CS 810: Complexity Theory	3/2/2007

Lecture 20: Alternation vs. Counting

Instructor: Dieter van Melkebeek — Scribe: Jeff Kinne

We introduced counting complexity classes in the previous lecture and gave some basic properties, including the relation between counting and decision classes. In this lecture we give results relating counting to the polynomial hierarchy. The first result shows that any #P function can be approximated in the second level of the polynomial hierarchy, giving evidence that approximate counting is not much more difficult than deciding. The second result gives evidence that exact counting is more difficult by showing that the entire polynomial hierarchy can be decided with a single query to a #P function.

The proofs of both results make use of families of universal hash functions. These are small function families that behave in certain respects as if they were random, allowing efficient random sampling. We first introduce universal hash functions, and then prove the two main results.

1 Universal Families of Hash Functions

A universal family of hash functions is a collection of functions. We wish the set of functions to be of small size while still behaving similarly to the set of all functions when we pick a member at random. This is made possible by choosing the appropriate notion of "behaving similarly".

Definition 1. *$\mathcal{H} = \{h : \{0,1\}^n \to \{0,1\}^m\}$ is a* universal family of hash functions *if the following holds. For all $x_1 \neq x_2 \in \{0,1\}^n$ and $y_1, y_2 \in \{0,1\}^m$,*

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

where h is chosen uniformly at random from the functions in $\mathcal{H}$.

Notice that the probability is the same as if we had picked h from all possible functions. If we fix x_1 and x_2 and pick $h \in \mathcal{H}$ at random, 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 definition above can be generalized to define k-universal hash functions that produce k-wise independent samples.

We will make use of a few 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 $y \in \{0,1\}^m$, $\Pr_{h\in\mathcal{H}}[h(x) = y] = \frac{1}{2^m}$.

To be able to efficiently sample from $\mathcal{H}$, we would like the family to be small. There are a number of ways to achieve this; we give two.

Example: Let $\mathcal{H} = \{h(x)|h(x) = Tx + v$ where T is some $m \times n$ Toeplitz matrix over GF(2) and v is some $m \times 1$ vector over GF(2)$\}$. A Toeplitz matrix is one where entries along each diagonal are all the same. So it takes $(m + n - 1) + m = 2m + n - 1$ bits to specify an element of $\mathcal{H}$, whereas the specification of a random function from all possible functions requires $m2^n$ bits to specify the truth table. If the correctness of a randomized algorithm only relies on pairwise independence, this universal family of hash functions can be used as a $O(\log r)$ length seed pseudorandom generator to derandomize the algorithm.

We leave it as an exercise to prove that $\mathcal{H}$ satisfies Definition 1. $\boxtimes$

Example: Another example is the set of affine functions over a finite field. We leave it as an exercise to prove that $\mathcal{H} = \{h(x) = (ax+b \pmod{2^m}) | a, b \in GF(2^m)\}$ is a universal family of hash functions from $\{0,1\}^n$ to $\{0,1\}^m$ for all $m \le n$, where by $\pmod{2^m}$ we mean that we are working over the field $GF(2^m)$ (correctness follows from the fact that $GF(2^m)$ is a field). $\boxtimes$

1.1 Applications

Before delving into the main results of this lecture, we give some intuition of how universal families of hash functions will be useful. Let $S \subseteq \{0,1\}^n$. Then intuitively, we expect that if we pick $2^m \approx |S|$, a randomly chosen hash function from n bits to m bits will map S to $\{0,1\}^m$ with few collisions. That is, for $h \in_U \mathcal{H}$, with high probability $h(S) \approx \{0,1\}^m$. In the first application, we wish to determine the size of S where S is an NP witness set. We will make use of the above intuition to derive a Σ_2^p predicate that allows us to approximately determine the size of S. In a later application, we wish to use randomness to reduce a satisfiable formula to another one that is uniquely satisfiable (satisfiable by only one assignment). We will use the above intuition to show that by choosing m appropriately and picking h at random, there will with high probability be a unique satisfying assignment that hashes to 0^m.

2 Approximate Counting

In this section we prove that any #P function can be approximated to within a polynomial factor with an oracle for the second level of the polynomial hierarchy.

