Recitation 12: Probablistic TMs, Arithmetization, Interactive Proofs

In this recitation, we'll cover the definition of BPP and its basic properties, the arithmetization-based algorithm for testing equivalence of read-once branching programs, and the definition of interactive proofs. These concepts are particularly tricky but they are very powerful techniques that you will likely encounter if you continue to learn about complexity theory or adjacent fields like algorithms and cryptography.

If you're curious about this or have questions about other complexity-related classes to take, please ask Prof. Sipser or the TAs, we're always happy to talk about this!

Probablistic Algorithms

In designing efficient algorithms, it is often much easier to give a *randomized* procedure that has a *pretty good chance of working* than it is to give a deterministic algorithm that works 100% of the time. The natural question for a complexity theorist to ask is: are there cases where this randomness is *necessary* to get an efficient algorithm?

We formalize this question by defining a ***probablistic Turing machine*** (PTM). Just like a nondeterministic Turing machine, a probabilistic Turing machine is allowed at any point to "branch" into two possibilities. In the case of NTMs we imagined that the algorithm always "got lucky", and would accept if (and only if) any of the branches accepted. For a PTM, we alternatively imagine the algorithm taking each side when branching with probability $1/2$, so that instead of always giving the same answer, it just has some *probability* of ending up accepting a particular input.

Definition 1. *A probablistic Turing machine is a Turing machine with the additional ability to "flip a coin". That is, at any point it can choose to generate a random bit. The acceptance probability of a PTM A on input x is the probability (over the coin tosses of the algorithm) that A accepts when run on x.*

We would like to talk about a PTMs as deciding a language — but a given PTM sometimes outputs different answers for the same input. To address this, we will focus on PTMs that get the right answer "most of the time":

Definition 2. *The class* BPP *consists of all languages L such that there exists a PTM A running in* ***polynomial time*** *(i.e. every branch takes at most polynomially many steps in the input length), and such that*

- *For $x \in L$, the acceptance probability of A on x is at least $\frac{2}{3}$.*
- *For $x \notin L$, the acceptance probability of A on x is at most $\frac{1}{3}$.*

The specific choice of $\frac{2}{3}$ and $\frac{1}{3}$ here may seem arbitrary — that's because it is. It turns out we get the exact same class if you replace $\frac{2}{3}$ and $\frac{1}{3}$ with any constants strictly greater and strictly less than $\frac{1}{2}$, respectively.

One fact worth noting is that not every polynomial-time PTM decides a BPP language: you could imagine a PTM which, on some inputs, accepts with probability $1/2$, which we don't count as either a YES or NO instance. If you wanted, you could instead define a language where you include every x with acceptance probability at least $1/2$, and exclude every x with acceptance probability less than $1/2$. However, the class of languages definable this way is *not* BPP — this is a seemingly much larger class, known as PP (which in particular is known to contain NP). It's worth thinking about why our amplification argument doesn't work if you only have a guarantee on whether the acceptance probability is at least $1/2$ or not (hint: consider a acceptance probability that varies based on the input length).

Lemma 1 (Amplification). *Suppose there exists a probabilistic poly-time Turing machine A that accepts $x \in L$ with probability at least* $.500001$*, and acceots $x \notin L$ with probability at most* $.499999$*. Then, there exists a probablistic poly-time Turing machine B that accepts $x \in L$ with probability at least* $.999999$*, and accepts $x \notin L$ with probability at most* $.000001$*.*

Proof. The idea is to just repeatedly run A on x, and take the majority answer. If we run A on x for $\mathsf{poly}(n)$ many times, tools from probability (see Chernoff bounds, or an alternative argument in Prof. Sipser's textbook) ensure that we are extremely unlikely to err on more than half of them — in fact, by running enough iterations of A, we can not only make B's error probability less than .000001, we can make it less than $2^{-\mathsf{poly}(n)}$. (And for some arguments it actually is useful to drive the error probability that low, although we will probably not see them in this class.) □

