CS 710: Complexity Theory 10/20/2011

Lecture 13: Randomness

Instructor: Dieter van Melkebeek Scribe: Brian Nixon

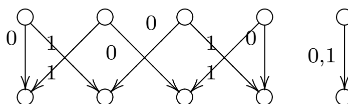
In this lecture we will wrap up our discussion of the relationship between NC^1 and branching programs by completing the proof we left unfinished last lecture. Moving forward, we begin a discussion on the power of randomness on the cost of solving problems by formally defining what we mean by randomness and noting some simple implications and known results. We will continue on the topic of randomness for the next few lectures.

1 Rest of proof from last lecture

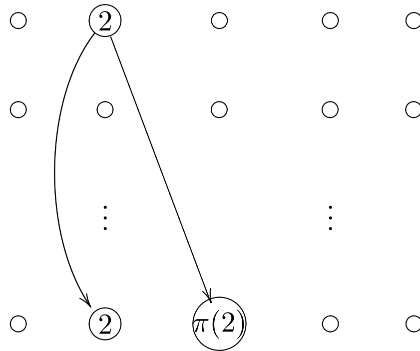
Theorem 1. *The following are equivalent:*

1. $f \in \mathrm{NC}^1$
2. f *has* poly-*size formulas*
3. f *has* log-*depth formulas*
4. f *has* poly-*size branching programs of constant width.*

Proof. Last lecture we proved the implications (1) $\Rightarrow$ (2), (2) $\Rightarrow$ (3), and started (3) $\Rightarrow$ (4). Let us present a compete proof of that claim here. Also, recall we noted that we can, in fact, prove the stronger claim that f has a poly-size oblivious permutation branching program (denoted pbp) with width 5. In a pbp, any two consecutive levels of the program represent a permutation when considering the arrows representing transitions of a single label as the figure shows. Different labels might induce different permutations.



Thus, no matter the input the overall effect is of a permutation as each level will be a permutation. f will yield a stricter program, a π-pbp with $\pi \neq e$ with e as the identity permutation. This means the permutation realized by the program is π if the input is accepted or e if it is rejected. Note that at least one starting vertex is mapped elsewhere if x is accepted and unchanged if x is rejected as in the following figure. We'll take this vertex to be the root node of our branching program.



One key property we can use is if f has a π-pbp and σ is conjugate to π (i.e. $\exists \tau$ such that $\tau^{-1}\pi\tau = \sigma$) then f has a σ-pbp of the same size. This follows as we take our π-pbp and permute the top level by τ and the bottom by τ^{-1}. If the input is accepted, the inner permutation will be π for a overall permutation of $\tau^{-1}\pi\tau = \sigma$. If the input is rejected, the inner permutation will be e for a overall permutation of $\tau^{-1}e\tau = \tau^{-1}\tau = e$. Consequently, we have flexibility in our choice of permutation π. Recall that π and σ are conjugate if they have the same cycle structure.

Let us consider the problem of transforming a fan-in 2 formula into a π-pbp for some $e \neq \pi \in S_w = \text{Sym}(w)$. Eventually we'll settle on $w = 5$ but for now let us proceed in generality by inducting on the size (equivalently, the depth).

The base case includes all formulas that consist of a single variable. Here, the branching program would merely be two layers with the 0 labelled arrows inducing the identity permutation and the 1 labelled arrows inducing a permutation π.

For the induction step, it is enough to prove for negation and AND as De Morgan's laws will suffice to prove the case of OR gate. For negation, we know that a permutation and its inverse are conjugate as they share the same cycle structure. By the induction hypothesis, we have a π^{-1}-pbp of the same size for the formula inside the not operation. Now we permute either the top or bottom by π. The action of the whole is $\pi e = \pi$ if the interior rejects and $\pi\pi^{-1} = e$ if the interior accepts.

For conjunction, we have two interior formulas f and g. Suppose we have a π-pbp for f (called M_f) and a σ-pbp for g (called M_g). Then there is a $\tau = [\pi, \sigma]$-pbp for $f \wedge g$ of size $\leq 2(\text{SIZE}(M_f) + \text{SIZE}(M_g))$. This is done by putting the four machines for $\pi^{-1}\sigma^{-1}\pi\sigma$ in sequence. If one machine rejects, it and its inverse act as the identity so the whole collapses to the identity. If both accept, it acts as the commutator τ. If all possible commutators reduce to the identity we have a problem, so will need to choose our permutation group appropriately. If τ is conjugate to π then we get a π-pbp in the end. Thus if there exist conjugates τ, π, σ where $\tau = [\pi, \sigma] \neq e$ then we can obtain the desired π-pbp of width w with size $\leq 2^d \, \text{SIZE}(f)$ for the formula f, letting d be the depth of the formula as the worst case is a conjunction on each level doubling the size. If the formula is log-depth then 2^d is polynomial in the input. If the formula size is polynomial then π-pbp is polynomial size.

