CS 710: Complexity Theory 2/16/2010

Lecture 7: Nonuniformity

Instructor: Dieter van Melkebeek Scribe: Mushfeq Khan & Chi Man Liu

The previous lecture introduced the polynomial-time hierarchy (PH). We encountered several characterizations of the complexity classes Σ_n^{P} and Π_n^{P} that make up PH, and one such characterization is in terms of alternating turing machines. In the first part of this lecture, we make use of alternation to obtain simultaneous time-space lower bound results for SAT.

In the second part of the lecture, we introduce the notion of nonuniform computation. Among the nonuniform models discussed are boolean circuits, branching programs, and uniform machines with advice.

1 Time-Space Lower Bounds for SAT

Although it is unlikely that SAT can be solved in linear time, we have yet to rule out even this trivial lower time bound. Similarly, we have yet to rule out the possibility that SAT is in L. However, if we take time and space into consideration simultaneously, we can obtain some nontrivial lower bounds for SAT, i.e. constraining space usage will impose nontrivial lower bounds for time usage and vice versa.

In what follows, t and s always denote functions from $\mathbb{N}$ to $\mathbb{N}$, where t is assumed to be time-constructible and s space-constructible.

Definition 1. $\mathrm{DTISP}(t, s)$ *is the class of languages that can be accepted by a DTM using at most* $t(n)$ *steps and* $s(n)$ *space, where* n *is the size of the input.*

Note that $\mathrm{DTISP}(t, s)$ is not the same as $\mathrm{DTIME}(t) \cap \mathrm{DSPACE}(s)$, since if a language L is in the former, then L must be recognized by a *single* machine with the given time and space characteristics.

In the following lemma, the assumption that SAT is "easy" allows us to obtain a limit on the time and space overhead of simulating a nondeterministic machine running in polynomial time on a deterministic machine.

Lemma 1. *Let* $c \geq 1$ *and* $d > 0$. *If* $SAT \in \mathrm{DTISP}(n^c, n^d)$ *then for* $a \geq 1$,

$$\mathrm{NTIME}(n^a) \subseteq \mathrm{DTISP}(n^{ac}\,\text{poly-log}(n), n^{ad}\,\text{poly-log}(n))$$

Proof. We first consider the case when $a = 1$. This amounts to showing that $\mathrm{NTIME}(n) \subseteq \mathrm{DTISP}(n^c\,\text{poly-log}(n), n^d\,\text{poly-log}(n))$.

In a previous lecture, we saw that SAT is complete for the class NQLIN under $\leq_m^{\mathrm{QLIN}}$ and that given such a reduction, we can actually compute each bit of the resulting boolean formula in polylogarithmic time and logarithmic space in the size of the original input. We shall exploit this fact for the current reduction. Given a language $L \in \mathrm{NTIME}(n) \subseteq \mathrm{NQLIN}$, let M be a quasi-linear time reduction of L to SAT. Also let N be a deterministic algorithm for SAT with the time and space characteristics assumed in the hypothesis. Now if x is a string of length n,

then $M(x)$ is a SAT instance of length $O(n \operatorname{poly-log}(n))$. Then the running time of N on $M(x)$ is $O(n^c \operatorname{poly-log}(n))$, while the space usage is $O(n^d \operatorname{poly-log}(n))$. Computing $M(x)$ all at one go, however, is not feasible, since this may exceed our space constraint (d may be less than 1). Instead we compute the bits of $M(x)$ on demand. Computing each bit requires $O(\operatorname{poly-log}(n))$ time, an overhead which is absorbed into the polylogarithmic factor of N's running time. Similarly, there is a logarithmic space overhead for computing each bit on demand, but this too is absorbed into the polylogarithmic factor of N's space usage.