Theorem 1. *For any $f \in$ #P and for all $a > 0$, there is a function g computable in polynomial time with oracle access to a Σ_2^p language such that for all x,*

$$|f(x) - g(x)| \le \frac{f(x)}{|x|^a}.$$

Notice that the goal in proving Theorem 1 is similar to the goal in showing that BPP $\subseteq \Sigma_2^p$. There, we needed to approximate the function counting the number of accepting random strings to within a constant factor. Now, we wish to approximate a #P function to within any polynomial factor.

Proof. Let the underlying NTM for f run in time n^c, and let us view it as a polynomial-time verifier. The NTM takes a certificate y of length n^c along with input x. We wish to determine the size of the set S_x of certificates causing the NTM to accept. Consider applying a randomly chosen hash function to the set of possible certificates, with $h : \{0,1\}^{n^c} \to \{0,1\}^m$ for some value m. If 2^m is large compared to S_x, we expect that $h(S_x)$ covers only a small portion of the range. If 2^m is small compared to S_x, we expect $h(S_x)$ to cover a large portion of the range. If we take a small collection of hash functions, they will collectively cover all of $\{0,1\}^m$ provided S_x is roughly the same size as 2^m. We will construct a Σ_2^p predicate that can be queried to determine if a small set of hash functions covers the range. By querying this language for increasing values of m until we get a negative answer, we get an estimate for $|S_x|$.

First attempt. We mimic the proof that BPP $\subseteq \Sigma_2^p$ to try to determine if $|S_x| \approx 2^m$. There, we showed that if x is accepted by a BPP machine, then there exists a small set of shift vectors $\sigma_1, ..., \sigma_t$ so that shifting the witness set covers the entire set of random strings. Here, we use hash functions rather than shift vectors, and want to see if hashing the witness set by a small set of hash functions covers all of $\{0,1\}^m$.

We first bound the probability that a fixed $z \in \{0,1\}^m$ is not hit by a randomly chosen h. We wish to upper bound $\Pr_{h\in\mathcal{H}}[(\sum_{y\in S_x} \chi_{h(y)=z}) = 0]$. We would like to use the fact that choosing h at random results in pairwise independent samples to use Chebyshev's inequality to bound this probability. To do this, we need to compute the expected value of the sum. By linearity of expectation and the properties of universal hash functions,

$$E_{h\in\mathcal{H}}[\sum_{y\in S_x} \chi_{h(y)=z}] = \sum_{y\in S_x} E_{h\in\mathcal{H}}[\chi_{h(y)=z}] = \sum_{y\in S_x} \frac{1}{2^m} = \frac{|S_x|}{2^m}.$$

Denote this value as E_m. By Chebyshev's inequality and pairwise independence,

$$\begin{aligned} \Pr_{h\in\mathcal{H}}[(\textstyle\sum_{y\in S_x} \chi_{h(y)=z}) = 0] &\le \Pr_{h\in\mathcal{H}}[|\textstyle\sum_{y\in S_x} \chi_{h(y)=z} - E_m| \ge E_m] \\ &\le \frac{\sigma^2(\sum_{y\in S_x} \chi_{h(y)=z})}{E_m^2} = \frac{|S_x|\frac{1}{2^m}(1-\frac{1}{2^m})}{E_m^2} < \frac{1}{E_m}. \end{aligned}$$

Then if $|S_x| \ge 2^{m+1}$, the probability that a fixed z is not hit by a randomly chosen h is at most $1/2$. By picking t hash functions independently, this probability is at most $1/2^t$. A union bound over all z shows that if $|S_x| \ge 2^{m+1}$ then $\{0,1\}^m$ is covered by t randomly chosen hash functions with probability at least $1 - \frac{2^m}{2^t}$. On the other side, each hash function covers at most $|S_x|$ elements of the range, so if $t \cdot |S_x| < 2^m$ the probability of covering the range is 0. To sum up,

$$\frac{|S_x|}{2^m} \ge 2 \text{ and } t \ge m \Rightarrow (\exists h_1, ..., h_t)(\forall z \in \{0,1\}^m)[z \in \bigcup_{i=1}^{t} h_i(S_x)] \tag{1}$$

$$\frac{|S_x|}{2^m} < \frac{1}{t} \Rightarrow \neg(\exists h_1, ..., h_t)(\forall z \in \{0,1\}^m)[z \in \bigcup_{i=1}^{t} h_i(S_x)]. \tag{2}$$

We can encode the RHS of the above as a language which we can query to determine if $|S_x| \approx 2^m$ or not. This can be used to get the approximation factor that we desired, but unfortunately evaluating the inner predicate ($z \in \bigcup_{i=1}^t h_i(S_x)$) requires an existential quantifier to guess a witness y that is mapped to z by some h_i - meaning we would need a Σ_3^p language.

