CS 710: Complexity Theory 12/1/2011

Lecture 24: Approximate Counting

Instructor: Dieter van Melkebeek Scribe: David Guild and Gautam Prakriya

Last time we introduced counting problems and defined the class #P. The output from a counting problem is not simply "yes" or "no" but rather a count of the number of "yes" instances. Some examples of counting problems are $\#SAT$, which counts the number of satisfying assignments of a boolean formula, and $\#PM$, which counts the number of perfect matchings in a bipartite graph.

This lecture will explore the relationship between counting and decision problems in more detail. Also, we study the relatioinship between #P and PH . We will show that it is possible to approximate #P functions in polynomial time when given oracle access to the second level the polynomial hierarchy.

1 Counting and Decision Problems

We showed last time that NP $\subseteq$ P$^{\#\text{P}[1]}$, so in some sense NP problems are no harder than #P problems. Although we cannot prove it, intuition suggests that this containment is proper and #P is more powerful than NP.

Conjecture 1. $\neg(\#SAT \leq_o^p SAT)$

One interesting observation is that $\#PM$ is $\leq_o^p$-complete for #P, but the corresponding decision problem $PM \in$ P. This lends some weight to our conjecture: if we could reduce $\#PM$ to its decision problem, then we could solve all of #P with a P-oracle and we would end up with NP = P. Of course, it might be the case that $\#SAT \leq_o^p SAT$ but $\neg(\#PM \leq_o^p PM)$. This would still be troublesome, because then PH would collapse to the second level.

2 Reducing Counting to a Decision Problem

The decision problem GI takes two graphs as input and accepts if they are isomorphic; that is, if there is a bijection between their vertices that preserves edges. We write $G_1 \simeq G_2$ if the graphs G_1 and G_2 are isomorphic. The related counting problem $\#GI$ returns the number of isomorphisms between two graphs. Note that $GI \in$ NP but it is not known if it is NP-complete. The following theorem is strong evidence that it is not.

Theorem 1. $\#GI \leq_o^p GI$

Proof. The proof of this theorem will proceed in two steps and make use of another counting problem, $\#GA$. This is the graph automorphism problem: given a graph, how many automorphisms (permutations of vertices that preserve edges) does it have? In this case the corresponding decision problem GA is trivial, because the identity permutation is always an automorphism.

1. $\#GI \leq_o^p \#GA$
2. $\#GA \leq_o^p GI$

For step 1, first observe that

$$\#GI(G_1, G_2) = \begin{cases} \#GA(G_1) & \text{if the graphs are isomorphic} \\ 0 & \text{if not} \end{cases}$$

To see why this is true, consider choosing some special isomorphism $f^* : G_1 \to G_2$. Then we can uniquely express any isomorphism $f_i : G_1 \to G_2$ as the composition of f^* and an automorphism $\pi_i : G_1 \to G_1$, that is, $f_i = f^* \circ \pi_i$. Then f^* defines a bijection between $\#GI(G_1, G_2)$ and $\#GA(G_1)$, and there are exactly as many isomorphisms between G_1 and G_2 as automorphisms of G_1.

As a corollary, two isomorphic graphs have the same number of automorphisms. We will not need to use this fact.

So we need to use $\#GA$ to determine whether the graphs are isomorphic. Note that since graph automorphisms preserve edges, they must also preserve connected components. Consider the disjoint union $H = G_1 \cup G_2$, and assume that the individual graphs are connected.

Because automorphisms preserve connected components, an automorphism of H either maps each graph to itself or maps the graphs to each other. We shall call these *graph-preserving* and *graph-swapping* automorphisms, respectively.

Note that graph-swapping automorphisms of H are also isomorphisms between G_1 and G_2. Then if $G_1 \not\simeq G_2$, all the automorphisms of H must be graph-preserving. Further, we can view the graph-preserving automorphisms of H as pairs of automorphisms of the component graphs. So there are $\#GA(G_1) \cdot \#GA(G_2)$ such graph-preserving automorphisms, and if $G_1 \not\simeq G_2$ then $\#GA(H) = \#GA(G_1) \cdot \#GA(G_2)$.

If $G_1 \simeq G_2$, then we can choose a special isomorphism f^*. This is also a graph-swapping automorphism of H. Using the same argument as before, we can uniquely express the graph-swapping automorphisms π_i as the composition of f^* and a graph-preserving automorphism σ_i. Then f^* defines a bijection (via $\pi_i = f^* \circ \sigma_i$) and there are as many graph-swapping automorphisms as graph-preserving automorphisms.

