CS 810: Introduction to Computational Complexity Theory 03/11/2003

Lecture 15: Furst-Saxe-Sipser Theorem

Instructor: Jin-Yi Cai Scribe: Mike DeCoster, Vinod Ganapathy and Charles Kahn

In the next few lectures, we will show that constant depth circuits of "small" size cannot computer parity. In this lecture, we will assume this result and show that there is an oracle that separates PH from PSPACE.

1 An Oracle to Separate PH from PSPACE

Furst-Saxe-Sipser considered the question of construing an oracle that separates PH from PSPACE. They described a framework where in, circuit lower bounds for the parity function can be used to construct the desired oracle. In that context, they proved that constant depth circuits of polynomial size cannot compute the parity function. This result shows that $\mathrm{AC}^0 \neq \mathrm{NC}^1$. But, stronger lower bounds were needed to construct the desired oracle. Subsequently, the required lower bounds were obtained by Yao, there by proving the following theorem.

Theorem 1. *There is an oracle A such that, for any $d \geq 0$, $\Sigma_d^{p,A} \neq \mathrm{PSPACE}^A$. That is, $\mathrm{PH}^A \neq \mathrm{PSPACE}^A$.*

Following these developments, Cai and, independently, Hastad improved the lower bounds further. Cai showed that constant depth circuits of "small" size cannot even approximate the parity function. As a consequence, he showed that, with probability one, a random oracle separates PH from PSPACE. Here, we shall assume certain circuit lower bounds for parity function and prove Theorem 1. (The assumed lower bounds will be proved in subsequent lectures.)

1.1 Overview

We start with the definition of parity function.

Definition 1. *The parity function over n boolean variables is given by* $\mathrm{Parity}_n(x_1, x_2, \ldots, x_n) = \sum x_i$ *(mod 2).*

The lower bound result we will use is as follows.

Theorem 2. *For sufficiently large n,* Parity_n *cannot be computed by a depth-d circuit of size $\leq 2^{(0.1)n^{1/d}}$.*

In the rest of the lecture, we will discuss the proof of Theorem 1. For any language A, we define an auxiliary language Parity_A as follows.

Definition 2. $\mathrm{Parity}_A = \{1^n | \#$ *of strings in A of length n is odd*$\}$.

Obviously, for any A, $\mathrm{Parity}_A \in \mathrm{PSPACE}^A$. Our goal is to construct an A such that $\mathrm{Parity}_A \notin \Sigma_d^{p,A}$, for any d. We will first overview the proof idea.
There is a close relationship between Σ_d^p oracle machines and constant depth circuit. Consider a Σ_d^p oracle machine and input 1^n. (To be formal, one can think of a polynomial time alternating machine with d alternations. Or equivalently, think of the d-alternating-quantifier characterization and

consider the underlying polynomial time oracle Turing machine with inputs 1^n and d "certificates".) Whether the machine accepts 1^n or not is determined by the answers to its oracle queries. Because of the polynomial time bound, the machine can ask only queries of polynomial length. In other words, there is a polynomial $p(n)$ such that the acceptance of 1^n is determined by the strings of length $\leq p(n)$ in the oracle set. Let $N = \sum_{i=0}^{p(n)} 2^i$ be the total number of strings of length $\leq p(n)$. Then, we can encode the relevant portion of the oracle set by an N-bit long string (bit i will be 1, if the lexicographically i^{th} string is in the oracle set). Thus, the acceptance of 1^n is a function with an N-bit long string as its argument. That is, the output of the machine is a function from $\{0,1\}^N$ to $\{0,1\}$. We shall argue that this function can be computed by a circuit of depth $d+1$ and size $N^{(\log N)^c}$, where c is a constant determined by the (polynomial) running time of the machine. Its now time to construct the required oracle A. First enumerate all Σ_d^p oracle machines (for all d). We then consider each machine, in turn and make sure that it does not compute Parity$_A$. Suppose we have a Σ_d^p oracle machine, for some d. It can ask only polynomially long queries, say $p(n)$. We choose an n such that $N^{(\log N)^c} \leq 2^{(0.1)2^{n/(d+1)}}$, where $N = 2^{p(n)}$. Since d is a constant and the function on the left hand side grows slower than the one on the right, it is clear that we can find such n and N. (Notice that the value of the right hand side is the lower bound from Theorem 2 for size of circuits of depth d computing Parity$_{2^n}$). As we said, the computation of the machine on 1^n can be captured by a circuit C of depth $d+1$ and size $N^{(\log N)^c}$. This circuit has N input bits, out of which 2^n bits correspond to the membership of strings of length n in the oracle. We will hard-wire the other bits appropriately (according to the oracle constructed so far) and consider the resulting circuit with 2^n inputs. By the lower bound theorem, this circuit of depth d is too small to compute the parity over 2^n bits. Thus, there exists a 2^n bit long string on which the circuit fails to compute parity. We construct the oracle A at length n according to this string (if i^{th} bit is 1, then the lexicographically i^{th} string of length n is added to the oracle). It should be clear that the Σ_d^p oracle machine under consideration fails to compute Parity$_A$. In particular, it would fail on input 1^n.