Second attempt. With a bit of work, we can reduce the complexity of the oracle from Σ_3^p to Σ_2^p. As the inner predicate needs an existential quantifier, this would be achieved if we could swap the order of the first two quantifiers in (1) and (2). Notice that this is not a problem for (1), but doesn't work for (2). As for (2), under the stronger assumption that $t \cdot |S_x| \le 2^{m-1}$, we have that

$$(\forall h_1, ..., h_t) \Pr_{z\in\{0,1\}^m}[z \in \bigcup_{i=1}^{t} h_i(S_x)] \le \frac{1}{2}.$$

Thus, if we pick ℓ z's at random (we abbreviate this as $Z = z_1, ..., z_\ell$),

$$(\forall h_1, ..., h_t) \Pr_Z[Z \subseteq \bigcup_{i=1}^{t} h_i(S_x)] \leq \frac{1}{2^\ell}.$$

So if $2^\ell >$ # choices for $h_1, ..., h_t$,

$$\Pr_Z[(\exists h_1, ...h_t)[Z \subseteq \bigcup_{i=1}^{t} h_i(S_x)]] < 1. \tag{3}$$

We conclude that

$$\begin{array}{rcl} \frac{|S_x|}{2^m} \geq 2 \text{ and } t \geq m & \Rightarrow & (\forall Z)(\exists h_1, .., h_t)[Z \subseteq \bigcup_{i=1}^{t} h_i(S_x)], \\ \frac{|S_x|}{2^m} \leq \frac{1}{2t} & \Rightarrow & \neg(\forall Z)(\exists h_1, ..., h_t)[Z \subseteq \bigcup_{i=1}^{t} h_i(S_x)]. \end{array} \tag{4}$$

We now set the parameters and verify that the RHS is a Π_2^p predicate. We set $t = m$ and need to set ℓ so that $2^\ell >$ # choices for $h_1, ..., h_t$. Notice that the running time of the predicate has a factor of ℓ to guess $Z = z_1, ..., z_\ell$, so we also need ℓ to be polynomial. This is where it is critical that we are drawing the h_i from a universal family of hash functions. The examples we gave earlier show that h_i can be specified with $O(n^c + m)$ bits, so setting $\ell = \Theta(m \cdot (n^c + m))$ is good enough to ensure that 2^ℓ is large enough. The inner existential quantifier is now side-by-side with the existential quantifier needed to evaluate the inner predicate, and we conclude that the RHS is a Π_2^p predicate. Finally, the Σ_2^p language that is the complement of the above is equivalent when used as an oracle.

We now see how to use an oracle to the Σ_2 language to approximate $|S_x|$. As mentioned earlier, we query the predicate for each value of $m = 1, 2, ..., n^c$ and determine the first value m^* where the answer to the predicate is negative. By (4), we know that for $m \leq (\log |S_x|) - 1$ the predicate is answered positively; and for $m \geq (\log |S_x| + \log \log |S_x|) + O(1)$ the predicate is answered negatively. Then $(\log |S_x|) - 1 \leq m^* \leq (\log |S_x| + \log \log |S_x|) \cdot O(1)$, which can be rewritten as

$$|S_x| \leq 2^{m^*+1} \leq O(|S_x| \log |S_x|).$$

As $\log |S_x| \leq n^c$, we have an approximation for $|S_x|$ that is within a fixed polynomial factor and is computable in $\mathrm{P}^{\Sigma_2^p}$.

But we would like to be within any polynomial factor, in particular within $1/n^a$ for some constant a. We obtain this by applying the above procedure on a modified predicate. If $f \in \#\mathrm{P}$ is the original function we are trying to approximate, we apply the above algorithm on the function $f' = f^{n^d}$ for a constant d we choose later. By the closure properties of #P, f' is a #P function whenever f is. By the above, our approximation for f' gives

$$f'(x) < 2^{m^*+1} < O(f'(x) \log(f'(x))).$$

Taking a $1/n^d$ power and rearranging, this becomes

$$f(x) \leq 2^{(m^*+1)/n^d} \leq f(x)(O(n^d \log f(x)))^{1/n^d} \leq f(x)(O(n^{d+c}))^{1/n^d}.$$

We have an approximation for $f(x)$ with relative error $< (O(n^{d+c}))^{1/n^d} = 2^{(O(1)+(d+c)\log n)/n^d}$. By looking at the Taylor expansion of this value, we see that it is $1 + \Theta(\frac{\log n}{n^d})$. We can set d large enough so the relative error is at most $1/n^a$. $\square$

A few notes about the above proof.