For the case when $a > 1$, we utilize a technique called "padding" to reduce the problem to the case when $a = 1$. Suppose that $L \in \text{NTIME}(n^a)$. We define:

$$L' = \{x\#0^{|x|^a} \mid x \in L\}$$

where $\#$ is a special symbol not in the alphabet of L. Clearly, $L' \in \text{NTIME}(n)$, since given a string y, we can check in linear time if it is of the form $x\#0^{|x|^a}$, then run the algorithm for recognizing L on the initial segment x. This takes $O(|x|^a)$ time, so the resulting algorithm runs in linear time. Now by our result in the case when $a = 1$, $L' \in \text{DTISP}(n^c \operatorname{poly-log}(n), n^d \operatorname{poly-log}(n))$. Let T be a deterministic machine that decides L' within these time and space bounds and let $\text{pad}(x)$ denote $x\#0^{|x|^a}$. Given a string x of length n, the length of $\text{pad}(x)$ is $O(n^a)$, so the running time of T on $\text{pad}(X)$ is $O((n^a)^c \operatorname{poly-log}(n^a)) = O(n^{ac} \operatorname{poly-log}(n))$ and similarly, the space usage is $O(n^{ad} \operatorname{poly-log}(n))$. As before, computing $\text{pad}(x)$ all at once is not feasible, so we compute it bit by bit on demand. It is not hard to see that this may be accomplished without incurring more than a polylogarithmic time overhead and a logarithmic space overhead, both of which may be absorbed into the polylogarithmic factors of T's running time and space usage. □

The next lemma shows that deterministic computation may be "sped up" by simulating it on an alternating turing machine.

Lemma 2. $\text{DTISP}(t, s) \subseteq \Sigma_2\text{TIME}(\sqrt{\text{ts}})$

Proof. Let L be a language in $\text{DTISP}(t, s)$, and let M be a machine that recognizes L within these time and space constraints. Without loss of generality, we may assume that M has a unique accepting configuration c_1. Fix an input string x of length n. The computation tableau of M on x has $t(n)$ rows and $O(s(n))$ columns. Let c_0 be the initial configuration of M on input x. Then $x \in L$ if and only if $c_0 \vdash_{M,x}^{t(n)} c_1$. In the proof of $\text{NSPACE}(s) \subseteq \text{DSPACE}(s^2(n))$, we guessed an intermediate configuration $c_{1/2}$ and checked recursively that $c_0 \vdash_{M,x} c_{1/2}$ and $c_{1/2} \vdash_{M,x} c_1$. We use a similar technique here, but this time, instead of splitting up the tableau into two parts, we split it up into b parts, where we defer fixing the value of b for now. Let $c^{(0)} = c_0$ and $c^{(b)} = c_1$. The computation proceeds by guessing $b - 1$ intermediate configurations $c^{(1)}$, $c^{(2)}$, ..., $c^{(b-1)}$, and by verifying for each $0 \leq i < b$, that $c^{(i)} \vdash_{M,x}^{t(n)/b} c^{(i+1)}$. We thus obtain:

$$x \in L \iff (\exists c^{(1)}, \ldots, c^{(b-1)})(\forall 0 \leq i < b) \left(c^{(i)} \vdash_{M,x}^{t(n)/b} c^{(i+1)}\right)$$

The matrix of this computation is computable in time $O(s(n) + t(n)/b)$: We simply simulate M starting from $c^{(i)}$ for $t(n)/b)$ steps. Assuming a constant simulation overhead, this takes $O(t(n)/b)$ steps plus an additional $O(s(n))$ steps to copy $c^{(i)}$ onto the work tape and to verify that the final configuration reached is $c^{(i+1)}$. So this is in fact a Σ_2 computation. Next, we analyze the time required by an alternating machine to perform this computation.

The machine first guesses the intermediate configurations (corresponding to the existential part of the above description). Each such configuration has size $O(s(n))$ and there are $b-1$ of them, so in total the machine spends $O(b.s(n))$ steps in an existential state. Next, the machine switches to a universal state, guessing a value for i, and this clearly takes $O(\log(b))$ steps. Combining the above with the time required to check the matrix, we obtain a total running time of $O(b.s(n) + t(n)/b)$. The value of b that optimizes this running time is $\sqrt{t(n)/s(n)}$. The result follows immediately. □

The preceding lemmas allow us to prove the main theorem:

Theorem 1. *If SAT in* $\mathrm{DTISP}(n^c, n^d)$ *then* $c(c+d) \geq 2$.

