IAS/PCMI Summer Session 2000
Clay Mathematics Undergraduate Program
Basic Course on Computational Complexity

Lecture 11: Proofs, Games, and Alternation

David Mix Barrington and Alexis Maciel
July 31, 2000

1. Overview

In this third week of the course we take a closer look at complexity classes above P, such as NP, PH, BPP, IP, and PSPACE. In each case the complexity class will consist of languages for which a skeptical person might be *convinced* that a string is in the language. Exactly what this means will differ for the various classes. In this lecture we consider NP as a class of provable languages, and look at *alternation*, a generalization of nondeterminism that defines languages in terms of games.

- We begin by surveying the many ways a prover might convince a verifier of some fact, given that one or both might be computationally limited. This will help motivate the complexity classes we will study this week.
- We describe NP as a proof system, and define the *polynomial hierarchy* PH, a complexity class between NP and PSPACE. The subclasses of PH will be based on increasingly complicated *proof games* between a prover and a verifier, here assumed to be smart enough to play optimally.
- We define the model of *alternating machines*, in terms of two-player games played on nondeterministic machines. We define the classes ASPACE(f) and ATIME(f) for arbitrary functions f, and in particular the complexity classes AP and AL.
- We prove the *Alternation Theorem*, in the special cases of AL = P and AP = PSPACE, and indicate how to prove the general cases in the exercises.
- We state the characterizations of the circuit classes NC^i and AC^i in terms of alternating machines, again leaving proofs for the exercises.

2. Different Ways of Proving

Our general setting is that we have two characters (called, following tradition, Alice and Bob) who are interested in some fact, such as whether a particular graph is three-colorable, or in general, whether a string x is in a particular language A. At the beginning of the scenario, Alice knows this fact and Bob does not. How can Alice convince Bob of the truth of the fact?

When we impose any set of conditions on Alice and Bob, we implicitly define a set of *provable languages* under those conditions. As we explore complexity classes above P, we will find that many classes can be defined by natural provability conditions:

- If the language in question is in P, Bob can satisfy himself as to whether $x \in A$ by a poly-time computation without any input from Alice. Since we will always allow Bob to carry out poly-time computations, P will be contained in all our provability classes.

- We have already used the language of presenting and checking proofs to describe the class NP. If $A \in$ NP, there is a way for Alice to *prove* that $x \in A$ by presenting some string y to Bob and letting Bob run a poly-time computation on x and y. Note that this proof requires Alice to figure out or otherwise find the proof string, something we believe to be computationally difficult. NP proofs assume a "clever Alice" who is able to do this.

- If we relax the assumption that Bob must be absolutely certain that $x \in A$, then he will be able to satisfy himself about membership questions for an apparently larger class of languages by using a *randomized algorithm*, even without Alice's help. We define BPP to be the class of languages for which there is a poly-time randomized decision algorithm whose error probability can be made less than any constant $\epsilon > 0$. (These terms will be defined in Advanced Lecture 12.)

- What if Alice and Bob are both clever? Bob can ask clever probing questions about the details of Alice's proof, and she can make clever responses to them. For many languages we can define a game based on x, such that Alice has a winning strategy for the game iff $x \in A$. In this lecture we will formalize this notion using *alternating machine*, and show that the resulting complexity class AP, where the games are played in polynomial time, is equal to the more familiar class PSPACE.

- Perhaps the most natural definition of proof has a clever Alice and a randomized Bob. Again Bob may ask questions, based on his random choices, and Alice

answers, forming an *interactive proof*. Bob is willing to tolerate a small probability of being convinced by a false proof or of falsely rejecting a valid proof, but we want him to accept valid proofs and reject false ones with high probability. If a poly-time proof system of this type exists we say that the language is in IP. In Advanced Lecture 14 we will show that IP is also equal to PSPACE. This is surprising as it says that the system with a randomized Bob can prove the same languages as that with a clever Bob. We will also give an example of a *zero-knowledge proof*, where Alice convinces Bob that $x \in A$ but tells him nothing that would allow him to convince anyone else of this fact.

- Finally in Advanced Lecture 15 we will see our strangest notion of proof, a *probabilistically checkable proof*. Here a clever Alice constructs a proof but Bob only looks at $O(1)$ bits of it, which he chooses based on $O(\log n)$ random choices. The resulting complexity class $\mathrm{PCP}(\log n, 1)$ has been proved equal to NP, a theorem with important implications for optimization problems.

