CS 810: Complexity Theory	2/5/2007

Lecture 7: Nonuniformity

Instructor: Dieter van Melkebeek — Scribe: Chi Man Liu

Last lecture we studied alternations and the related polynomial-time hierarchy (PH). We gave four different characterizations of the alternation model, and also introduced some alternating time and space complexity classes. In the first part of this lecture, we make use of alternations and prove some time-space lower bound results for SAT.

In the second part of the lecture, we introduce the notion of nonuniform computation. Nonuniform models we discuss include boolean circuits, branching programs and uniform machines with advice. We conclude the lecture by looking at some connections between uniform and nonuniform models.

1 Time-Space Lower Bounds for SAT

Whether SAT can be solved in polynomial time remains an open question until the P = NP question has been resolved. Although it is very unlikely that SAT can be solved in linear time, none of the known results have ruled out this seemingly ridiculous possibility. More strikingly, it is possible (though not likely) that SAT is in L. However, if we put both time and space into consideration, we can prove some nontrivial lower bounds for SAT.

Definition 1. *Let $t(n)$ and $s(n)$ be functions from $\mathbb{N}$ to $\mathbb{N}$. The class* $\mathrm{DTISP}(t(n), s(n))$ *is defined to consist of all languages L that can be accepted by a DTM using at most $t(n)$ steps and $s(n)$ space.*

Note that $\mathrm{DTISP}(t(n), s(n))$ is not the same as $\mathrm{DTIME}(t(n)) \cap \mathrm{DSPACE}(s(n))$ in general.

Our first theorem says that no algorithm for SAT is both time- and space-efficient. The theorem is proved by considering nondeterministic linear time. We first prove a lemma that says that SAT essentially captures the power of nondeterministic linear time.

Lemma 1. *If $SAT \in \mathrm{DTISP}(n^c, n^d)$ then $\mathrm{NTIME}(n) \subseteq \mathrm{DTISP}(n^c \text{ poly-log}(n), n^d \text{ poly-log}(n))$.*

Proof. Recall that in Lecture 3 we proved that SAT is complete for NQLIN under $\leq_m^{QLIN}$ and each bit of the reduction can be computed in polylogarithmic time and logarithmic space. Given a language $L \in \mathrm{NTIME}(n) \subseteq NQLIN$ and an input of length n, we compute its reduction to SAT. The reduction has length $O(n \text{ poly-log}(n))$. Once we have the reduction we can solve it in deterministic $O(n^c \text{ poly-log}(n))$ time using our deterministic algorithm for SAT in the hypothesis. However, if we computed the reduction in a straightforward manner it would take up too much space — $O(n \text{ poly-log}(n))$ space just for storing the reduction (note: d may be less than 1). Instead, we can modify our SAT algorithm to run directly on inputs for L: whenever the algorithm requires a bit of the reduction, that bit is computed from the original input. This computation of a single bit can be done in polylogarithmic time and logarithmic space. This incurs only a polylogarithmic blow-up in time. The space requirement is clearly $O(n^d \text{ poly-log}(n))$ (the $\log n$ factor for computing the reduction on-the-fly is absorbed). ☐

Theorem 1. *SAT* $\notin$ DTISP(n^c, n^d) *for constants c and d such that $c(c+d) < 2$.*

Proof. By Lemma 1, it suffices to show that NTIME$(n) \not\subseteq$ DTISP$(n^c \,\text{poly-log}(n), n^d \,\text{poly-log}(n))$. We prove this by contradiction. In this proof we only assume that NTIME$(n) \subseteq$ DTISP(n^c, n^d) and show that this leads to a violation of the nondeterministic time hierarchy theorem. The polylogarithmic factors add a $o(1)$ term in the exponent of the time and space usage. We will see that we reach the same contradiction even when we include these factors.

Our proof involves alternations which we discussed in the previous lecture. We split the proof into two parts. First we speed up deterministic computations by introducing alternations (going from DTISP to Σ_2-TIME). Then we eliminate these alternations (going from Σ_2-TIME to NTIME) using our hypothesis.

Part 1. Let L be a language in DTISP(t, s). Fix an input string x, and consider the computation tableau which has $t = t(|x|)$ rows and $s = s(|x|)$ columns. Each row describes a configuration of the Turing machine at some point in time. We call the initial configuration c and the (unique) accepting configuration c'. Then $x \in L$ if and only if c' can be reached from c in t steps. In the proof of NSPACE$(s(n)) \subseteq$ DSPACE$(s^2(n))$ in Lecture 5, we guessed an intermediate state $c_{\frac{1}{2}}$ and verified independently that we could go from c to $c_{\frac{1}{2}}$ and $c_{\frac{1}{2}}$ to c'. We will use the same technique here, but this time instead of splitting up the tableau into two parts, we split it up into b parts (value of b will be determined later). We can guess $b-1$ configurations $c_1, c_2, \ldots, c_{b-1}$, and verify that we can go from c_{i-1} to c_i in t/b steps, for $1 \le i \le b$, $c_0 = c$, and $c_b = c'$. This computation can be stated as:

$$x \in L \iff (\exists c_1, \ldots, c_{b-1})(\forall 1 \le i \le b) c_{i-1} \vdash^{t/b} c_i$$