Proof. Assume that $\mathrm{SAT} \in DTISP(n^c, n^d)$. By Lemmas 1 and 2, we have:

$$\begin{aligned} \mathrm{NTIME}(n^a) &\subseteq \mathrm{DTISP}(n^{ac}\,\text{poly-log}(n), n^{ad}\,\text{poly-log}(n)) \\ &\subseteq \Sigma_2\mathrm{TIME}(\mathrm{n}^{\frac{\mathrm{a(c+d)}}{2}}\,\text{poly-log(n)}) \\ &\subseteq \Sigma_2\mathrm{TIME}(\mathrm{n}^{\frac{\mathrm{a(c+d)}}{2}+\mathrm{o}(1)}) \end{aligned}$$

Let $\gamma(a) = \frac{a(c+d)}{2}$ and suppose $L \in \Sigma_2\mathrm{TIME}(\mathrm{n}^{\gamma(\mathrm{a})+\mathrm{o}(1)})$. Then,

$$x \in L \iff \exists y \in \{0,1\}^{|x|^{\gamma(a)+o(1)}} \forall z \in \{0,1\}^{|x|^{\gamma(a)+o(1)}} (R(x,y,z)) \tag{1}$$

where $R(x,y,z)$ is a predicate computable in time linear in its inputs. Now the predicate in indeterminates x and y given by

$$\forall z \in \{0,1\}^{|x|^{\gamma(a)+o(1)}} (R(x,y,z))$$

determines a Π_2 language L' of pairs $\langle x, y\rangle$ and is recognized by a Π_2 alternating machine in time $O(|x|+|y|) = O(|x| + |x|^{\gamma(a)+o(1)})$ (i.e. linear in the length of its inputs, but not necessarily linear in $|x|$ alone). The complement of L' is a Σ_1 language recognizable in linear time and, by another application of Lemma 1, is in $\mathrm{DTIME}(n^c, n^d)$. The latter class is closed under complementation, so L' itself is in $\mathrm{DTIME}(n^c\,\text{poly-log}(n), n^d\,\text{poly-log}(n)) \subseteq \mathrm{DTIME}(n^{c+o(1)}, n^{d+o(1)})$. So L' is recognizable by a deterministic machine running in time $O((|x| + |x|^{\gamma(a)+o(1)})^{c+o(1)})$. Restoring the existential quantification of z in the equivalence 1 brings us back to a description of L and also shows that L is a Σ_1 language recognizable in $O((n + n^{\gamma(a)+o(1)})^{c+o(1)}$ time. Now we pick a_0 large enough so that

$$\Sigma_1\mathrm{TIME}((\mathrm{n} + \mathrm{n}^{\gamma(\mathrm{a}_0)+\mathrm{o}(1)})^{\mathrm{c}+\mathrm{o}(1)}) = \Sigma_1\mathrm{TIME}(\mathrm{n}^{\gamma(\mathrm{a}_\mathrm{o})\mathrm{c}+\mathrm{o}(1)})$$

In so doing we obtain the containment $\mathrm{NTIME}(n^{a_0}) \subseteq \mathrm{NTIME}(n^{\gamma(a_0)c+o(1)})$. By the Nondeterministic Time Hierarchy Theorem, $\gamma(a_0)c \geq a_0$. By the definition of γ, this implies that $c(c+d) \geq 2$. □

Corollary 1. *Suppose* $SAT \in DTISP(n^c, n^d)$ *for constants* c *and* d.

1. *If* $d < 1$ *then* $c > 1$;
2. *if* $c < \sqrt{2}$ *then* $d > 0$.

The first statement of the corollary says that if a deterministic algorithm for SAT were to run in sub-linear space (e.g. logarithmic space), then it would necessarily require super-linear time. The second statement shows, in particular, that if a deterministic algorithm for SAT were to run in linear time, then its space usage would necessarily be super-logarithmic.

We do, in fact know how to prove a slighly stronger statement - if there were a deterministic algorithm for SAT that ran in logarithmic space, then this would preclude the possibility of any deterministic algorithm for SAT that runs in linear time.