1.2 Equivalence of $(\Sigma_d^p)^A$ and constant depth circuits

We mentioned that circuits can be used to model the computations of Σ_d^p oracle machines This connection is discussed in detail here.

We will describe the idea using Σ_1^p oracle machines. Consider any such machine with input 1^n. Its running time is bounded by some polynomial function on input size. Such a machine can ask queries to the oracle, and use the answer to decide the next state that it is going to enter. However, we can alter its operation a little bit – when the machine enters a "query" state, we will not consult the oracle A to answer to query. Instead, we will guess 0 or 1 corresponding to non-membership and membership respectively of the query string in the oracle A. Finally, at the end of the computation, we will check to see if the answers to the queries that were guessed were indeed correct.

More specifically, consider the diagram in Figure 1 below: since we are dealing with a Σ_1^p oracle machine, at the root of the computation tree, there are $2^{p(n)}$ choices, for some polynomial $p(n)$, to be checked before the computation can be accepted (recall that Σ_1^p is $x|\exists^P y P(x,y) = 1$, the choice at the root of the computation tree is the choice that can be made for y). At each point in the computation where a query was asked, we do not ask a query, but instead guess the answer, and remember the answer we had guessed. This is shown in Figure 1 by the branch with both possible answers for the query string q. At the end of a computational path, all the queries are checked

with the oracle to see if the guesses were correct. The computational path accepts if and only if all queries were answered correctly.

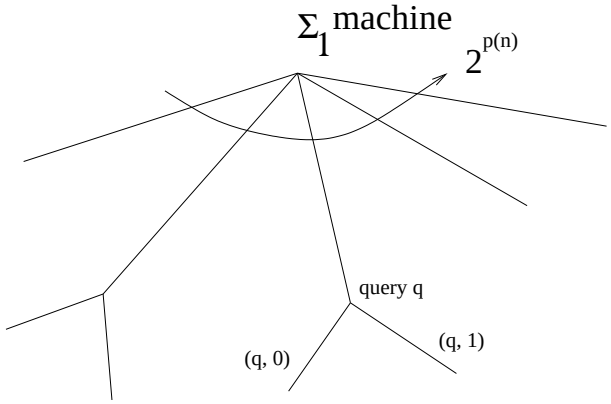

Figure 1: A $(\Sigma_1^p)^A$ machine with delayed queries

We can now construct the required circuit. Input to the circuit are the bits encoding the oracle (upto the relevant length). Refer to Figure 2 for a pictorial depiction. The gate on top is an OR-gate because a Σ_1^p machine accepts if one of the computational paths accepts. The gate at the second level is an AND-gate because we need to look at the input bits (that encode the oracle) and check whether the guessed answers are indeed correct. All guesses have to be correct for a computational path to accept. The original NP Turing machine with a certain oracle accepts 1^n iff the circuit accepts the encoding of the oracle given as input. Note that the fan-in of these AND gates need to be only polynomial in n, though the OR gate should have exponential fan-in. Depth of the circuit is 2 and its size is $2^{\text{poly}(n)}$. But, size of the input to the circuit is $N = 2^{\text{poly}(n)}$ and thus, its size in terms of its input length is $N^{(\log N)^c}$, for some constant c.

