a counting problem. For example, if $p_1 = p_2 = \cdots = p_n = 1/2$, then

$$\Pr[\varphi(x_1, \ldots, x_n)] = \frac{\#\text{ satisfying assignments}}{2^n}$$

(The case of arbitrary p_i's amounts to a weighted counting problem.)

2 Formalizing the notion

In order to talk about counting formally, we need to be precise about what we mean by the concept. In this course we will deal with only one counting class: $\#\mathbf{P}$:

Definition 1 *$f : \Sigma^* \to \mathbb{N}$ is in $\#\mathbf{P}$ iff $\exists$ $\mathbf{NP}$ relation R (i.e. polynomially balanced and polynomially decidable) such that for all x, $f(x) = \#\{y : (x, y) \in R\}$.*

That is, the function basically counts the number of $\mathbf{NP}$ witnesses for the given input x.
An example of this is

$$R = \{((G = (V, E), s, t), S) : S \subseteq E \text{ subgraph such that } s \text{ and } t \text{ are connected in } S\}.$$

This is the network reliability problem seen before.

Note: An equivalent way to formalize $\#\mathbf{P}$ is that it is the set of functions f such that $f(x) = \#$ accepting computations of a nondeterministic polynomial time TM on x.

A reasonable question to ask at this point is the following: which functions in $\#\mathbf{P}$ are in $(\mathbf{F})\mathbf{P} = \{$ functions computable in polynomial time$\}$? That the question might be resolvable any time soon is quickly put down by the following observation:

Observation 2 *$L \in \mathbf{NP}$ iff $\exists f \in \#\mathbf{P}$ such that $x \in L \Leftrightarrow f(x) \neq 0$ (where f is not necessarily unique).*

It follows immediately that counting is harder than decision. Indeed, if R defines an $\mathbf{NP}$-complete language L, then the counting problem defined by R is $\mathbf{NP}$-hard. As a result, counting is really most interesting when the language in question is easy, but the corresponding counting problem is hard. We'll see a couple examples of this shortly.

3 Relating counting problems

There are two types of reductions one might try to use when considering counting problems:

1. **Parsimonious reductions:** f reduces to g iff there exists a logspace-computable function $h : \Sigma^* \to \Sigma^*$ such that for all x, $g(h(x)) = f(x)$. That is, the reduction preserves the number of solutions.

2. **Cook reductions:** f reduces to g iff $g \in \mathbf{FP}^f$.

Parsimonious reductions, while nice when available, are far too restrictive for our purposes. Thus, we will use Cook reductions for our notions of completeness. This yields the following definition:

Definition 3 *f is* $\#\mathbf{P}$-complete *iff $f \in \#\mathbf{P}$ and $\forall g \in \#\mathbf{P}, g \in \mathbf{FP}^f$.*

It is natural now to try to find such problems, and indeed a variant of a standard **NP**-complete problem does the trick:

Theorem 4 #3SAT *is $\#\mathbf{P}$-complete.*

Proof: That $\#3\mathrm{SAT} \in \#\mathbf{P}$ is obvious—use the same R that defines 3SAT. The Cook-Levin reduction is actually parsimonious, and so applying the same reduction from any $\#\mathbf{P}$ problem to #3SAT we find that the result follows immediately. To see that the reduction is parsimonious, recall that the reduction proceeded in two steps:

1. First, given relation R (and an input length n), we construct a circuit C such that $(x, y) \in R$ iff $C(x, y) = 1$. Since the same y that satisfies R will make the circuit value 1, this is certainly parsimonious.

2. Second, we convert C into a formula $\varphi(x, y, z)$ such that if $C(x, y) = 1$ then there exists a unique z such that $\varphi(x, y, z) = 1$, and if $C(x, y) = 0$ then $\nexists z \varphi(x, y, z) = 1$. Obviously x and y are again the same, and z just consists of variables whose possible satisfying values are uniquely determined by x and y. Thus, the number of solutions is again maintained.

■

Theorem 5 #3DNF *is $\#\mathbf{P}$-complete.*

Proof: That $\#3\mathrm{DNF} \in \#\mathbf{P}$ is clear: the decision problem is certainly in **NP** (can guess the satisfying assignment and check it), so just use the same R.

To prove it is $\#\mathbf{P}$-hard, we will reduce from #3SAT. Given $\varphi(x_1 \ldots, x_n)$, just consider $\neg\varphi$. This can be written efficiently in DNF form, and the number of assignments that satisfy it is simply the number *not* satisfying φ, which is just 2^n - # satisfying assigments of φ.

Thus, the following algorithm shows $\#3\mathrm{SAT} \in \mathbf{FP}^{\#3\mathrm{DNF}}$.