3. NP and the Polynomial Hierarchy

In the ordinary prover-verifier setting for NP, a clever Alice provides a proof and a poly-time Bob checks it. A language A is in NP iff we can design a system of this sort such that $x \in A$ iff *Alice can provide a proof*. An example is 3-COLORABILITY of a graph, where the proof consists of a three-coloring.

Suppose that Alice and Bob have a graph that is *not* three-colorable, and that Alice knows this. As far as we know, there is no polynomial-length string she could give to Bob that would convince him that no three-coloring exists. The language $\overline{\text{3-COLORABLE}}$ of non-three-colorable graphs is not known to be in NP, but in the class called "coNP" that consists of the *complements* of the languages in NP.

There is a proof system that can handle this language. Suppose that *Bob* is the clever one and wants to convince *Alice* that the graph is *not* in $\overline{\text{3-COLORABLE}}$. Since he's really trying to convince her that it is in 3-COLORABLE, all he needs to do is provide a three-coloring. Similarly, Bob could convince Alice that a number is *not* in PRIMES by giving her nontrivial factors of the number, so PRIMES is in co-NP as well. (PRIMES is actually in NP and in coNP, see the Exercises.)

The complexity class PH, called the *polynomial hierarchy*, consists of languages for which we can define a game with $O(1)$ rounds between a clever Alice, who wants to establish that $x \in A$, and a clever Bob, who wants to establish $x \notin A$. In each round, one of the players names a string whose length is polynomial in the length

of x. After all rounds are done, a poly-time deterministic machine runs on an input consisting of the original input and all of Alice's and Bob's moves. It decides who wins the game, and we say that the game defines A if Alice has a winning strategy for the game iff $x \in A$.

Here is an example. Say that a circuit C is *optimal* if no circuit D with fewer gates computes exactly the same function as D. (Assume that both C and D have the same input variables $x_1, \ldots, x_n$.) We can give the formal definition:

$$C \in \text{OPTIMAL} \leftrightarrow \forall D : \exists x : [(|D| \geq |C|) \vee (C(x) \neq D(x))]$$

How can we define the language OPTIMAL by a game? The input to the game is the circuit C. Bob provides the circuit D. Alice replies with a string x of length n. Then Alice wins iff either $|D| > |C|$ or $C(x) \neq D(x)$. Clearly, if the first-order formula above is true, Alice will win under optimal play, as either $|D| > |C|$ already or she can find an x such that $C(x) \neq D(x)$. Just as clearly, if the formula above is false, Bob can win the game by providing the D that is the counterexample to the optimality of C.

The game to put a language A in PH exists iff there is a first-order formula of the type above defining A. (This is not hard to prove.) Specifically, the quantifiers bind strings of length polynomial in the input, and the quantifier-free part of the formula must be checkable in polynomial time.

One historical note: the complexity class AC^0 was first studied in large part in order to find out more about PH. In particular, the "main result" of the famous 1984 Furst-Saxe-Sipser paper, proving that PARITY is not in AC^0, is that because of this lower bound, certain proof techniques ("relativizable" ones) are unable to show that PH = PSPACE even should this be true.

4. Alternating Machines

We can define a more general "model of computation" that extends the notion of "proof by game" beyond a constant number of moves by Alice and Bob. Suppose again that Alice wants to establish that $x \in A$, Bob wants to establish $x \notin A$, and a poly-time referee will decide who wins after they have together written a polynomial-length string. We could allow them to *alternate* bits between them, so that Alice writes y_1, Bob writes y_2, Alice writes y_3, and so on until polynomially many moves have been made.

An equivalent game is played on a poly-time *nondeterministic* machine M. On each odd-numbered move of M, Alice decides which of the nondeterministic choices

M will make. On each even-numbered move, Bob decides. At the end, Alice wins if the machine accepts and Bob wins if it rejects. (We can put a clock on M so that it is guaranteed to halt.) It is not hard to show that these two formulations are equivalent — that there exists a poly-time referee and rules to define a language A iff there is a nondeterministic machine whose alternation game defines A.

It is not crucial that the players alternate moves as long as it is clear who is supposed to move next. (For example, in the machine setting there could be Alice-control states and Bob-control states.) In fact a third resource for alternating machines, along with time and space, is the *number of alternations*, that is, the number of changes from Alice-control to Bob-control states or vice versa.

If the protocol for who should move next is not something obvious like strict alternation of bits, we can enforce the protocol as long as the referee can determine when a violation has occurred and punish the offender by making them forfeit the game. Then, under optimal play by both players, such rule violations will never occur. We have to worry about the resources needed to decide a question about the legality of a move. But note that we need to decide such a question *only once* per game, because whichever way the challenge goes the game will end. (We can say that if a player incorrectly challenges a move, they lose.)

