Notes on Complexity Theory	Last updated: November, 2011
Lecture 23	
Jonathan Katz	

1 The Complexity of Counting

1.1 The Class $\#\mathcal{P}$

$\mathcal{P}$ captures problems where we can efficiently find an answer; $\mathcal{NP}$ captures problems where we can efficiently verify an answer. *Counting* the number of answers gives rise to the class $\#\mathcal{P}$.

Recall that $L \in \mathcal{NP}$ if there is a (deterministic) Turing machine M running in time polynomial in its first input such that

$$x \in L \Leftrightarrow \exists w \; M(x, w) = 1. \tag{1}$$

The corresponding counting problem is: given x, determine the *number* of strings w for which $M(x, w) = 1$. (Note that $x \in L$ iff this number is greater than 0.) An important point is that for a given L, there might be several (different) machines for which Eq. (1) holds; when specifying the counting problem, we need to fix not only L but also a specific machine M. Sometimes, however, we abuse notation when there is a "canonical" M for some L.

We let $\#\mathcal{P}$ denote the class of counting problems corresponding to polynomial-time M as above. The class $\#\mathcal{P}$ can be defined as a function class or a language class; we will follow the book and speak about it as a function class. Let M be a (two-input) Turing machine M that halts on all inputs, and say M runs in time $t(n)$ where n denotes the length of its first input. Let $\#M(x) \stackrel{\text{def}}{=} \left|\{w \in \{0,1\}^{t(|x|)} \mid M(x,w) = 1\}\right|$. Then:[1]

Definition 1 *A function $f : \{0,1\}^* \to \mathbb{N}$ is in $\#\mathcal{P}$ if there is a Turing machine M running in time polynomial in its first input such that $f(x) = \#M(x)$.*

We let $\mathcal{FP}$ denote the class of functions computable in polynomial time; this corresponds to the language class $\mathcal{P}$.

Any $f \in \#\mathcal{P}$ defines a natural language $L \in \mathcal{NP}$: letting M be the Turing machine for which $f(x) = \#M(x)$, we can define

$$L = \{x \mid f(x) > 0\}.$$

This view can be used to show that $\#\mathcal{P}$ is at least as hard as $\mathcal{NP}$. Consider, for example, the problem $\#\mathsf{SAT}$ of counting the number of satisfying assignments of a boolean formula. It is easy to see that $\#\mathsf{SAT} \in \#\mathcal{P}$, but $\#\mathsf{SAT}$ is not in $\mathcal{FP}$ unless $\mathcal{P} = \mathcal{NP}$ (since being able to count the number of solutions clearly implies the ability to determine existence of a solution). Interestingly, it is also possible for a counting problem to be hard even when the corresponding decision problem is easy. (Actually, it is trivial to come up with "cooked up" examples where this is true. What is

[1] For completeness, we also discuss how $\#\mathcal{P}$ can be defined as a language class. For the purposes of this footnote only, let $\#\mathcal{FP}$ denote the function class (as defined above). Then language class $\#\mathcal{P}$ can be defined as: $L \in \#\mathcal{P}$ if there is a Turing machine M running in time polynomial in its first input such that $L = \{(x,k) \mid \#M(x) \leq k\}$. We use inequality rather than equality in this definition to ensure that $\#\mathcal{P} = \mathcal{P}^{\#\mathcal{FP}}$ and $\#\mathcal{FP} = \mathcal{FP}^{\#\mathcal{P}}$.

interesting is that there are many natural examples.) For example, let $\#\mathsf{cycle}$ be the problem of counting the number of cycles in a directed graph. Note that $\#\mathsf{cycle} \in \#\mathcal{P}$, and the corresponding decision problem is in $\mathcal{P}$. But:

Claim 1 *If $\mathcal{P} \neq \mathcal{NP}$, then $\#\mathsf{cycle} \notin \mathcal{FP}$.*

Proof (Sketch) If $\#\mathsf{cycle} \in \mathcal{FP}$ then we can detect the existence of Hamiltonian cycles in polynomial time. (Deciding Hamiltonicity is a classic $\mathcal{NP}$-complete problem.) Given a graph G, form a new graph G' by replacing each edge (u, v) with a "gadget" that introduces $2^{n \log n}$ paths from u to v. If G has a Hamiltonian cycle, then G' has at least $\left(2^{n \log n}\right)^n = n^{n^2}$ cycles; if G does not have a Hamiltonian cycle then its longest cycle has length at most $n-1$, and it has at most n^{n-1} cycles; thus, G' has at most $\left(2^{n \log n}\right)^{n-1} \cdot n^{n-1} < n^{n^2}$ cycles. ∎