Theorem 2. *$SAT \notin \mathrm{L} \cap \mathrm{DTIME}(n)$.*

2 Nonuniformity: Motivation

The computational models we have seen thus far, such as Turing machines (deterministic or not), are capable of handling problems where the same algorithm works for inputs of all lengths. We call these *uniform* models of computation. In this section, we introduce models capable of handling problems where for different input lengths, we may require a different algorithm for recognizing strings of that length. In general, we will have no computable way (or at least, not computable within certain resource bounds) of obtaining from a length $n \in \mathbb{N}$ the algorithm that works for strings of that length. We call such models *nonuniform*.

Nonuniformity may seem a bit odd at first sight. Indeed, uniformity is a more natural form of computation because it resembles algorithms – finite procedures for all possible inputs. So why are we interested in nonuniform models? There are in fact close relationships between uniform and nonuniform models. By studying nonuniform models, we may be able to derive lower bound or hardness results for uniform models. We will see some of these relationships in Section 4.

3 Nonuniform Models of Computation

We introduce three forms of nonuniform computation. The first two are nonuniform models — boolean circuits and branching programs. Both of them solve instances of a specific problem with a fixed input length. For simplicity we assume that languages are defined over the binary alphabet $\{0, 1\}$. Boolean circuits are useful in analyzing the uniform time complexity of problems; branching programs are more useful for space complexity.

The third model is an extension of the uniform model, where we allow a nonuniform ingredient known as *advice*. Advices are similar to certificates – both are additional information which speed up computation. The difference between certificates and advices is that while certificates can vary from input to input, all inputs of the same length share the same advice. In other words, the advice for an input only depends on its length.

We briefly discuss the three models in the following.

3.1 Boolean Circuits

We define boolean circuits similarly to real-world electronic circuits. A boolean circuit is a directed acyclic graph where each node is either a logic gate or an input. An input node has no incoming edges. One of the gate nodes is designated as the output node which has no outgoing edges. Label the input nodes x_1, x_2, ..., x_n. Given input string $x \in \{0, 1\}^n$, the boolean circuit computes its output (a single bit) as follows. The bits of x are first copied to the corresponding input nodes.

Then, in topological order, each logic gate receives bits from its incoming edges, performs the boolean operation on the bits, and sends the output bit along all its outgoing edges. The output of the circuit is the bit output by the output node.

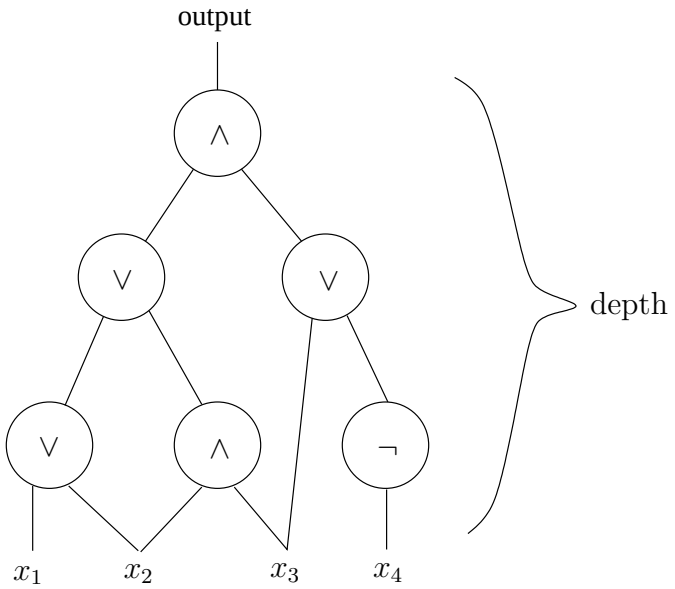

Figure 1: A boolean circuit.

For any function $f : \{0,1\}^n \to \{0,1\}$, we say that a boolean circuit B realizes f if the output of B matches $f(x)$ for every input $x \in \{0,1\}^n$. In this course we consider circuits with AND, OR, and NOT gates with a bounded (say 2) fan-in. We define the *circuit size* $C(f)$ of a function f to be the size of the smallest (in terms of number of nodes) circuit realizing f.