Let's also note a couple of small observations about probabilistic Turing machines. The following question was addressed in recitation:

Question 1. *When we describe randomized algorithms, sometimes we want to choose a uniformly random element from some set — for instance, the set of integers mod some large prime. But our definition of PTMs only allow access to coin tosses — how do we choose a uniform random element of a larger set?*

Answer 1. *The basic answer is that, to generate a uniformly random element of a set of size 2^k, it suffices to flip k coins, since there are 2^k equally-likely outcomes for the sequence of bits you get as a result. But this question is getting at a more subtle point: what if the set S you want to choose from has a size that isn't a power of 2? For instance, what if you wanted to choose a random value from $\{1, 2, 3\}$? The idea is still to flip $\lceil \log |S| \rceil$ coins, and associate each element of S with some sequence of resulting bits — but now, up* *to half of the resulting sequences of flips might not correspond to any element of S. So, with probability up to $1/2$, this procedure might fail to generate an element of S. But, fine, in that case we'll just run it again over and over again up to n times until it does. The probability that we still haven't found one after n attempts is at most $\frac{1}{2^n}$, at that point we can just give up, since getting this branch wrong will only give a tiny increase to the algorithm's overall error probability.*

Another good question is "where does BPP lie in relation to the other classes we know about?". It turns out not a whole lot is known — for instance, we don't know whether either of BPP or NP contains the other. However, we do know the following:

Claim 1. $\mathsf{P} \subseteq \mathsf{BPP} \subseteq \mathsf{PSPACE}$.

Proof. The inclusion $\mathsf{P} \subseteq \mathsf{BPP}$ is easy: any deterministic TM deciding A can be thought of as a PTM that just happens to make no coin flips, and happens to have 0% error rate, so $A \in \mathsf{P} \implies A \in \mathsf{BPP}$. To show $\mathsf{BPP} \subseteq \mathsf{PSPACE}$, observe that in polynomial space we can just simulate every branch of a poly-time PTM one at a time, and keep track of the total acceptance probability. □

In contrast to the case of P and NP, which are strongly suspected to be unequal, perhaps somewhat surprisingly there is good evidence[1] to believe that $\mathsf{P} = \mathsf{BPP}$, although this is not known for sure.

[1] If you make some assumptions about which problems are hard for small circuits of ands/ors/nots to solve, you can construct ***pseudorandom generators***, which output sequences of bits that "look random" to poly-time algorithms, letting you simulate randomized algorithms deterministically. If you want to hear this story properly, consider taking 18.405 in the spring!

Branching Programs

Now, let's talk about one of the few problems known to be in BPP but not known to be in P. First, recall the model of ***branching programs*** (BPs). A branching program consists of a directed acyclic graph, with a starting node, and two output nodes labeled 0 and 1, respectively. All nodes except those output nodes are labeled with the name of some variable x_i, and have exactly two outgoing edges, labeled 0 and 1. We call these nodes *query nodes* since the computation of the BP takes one of the two edges depending on the value of x_i in the input.

We now describe the computation of a BP B on some input x. From the starting node, the B takes either the 0-edge or the 1-edge depending on the value of its label x_i. From there, at each query node, the BP continues to follow one of the two outgoing edges based on the input until it reaches an output node. At this point, B outputs either 0 or 1 depending on which output node it is in. This defines some function $f\colon \{0,1\}^n \to \{0,1\}$.

As with other computation models, we are often interested in the question of whether two BPs are *equivalent*. Even if two BPs B_1 and B_2 have different underlying graphs, they might still agree at every possible input (we'll see an example of this later).

Definition 3. *We say that two BPs B_1 and B_2 are equivalent if for all inputs $x \in \{0,1\}^n$, $B_1(x) = B_2(x)$. In other words, they compute the same boolean function.*

Based on this, we define the following language.