We claim that such π, σ, τ exist for $w = 5$. This is true as S_5 is not solvable, and in fact is the smallest permutation group that isn't solvable. Examples would be $\pi = (12345)$, $\sigma = (13542)$, and $\tau = (12534)$.

Finally, let us prove (4) $\Rightarrow$ (1), that poly-size branching programs of constant width can be implemented with NC^1 circuits. To do this we use the same technique as in the proof $\text{NSPACE}(n) \subseteq DSPACE(n^2)$. Instead of dividing a computational tableau, here we divide the given branching program itself.

Let $F(a, b)$ return true if the execution of the branching program on its given input enters node a and eventually transitions to the node b and return false otherwise. As the execution of the program must pass through all layers, computing the value of $F(a, b)$ can be broken up from the question of whether "a transitions to b" into deciding over all c_i in the middle layer, halfway between the layer of a and the layer of b, whether the scenario "a transitions to b passing through node c_i" holds. The width of the program is a constant k, so this can be done with an OR gate of fan-in k. Each scenario can be returned to our initial form using an AND gate of fan-in 2, evaluating "a transitions to c_i" and "c_i transitions to b.

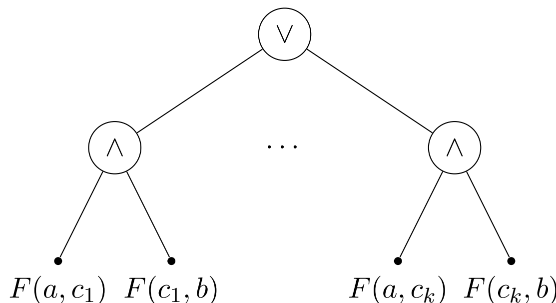


Figure 1: F(a,b)

We repeat the process to determine the validity of each leaf node. This terminates at the trivial case when a and b are in adjacent layers where we evaluate the single transition. We know the initial start state and, if the program accepts, the final state so to check if a branching program accepts we simply evaluate $F(\text{start state}, \text{accept state})$.

Let us check the properties of the resulting circuit. The fan-in is bounded by k, a constant. Adding two layers to the circuit reduced the size of the problem by a factor of two so the circuit depth will be log in the depth of the branching program. the program depth is polynomial so the circuit will have log depth. Thus the circuit is in NC^1.

Another similar proof is $\mathrm{L} \subseteq \mathrm{NC}^2$. Here the number of choices for intermediate node c_i in the reduction step is polynomial rather than constant. In order to bring our fan-in down from a polynomial, we must use a log depth circuit of OR gates, each with constant fan-in. Avoiding this extra log factor in the induction step is the critical element to bringing the circuit from the above proof into NC^1 rather than NC^2.

□

We note that it is still possible that $\mathrm{NP} \subseteq \mathrm{NC}^1$. We know AC^0 is much more restricted.

2 Randomness

How does the computing power of a machine change if we allow it to flip unbiased coins to generate random bits? Assuming the existence of a suitable source of randomness and allowing a machine to make use of the resulting bit strings has resulted in simpler algorithms in a variety of settings. There are some settings where we know how to solve problems using randomness for which we have no deterministic algorithms. One example is the dining philosopher's problem in distributed computing. Cryptography seems to rely crucially on the use of randomness to disguise the particular

form of the cipher being used through the generation of keys. Without randomness, the strength of a cipher would correspond to how secret the algorithm itself remained.

2.1 Standard Model

In the standard setting where the objects under consideration are mappings from inputs to outputs, it is still an open question whether randomized algorithms enjoy asymptotic complexity gains that cannot also be realized by a well chosen deterministic algorithm. The current conjecture is randomness induces at most a polynomial speedup in time or at most a constant reduction in space.

Formally, we include randomness in a Turing machine by allowing the machine to generate random bits and base decisions on the results, making the configuration at any time a random variable. As a consequence, the output of the program will be a random variable, introducing the possibility that our machine will return an incorrect result. For the class of decision problems, we consider three types of algorithms distinguished by the types of error they allow.

- 2-sided error. False positives and false negatives are both allowed.
- 1-sided error. Only false negatives allowed.
- 0-sided error. Program allowed to output accept, reject, or "unknown". When it accepts or rejects it is correct in its decision.

Machines that aren't solving decision problems can use randomness to get benefits without the possiblity of error in the output. Quicksort would be one such example. It always returns a sorted set and never returns "unknown" but the running time is affected by the randomness.