This gives us a way to determine if two connected graphs are isomorphic with $\#GA$ queries. Specifically,

$$G_1 \not\simeq G_2 \iff \#GA(G_1 \cup G_2) = \big(\#GA(G_1) \cdot \#GA(G_2)\big)$$
$$G_1 \simeq G_2 \iff \#GA(G_1 \cup G_2) = 2\big(\#GA(G_1) \cdot \#GA(G_2)\big)$$

We still need to generalize this to unconnected graphs. This is not too difficult. We can modify the graphs by adding an extra vertex that is connected to all the original vertices, making the graphs connected. In order to preserve the original automorphisms and isomorphisms, we need to make sure that this added vertex can only map to itself or the corresponding extra vertex in the other graph. This can be achieved by "coloring" the vertex.

2.1 Coloring

We can consider coloring to be an extension of the original graph isomorphism and automorphism problems. In other words, the colored version of these problems asks about mappings that preserve edges and colors, where *color* is just an arbitrary labeling. It turns out that these colored problems reduce to their non-colored versions quite easily by means of rigid graphs.

A *rigid graph* is a graph that has only the trivial automorphism. (We leave the existence and construction of such graphs to other lectures.) For a graph of size n, we can color a vertex by attaching a rigid graph of size $n + 1$ to it. Then any automorphism of the new graph must map

that vertex to itself (or another similarly-colored vertex). Note that while we have increased the size of the graph, we have not added any new mappings. If we wish to add additional colors, we can use rigid graphs of size $n+2$, $n+3$, etc. In the worst case the number of vertices will increase to n^3, so this is a $\leq_m^p$-reduction.

The proof of step 2 will make use of the colored GI problem. We need to determine the number of automorphisms of an arbitrary graph G, given oracle queries about colored isomorphisms. The general idea is to make two copies of G, color some vertices in each copy, and then ask if the two are isomorphic. We shall use G_{v_i} to denote the graph G where vertex v has been given the color i, and $G_{v'}$ if v has been given some unique color.

Given a vertex $v \in G$, we define the *orbit* of v to be the set of vertices that v can be mapped to under some automorphism.

$$Orbit(v) = \{w \mid (\exists \pi \in Aut(G))\ \pi(v) = w\}$$

This is actually an equivalence relation among the vertices of G. We can compute these orbits by observing that $w \in Orbit(v) \iff G_{v_i} \simeq G_{w_i}$ and the right side is just a colored graph isomorphism problem.

We can make use of these orbits by observing that for any $v \in G$,

$$\#GA(G) = |Orbit(v)| \cdot \#GA(G_{v'})$$

The argument for this is the same as we used for similar statements earlier. For each $w \in Orbit(v)$, pick some $\pi_w \in Aut(G)$ such that $\pi_w(v) = w$. Note that this defines a bijection between the automorphisms that map v to w and those that map v to itself; if $f_{\pi_w}(\sigma) = \pi_w \circ \sigma$ then

$$f_{\pi_w} : \{\sigma \in Aut(G) \mid \sigma(v) = v\} \xrightarrow{bijection} \{\pi \in Aut(G) \mid \pi(v) = w\}$$

Since

$$Aut(G) = \bigcup_{w \in G} \{\pi \in Aut(G) \mid \pi(v) = w\}$$

this proves the equality above.

Then we can compute $\#GA(G)$ recursively with the following algorithm for computing the automorphisms of a partially colored graph G^*:

- If all vertices in G^* are colored, return 1.
- Otherwise, choose an uncolored vertex v and compute its orbit. Return $|Orbit(v)| \cdot \#GA(G^*_{v'})$.

Note that many of the $|Orbit(v)|$ factors will turn out to be one, because they are orbits in G^*, not the original graph G. In particular, once v has been uniquely colored the other vertices in its orbit will become fixed and thus have only trivial orbits when chosen in a later recursive step.

Another way to think about this is that we are actually computing the size of each orbit equivalence class, and multiplying those together. We could modify the algorithm above by giving a unique color to each vertex in $Orbit(v)$ instead of just v. Proof that this is the same algorithm is left as an exercise.

This proves that $\#GA \leq_o^p$ colored-GI. Combined with our claim that colored-$GI \leq_m^p GI$ via rigid graphs, we get the desired result of $\#GA \leq_o^p GI$. □

3 PH vs. #P

We now shift our attention to the relationship between PH and #P. We will be interested in the following results.