There are two approaches, both frequently encountered, that can be used to define (different notions of) $\#\mathcal{P}$-completeness. We say a function $g \in \#\mathcal{P}$ is *$\#\mathcal{P}$-complete under parsimonious reductions* if for every $f \in \#\mathcal{P}$ there is a polynomial-time computable function ϕ such that $f(x) = g(\phi(x))$ for all x. (A more general, but less standard, definition would allow for two polynomial-time computable functions ϕ, ϕ' such that $f(x) = \phi'(g(\phi(x)))$.) This is roughly analogous to a Karp reduction. An alternative definition is that $g \in \#\mathcal{P}$ is *$\#\mathcal{P}$-complete under oracle reductions* if for every $f \in \#\mathcal{P}$ there is a polynomial-time Turing machine M such that f is computable by M^g. (In other words, $\#\mathcal{P} \subseteq \mathcal{FP}^g$.) This is analogous to a Cook reduction.

Given some $g \in \#\mathcal{P}$, denote by L_g the $\mathcal{NP}$-language corresponding to g (see above). It is not hard to see that if g is $\#\mathcal{P}$-complete under parsimonious reductions then L_g is $\mathcal{NP}$-complete. As for the converse, although no general result is known, one can observe that most Karp reductions are parsimonious; in particular, $\#\mathsf{SAT}$ is $\#\mathcal{P}$-complete under parsimonious reductions. $\#\mathcal{P}$-completeness under oracle reductions is a much more liberal definition; as we will see in the next section, it is possible for g to be $\#\mathcal{P}$-complete under Cook reductions even when $L_g \in \mathcal{P}$.

1.2 $\#\mathcal{P}$-Completeness of Computing the Permanent

Let $A = \{a_{i,j}\}$ be an $n \times n$ matrix over the integers. The *permanent* of A is defined as:

$$\mathrm{perm}(A) \stackrel{\text{def}}{=} \sum_{\pi \in S_n} \prod_{i=1}^{n} a_{i,\pi(i)}\,,$$

where S_n is the set of all permutations on n elements. This formula is very similar to the formula defining the *determinant* of a matrix; the difference is that in the case of the determinant there is an extra factor of $(-1)^{\mathrm{sign}(\pi)}$. Nevertheless, although the determinant can be computed in polynomial time, computing the permanent (even of boolean matrices) is $\#\mathcal{P}$-complete.

We should say a word about why computing the permanent is in $\#\mathcal{P}$ (since it does not seem to directly correspond to a counting problem). The reason is that computing the permanent is equivalent to (at least) two other problems on graphs. For the case when A is a boolean matrix, we may associate A with a bipartite graph G_A having n vertices in each component, where there is an edge from vertex i (in the left component) to vertex j (in the right component) iff $a_{i,j} = 1$. Then $\mathrm{perm}(A)$ is equal to the number of perfect matchings in G_A. For the case of a general integer matrices, we may associate any such matrix A with an n-vertex weighted, directed graph G_A (allowing self-loops) by viewing A as a standard adjacency matrix. A *cycle cover* in G_A is a set of edges such that each vertex has exactly one incoming and outgoing edge in this set. (Any cycle cover corresponds to a permutation π on $[n]$ such that $(i, \pi(i))$ is an edge for all i.) The *weight* of a cycle cover is the product of the weight of the edges it contains. Then $\mathrm{perm}(A)$ is equal to the sum of the weights of the cycle covers of G_A. (For boolean matrices, $\mathrm{perm}(A)$ is just the number of cycle covers of G_A.)

Determining *existence* of a perfect matching, or of a cycle cover, can be done in polynomial time; it is *counting* the number of solutions that is hard:

Theorem 2 *Permanent for boolean matrices is $\#\mathcal{P}$-complete under oracle reductions.*

The proof is quite involved and so we skip it; a full proof can be found in [1, Section 17.3.1].

References

[1] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.