Definition 4. $EQ_{BP} = \{\langle B_1, B_2\rangle \mid B_1 \textit{ and } B_2 \textit{ are equivalent BPs}\}$

This problem turns out to be quite hard. In fact, you saw on the pset that it is coNP-complete!

So let's instead look at a restricted form of branching program. We'll call a branching program a ***read-once branching program*** (ROBP) if it only queries each variable at most once on any given computation branch. Note that this does not mean that each variable only appears in one query node of the DAG, it just means that each variable only appears once on each path from the starting node to an output node. It is also worth noting that for different inputs x and x', the variables might get queried in different orders. So even though this model is more restricted than a regular BP, it is still quite expressive.

We can now define a language for the problem of testing equivalence for ROBPs.

Definition 5. $EQ_{ROBP} = \{\langle B_1, B_2\rangle \mid B_1 \textit{ and } B_2 \textit{ are equivalent ROBPs}\}$

This problem turns out to be really interesting from a complexity perspective. Similarly to EQ_{BP}, this language is not known to be in P. Nonetheless, we can actually show that this language is in BPP! This is particularly surprising since it means we are able to determine if two ROBPs agree at *every possible input* without actually having to check all of them!

What would happen if $EQ_{BP} \in \mathsf{P}$?

Arithmetization

Given two branching programs with inputs of length n, it is entirely possible for them to agree at $2^n - 1$ possible inputs, but only disagree at one. So, to check whether two ROBPs are different, we can't just guess random boolean inputs and try them both, since the probability that we pick the one at which they disagree could be exponentially low.

Instead, we want to convert the two ROBPs to some other mathematical object where testing equivalence randomly is actually feasible. At a high level, this means we want an object where two instances are either equivalent, or *very different*. It turns out a very common mathematical object satisfies just this: low-degree polynomials. In particular, we will focus on polynomials over finite fields (which you can just think of as working over the integers modulo some prime a).

For single variable polynomials, we know that any polynomial of degree d is either equal to 0 for at most d inputs, or it is 0 everywhere. Note that this fact can be used to check if two polynomials of degree d are equivalent by picking a random input r and then checking if $p(r) - q(r) = 0$. If they are equivalent, $p - q$ will always be 0 but if they are not, this will happen with probability at most d/a.

While this exact property does not hold for multivariate polynomials, we have a very similar result. You are not responsible for the proof of this result, but the statement can be very useful!

Definition 6. *(Schwartz-Zippel) Let p be a nonzero degree-d polynomial over m variables. If we randomly pick inputs $r_1, \cdots, r_m$ from a finite field of size a, then*

$$\Pr[p(r_1, \cdots, r_m) = 0] \leq md/a.$$

Now that we have seen how to efficiently use randomness to check equivalence of polynomials, we need to find a way to convert BPs to appropriate polynomials. Our goal will be to create an n-variable polynomial p over a field $\mathbb{F}_a$ that agrees with the output of some BP B on all inputs $x_1, \cdots, x_n \in \{0, 1\}^n$. We will do this by sequentially assigning polynomials to the nodes and edges as follows:

- The starting node gets the polynomial 1
- The 0 outgoing edge from a query node with variable x_i and polynomial p is assigned the polynomial $p(1 - x_i)$
- The 1 outgoing edge from a query node with variable x_i and polynomial p is assigned the polynomial px_i
- A query node with k incoming edges that have been assigned polynomials $p_1, \cdots, p_k$ is assigned the polynomial $p_1 + \cdots + p_k$.

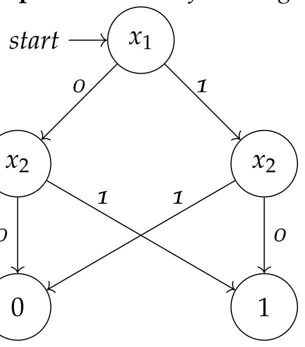
Then, the polynomial at the output node labelled 1 will be the polynomial that corresponds to the BP.

