CS 710: Complexity Theory 11/29/2011

Lecture 23: Counting

Instructor: Dieter van Melkebeek Scribe: Li-Hsiang Kuo

Last lecture we introduced extractors. We gave an application of extractors: running BPP algorithm with weak sources. In the first part of today's lecture, we give another application of extractors. In particular, an alternative proof of BPP $\subseteq \Sigma_2^p \cap \Pi_2^p$ by extractors. Then we discuss constructions of polynomial time computable extractors. We give a construction based on the construction of pseudorandom generators.

In the second part of this lecture we introduce the class #P of counting problems. Problems we have seen in class so far are decision problems where the output is a single bit. On the contrary, counting problems require integers (in the form of binary strings) as output. We look at some properties of #P, as well as relations between #P and other complexity classes.

1 Extractors

We restate the definition of an extractor here as reference.

Definition 1 (Extractor)**.** *The function $E : \{0,1\}^n \times \{0,1\}^\ell \to \{0,1\}^m$ is a (k,ϵ)-extractor if for all X on $\{0,1\}^n$ with $H_\infty(X) \geq k$,*

$$\| E(X, U_\ell) - U_m \|_1 < 2\epsilon,$$

where U_ℓ is a uniform variable on ℓ bits and U_m is uniform on m bits.

1.1 Applications

We give two applications of extractors.

1. Simulating BPP algorithm with weak random sources.
2. An alternate proof that BPP $\subseteq \Sigma_2^p \cap \Pi_2^p$.

We have shown the first application in the last lecture. The second application we refer to the note from previous run of this course.

1.2 Constructions

There are several different constructions of polynomial time computable extractors. We give one which is based on the construction of pseudorandom generators. The others we refer to the note from previous run of this course.

Theorem 1. *If $(\exists L \in E) C_L(n) = 2^{\Omega(n)}$, then there exists polynomial time $\frac{1}{r}$-PRG*

$$G : \{0,1\}^{O(\log r)} \to \{0,1\}^r$$

for circuits of size $\leq r$.

Recall the pseudorandom generator that we gave in a previous lecture that is secure against circuits; that is, the generator is indistinguishable to circuits of size n. Given a function $f : \{0,1\}^{O(\log r)} \to \{0,1\}$ with non-uniform circuit complexity $C_f \geq r^c$ with a constant c, we were able to construct a generator $G_f : \{0,1\}^{O(\log r)} \to \{0,1\}^r$ such that for all circuits D of size $\leq r$,

$$|\Pr_\rho[D(\rho) = 1] - \Pr_\sigma[D(G_f(\sigma)) = 1]| \leq \frac{1}{r}.$$

An examination of the proof reveals that the proof relativizes, so that if the function has circuit complexity $C_f^A \geq r^c$ even for circuits that are given an A oracle (i.e., circuits may have gates that query A), then the output is indistinguishable even for circuits D^A with oracle access to A. In particular, ($\forall$ oracle A)($\forall f : \{0,1\}^{O(\log r)} \to \{0,1\}$), if f has $C_f^A \geq r^c$, then

$$|\Pr_\rho[D^A(\rho) = 1] - \Pr_\sigma[D^A(G_f(\sigma)) = 1]| \leq \frac{1}{r}.$$

This inequality already makes use of a short random seed, but only works for a fixed f. So to view G_f as an extractor, we use the weak random source to pick a function f at random, allowing the sample to specify the truth table of f. Since f is a function on $O(\log r)$ bits, this can be specified in $r^{O(1)}$ bits.

We are now ready to define the extractor. We view x as the truth table of a function f from $O(\log r)$ bits to 1 bit, and view y as the seed σ for the pseudorandom generator G_f, then

$$E(x, y) = G_f(\sigma).$$

Now consider the difference in probability assigned to a set $A \subseteq \{0,1\}^r$ by this extractor and the uniform distribution. By the fact that x comes from a weak random source with min-entropy at least k, we split the difference in probability assigned to A based on whether or not f has high circuit complexity. We have

$$\begin{aligned} |\Pr[E(x,y) \in A] - \Pr[\rho \in A]| &= |\Pr[G_f(\sigma) \in A] - Pr[\rho \in A]| \\ &\leq \Pr[C_f^A < r^c] \cdot 1 + \tfrac{1}{r} \\ &\leq (\#\text{ of } f \text{ such that } C_f^A < r^c) \cdot 2^{-k} + \tfrac{1}{r} \\ &\leq 2^{r^{2c}} \cdot 2^{-k} + \tfrac{1}{r}. \end{aligned}$$

This means we only need to have $k = r^{\Omega(1)}$ to ensure the above probability is less than some given constant ϵ, showing that $E(x, y)$ is a (k, ϵ)-extractor.