Note that this is in fact a Σ_2 computation. We now analyze the time taken by this computation. Guessing the existential part takes time $O(bs)$ since each c_i is of length s. Guessing the universal part takes time $O(\log b)$. The verification part takes time $O(s + t/b)$ since we can easily simulate each step of the deterministic computation in constant time. Combining we get a $O(bs + t/b)$ running time. To achieve the best speedup, we minimize this value by setting b appropriately. Setting $bs = t/b$ achieves a value within a constant factor of the minimum. So $b = \sqrt{t/s}$ and the running time becomes $O(\sqrt{ts})$. Thus we have

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

Part 2. Let L be a language in Σ_2-TIME(n^a) for some $a \ge 1$. Computation for L can be written as

$$x \in L \iff (\exists y \in \Sigma^{|x|^a})(\forall z \in \Sigma^{|x|^a}) R(\langle x, y, z\rangle) \tag{1}$$

where R is a deterministic linear time computable predicate. Note that $(\forall z \in \Sigma^{|x|^a}) R(\langle x, y, z\rangle)$ is in co-NTIME(N) for fixed y with $|y| = |x|^a$, where $N = |x| + |x|^a$ (length of $\langle x, z\rangle$). Since NTIME$(N) \subseteq$ DTISP(t, s), so is co-NTIME(N) as DTISP is closed under complement. Hence $(\forall z \in \Sigma^{|x|^a}) R(\langle x, y, z\rangle)$ is in DTIME(N^c). It follows that (1) is in NTIME(N^c) = NTIME(n^{ac}), and thus

$$\Sigma_2\text{–TIME}(n^a) \subseteq \text{NTIME}(n^{ac})$$

Combining parts 1 and 2 gives us DTISP$(n^{2c}, n^{2d}) \subseteq \Sigma_2$-TIME$(n^{c+d}) \subseteq$ NTIME$(n^{(c+d)c})$. The final step is showing NTIME$(n^2) \subseteq$ DTISP(n^{2c}, n^{2d}), from which follows NTIME$(n^2) \subseteq$ NTIME$(n^{(c+d)c})$, contradicting the nondeterministic time hierarchy theorem. This inclusion follows from the hypothesis and a technique called *padding*. Notice that if we had included the polylogarithmic factors in the above, we would end up with an addition $o(1)$ term on the final running time - i.e. we would reach the conclusion that $\mathrm{NTIME}(n^2) \subseteq \mathrm{NTIME}(n^{(c+d)c+o(1)})$. This remains contradictory to the nondeterministic time hierarchy for c and d such that $(c+d)c < 2$.

Padding. Let L be a language in $\mathrm{NTIME}(n^2)$. Let $L' = \{0^{|x|^2-|x|-1}1x \mid x \in L\}$. We claim that $L' \in \mathrm{NTIME}(n)$. Let M be a NTM accepting L that runs in quadratic time. We construct a NTM M' accepting L' as follows. On input x', L' checks whether $|x'|$ is a possible member of L' by counting the number of leading zeroes. It then extracts x from x' by removing the artificially padded prefix, and simulates the computation of L on x. $|x| = \sqrt{|x'|}$, therefore computation takes time $O(|x|^2) = O(|x'|)$. So $L' \in NTIME(n)$ and therefore also $L' \in \mathrm{DTISP}(n^c, n^d)$.

We now use the $\mathrm{DTISP}(n^c, n^d)$ algorithm for L' to decide L in $\mathrm{DTISP}(n^{2c}, n^{2d})$. Given x, a possible instance of L, we can pad it to x' and decide in $\mathrm{DTISP}((n')^c, (n')^d)$ if $x' \in L'$ (equivalent, $x \in L$). Since $n' = n^2$, we have solved L in $\mathrm{DTISP}(n^{2c}, n^{2d})$. □

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 an algorithm for SAT is space-efficient (logarithmic space in particular), then it runs in super-linear time. The second statement says that if an algorithm for SAT is time-efficient (linear time in particular), then it uses polynomial space – logarithmic space is not sufficient.

The above theorem and its corollary show inefficiency in either time or space of a single algorithm for SAT. It might be possible that there exists a linear-time linear-space algorithm, and another log-space quadratic-time algorithm for SAT. The following lower bound result rules out this possibility.

Theorem 2. *SAT* $\notin \mathrm{L} \cap \text{co-NTIME}(n^{1+o(n)})$.

Proof. Exercise. □

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

2 Nonuniformity: Motivation

Computational models we have seen so far, such as Turing machines and random access machines, work for inputs of all lengths. We call these *uniform* models of computation. In this section, we introduce *nonuniform* models of computation.