- If the condition for (3) is strengthened to $2^\ell > 2 \cdot \#$ choices for $h_1, ..., h_t$, then we have

$$\frac{|S_x|}{2^m} \geq 2 \text{ and } t \geq m \Rightarrow \Pr_Z[(\exists h_1, ..., h_t)[Z \subseteq \bigcup_{i=1}^{t} h_i(S_x)]] = 1,$$

$$\frac{|S_x|}{2^m} \leq \frac{1}{2t} \Rightarrow \Pr_Z[(\exists h_1, ..., h_t)[Z \subseteq \bigcup_{i=1}^{t} h_i(S_x)]] \leq \frac{1}{2}.$$

  Plugging this into the argument shows that the approximation can be "computed in $\mathrm{RP}^{\mathrm{NP}}$" in the following sense: there is a randomized machine with oracle access to an NP language which computes an approximation to $f(x)$ where the estimate is never smaller than the true value of $f(x)$ and is never larger than $(1 + \frac{1}{|x|^a}) \cdot f(x)$.

- The language that is used as an oracle is a Π_2^p predicate and remains so even if checking whether $y \in S_x$ requires nondeterminism.

Both of these will play a role in an application of approximate counting (to AM games) later in the semester.

3 Exact Counting

For our second main result, we show the following.

Theorem 2. *Any language in the polynomial hierarchy can be decided in polynomial time with a single oracle query to a* $\#\mathrm{P}$ *function, namely* $\mathrm{PH} \subseteq \mathrm{P}^{\#\mathrm{P}[1]}$.

We prove this theorem in three parts.

1. We first show $\mathrm{NP} \subseteq \mathrm{RP}^{\text{UNIQUE-SAT}}$. UNIQUE-SAT is the promise problem defined on formulae that have exactly either one or zero satisfying assignments, with the positive instance being uniquely satisfiable formulas. The UNIQUE-SAT oracle is guaranteed to give the correct answer on such formulae, and can act arbitrarily on others. In fact, the RP algorithm given is correct even if the oracle gives inconsistent answers on queries that are outside of the promise.

2. We next use the first part to show that $\mathrm{PH} \subseteq \mathrm{BPP}^{\oplus\mathrm{P}}$.

3. We finish the proof by showing that $\mathrm{BPP}^{\oplus\mathrm{P}} \subseteq \mathrm{P}^{\#\mathrm{P}[1]}$.

3.1 Solving NP with randomness and UNIQUE-SAT oracle

We again use hash functions for this theorem. Consider the NTM for SAT running in time n^c. We look at the set of all possible assignments for the formula given as input and consider applying a hash function on these. The idea is to try to choose the range of the hash function about the same size as S_x. If we can achieve this, we show that a randomly chosen hash function with high probability maps a unique satisfying assignment to 0^m. This gives us a potential UNIQUE-SAT query for the oracle.

We first bound the probability that a randomly chosen hash function maps a unique satisfying assignment to 0^m. Let S_x be the set of satisfying assignments, and let $\mathcal{H}_m$ be a universal family of hash functions from $\{0,1\}^{n^c}$ to $\{0,1\}^m$. The probability that h maps a unique satisfying assignment to 0^m is given by

$$\Pr_{h\in\mathcal{H}_m}[\sum_{y\in S_x} \chi_{h(y)=0^m} = 1] = \Pr_{h\in\mathcal{H}_m}[\sum_{y\in S_x} \chi_{h(y)=0^m} \geq 1] - \Pr_{h\in\mathcal{H}_m}[\sum_{y\in S_x} \chi_{h(y)=0^m} \geq 2].$$