We define AP to be the set of languages for which there exist poly-time alternation games or poly-time alternating machines defining them. Similarly AL is the set of languages defined by log-space alternation games or log-space alternating machines. More generally, we can define classes ATIME(f) and ASPACE(f) by analogy to the deterministic and nondeterministic classes.

Note that here we have defined alternating machines and their corresponding complexity classes using a semantics where "game" and "winning strategy" are considered to be primitive notions. This is of course not the only way to define these concepts, and historically it is not the way they were developed. Both Sipser and Papadimitriou give good explanations of the more conventional semantics.

We defined the problem QBF in Basic Lecture 8 as the set of true quantified boolean sentences, where the quantifiers bind boolean variables and the quantifier-free part consists of a boolean formula. We asserted there (and asked you to prove in the exercises) that QBF is complete for PSPACE. Here we prove an easier fact:

Theorem 1 *QBF is complete for* AP *under poly-time reductions.*

Proof To see that QBF is in AP, define a game as follows. Given a quantified formula Φ, have Alice and Bob choose the value of each quantified variable in turn,

with Alice choosing for $\exists$ variables and Bob choosing for $\forall$ variables. After all values are chosen, the referee evaluates the formula and Alice wins iff it is true. From the definition of the meaning of the quantifiers, it should be clear that Alice wins this game under optimal play iff the quantified formula Φ is true.

To show completeness, we must take any poly-time alternating game to define whether $x \in A$ and produce, in deterministic polynomial time, a quantified formula that is true iff Alice wins the game under optimal play. Assume for simplicity that the players alternate one-bit moves for $p(n)$ rounds to create a string y and then the referee decides who wins based on x and y. We model each of the rounds by a quantifier binding a separate boolean variable, $\exists y_i$ for Alice's moves and $\forall y_i$ for Bob's. Then we need to write a quantified boolean formula in these variables that is true iff the referee accepts the resulting string. Now we follow the proof of the Cook-Levin Theorem (that SAT is NP-complete) in Basic Lecture 7. We write an unquantified boolean formula $\phi(x, y_1, \ldots, y_{p(n)}, z_1, \ldots, z_{q(n)})$ that is satisfiable (by some values of the z_i's) iff the poly-time referee machine accepts $x, y_1, \ldots, y_{p(n)}$. This formula is satisfiable iff the quantified formula $\exists z_1 \exists z_2 \ldots \exists z_{q(n)} \phi(x, y, z)$ is true. So inserting this quantified formula after the $p(n)$ quantifiers gives us an overall quantified formula

$$\exists y_1 \forall y_2 \ldots \forall y_{p(n)} \exists z_1 \exists z_2 \ldots \exists z_{q(n)} \phi(x, y_1, \ldots, y_{p(n)}, z_1, \ldots, z_{q(n)})$$

that is true iff Alice wins the game. This is a reduction from an arbitrary AP language to QBF, completing the proof that QBF is AP-complete. □

5. The Alternation Theorem

Alternating machine classes turn out to be *equal* to corresponding deterministic machine classes, with the odd behavior that alternating time classes turn into deterministic space classes and vice versa. There is a general form of this result, called the *Alternation Theorem*, that you can prove in the exercises (or look up in Papadimitriou or in Sipser). Here we prove two special cases of the theorem. The two directions of each proof provide four arguments that can be generalized to prove the entire theorem. It is interesting that to a large extent *we have seen each of these four arguments before* in dealing with the traditional machine and circuit complexity classes.

Theorem 2 *(Chandra-Kozen-Stockmeyer)* $\mathrm{AL} = \mathrm{P}$.

Proof We begin by showing $\mathrm{AL} \subseteq \mathrm{P}$, using an argument called the *configuration simulation*. Assume that $A \in \mathrm{AL}$ and that A is defined by the alternating machine

game on a particular nondeterministic machine M that always halts using no more than $c \log n$ space on inputs of length n. Also assume that M has a clock, so that it can never repeat a configuration. Let G_n be a graph with a node for each possible configuration of M on an input of length n. Configurations include state, memory contents, and head positions, so G has only polynomially many nodes.