We can also use boolean circuits to accept a language L. Instead of using one circuit for all possible inputs as in uniform computation, we must use a different circuit for inputs of different lengths. Formally, we say that a family of circuits $\{B_i\}$ accepts a language L if for every $n \in \mathbb{N}$, B_n realizes L_n, where L_n is the characteristic function of L restricted to inputs of length n. We define the *circuit complexity* $C_L(n)$ of L by $C_L(i) = C(L_i)$ for all $i \in \mathbb{N}$.

The circuit complexity of a language can be a good measure of its uniform time complexity. The reason is that, given a boolean circuit, we can simulate its computation by a uniform machine in linear time. Likewise, given a Turing machine that runs in time t, we can "encode" its transition function into a boolean circuit of size quadratic in t. Another measure is the *depth* of a circuit, which equals the length of the longest path from an input to the output node. Circuit depth is not as comparable to uniform time complexity as circuit size, since for any specific language and an input length n, we can convert membership into a CNF or DNF, and build a constant-depth circuit with fan-in 2^n. This circuit can be converted into an equivalent linear-depth circuit with a bounded fan-in of 2.

Note: For a language L, its circuit complexity $C_L(n)$ can be considerably smaller than its uniform time complexity $t_L(n)$, due to the "one algorithm for all inputs" restriction imposed on uniform computation.

3.2 Branching Programs

A branching program P is a directed acyclic graph where each node is labeled x_1, x_2, ..., x_n, ACCEPT, or REJECT. Each node (except those labeled ACCEPT or REJECT) has exactly two outgoing edges where one of them is marked 0 and the other 1. One of the nodes is designated as the start node. The computation of P on input $x \in \{0,1\}^n$ is as follows: Starting from the start node, look at its label x_i and follow the appropriate edge to the next node by looking at the input string. Repeat until it reaches an ACCEPT node or a REJECT node. Note that similar to a boolean circuit, a branching program only works for inputs with a specific length. For a function $f : \mathbb{N} \to \mathbb{N}$, we define its *branching program complexity* $BP(f)$ to be the size of the smallest branching program that accepts f.

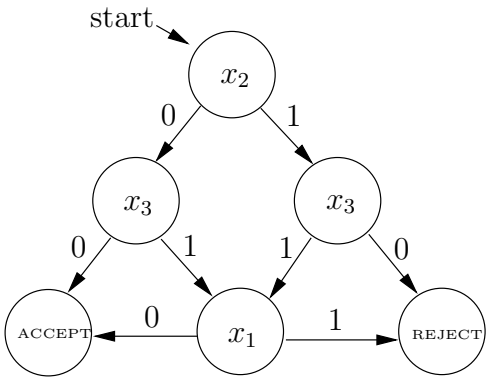

Figure 2: A branching program.

We can use a family of branching programs to accept a language. The branching program complexity $BP_L(n)$ of a language L is defined by $BP_L(i) = BP(L_i)$ for all $i \in \mathbb{N}$.

The branching program complexity of a language can be a good measure of its uniform space complexity. Suppose we are given a branching program with v nodes. We can simulate its computation on a Turing machine using $O(\log v)$ space — the space used to store the index of the current node. Now suppose we have a Turing machine that uses space s and runs in time t. We can construct a layered branching program (a branching program whose nodes can be partitioned into layers such that every edge goes from one layer to the next layer) with $t+1$ layers and $O(2^s)$ nodes in each layer. Each node represents a machine configuration. The first layer contains a single node representing the initial configuration. This node is the start node. Each node in subsequent layers is labeled by the variable corresponding to the current input tape head position in its configuration. Edges indicate valid transitions between successive configurations. A node becomes an ACCEPT (respectively REJECT) node if it represents an accepting (respectively rejecting) configuration. The size of this branching program is $O(2^s t)$. We see from the above simulations that there is a roughly logarithmic relationship between the size of a branching program and space usage of its corresponding Turing machine. Another possible candidate for measurement is the *width* of a (layered) branching program, which equals the maximum number of nodes in a layer.