For the first term we have

$$\Pr_{h\in\mathcal{H}_m}[\sum_{y\in S_x} \chi_{h(y)=0^m} \geq 1] \geq \frac{|S_x|}{2^m} - \binom{|S_x|}{2}\frac{1}{2^{2m}}$$

by considering the first two terms of the inclusion-exclusion principle expansion of the probability and using pairwise independence of the hash functions. For the second term we have

$$\Pr_{h\in\mathcal{H}_m}[\sum_{y\in S_x} \chi_{h(y)=0^m} \geq 2] \leq \binom{|S_x|}{2}\frac{1}{2^{2m}}$$

by union bound and pairwise independence. Putting these two together and using the fact that $\binom{|S_x|}{2} \leq \frac{|S_x|^2}{2}$ gives us

$$\Pr_{h\in\mathcal{H}_m}[\sum_{y\in S_x} \chi_{h(y)=0^m} = 1] \geq \frac{|S_x|}{2^m}\left(1 - \frac{|S_x|}{2^m}\right)$$

which is equal to $X(1-X)$ for $X = \frac{|S_x|}{2^m}$. This value is symmetric around $X = 1/2$ and achieves its maximum of $1/4$ here. As $\frac{|S_x|}{2^m}$ increases by 2 for each value of m, there is some choice of m causing the probability to be in the range $[1/3, 2/3]$ providing $S_x \neq \emptyset$. For that value of m, the probability that a randomly chosen hash function maps a unique satisfying assignment to 0^m is at least $2/9$.

Given the above analysis, the following is the $\mathrm{RP}^{\text{UNIQUE-SAT}}$ algorithm for SAT.

INPUT: formula ϕ.
(2) **foreach** $m = 0, 1, 2, ..., n^c$
(3) Pick $h \in \mathcal{H}_m$ at random.
(4) Convert the following into a SAT query and ask the UNIQUE-SAT oracle: is there an assignment y that both satisfies ϕ and $h(y) = 0^m$?
(5) **if** Oracle says yes **then** Use self-reducibility to find y, and verify $\phi(y) = 1$. If yes, then output "Yes".
(6) Output "No".

Because we are choosing h from a universal family of hash functions, choosing the hash function can be done in polynomial time. The rest of the algorithm also runs in polynomial time. Suppose ϕ is satisfiable. Then for at least one choice of m, with probability at least $2/9$ line (4) corresponds to a uniquely satisfiable formula. Notice that the formula remains uniquely satisfiable when using self-reducibility, so in this case, the algorithm correctly outputs "Yes". If ϕ is not satisfiable, the algorithm always outputs "No". The probability of success on satisfiable formulas can be amplified by repeating the above, so the algorithm is $\mathrm{RP}^{\text{UNIQUE-SAT}}$.

3.2 Randomized reduction of PH to $\oplus$P

Now we can use the fact that NP $\subseteq$ RP$^{\text{UNIQUE-SAT}}$. Because UNIQUE-SAT only cares about formulas with exactly 0 or 1 satisfying assignments, $\oplus$SAT solves UNIQUE-SAT on formulas within the promise. Then the previous section shows that NP $\subseteq$ BPP$^{\oplus\text{P}}$.

Given this, we use the fact that a problem from the first homework relativizes. Namely, NP $\subseteq$ BPP $\Rightarrow$ PH $\subseteq$ BPP relativizes. So NP$^{\oplus\text{P}}$ $\subseteq$ BPP$^{\oplus\text{P}}$ $\Rightarrow$ PH$^{\oplus\text{P}}$ $\subseteq$ BPP$^{\oplus\text{P}}$. We want to show the hypothesis of this. To get this, we notice that NP $\subseteq$ BPP$^{\oplus\text{P}}$ relativizes as well, giving NP$^{\oplus\text{P}}$ $\subseteq$ (BPP$^{\oplus\text{P}}$)$^{\oplus\text{P}}$.

In general, giving an additional oracle O_2 to an oracle machine with access to O_1 can be solved by the base machine by giving it access to oracles O_1 and $O_1^{O_2}$. For the case of (BPP$^{\oplus\text{P}}$)$^{\oplus\text{P}}$, we get that a BPP machine requires access to a ($\oplus$P)$^{\oplus\text{P}}$ oracle in addition to a $\oplus$P oracle. These can be combined into a single ($\oplus$P)$^{\oplus\text{P}}$ oracle using the completeness of $\oplus$SAT, and using the fact that ($\oplus$P)$^{\oplus\text{P}}$ = $\oplus$P, we get NP$^{\oplus\text{P}}$ $\subseteq$ BPP$^{(\oplus\text{P}^{\oplus\text{P}})}$ = BPP$^{\oplus\text{P}}$. Having achieved NP$^{\oplus\text{P}}$ $\subseteq$ BPP$^{\oplus\text{P}}$, we conclude that PH $\subseteq$ PH$^{\oplus\text{P}}$ $\subseteq$ BPP$^{\oplus\text{P}}$.