We convert G into a circuit with a gate for each node. Suppose a node c represents a configuration where Alice controls the machine. The machine in this configuration is looking at exactly one input bit x_i. Say that $d_1, \ldots, d_a$ are the possible configurations to which Alice could move with $x_i = 0$, and $e_1, \ldots, e_b$ are the places she could move with $x_i = 1$. We make the node for c an OR gate with $a + b$ children, each a new binary AND gate. The children of the first a AND gates are $\neg x_i$ and d_j for each j from 1 to a, and the children of the others are x_i and e_j for each j from 1 to b. We do the same for Bob-control configurations, making them AND gates whose children are binary ANDs of literals and other configurations. We make accepting configurations into constant 1-gates, and rejecting into constant 0-gates. We make the start configuration into the output gate. We have designed a circuit, and this circuit accepts the input x exactly if Alice wins the game on M with input x, that is to say, exactly if $x \in A$.

For the other direction of the proof we must show P $\subseteq$ AL. Using a result in Basic Lecture 5, we assume that language A is defined by an L-uniform circuit family where the circuit C_n with n inputs has size polynomial in n. We will define a game called the *circuit game*, based on this circuit, such that the referee can operate in $O(\log n)$ space and that Alice will win the game with input x under optimal play iff $x \in A$.

At any point in the game the *current position* will be a node in C_n. We begin the game at the output gate. If the current position is an AND gate, Bob makes the next move by selecting a child of the current gate to be the new current position. If the current position is an OR gate, Alice chooses a child to be the new current position. If the current position is an input gate, the game ends, with Alice winning if the gate has value 1 and Bob winning if it has value 0.

Clearly this game terminates (as the circuit has no cycles) and can be refereed in log space. (Alice and Bob need to agree on whose turn it is to move, and that each of their moves is actually to a child of the current gate. But because the circuit is L-uniform, these questions can be decided by the log-space referee if a challenge occurs.) We need to show that Alice wins iff the circuit accepts the input x. But if this is true iff the game starts on a node that will evaluate to 1 when x is the input. Alice has only to make sure that whenever she moves, she always moves to a node where this is also true. She can do this because her current OR gate has value 1, so one of its children has value 1. Bob, on the other hand, can never by his moves

change the fact that the current gate has value 1. This is because his AND gate has value 1, so all of its children have value 1. If Alice plays correctly, then, the input gate that is reached will have value 1 and Alice will win. Conversely, if the output gate has value 0, Bob can maintain this fact with the right moves and Alice can do nothing about it with hers, so Bob will reach an input gate with value 0 and he will win. □

Theorem 3 *(Chandra-Kozen-Stockmeyer)* AP = PSPACE.

Proof We begin by proving AP $\subseteq$ PSPACE, using an argument that generalizes the proof that NP $\subseteq$ PSPACE. In any game we have a *game tree*, where nodes represent states of the game and edges represent possible moves. If $A \in AP$, the game to determine whether $x \in A$ has a depth polynomial in the length of x, and nodes that can be labelled by poly-length strings representing configurations of the nondeterministic machine M on which the game is played.

We define a recursive algorithm to search the game tree exhaustively and determine the winner. If the current node of the tree is a leaf (a state of the game where Alice or Bob has just won), simply report the result. Otherwise recursively determine the winner for the subtrees corresponding to each of the children of the current node. If the current node is an Alice-move state, report Alice as the winner iff she wins at least one of the subgames. If it is a Bob-move state, report Bob as the winner iff he wins at least one of the subgames. This algorithm clearly reports the correct winner of the original game. It runs in polynomial space because it recurses to at most polynomial depth (the depth of the tree) and each time it recurses it needs to store only the name of the current node (a poly-length string) on the stack. We have thus decided $x \in A$ in PSPACE.

For the other direction, we have a language $A \in$ PSPACE and we need to define a polynomial-time game that Alice will win iff $x \in A$. Let M be a PSPACE machine deciding A and let G_x be the configuration graph of M on input x. (This graph has a node for each configuration and an edge from each configuration to the configuration that M reaches in one step on input x.) Let $s = 2^{p(n)}$ be the number of nodes in G_x.