In short, a nonuniform model is a computational model in which a different machine (more generally, a computing device) is used for each input length. For example, you can use a machine M_0 for inputs of length 0, another machine M_1 for inputs of length 2, and yet another machine M_2 for length 2, and so on. The machines M_0, M_1, M_2, ... form an infinite family of machines.

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 in fact our old uniform model, but with 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. Next section we will relate nonuniform models to uniform models.

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 $x_1, x_2, \ldots, x_n$. 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.

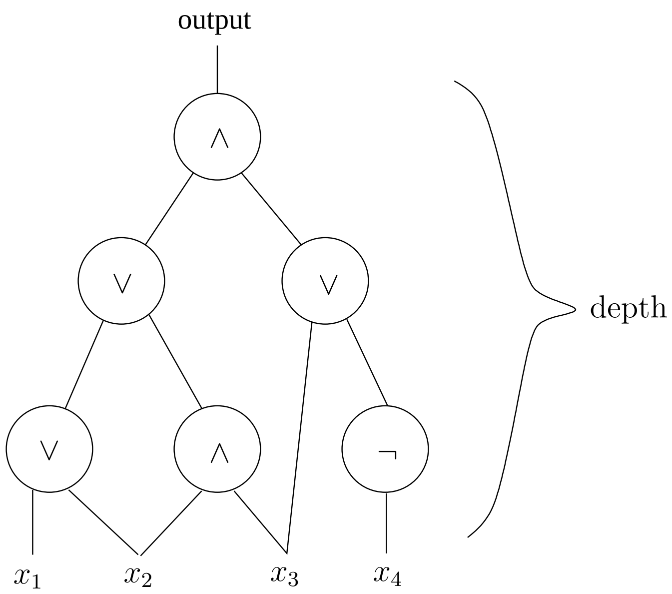

Figure 1: A boolean circuit.

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, $\ldots$, 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.

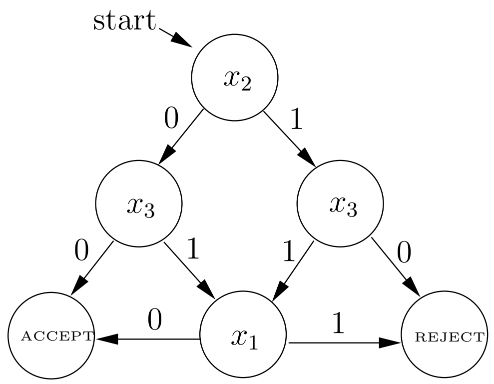

Figure 2: A branching program.

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 $i \in \mathbb{N}$. (See subsection on boolean circuits).

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 (resp. REJECT) node if it represents an accepting (resp. 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*, a piece of additional information which may help improve the efficiency of computation.

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 $\mathcal{C}/a(n) = \{L|$ there exists $L' \in \mathcal{C}$ and $y_0, y_1, y_2, \ldots \in \Sigma^*$ such that $x \in L \iff \langle x, y_{|x|}\rangle \in L'$, where $|y_n| \leq a(n)$ for $n = 0, 1, 2, \ldots\}$.*

The sequence $\{y_0, y_1, y_2, \ldots\}$ in the above definition is called the *advice sequence* and $a(n)$ is an upper bound for the length of advice. Note that the advice taken by the machine only depends on the length of the input. This differs from *certificates* where each input may have a different certificate and there is no restriction on certificates for inputs of equal length.

Advice can be very powerful, even when its length is bounded by one bit. In fact, there exist uncomputable languages which can be computed using one-bit advice. As an example we consider the halting problem HALT $= \{x \,|\, x$ encodes a DTM M and M halts on input $x\} \subseteq \{0,1\}^*$. HALT is known to be uncomputable (by any Turing machine). Consider the language $L' = \{0^z \,|\, z$ is the lexicographic rank of x and $x \in$ HALT $\} \subseteq \{0\}^*$. It is clear that HALT$\leq_m L'$ and hence L' is also uncomputable. If we take the advice sequence $\{y_0, y_1, y_2, \ldots\}$ to be the characteristic function of HALT, then L' can be computed by a machine that simply outputs the advice bit y_z on input 0^z.

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. P/ poly $= \{L | C_L(n)$ *is polynomially bounded* $\}$.

Proof Sketch. $\subseteq$: We can construct polynomial-size circuits accepting a language in P/ poly by hardwiring the polynomial-length advice strings as "additional inputs". $\supseteq$: If L has polynomial-size circuits, we can use the descriptions of the circuits as advices. □

The proof of the following theorem is similar and is left as an exercise.

Theorem 4. L/ poly $= \{L | BP_L(n)$ *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. P $= \{L \mid L$ *has uniform polynomial-size circuits* $\}$.

Proof Sketch. $\subseteq$: Let $L \in$ 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. L $= \{L \mid L$ *has uniform polynomial-size branching programs* $\}$.

5 Next Time

Consider the class 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 NP $\subseteq$ P/ poly, then the polynomial-time hierarchy collapses to the second level. Then we will move on to talk about constant-depth boolean circuits.