1. $\text{PH} \subseteq \text{P}^{\#\text{P}[1]}$.
2. Approximate counting is in $\text{P}^{\Sigma_2^P}$, i.e, one can approximate a #P function in polynomial time with a Σ_2^P oracle.

It follows from these statements that unless the PH collapses to the second level, exact counting is strictly harder than approximate counting. Therefore, Theorem 1 is strong evidence that GI is not NP-complete.

We now give a brief introduction to universal hash families since they are used to prove both results stated above.

3.1 Universal hash families

Definition 1. $\mathcal{H} \in \{h : \{0,1\}^n \to \{0,1\}^m\}$ *is a Universal hash family if* $(\forall x_1 \neq x_2 \in \{0,1\}^n)(\forall y_1, y_2 \in \{0,1\}^n)$

$$\Pr_{h \in_R \mathcal{H}}[h_1(x_1) = y_1 \wedge h_2(x_2) = y_2] = \frac{1}{2^{2m}}$$

This essentially says that if we pick $h \in \mathcal{H}$ at random and fix x_1 and x_2, then the random variables $h(x_1)$ and $h(x_2)$ are independent. So we can think of universal hash functions as giving us the ability to produce uniform pairwise independent samples.

The following are immediate consequences of the above definition.

- For any $x_1 \neq x_2 \in \{0,1\}^n$ $\Pr_{h \in \mathcal{H}}[h(x_1) = h(x_2)] = \frac{1}{2^m}$.
- For any $x \in \{0,1\}^n$ and $y \in \{0,1\}^m$, $\Pr_{h \in \mathcal{H}}[h(x) = y] = \frac{1}{2^m}$.

Clearly, the set of all functions $\mathcal{H}$ forms a universal family. However this is of no use to us, since, we are interested in universal hash families where the hash functions can be specified with a short string and are efficiently computable. Below are a couple of examples of such families.

Example: $\mathcal{H}_1 = \{Tx + a | T$ is a $m \times n$ Toeplitz matrix and $a \in \{0,1\}^m\}$

A Toeplitz matrix is a matrix in which all the enteries in each diagonal are the same. Each hash function can be specified with $2m + n$ bits ($m + n$ for the matrix and m for a) ⊠

Example: $\mathcal{H}_2 = \{$ The first m bits of $ax + b | a, b \in GF(2^n)\}$ Clearly, each hash function in $\mathcal{H}_2$ can be specified with $2n$ bits.

⊠

Exercise 1. *Show that $\mathcal{H}_1$ and $\mathcal{H}_2$ are universal families of hash functions.*

4 Approximate Counting

Let f be a #P function and M its underlying NP machine. Let S_x be the set of accepting paths of M on input x. We would like to obtain an estimate of $|S_x|$. A result we have already seen of a similar flavor is BPP $\subseteq \Sigma_2^P \cap \Pi_2^P$, where we estimated $Acc(x)$ under the promise that $Acc(x)$ occupies either a large fraction of the universe or a small fraction. We used this fact to argue that a given x is in the BPP language iff there exist a small number of shifts of $Acc(x)$ that cover the universe. In the current setting we don't have this promise however we can use random functions to obtain something similar.

Let the running time of the NP machine be n^c ($S_x \subseteq \{0,1\}^{n^c}$). The intuition behind our approach is that if we pick an m s.t. $2^m \approx |S_x|$ then given a randomly chosen function g from $\{0,1\}^{n^c}$ to $\{0,1\}^m$, it is likely that $f(S_x)$ will cover a significant fraction of $\{0,1\}^m$ without too many collisions. However, this does not help us obtain a Σ_2^P formula - Specifying an arbitrary function may take too many bits. A key observation is that our intution works even with a randomly picked hash function from a universal hash family. As we will see, this will help us obtain the required Σ_2^P formula.

We will now prove a lemma which states that

$$2^m \approx |S_x| \Leftrightarrow (\exists h_1, h_2, \ldots h_t \in \mathcal{H})(\forall z \in \{0,1\}^m)[z \in \cup_{i=1}^t h_i(S_x)]$$

This is not a Σ_2^P formula since the inner predicate requires a $\exists$ quantifier. We will see later that one can obtain a Σ_2^P formula from the formula above. Note that this gives us the Approximate counting algorithm, since, we can run through the polynomially many values m can take to find an m s.t. $2^m \approx |S_x|$ in $\mathrm{P}^{\Sigma_2^P}$.