Note: For a language L, its branching program complexity $BP_L(n)$ can be considerably smaller than its uniform space complexity $s_L(n)$.

3.3 Uniform Models with Advice

The two models discussed above use a different machine for each input length, resulting in an infinite family of machines. Our third model is similar to uniform models in that it uses a single machine for all input lengths. We give extra power (nonuniformity) to the uniform machine by allowing access to *advice*, pieces of additional information which enable the machine to handle input strings of a particular length.

Definition 2. *Let $a(n)$ be a function from $\mathbb{N}$ to $\mathbb{N}$. Let $\mathcal{C}$ be a class of languages. We define the class*

$$\mathcal{C}/a(n) = \{L \mid \text{ There exist } L' \in \mathcal{C} \text{ and a sequence of strings } (y_n)_{n\in\mathbb{N}}, \text{ such that } |y_n| < a(n) \text{ and } x \in L \iff \langle x, y_{|x|}\rangle \in L'\}$$

The sequence $(y_n)_{n\in\mathbb{N}}$ in the above definition is called the *advice sequence*.

To illustrate the power of machines which have access to advice strings, let's consider the language

$$L = \left\{0^{\overline{x}} \mid x \in \text{HALT}\right\}$$

where $\overline{x}$ denotes the natural number represented by the binary string $1x$. This is clearly an incomputable set. However, a machine with access to advice can compute it: let the advice sequence encode the characteristic function of the set of natural numbers that are lengths of strings in L. In fact, each advice string in such a sequence would be of length 1.

4 Connections Between Uniform and Nonuniform Models

In this section, we present a few results relating nonuniform models to uniform complexity classes.

P/poly is the class of all languages which can be computed in polynomial time using polynomial-length advice. Similarly, L/poly is the class of all languages computable in logarithmic space using polynomial-length advice. The next two theorems show relationships between nonuniform models and these complexity classes.

Theorem 3. $\text{P/poly} = \{L \mid C_L(n) \text{ is polynomially bounded }\}$.

Proof Sketch. $\subseteq$: Suppose $L \in \text{P/poly}$. Let M be the polynomial-time algorithm with access to advice that recognizes L. For a given input length n, let C_n be a circuit that computes M. The size of C_n is polynomially bounded in n. Let a_n be the advice string for input length n. By hardcoding a_n into C_n, we obtain the desired circuit.
$\supseteq$: If L has polynomial-size circuits, we can use the descriptions of the circuits as advice. □

A similar argument leads to the following:

Theorem 4. $\text{L/poly} = \{L \mid BP_L(n) \text{ is polynomially bounded }\}$.

We may also consider uniform boolean circuits. A family of circuits $\{B_i\}$ is *uniform* if there exists a uniform machine which, given n, outputs the description of B_n in time polynomial in n. Uniform branching programs are defined similarly, except that the notion of uniformity here is not standardized and may differ by context. The following two theorems say that uniform circuits and branching programs are indeed uniform.

Theorem 5. $\mathrm{P} = \{L \mid L \textit{ has uniform polynomial-size circuits }\}$.

Proof Sketch. $\subseteq$: Let $L \in \mathrm{P}$ and M be a DTM accepting L. We can construct a circuit for inputs of length n by hardwiring valid transitions and constraints in the computation tableau of M. Since M runs in polynomial time, its computation tableau has polynomial size. The resulting circuit also has polynomial size and can be computed in polynomial time. $\supseteq$: On input x, the uniform machine computes $B_{|x|}$ in polynomial time, then simulates the computation of $B_{|x|}$ on x also in polynomial time. □

Theorem 6. $\mathrm{L} = \{L \mid L \textit{ has uniform polynomial-size branching programs }\}$.

5 Next Time

Consider the class $\mathrm{P/poly}$ of problems which are solvable in polynomial time given some advice with polynomial length. If for some NP-complete problems, we happened to have computed the advice strings as well as polynomial-time algorithms making use of them, then we could solve these NP-complete problems efficiently. Next lecture we will show that this is unlikely to be the case; in particular, we will prove that if $\mathrm{NP} \subseteq \mathrm{P/poly}$, then the polynomial-time hierarchy collapses to the second level.