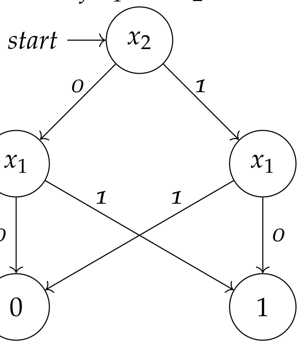
Now we show some explicit examples of this arithmetization for special Branching Programs that compute the XOR and OR of two boolean variables.

First we just give the branching programs.

Example 1. *The two following BPs compute the XOR of x_1 and x_2:*

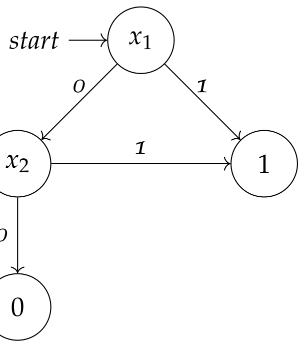
As an exercise, check that these actually both compute XOR!



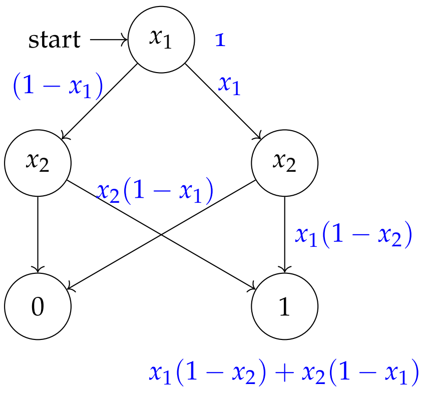


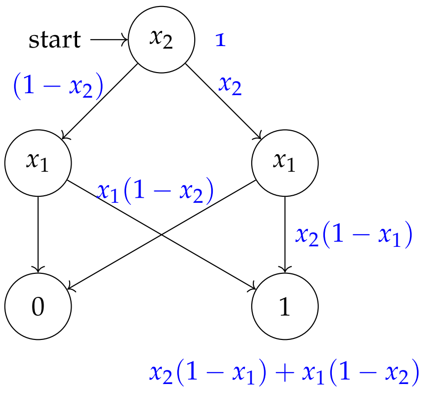
Example 2. *The following branching program computes the OR function:*

Another exercise! Check that this corresponds to the OR function.

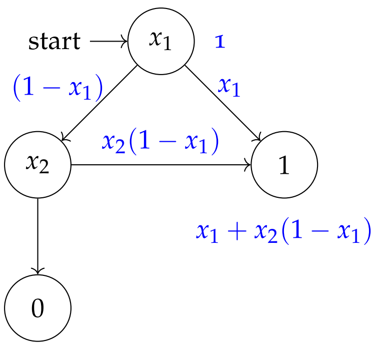


Now, let's arithmetize the BPs for XOR. Since we only need the polynomial at the 1 output node, we only compute the relevant polynomials, which we write in blue:





We can also arithmetize the BP for OR:



As an exercise, pick a prime number, then two random inputs from that field and evaluate the three polynomials on these. If you repeat this a few times, you should notice that you tend to get different results.

Note that the technique of arithmetization is applicable in more cases than just ROBPs. For instance, we can arithmetize any boolean formula by simply describing how to convert NOT, AND and OR. One way to do this is as follows:

1. $\bar{a} \to (1-a)$
2. $a \wedge b \to a \cdot b$
3. $a \vee b \to a + b - a \cdot b$

It is a good exercise to check that these arithmetizations agree with the original expression at all boolean inputs.

This gives us our algorithm for deciding equivalence of ROBPs: we'll think of both ROBPs as polynomials, and evaluate both of them on random points from a large finite field. The reason this works is as follows:

1. If the two ROBPs are different, their associated polynomials will be different. These are degree-n polynomials over n variables, so their difference is as well. As long as our finite field has size larger than n^2, Schwartz–Zippel guarantees that we're likely to find a nonzero difference.