Notice that in machines using randomness, running time and space are also random variables. We are interested in improving the bounds on these variables while controlling the error in the output. By controlling the error, we mean bounded away from trivial. For example, it is easy to be right in a decision problem $\frac{1}{2}$ of the time by flipping a coin and outputing the result. A non-trivial error bound would be $\epsilon \leq \frac{1}{2} - \delta$ with $\delta > 0$. Once error is restricted away from $\frac{1}{2}$ we can reduce it further by running k independant instances in parallel and outputting the majority answer.

$$\begin{aligned}
\Pr[\text{majority vote is wrong}] &= \sum_{i=k/2}^{k} \Pr[\text{Exactly } i \text{ trials are wrong}] \\
&= \sum_{i=k/2}^{k} \binom{k}{i} \epsilon^i (1-\epsilon)^{k-i} \\
&\leq (\frac{1}{2} - \delta)^{k/2} (\frac{1}{2} + \delta)^{k/2} \sum_{i=k/2}^{k} \binom{k}{i} \\
&\leq (\frac{1}{2} - \delta)^{k/2} (\frac{1}{2} + \delta)^{k/2} 2^k \\
&= (1 - 4\delta^2)^{k/2} \\
&\leq (e^{-4\delta^2})^{k/2} \\
&= e^{-2k\delta^2}
\end{aligned}$$

Exercise 1. *Prove the inequalities.*

- $\epsilon^i(1-\epsilon)^{k-i} \leq \epsilon^{k/2}(1-\epsilon)^{k/2}$ *for* $i \geq k/2$.
- $e^x \geq 1 + x$. *Consider the tangent line at* $x = 0$.

This tells us if δ is $\frac{1}{\text{poly}}$ then some poly k will make δ really small. Thus we can control the 2-sided error case as if the original probability of error is not too close to $\frac{1}{2}$ then it is possible to produce an exponentially small probability of error through a majority vote over a polynomial number of runs. For 1-sided error, the bounding calculation is easier as we are only searching for at least one "yes" vote. As machines with 1-sided error or 0-sided error can be viewed as into 2-sided error machines with comparable terms, our analysis above suffices to prove we can control error rates on all machines.

2.2 Examples of Randomized Algorithms

In the time bounded setting, the traditional example of the power of randomization has been primality testing where we have simple polynomial time algorithms that use randomness. However, we now know there exists a polynomial time deterministic algorithm that also performs primality testing.

Instead, consider polynomial identity testing. To create an arithmatic formula we are allowed to add, multiply, and subtract variables and constants with the use of brackets allowed to control the order of operations. It is not clear when the resulting multivariate polynomial is identically zero for all variable settings (the variables can be drawn from domains such as the integers or finite fields, constrained such that all variables draw form the same domain). One method would be to expand all the terms and collect the resulting terms one monomial at a time. However, the number of monomials can be exponential in the size of the formula. In fact, all known deterministic algorithms for polynomial identity testing run in exponential time.

By switching the question to ask when a multivariate polynomial is not identically zero we can get a simple 1-sided error algorithm by performing test and check at points chosen independently and uniformly at random over a sufficiently large interval I in the domain. What does sufficiently large mean in this instance?

Lemma 1. $\Pr[P(\vec{x}) = 0 | P \neq 0] \leq \deg(P)/|I|$ *where* P *is the multivariate polynomial and elements of* $\vec{x}$ *are chosen from* I.

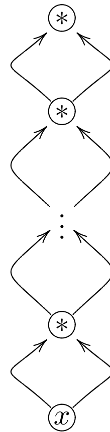
We note that considering the polynomial as the circuit generated by the arithmetic formula yields the bound $\deg(P) \leq$ (size of formula). This is apparent as the addition gates don't add to the total degree but just return the maximum degree of their children and the multiplication gates return the degree that is the sum of the degrees of their children. Thus we get nontrivial bounds using $|I|$ of order the size of the formula. Controlling the size of our inputs is important because it allows us to control the time it takes to evaluate the formula. Letting N be the size of the formula, each value in $\vec{x}$ is from I so can be specified with $O(\log N)$ bits. Evaluating the formula raises the variables to no more than the power N, thus the intermediate numbers have no more than $O(N \log N)$ bits. All the arithmetic operations can be performed in polynomial time in the bit length of the input numbers so the formula can be evaluated in time polynomial in N.

Exercise 2. *Prove lemma 1. This can be done by induction on either the degree or the number of variables.*

For multivariate polynomials, there is no known efficient deterministic algorithm. The existence of one would have major implications for circuit complexity questions that are approximately 40 to 50 years old now.

3 Looking Ahead

Next time we will examine arithmetic circuits instead of formulas. These will have similar construction as we saw in the Boolean setting but use different gates (addition and multiplication instead of AND and NOT). Unfortunately, here we won't have total degree bounded by circuit size. Consider the following figure where the degree is approximately $2^{\text{size of circuit}}$.



In this case we have to compute $x^{2^{\text{size}}}$ which is expensive. Our $|I|$ being exponential size isn't a problem as it still has only a polynomial number of bits but the evaluation step is too expensive as it involves numbers with an exponential number of bits. We can control this by taking operations modulo some m.

Acknowledgements

In writing the notes for this lecture, I sampled from the notes by Brian Rice and Jake Rosin for lecture 11 from the Spring 2010 offering of CS 710 to revise the section on randomness. I similarly used the notes by Jake Rosin for lecture 11 from the Spring 2007 offering of CS 710 to improve my section on randomness.