2 Counting Class

All problems we have studied so far are decision problems — problems that require only "yes" or "no" answers. In this section, we introduce a new class of problems that require multiple-bit outputs. In particular, we consider problems of the following form: Given an instance x, what is the number of solutions (or witnesses) to x? This leads us to exploring the class #P of counting problems.

2.1 Definitions and Examples

First we give the definition of the complexity class #P.

Definition 2. $\#\mathrm{P} = \{f : \{0,1\}^* \to \mathbb{N} \mid$ *There exists a polynomial-time NTM* M *such that for all inputs* x, $f(x)$ *equals the number of accepting computation paths of* M *on* x $\}$.

Here are some examples of problems in #P.

1. #SAT is in #P. Recall that the corresponding decision problem SAT, which asks whether there exists a satisfying assignment, is in NP. We can construct a NTM M which, on input ϕ, guesses an assignment and accepts if it satisfies ϕ. Note that the number of accepting computation paths of M on ϕ is exactly the number of satisfying assignments for ϕ.

2. The problem #PM, which asks for the number of perfect matchings in a bipartite graph, is in #P. Note that the corresponding decision problem PM (existence of perfect matchings in bipartite graphs) can be solved in polynomial time. However, #PM = #SAT under $\leq_o^p$ reductions. This problem arises from statistical physics.

3. For any polynomial-time decidable graph property (for example, connectivity and acyclicity), counting the number of graphs of a given size n that satisfy the property is a problem in #P. These problems appear very often in enumerative combinatorics.

4. Let A be an $n \times n$ matrix with coefficients in $\mathbb{N}$. Define the *permanent* of A as

$$perm(A) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} A_{i,\sigma(i)},$$

   where S_n is the set of all permutations on $\{1, \ldots, n\}$. Note that the permanent of a matrix is very similar to the determinant except that a factor of $(-1)^{sign(\sigma)}$ is multiplied to each permutation for the determinant, where $sign(\sigma)$ denotes the sign of the permutation σ. The problem of computing the permanent of a given matrix A is in #P. This might not be obvious at first sight. In fact, we can construct a NTM M that generates computation paths according to the entries of A as follows. On input A, M guesses a permutation σ. For each entry b captured by σ, M creates b computation paths. (If $b = 0$, reject immediately.) Each of these computation paths keeps on expanding, until all n entries have been processed. The total number of computation paths generated is exactly the product of the entries captured by σ. Summing over all permutations, we get the permanent of A. We now argue that M runs in polynomial time. Guessing a permutation takes $O(n \log n)$ steps since we need to specify n numbers and each number has $O(\log n)$ bits. Let m be the largest integer entry in A. Spawning m computation paths takes $O(\log m)$ steps since only a constant number of computation paths can be spawned in each step. Thus, the total running time of M is $O(n \log n + n \log m)$, which is polynomial in the input length $\Omega(n^2 + \log m)$. As an aside, the number of perfect matchings in a bipartite graph G is equal to the permanent of the "adjacency matrix" of G. [1]

[1] The "adjacency matrix" here is the same as usual adjacency matrices except that the rows represent vertices in one partition and the columns represent vertices in the other.

In the above, we restricted our attention to matrices with non-negative entries. If we relaxed this constraint and let the matrices have negative entries, then computing their permanents would not be a problem in #P, simply because the permanents could be negative. In fact, there is another complexity class that captures this problem:

$$\text{GapP} = \{f - g \mid f, g \in \#\text{P}\}.$$

We can show that the unrestricted permanent problem lies in GapP by constructing a NTM M^+ that only accepts permutations giving positive products, and another NTM M^- that only accepts permutations giving negative products.

2.2 Properties

The complexity class #P exhibits some algebraic properties:

1. #P is closed under addition, i.e. for any two functions f and g in #P, their sum $f + g$ also lies in #P. To show that this holds, let M and N be NTMs inducing f and g respectively. Let P be an NTM which, on input x, immediately creates two computation paths. One of the paths runs the computation of M on x; the other runs the computation of N on x. It is easy to see that the total number of accepting computation paths is $f(x) + g(x)$. More generally, uniform exponential sums of #P functions are also in #P, i.e., for any $g \in \#\text{P}$ and constant c, the function
$$f(x) = \sum_{|y|=|x|^c} g(x, y)$$
is also in #P.