As in the proof of Savitch's Theorem, we know that G_x has a path of length at most s from the start configuration to the accepting configuration (assume that M cleans up its memory) iff $x_i nA$. We define a game called the *Savitch game* that Alice will win under optimal play iff this is true. The current position will be two nodes u and v, and a number ℓ for the maximum length of the path that Alice asserts to exist from u to v. Alice's move is to name a node w that she asserts to be the midpoint of this path. Bob's move is to challenge either the assertion that there is a path

from u to w of length at most $\ell/2$, or the assertion that there is such a path from w to v. The challenged assertion becomes the basis of the new current position. After $O(\log s)$ moves, a polynomial number, ℓ becomes 1 and we can decide the game. Alice wins iff there is an edge from u to v, or if $u = v$. The fact that Alice wins iff her path exists follows from the analysis in Savitch's Theorem. To see that we can play in polynomial time, note that there are only polynomially many moves, that each move is a configuration of the PSPACE machine M and is thus polynomial length, and finally that a deterministic poly-time referee can easily check whether there is an edge from u to v or whether $u = v$ at the end. □

6. Alternating Machines and Circuits

The correspondence between alternating logspace machines and poly-size circuits is actually quite robust. Our circuit-based subclasses of P have alternate definitions in terms of alternating machines, which you are asked to verify in the exercises. These were discovered by Ruzzo and were a major motivating factor in the development of the NC hierarchy.

The *depth* of a circuit with bounded fan-in corresponds to the *time* taken by the alternating machine, so that NC^i is the class of languages definable by alternating machines with $O(\log n)$ space and $O(\log^i)$ time. Since for the NC^i classes the depth is sublinear, and we want our languages to depend on the entire input, it is important that we use the *random-access machine* model defined in one of the exercises in Advanced Lecture 4. In this model the machine can write a number from 0 to $n-1$ on a special section of memory and access the bit of the input with that number in one step.

Similarly the depth of an *unbounded* fan-in circuit corresponds to the *number of alternations* of the alternating machine. Thus AC^i is the set of languages definable by alternating machines with $O(\log n)$ space and $O(\log^i n)$ alternations (changes in control from Alice to Bob or vice versa).

Actually working with these models becomes simpler if we prove that without loss of generality, alternating machines can look at their input only once, at the end of the game. That is, the moves of Alice and Bob eventually determine the location of one of the input bits, and the value that Alice asserts it to have. The referee then looks up this bit and decides the game accordingly. It can be shown that such one-look machines can simulate general random-access alternating machines with either the same time bound or the same bound on alternations. Oddly, each of these results requires a different proof and the simulation is apparently impossible if both resources

are to be preserved.

7. Exercises

1. Prove that if any language is complete for PH, then PH collapses.

2. It is obvious that the language PRIMES (the binary encodings of prime numbers) is in coNP. Prove the less obvious fact that it is in NP as well. (Hint: We need a proof of the primality of x that can be written down as a string whose length is polynomial in n, the number of bits in x. We know that x is prime iff there exists a number g such that $1 < g < x$, $g^{x-1} = 1$ modulo x, and $g^j \neq 1$ modulo x whenever $0 < j < p-1$. Can you find a short string that proves these facts?)

3. A boolean circuit C *projection simulates* another circuit D if it is possible to put constants, D-inputs, and negated D-inputs into the inputs of C and get the same answer as D on all possible settings of the D-inputs. Show that the language of pairs of circuits $\langle C, D \rangle$ such that C projection-simulates D is in PH.

4. Prove that if $s(n) = \Omega(\log n)$ and $s(n)$ is space constructible, $\text{ASPACE}(s) \subseteq \text{DTIME}(2^{O(s)})$. (Hint: Use a marking algorithm.)

5. Prove that with the same conditions on $s(n)$, $\text{DTIME}(2^{O(s)}) \subseteq \text{ASPACE}(s)$. (Hint: Play the circuit game.)

6. Prove that for $t(n) = n^{\Omega(1)}$, $\text{ATIME}(t) \subseteq \text{DSPACE}(t)$. (Hint: Exhaustively search the game tree.)

7. Prove that for the conditions on $s(n)$ above, $\text{DSPACE}(s) \subseteq \text{ATIME}(s^2)$. (Hint: Play the Savitch game.)

8. Show that for $i \geq 2$, NC^i consists of the languages of alternating machines that run in $O(\log n)$ space and $O(\log^i n)$ time. Assume L-uniformity. (Hint: In one direction, play the circuit game, erasing your work as needed. In the other, use the configuration simulation but make sure the machine only queries the input once.)

9. (Extra Credit) Prove the $i = 1$ case of the result above. Now you should assume that the circuits are to have their extended connection language (alluded to in Advanced Lecture 5) in $\text{DTIME}(\log n)$.

10. Show that for $i \geq 1$, AC^i consists of the languages of alternating machines that run in $O(\log n)$ space and $O(\log^i n)$ alternations between control by the two players. (Hint: The circuit game and the configuration simulation will both work with the necessary adjustments. But note that we can't afford to simulate every configuration of the machine by a node.)

11. (Extra Credit) Show the $i = 0$ case of the result above, using DLOGTIME uniformity.