CS 810: Complexity Theory	1/24/2007

Lecture 2: Determinism and Nondeterminism

Instructor: Dieter van Melkebeek — Scribe: Matt Elder

In this lecture, we discuss two applications of efficient, deterministic, universal Turing machines: deterministic time and spaces hierarchy theorems, and completeness results. We also introduce the nondeterministic Turing machine, our model for nondeterministic computation.

1 Hierarchy Results

A *hierarchy result* is a theorem that imposes a strict-subset relation between two complexity classes; that is, for some two classes of problems $\mathcal{C}$ and $\mathcal{C}'$, a hierarchy result might yield that $\mathcal{C}' \subsetneq \mathcal{C}$. Thus, class $\mathcal{C}$ is computationally more powerful than class $\mathcal{C}'$.

For example, we can show that if the function t' is sufficiently smaller than the function t, then $\mathrm{DTIME}(t') \subsetneq \mathrm{DTIME}(t)$. We prove this using the technique of *diagonalization*, which originated in Cantor's proof that $[0, 1]$ is uncountable.

Theorem 1 (Cantor)**.** *The interval* $[0, 1]$ *is uncountably infinite.*

Proof. We prove this theorem by contradiction. Given any enumeration of numbers from $[0, 1]$, we show that it cannot contain all numbers from $[0, 1]$ by constructing a number r' from $[0, 1]$ that it cannot contain.

Assume that $[0, 1]$ is countable. If the interval $[0, 1]$ is countable, then its individual elements can be enumerated. Suppose we somehow enumerate all numbers in $[0, 1]$. Let r_i be the infinite binary representation of the i^{th} number in this enumeration, and let $b_{i,j}$ be the j^{th} bit in r_i.

	b_1	b_2	b_3	$\cdots$
r_1	**0**	1	1	$\cdots$
r_2	1	**0**	0	$\cdots$
r_3	1	0	**1**	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$
r'	1	1	0	$\cdots$

Figure 1: r' is constructed to complement the diagonal elements of each r_i, used to prove Theorem 1.

Now, we consider the diagonal elements $b_{i,i}$, the bold elements in Figure 1. To build r' from these diagonal elements, set bit i of r' to the complement of $b_{i,i}$. Since the i^{th} bit of r' differs from the i^{th} bit of r_i, we know that $r' \neq r_i$ for all i. Thus, r' represents a number x that is in $[0, 1]$ but not already enumerated as some r_i. Actually, this isn't *quite* true, because every rational number has two infinite binary representations: one terminating with 000... and one terminating with 111.... We have several ways to avoid this issue. For example, we could let all representations be in base 4, and set digit i of r' to be 1 if $b_{i,i}$ equals 2 or 3, and set it to 2 if $b_{i,i}$ equals 0 or 1. We could instead ensure that r' does not end in an infinite sequence of zeroes or ones, a construction that is more complicated.

Provided we have taken care of the issue described above, the construction of r' contradicts the fact that our enumeration contains every element of $[0, 1]$, so our initial assumption is false, and $[0, 1]$ is not countable. $\square$

The proof of the deterministic time and space hierarchies emulate Cantor's diagonalization proof. The theorem requires a condition on the time bounds known as time-constructibility which we describe as it appears in the proof.

Theorem 2. *If t and t' are functions from $\mathbb{N}$ to $\mathbb{N}$, t is time-constructible, and $t(n) = \omega\left(t'(n)\log t'(n)\right)$, then* $\mathrm{DTIME}(t') \subsetneq \mathrm{DTIME}(t)$*.*

Proof. We will use a universal Turing machine to construct a contrary machine M. We build M so that for each Turing machine M_i running in time t', M's output differs from the output of M_i on some input x_i. In this way, our contrary machine M will accept a language that no machine can accept in time t'. We must also ensure M runs in time t.

Let μ be a function mapping inputs to descriptions of Turing machines with the following properties: i) μ is computable in linear time, and ii) each machine M' appears infinitely often as an image of μ. We leave it to the reader to verify that such a μ can be constructed from any computable enumeration of deterministic Turing machines. Let x_i denote the i^{th} possible input, and $\mu(x_i) = \langle M_i \rangle$. Then the code for M is as follows.

(1) Read input x_i.
(2) Compute $\mu(x_i) = \langle M_i \rangle$.
(3) Pass $\langle M_i, x_i \rangle$ to a universal Turing machine, and run the simulation as long as the total time used by M is at most $t(|x_i|)$.
(4) If the universal Turing machine halts and rejects, accept x_i. Otherwise reject x_i.

In the above simulation, M must be able to keep track of its time usage: M wishes to use at most $t(|x_i|)$ time in total. This is the place where we use the time-constructibility of t. We define this notion after the proof. The property we require is that M can in time $O(t(|x_i|))$ write down $t(|x_i|)$ many zeroes on a work tape. We use these zeroes as a clock by erasing one for each step of execution and halting if all of them are ever erased.