We point out that the proof given here of this result is simpler than the original proof. This is a demonstration of the power of relativization.

3.3 Deterministic reduction of BPP$^{\oplus\text{P}}$ to #P

Given a language L in BPP$^{\oplus\text{P}}$, we wish to determine if $x \in L$ by making a single query to a #P function. We first show that we can separate out the randomized portion of L from the counting portion. This will be useful in performing the reduction.

Definition 2. *Let $\mathcal{C}$ be a complexity class. We define the* BP *operator to give a new complexity class of languages, where* $\text{BP} \cdot \mathcal{C} = \{L | (\exists c > 0)(\exists L' \in \mathcal{C})$

$$x \in L \Rightarrow \Pr_{y \in |x|^c}[\langle x, y\rangle \in L'] > 2/3,$$
$$x \notin L \Rightarrow \Pr_{y \in |x|^c}[\langle x, y\rangle \in L'] < 1/3 \quad \}$$

Notice that BP $\cdot$ P = BPP, so this is a reasonable definition of the BP operator. The alternative characterization of BPP$^{\oplus\text{P}}$ we use is given by the following.

Claim 1. BPP$^{\oplus\text{P}}$ = BP $\cdot$ $\oplus$P.

Proof. We show that for any complexity class $\mathcal{C}$, BPP$^{\mathcal{C}}$ = BP $\cdot$ P$^{\mathcal{C}}$. The result then follows by using $\oplus$P as $\mathcal{C}$ and the fact that P$^{\oplus\text{P}}$ = $\oplus$P.

Consider a BPP machine that can ask queries to a $\mathcal{C}$ language. Without changing the computation, the machine can guess its random bits at the beginning the computation before proceeding. What is left is a P$^{\mathcal{C}}$ predicate. Now consider a BP $\cdot$ P$^{\mathcal{C}}$ language. The BPP$^{\mathcal{C}}$ machine computing the same language simply generates enough random bits to be the second input of the P$^{\mathcal{C}}$ machine and then simulates that machine. ∎

Now consider a language $L \in$ BP $\cdot$ $\oplus$P which we hope to solve in P$^{\#\text{P}[1]}$. By definition, there is some $f \in$ #P such that

$$x \in L \Rightarrow \Pr_{|y|=n^c}[f(x, y) \equiv 1 \pmod 2] > 2/3$$
$$x \notin L \Rightarrow \Pr_{|y|=n^c}[f(x, y) \equiv 1 \pmod 2] < 1/3$$

We would like to create a #P function which sums f over all y such that we can detect the gap in probability. The following claim is the main ingredient to make this happen.

Claim 2. *Let $f \in$ #P. Then there is a function $g \in$ #P such that*

$$\begin{array}{l} f(z) \equiv 1 \pmod 2 \Rightarrow g(z, 0^M) \equiv -1 \pmod{2^M} \\ f(z) \equiv 0 \pmod 2 \Rightarrow g(z, 0^M) \equiv 0 \pmod{2^M} \end{array}$$

where $M = 2^m$ is some power of 2.

Before proving this claim, we see how it allows us to complete the construction. Notice that if $f(z) \equiv 1 \pmod 2$, then the last M bits of $-g(z, 0^M)$ are 000...001; if $f(z) \equiv 0 \pmod 2$, the last M bits are 000...000. We would like to sum up $-g(z)$ for all $z = (x, y)$ in such a way that the results do not "spill over" past the last M bits. Because we must sum over 2^{n^c} many y, we need to pick $M > n^c$ to ensure there is no spill over. Namely notice that if $M > n^c$, then

$$\begin{array}{l} x \in L \Rightarrow \sum_{|y|=n^c} -g(x, y, 0^M) \pmod{2^M} > \frac{2}{3}2^{|x|^c} \\ x \notin L \Rightarrow \sum_{|y|=n^c} -g(x, y, 0^M) \pmod{2^M} < \frac{1}{3}2^{|x|^c}. \end{array}$$

In fact, by picking $M > n^c$ we ensure there is a single bit of the sum that we can check to distinguish between the two cases. Thus, $g'(x) = \sum_{|y|=n^c} g(x, y, 0^M)$ is the #P function that we query once to determine if $x \in L$ or not. That is, we compute $g'(x)$, and check if $-g'(x) \pmod{2^M}$ is $> \frac{2}{3}2^{|x|^c}$ or not. Notice that g' is in fact a #P function because a machine can branch on all y and then compute $g(x, y, 0^M)$ on each branch.