2. #P is closed under multiplication. This can be done by running the second NTM at the end of every accepting computation path of the first NTM. More generally, uniform polynomial products of #P functions are also in #P, i.e., for any $g \in \#\text{P}$ and constant c, the function
$$f(x) = \prod_{|y|=c\cdot\log|x|} g(x, y)$$
is also in #P.

3. The complexity class GapP also has the above two properties. Furthermore, it is closed under subtraction, a property which #P does not possess.

Remark. It follows directly from properties (1) and (2) that computing permanents over $\mathbb{N}$ is in #P.

2.3 Relationships

Most of the complexity classes we have encountered are classes of decision problems. In order to compare #P with them, we need to make #P an oracle. Denote by $\text{P}^{\#\text{P}}$ the class of decision problems solvable using a polynomial-time DTM with access to a #P oracle, and $\text{P}^{\#\text{P}[1]}$ the class of decision problems solvable using a polynomial-time DTM that makes at most one query to a #P oracle.

Proposition 1. *The following relations hold:*

(a) $\mathrm{NP} \subseteq \mathrm{P}^{\#\mathrm{P}[1]}$.

(b) $\mathrm{BPP} \subseteq \mathrm{P}^{\#\mathrm{P}[1]}$.

(c) $\mathrm{PH} \subseteq \mathrm{P}^{\#\mathrm{P}}$.

(d) $\oplus\mathrm{P} \subseteq \mathrm{P}^{\#\mathrm{P}[1]}$

(e) $\mathrm{P}^{\#\mathrm{P}} \subseteq \mathrm{PSPACE}$.

Proof. Let L be a problem in NP, and M be a polynomial-time NTM solving L. Then the $\mathrm{P}^{\#\mathrm{P}}$ oracle used in the reduction is simply the function f_M induced by M, and $x \in L$ if and only if $f_M(x) > 0$. This completes part (a).

Part (b) can be proved similarly. Given a polynomial-time probabilistic machine P, we construct a NTM P' that guesses a random bit string (which is polynomial in length) and simulates the computation of P. The oracle used in the reduction is $f_{P'}$, and the oracle machine accepts if and only if the number of accepting computation paths of P' is at least $2/3$ of the total number of computation paths.

Part (c) will be proved in the next lecture.

$\oplus$P denotes the parity of P. Part (d) is quite obvious. The following proposition shows a more interesting property of $\oplus$P, which is also seen in P and BPP. The proof is left as an exercise.

Part (e) follows from the fact that the entire computation tree of an NTM can be traversed deterministically in polynomial space. □

Exercise 1. $\oplus\mathrm{P}^{\oplus\mathrm{P}} = \oplus\mathrm{P}$.

2.4 Complete problems

In this section, we present some #P-complete problems.

Theorem 2. #SAT *is* #P*-complete under* $\leq_m^p$.

Proof sketch. In a previous lecture we proved that SAT is NP-complete. In that proof we took a polynomial-time NTM M and constructed a Boolean formula ϕ_x capturing the computation of M on some input x. It can be verified that the number of assignments satisfying ϕ_x is the same as the number of accepting computation paths of M on input x. □

Theorem 3. *Computing the permanent of an integer matrix is* #P*-hard under* $\leq_o^p$.

Proof sketch. It can be shown that given a Boolean formula ϕ with ℓ occurrences of literals, we can construct in polynomial time an integer matrix A such that

$$perm(A) = 4^\ell \cdot \#(\phi),$$

where $\#(\phi)$ denotes the number of assignments satisfying ϕ. Given this construction, a single oracle call is needed to determine $\#(\phi)$. □

Theorem 4. #PM *is* #P*-complete under* $\leq_o^p$.

Proof sketch. This theorem can be proved by reducing integer matrix permanents to #PM, then applying Theorem 3. The reduction has two steps. In the first step, we get rid of all negative entries in the matrix using modular arithmetic. In the second step, we transform the nonnegative integer matrix into a 0/1-matrix. □

3 Next Time

Next lecture we will prove that #GI $\leq_o^p$ #GI. Then we will discuss the relations between #P and PH. In particular, we will prove that PH $\subseteq$ $P^{\#P[1]}$. We will also see that although it is unlikely that PH can handle exact counting, it can be shown that we can do approximate counting in the second level of PH.

Acknowledgements

In writing the notes for this lecture, I perused the notes by Beth Skubak and Phil Rydzewski for lecture 21 and 22 from the Spring 2010 offering of CS 710 and the notes by Chi Man Liu for lecture 19 from the Spring 2007 offering of CS 810.