Lemma 1. $2^m \approx |S_x| \Leftrightarrow (\exists h_1, h_2, \ldots h_t \in \mathcal{H})(\forall z \in \{0,1\}^m)[z \in \cup_{i=1}^t h_i(S_x)]$ *more precisely,*

- *If* $|S_x| \geq 2^{m+1}$ *and* $t \geq m$ *then* $(\exists h_1, h_2, \ldots h_t \in \mathcal{H})(\forall z \in \{0,1\}^m)[z \in \cup_{i=1}^t h_i(S_x)]$
- *If* $|S_x| t < 2^m$ *then* $\neg((\exists h_1, h_2, \ldots h_t \in \mathcal{H})(\forall z \in \{0,1\}^m)[z \in \cup_{i=1}^t h_i(S_x)])$

Proof. Fix $z \in \{0,1\}^m$

As in the BPP $\subseteq \Sigma_2^P \cap \Pi_2^P$ proof, we would like to give an upper bound for

$$\Pr_{h_1,h_2,\ldots h_t \in \mathcal{H}}[\sum_{i=1}^{t} X(z \in h_i(S_x)) = 0].$$

Where $X(A)$ is the indicator random variable for event A. Now,

$$\Pr_{h \in \mathcal{H}}[z \notin h(S_x)] = \Pr_{h \in \mathcal{H}}[\sum_{y \in S_x} X(h(y) = z) = 0]$$

Note that

$$\mathbb{E}[\sum_{y \in S_x} X(h(y) = z)] = \frac{|S_x|}{2^m}$$

By Chebyshev,

$$\Pr[\sum_{y\in S_x} X(h(y)=z)=0] \leq \Pr[|\sum_{y\in S_x} X(h(y)=z)-\mu| \geq \mu] \leq \frac{\sigma^2(\sum_{y\in S_x} X(h(y)=z))}{\mu^2}$$

For the sum to be zero it has to be bounded away from μ by μ.

Since the hash function is picked from a universal family, the indicator random variables under consideration are pair-wise independent. Pair-wise independence implies that the variance is additive.

Therefore,

$$\frac{\sigma^2(\sum_{y\in S_x} X(h(y)=z))}{\mu^2} = \sum_{y\in S_x} \frac{\sigma^2(X(h(y)=2))}{\mu^2} = \frac{|S_X|\frac{1}{2^m}.(1-\frac{1}{2^m})}{\mu^2}$$

$$\Pr[\sum_{y\in S_x} X(h(y)=z)=0] < \frac{1}{\mu}$$

This implies that if m is s.t. $\mu = \frac{|S_x|}{2^m} \geq 2$, then $\Pr_{h\in\mathcal{H}}[z \notin h(S_x)] < \frac{1}{2}$.

Which gives

$$\Pr_{h_1,h_2,\ldots h_t\in\mathcal{H}}[z \in \cup_{i=1}^t h_i(S_x)] < \frac{1}{2^t}$$

This is for a fixed z. By a union bound,

$$\Pr_{h_1,h_2,\ldots h_t\in\mathcal{H}}[\neg((\forall z\in\{0,1\}^m)[z\in\cup_{i=1}^t h_i(S_x))] < \frac{2^m}{2^t}$$

If $|S_x| \geq 2^{m+1}$ and $t \geq m$ then , $\Pr_{h_1,h_2,\ldots h_t\in\mathcal{H}}[\neg((\forall z\in\{0,1\}^m)[z\in\cup_{i=1}^t h_i(S_x))] < 1$. Which tells us that we can pick $h_1, h_2, \ldots h_t \in \mathcal{H}$ s.t. $(\forall z\in\{0,1\}^m)[z\in\cup_{i=1}^t h_i(S_x)]$. i.e, the Σ_3^P formula is satisfied.

On the other hand, if $|S_x|t < 2^m$ then no choice of t hash functions can cover $\{0,1\}^m$.

□

Next time

In the next lecture we describe how one can get a Σ_2^P formula. We will also make precise what we mean by $\{0,1\}^m \approx |S_x|$. Besides this we will also begin to prove that PH $\in$ P$^{\#\mathrm{P}}$.

Acknowledgements

In writing the notes for this lecture, we perused the notes by Chi Man Liu and Jeff Kinne for lectures 19,20 from the Spring 2007 offering of CS 810, and the notes by Phil Rydzewski and Mushfeq Khan for lectures 22,23 from the Spring 2010 offering of CS 710.