All that remains is the proof of the claim.

Proof of Claim 2. The base of the construction is the function $h(z) = 4z^3 + 3z^4$ which has the nice property that

$$\begin{array}{l} z \equiv -1 \pmod{2^M} \Rightarrow h(z) \equiv -1 \pmod{2^{2M}} \\ z \equiv 0 \pmod{2^M} \Rightarrow h(z) \equiv 0 \pmod{2^{2M}} \end{array}$$

Given this property of h, $g(z)$ is the result of applying h recursively $m = \log M$ times to $f(z)$. The claimed property of $g(z)$ follows from the property of h. Consider the running time of the underlying NTM machine whose number of accepting paths is g. Each application of h increases the running time by a constant factor, so overall g is a factor $2^{O(m)} = M^{O(1)}$ slower than f - thus $g \in$ #P.

All that remains is to prove the claimed property of h. Suppose $z \equiv b \pmod{2^M}$, that is $z = q2^M + b$ for some integer q. Then if we look at the expansion of $h(z)$ and remove all terms with a factor of 2^{2M} or higher power of 2^M, we get $h(z) \equiv 4(3b^2q2^M + b^3) + 3(4b^3q2^M + b^4) \pmod{2^{2M}}$. For $b = 0$, this gives us $0 \pmod{2^{2M}}$, and for $b = -1$, $-1 \pmod{2^{2M}}$. □

4 Next time

This lecture concludes the first part of the course that has focused on the traditional setting of complexity theory - namely efficiently computing relations within various computational models. We focus in the remainder of the course on alternative goals within complexity theory. Many of the lectures will take a brief look at an area within complexity theory - each of which could be studied within an entire course. These lectures will often be something like surveys of the area. In fact, we have one more lecture where we are interested in the efficiency of computing relations: next lecture we discuss the power of quantum computing.

Appendix

A Alternate proof of approximate counting

Here we given an alternate proof of Theorem 1 that uses the notion of isolation.

Proof. Let the underlying NTM for f run in time n^c, and let us view it as a polynomial-time verifier. The NTM takes a certificate y of length n^c along with input x. We wish to determine the size of the set S_x of certificates causing the NTM to accept. Consider applying a randomly chosen hash function to the set of possible certificates, with $h : \{0,1\}^{n^c} \to \{0,1\}^m$ for some value m. If 2^m is small compared to S_x, we expect that there are many collisions between members of S_x. If 2^m is large compared to S_x, we expect few collisions. If we are able to determine the relative number of collisions for each value of $m = 1, 2, ..., n^c$, we can come up with an estimate for $|S_x|$.

These ideas are formalized by using the concept of isolation. Let $\mathcal{H}$ be a universal family of hash functions from $\{0,1\}^{n^c}$ to $\{0,1\}^m$. $y \in S_x$ is *isolated* by $h \in \mathcal{H}$ if for all $y' \in S_x$ not equal to y, $h(y) \neq h(y')$. If S_x is small compared to 2^m, then a large portion of S_x should intuitively be isolated by a randomly chosen h. In this case, only a small number of hash functions should be required to guarantee that each $y \in S_x$ is isolated by at least one of them. On the other hand, if S_x is large compared to 2^m, we will show that no small set of hash functions can isolate each $y \in S_x$.

We now quantify these ideas. We first bound the probability that a fixed $y \in S_x$ is not isolated by a random h.

$$\begin{aligned}\Pr_{h\in\mathcal{H}}[y \text{ not isolated by } h] &\overset{(a)}{=} \Pr_{h\in\mathcal{H}}[\bigvee_{y'\in S_x, y'\neq y} h(y') = h(y)] \\ &\overset{(b)}{\leq} \sum_{y'\neq y\in S_x} \Pr_{h\in\mathcal{H}}[h(y') = h(y)] \overset{(c)}{=} \sum_{y'\neq y\in S_x} \frac{1}{2^m} \overset{(d)}{<} \frac{|S_x|}{2^m}.\end{aligned} \tag{5}$$

(a) is by definition of isolation; (b) is by union bound; (c) is because h is chosen at random from a universal family of hash functions; (d) is summing over all $y' \neq y \in S_x$.