On input 3-CNF φ:

1. Convert $\neg\varphi$ to a 3DNF and send this to the oracle, receiving back N.

2. Output $2^n - N$.

■

#3DNF is interesting because, unlike #3SAT, it's decision problem is in **P**. Another such example follows:

Theorem 6 #MON *(# of satisfying assignments to a monotone formula) is $\#\mathbf{P}$-complete.*

Proof: Once again, membership in #**P** is trivial in the same way as before. We will show #SAT $\in$ $\mathbf{FP}^{\#\mathbf{MON}}$ (note that the fact that #SAT is #**P**-complete follows trivially from the #**P**-completeness of #3SAT). Given $\varphi(x_1, \ldots, x_n)$, map it to ψ as follows:

$$\varphi(x_1, \ldots, x_n) \mapsto \psi(x_1, \ldots, x_n, y_1, \ldots, y_n) = \varphi \text{ where } \neg x_i \text{ replaced by } y_i.$$

This gives

$$\begin{aligned}
\#\varphi(\overline{x}) &= \# \left[\psi(\overline{x}, \overline{y}) \wedge \left(\bigwedge_{i=1}^{n} (x_i \vee y_i) \right) \wedge \neg \bigvee_{i=1}^{n} (x_i \wedge y_i) \right] && (1) \\
&= \# \left[\alpha(\overline{x}, \overline{y}) \wedge \neg \beta(\overline{x}, \overline{y} \right], \text{ where } \alpha, \beta \text{ are monotone formulae} && (2) \\
&= \#\alpha - \#(\alpha \wedge \beta) && (3)
\end{aligned}$$

For (1), basically just note the satisfying assignments of φ are the "same" as assignments satisfying ψ such that either x_i or y_i is true, and it is not the case that both are true, for all i (since this simulates $y_i = \neg x_i$).

For (2), just let $\alpha(\overline{x}, \overline{y}) \triangleq \psi(\overline{x}, \overline{y}) \wedge \bigwedge_{i=1}^{n} (x_i \vee y_i)$ and $\beta(\overline{x}, \overline{y}) \triangleq \bigvee_{i=1}^{n} (x_i \wedge y_i)$.

For (3), a simple Venn diagram makes this pretty clear: in words, if an assignment satisfies α, but does not satisfy $\alpha \wedge \beta$, it must not satisfy β, so it satisfies $\alpha \wedge \neg\beta$. Conversely, if something satisfies $\alpha \wedge \neg\beta$, it must satisfy α, and it can't satisfy β, so it can't satisfy $\alpha \wedge \beta$.

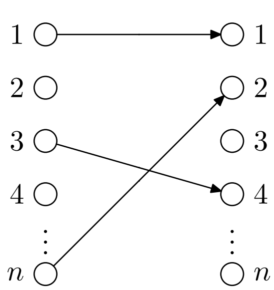
■

An important problem in mathematics studied since the 1800's is that of the *permanent*. It turns out that the problem of calculating it is #**P**-complete, which has interesting implications for the class #**P**, given the algebraic structure of the permanent.

Definition 7 *The* ***permanent*** *of an $n \times n$ matrix A is* $\mathrm{Perm}(A) \triangleq \sum_{\pi} A_{1\pi(1)} A_{2\pi(2)} \ldots A_{n\pi(n)}$*, where the sum is over all permutations $\pi : [n] \to [n]$.*

This definition bears striking similarities to that of the determinant–indeed, the determinant differs only in multiplying each term of the sum by the sign of the permutation π. Computing the determinant is easy—it can be done in polynomial time using Gaussian elimination. Thus, it is quite surprising that computing the permanent, which seems all the simpler without the complexity of the signs, is in fact hard.

As of now it is not clear why we would talk about computing the permanent as a *counting* problem. To change this, assume that A has 0/1 entries. Now we can think of A as the adjacency matrix of a bipartite graph G. Thus, G would look like the following:



(where we put an edge between i and j i $A_{ij} = 1$)

We first need the following definition:

Definition 8 *A* ***perfect matching*** *in a graph is a subset of edges such that each vertex touches exactly one edge. (In particular, in a bipartite graph, the edges give a one-to-one correspondence between left and right vertices.)*

Now, we may assert a connection between counting and the permanent:

Proposition 9 $\mathrm{Perm}(A) = \#$ *perfect matchings in* G.

To prove the proposition, one would first show the following lemma:

$$\text{A term in } \mathrm{Perm}(A) \text{ is nonzero} \Leftrightarrow \{(i, \pi(i))\} \text{ is a "perfect matching" in } G$$

From here, the result follows, because a term in the permanent is nonzero iff all the terms $A_{i\pi(i)} = 1$, which corresponds exactly to a perfect matching in G, since this means that there are edges from each left vertex to some right vertex (π is a bijection). So the number of nonzero terms equal the number of perfect matchings, but the number of nonzero terms is simply the permanent, since each nonzero term is 1.

This proposition thus shows why computing the permanent can be seen as a counting problem. As indicated before, however, we know much more than just this:

Theorem 10 (Valiant) *: The problem of computing the permanent of* $0/1$ *matrices is* #**P**-*complete.*

We won't prove this theorem in class, but it's in Papadimitriou (Ch. 18).It's worth noting that the decision problem corresponding to the permanent—that of finding a perfect matching—is in **P**. This is shown in AM 107 (via alternating paths) and CS 226r (via max flow).