2. If the two ROBPs are the same, their associated polynomials will be the same, and so we will always get the same result. This is the part that crucially relies on the fact that they are read-once: because they're read-once, the associated polynomial will be multilinear, and there's only one multilinear polynomial agreeing with any function over $\{0,1\}^n$ (this particular case is explained in more detail in Sipser's book). However, if you arithmetize arbitrary branching programs, their associated polynomials might have terms with higher powers such as x_1^2. This equals x_1 over boolean inputs but not over arbitrary fields, so equivalence of arbitrary BPs does not imply equality of their arithmetized polynomials.

One crucial point to emphasize is that, in this procedure, we do *not* explicitly write down the polynomials associated with each ROBP. There are examples where doing so would require writing out exponentially many coefficients. We're making use here of the fact that we can *evaluate* the polynomial without writing down the coefficients, simply by plugging in the variables and working down the DAG, as may have done to evaluate those examples.

Interactive Proof Systems

Time permitting, we finished the recitation with a quick review of Interactive Proof Systems and the corresponding language class IP, which can be thought of as a more powerful version of NP. We can restate our definition of NP in the following terms: a poly-time verifier V has input x, and an unbounded prover P wants to convince V that $x \in L$ (regardless of whether x is actually in L!). The prover will send the verifier some "certificate" of membership, and the verifier must have some procedure that accepts if and only the certificate is good. That is, for any $x \in L$, there exists some certificate P can send to convince V to accept, but for any $x \notin L$, there's nothing P could send that would "trick" V into accepting. The languages L that admit such a verifier strategy V are exactly those in NP.

To modify this definition, imagine that instead of just sending a single message, P allows V to respond with questions, which P then answers. Intuitively, one might expect that this kind of "back-and-forth" helps V examine places where P's story might be suspicious, and become more fully convinced. Something like this turns out to be true, but in order for things to be interesting we need to allow V to be randomized. The reason is simple: suppose V was following some deterministic procedure. Then, P could just predict the entire transcript of the interaction and send that in a single message. That is, instead of actually having a back-and-forth conversation, P could just say "here's my proof. You're going to want to respond with questions blah blah and blah, to which I would respond blah". But with randomness, we get a more interesting notion:

Note: this is also good justification for why you should ask question in lecture!

Definition 7. *We say a language $A \in IP$ if there exists a probabilistic polynomial time verifier V such that:*

- *There exists an honest prover P satisfying that for all $x \in A$:*

$$\Pr[V \leftrightarrow P \textit{ accepts } x] \geq 2/3$$

- *For any $x \notin A$ and any prover $\tilde{P}$:*

$$\Pr[V \leftrightarrow \tilde{P} \textit{ accepts } x] < 1/3$$

As in the case of NP, we're imagining a prover that always really wants to convince the verifier that $x \in A$, regardless of what is actually true. It's a valid protocol if, for any $x \in A$, the prover has a strategy that is likely to be convincing — but, for any $x \notin A$, no prover strategy is likely to trick the verifier.

It turns out that IP protocols are incredibly powerful. In class on Tuesday, we'll show that coNP $\subseteq$ IP. This can actually be strengthened, and although we do not cover the proof in class, you should know that the following relationship is true:

Theorem 1. IP $=$ PSPACE

This is an impressive fact, and was considered rather surprising at the time it was proved. This proof will again make use of the techniques of arithmetization we discussed in the previous section. This time, the prover sends certain polynomials related to the problem, using the same kind of Schwartz–Zippel polynomial identity testing tricks to make sure that those polynomials are actually good. (Note: it's a little bit of a tricky argument, and we won't have a recitation covering it, so if you're totally baffled after Tuesday's lecture please come to the review session and we'll be happy to talk through things again!)

Closing Remarks

Since these are the last recitation notes, we just wanted to thank you for such a fun semester and really hope you enjoyed learning about computability and complexity. Good luck on the final!