Now consider the probability that for a random choice of t hash functions, $y \in S_x$ is not isolated by any of them. Because the events (y not isolated by h_i) for a fixed y are independent for independently chosen h_i, we have

$$\Pr_{h_1,...,h_t\in\mathcal{H}}[y \text{ not isolated by any of } h_1, ..., h_t] = \left(\Pr_{h\in\mathcal{H}}[y \text{ not isolated by } h]\right)^t < \left(\frac{S_x}{2^m}\right)^t. \tag{6}$$

Now consider the probability that there is at least one $y \in S_x$ not isolated by any of $h_1, ..., h_t$. A union bound gives

$$\begin{aligned}\Pr_{h_1,...,h_t\in\mathcal{H}}[S_x \text{ not isolated by } h_1, ..., h_t] &\leq \sum_{y\in S_x} \Pr_{h_1,...,h_t\in\mathcal{H}}[y \text{ not isolated by } h_1, ..., h_t] \\ &< |S_x|\left(\frac{|S_x|}{2^m}\right)^t = \frac{|S_x|^{t+1}}{2^{mt}}\end{aligned}$$

meaning the probability is less than 1 when $2^m > |S_x|^{(t+1)/t}$. This gives us a method for testing whether 2^m is roughly at least as large as S_x. Namely, for all large enough m, we know there are a choice of $h_1, ..., h_t$ isolating all of S_x. We also would like a method for testing whether 2^m is roughly at most as large as S_x. Notice that each h_i can isolate at most 2^m elements of S_x, so $h_1, ..., h_t$ can isolate at most $t2^m$. Then if $t2^m < |S_x|$, there can be no $h_1, ..., h_t$ isolating all of S_x.

Now let $t = m$ for simplicity. By the above discussion, for all $m = 1, 2, ..., \log(|S_x|) - \log\log(|S_x|)$ there can be no set of hash functions $h_1, ..., h_m$ isolating all of S_x; while for all $m = 1 + \log(|S_x|), 2 + \log(|S_x|), ..., n^c$ there do exist $h_1, ..., h_m$ isolating all of S_x. This gives us a method to estimate the size of S_x: test the predicate

$$(\exists h_1, ...h_m \in \mathcal{H})(\forall y \in S_x)[\vee_{i=1}^m (h_i \text{ isolates } y)] \tag{7}$$

for each value $1, 2, ..., n^c$ and determine the first value m^* for which the predicate evaluates to true.

Claim 3. *(7) is a Σ_2^p predicate.*

We finish the analysis given this claim, then prove the claim. From the discussion above, we know $\log|S_x| - \log\log|S_x| < m^* < 1 + \log|S_x|$ which can be rewritten as

$$\frac{|S_x|}{2\log|S_x|} < 2^{m^*-1} < |S_x|. \tag{8}$$

As $\log|S_x| \le n^c$, we have an approximation for $|S_x|$ that is within a fixed polynomial factor and is computable in $\mathrm{P}^{\Sigma_2^p}$. We can then use the same method as given in section 2 to make the approximation ratio any polynomial.

All that remains is to verify that (7) is in Σ_2^p. This is the point where we use the fact that we are choosing the h_i from a universal family of hash functions rather than at random. Because we are choosing from $\mathcal{H}$, the initial existential guesses are polynomial in size. We claim that the remaining predicate $(\forall y \in S_x)[\vee_{i=1}^m h_i \text{ isolates } y]$ is a coNP predicate. This is realized with the predicate

$$(\forall y \in \{0,1\}^{n^c})(\exists i \in \{1, ..., m\})(\forall y' \in \{0,1\}^{n^c})[y \in S_x \wedge y' \in S_x \Rightarrow h_i(y) \neq h_i(y')].$$

Testing $y \in S_x$ is done in polynomial time by evaluating the NTM when given y as a certificate, and the existential phase can be pushed inside since it is of polynomial size. Hence, this is a Σ_2^p predicate.

We have shown that we can approximate $f(x)$ deterministically using a Σ_2^p oracle. In fact, this can be done using a randomized algorithm with a NP oracle. Consider (7), and let us pick the hash functions at random. For large enough values of m, most hash functions satisfy the inside coNP predicate, while for small enough values of m no set of hash functions can satisfy the predicate. Then the m^* derived by randomly selecting hash functions and querying the inside coNP predicate with high probability still satisfies (8). Notice that the estimate for $|S_x|$ derived errors only in one direction - no choice of random functions can satisfy the inside coNP predicate of (7) for small values of m. □