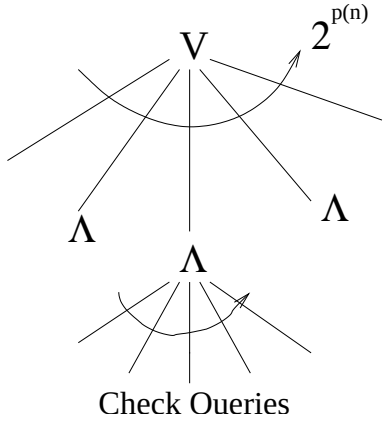

Figure 2: A boolean circuit equivalent of the $(\Sigma_1^p)^A$ machine

Now consider a (Π_1^p) oracle machine with input 1^n. Just as in the case of a (Σ_1^p) machines, we have $2^{p(n)}$ computational paths to check at the root of the computational tree, however all the paths have to accept in this case. To understand how the boolean circuit is constructed for the (Π_1^p) machine, all we need is that the Π_1 language is the complement of the Σ_1 language. Hence, constructing the boolean circuit for the Π_1 is just like constructing the boolean circuit for the Σ_1 machine, except

that we have a negation at the very top. If we push the negation down to the leaves, we have a boolean circuit of the kind shown in Figure 3 which is the boolean circuit for the Π_1 machine. The basic idea is that the queries that are checked at the leaves (the "delayed" queries) are the negations of the queries that would have been asked in case of a Σ_1 machine. Hence, by OR-ing together negations of these queries, we have ensured that the circuit representing the Π_1 machine accepts an input if and only if the circuit representing the Σ_1 machine rejects. In other words, the queries that are asked are as though we wanted to test membership of the input string in the Σ_1^P language. By "and"ing together the negations of these queries, we ensure that we reject a computational path if and only if the queries were both answered correctly, which is the requirement for the string to belong to the Σ_1 language.

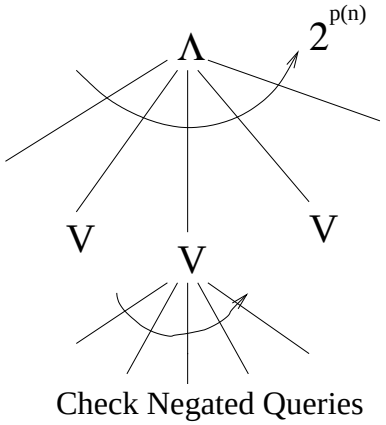

Figure 3: A boolean circuit equivalent of the $(\Pi_1^P)^A$ machine

Idea behind constructing circuits for general Σ_d^p oracle machines is similar. Computation of the Σ_d^p machines on input 1^n can be captured by circuits of depth $d+1$. At the root of the circuit will be an OR gate, the next level will be made of AND gates. AND gates and OR gates will alternate as we go down to the leaf level. The gates in non-leaf levels will have fanin-in exponential in n. At the leaf level we need "smaller" OR gates with fan-in polynomial in n. Size of the circuit will be $2^{\text{poly}(n)}$. Input to the circuit is of length $2^{\text{poly}(n)}$. (The textbook gives a formal proof using induction.)

1.3 Constructing the Oracle

In our outline, we discussed how to construct the required oracle. A more formal description is given below. Let $M_1, M_2, M_3, \ldots$ be an enumeration of all Σ_d^p machines, for $d \geq 0$. (Formally, one can consider all alternating Turing machines with some constant number of alternations). We construct an oracle A such that none of these machine compute Parity_A.

1. $n_0 \leftarrow 1$ and $A \leftarrow$ emptyset.
2. For each machine M_i do
   (a) Let $p(n)$ be the (polynomial) running time of M_i. Let $s(n)$ be the bound on the size of the circuit that "captures" computation of M_i on input 1^n (as discussed in Section 1.2). Note that $s(n) = 2^{\text{poly}(n)}$.

(b) Choose an n' such that $n' > n_0$ and $s(n') \geq 2^{(0.1)2^{n'/(d+1)}}$, where d is the number of alternations used by M_i. (So that, circuits of size $s(n')$ and depth $d+1$ cannot compute parity on $2^{n'}$ bits.)

(c) Construct the circuit C that "captures" computation of M_i on input $1^{n'}$. This circuit has $2^{\text{poly}(n')}$ input bits (encoding of relevant segment of the oracle). Of these, $2^{n'}$ bits correspond to strings of length n'. Of the remaining, some correspond to strings of length $< n'$ and others correspond to strings of length $> n'$. Hard-wire first kind of bits according to the oracle constructed so far. Hard-wire the second kind of bits to zero.

(d) The circuit in hand has $2^{n'}$ bits. Its size is too small to compute parity on $2^{n'}$ bits. Find a $2^{n'}$-bit string w where this circuit fails to compute parity.

(e) For each $1 \leq i \leq 2^{n'}$, if the i^{th} bit of w is 1, add the n-bit binary representation of i. to the set A.

(f) Machine M_i could have asked for queries of length at most $p(n')$. So set $n_0 \leftarrow p(n')$ (this will ensure that we answered all the machines consistently).