As t is time-constructible, the discussion above shows that M can keep track of its time usage to ensure the entire execution is $O(t(|x|))$. Because $t(n) = \omega(t'(n)\log t'(n))$ and the efficiency of the universal Turing machine proved in the previous lecture, M has enough time to perform the simulation for machine M' running in t' time for all but finitely many inputs. Because each Turing machine M' appears infinitely often as an image of μ, there is an input for which M has enough time to complete the simulation and complement the behavior M' so long as M' runs in t' time.

Thus, $L(M) \notin \mathrm{DTIME}(t')$ whereas $L(M) \in DTIME(t)$, and $\mathrm{DTIME}(t') \subsetneq \mathrm{DTIME}(t)$. $\square$

The condition required of the time bound in the theorem is stated formally in the following definition. It can be shown that all of the functions we are used to dealing with (polynomials, exponentials, logarithms, etc.) that are at least linear are time-constructible and those that are at least logarithmic are space-constructible.

Definition 1. *A function $t : \mathbb{N} \to \mathbb{N}$ is* time-constructible *if the function that maps the string 0^n to the string $0^{t(n)}$ can be computed in time $O(t(n))$. Similarly, a function $s : \mathbb{N} \to \mathbb{N}$ is* space-constructible *if the function that maps 0^n to $0^{s(n)}$ can be computed in space $O(s(n))$.*

We can use an identical proof to derive a hierarchy theorem for the amount of space used by deterministic machines, giving the following theorem. Due to the fact that the overhead in space for a universal Turing machine is smaller than the overhead for time, the space hierarchy is tighter than the time hierarchy.

Theorem 3. *If $s(n) = \omega(s'(n))$ and s is space-constructable, then* $\mathrm{DSPACE}(s') \subsetneq \mathrm{DSPACE}(s)$.

Notice that the separation in space bounds above is as tight as we could hope for, as we have already discussed that constant-multiple differences between two functions do not change their computational power.

As a corollary of these theorems, we know that $\mathrm{P} \subsetneq E \subsetneq EXP$, and that $L \subsetneq PSPACE$.

2 Reductions and Completeness

Problem reducibility is central to the study of complexity. Reductions allow us to determine the relative complexity of problems without knowing the absolute complexity of those problems. If A and B are two problems, then $A \le B$ denotes that A reduces to B. So, A can be efficiently solved if B can be efficiently solved. This implies that the complexity of A is no greater than the complexity of B (modulo the complexity of the reduction itself), so the notation $A \le B$ is sensible in this context.

We consider two types of reductions, mapping and oracle reductions. Mapping reductions are more restrictive.

Definition 2. *A* mapping reduction*, also called a many-one reduction or a Karp reduction, is a function that maps instances of one problem to instances of another, preserving their outcome.*

More specifically, if A and B are decision problems, then $A \le_m B$ if there exists some function $f : \Sigma_A^* \to \Sigma_B^*$ such that, for all problem instances $x \in \Sigma_A^*$, $x \in A \iff f(x) \in B$. That is, $f(x)$ is in the language of B iff x is in the language of A.

To capture the intended meaning of a reduction, we would like the function f to be efficient. The efficiency requirement varies by context - we describe the two most common settings in a moment.

Definition 3. *An* oracle reduction*, also called a Turing reduction or Cook reduction, is an algorithm to solve one problem given a solver to a second problem as an instantaneous subroutine.*

*A B-*oracle *is a hypothetical machine that can solve any instance of problem B and return its answer in one step. An* Oracle Turing Machine *(OTM) is a Turing Machine with a special oracle tape, and a specific query state. When the machine reaches the query state, it invokes its oracle on the current content of the oracle tape. In this state, the oracle's input is erased, the oracle's output is placed on the oracle tape, and the oracle tape head is placed on the leftmost position.*

For a given OTM M, M^B is that machine with a B oracle. We say that A oracle-reduces *to B, denoted $A \le_o B$, if there exists some efficient OTM M such that M^B solves A.*

Given a mapping reduction f from A to B, we can give an oracle-reduction of A to B in the following OTM: "Given input x, compute $f(x)$ and write it to the oracle tape. Then, query the oracle, and return its output." Since any mapping reduction may be thus reduced to an oracle reduction, oracle reductions are at least as powerful as mapping reductions.

We denote time or space constraints on the efficiency of a reduction via superscripts on the $\leq$ symbol. Polynomial-time reducibility is denoted $\leq^{\mathrm{P}}$, and log-space reducibility is denoted $\leq^{\log}$. log-space oracle reductions have a slightly complicated specification, which may differ by author and context. For our purposes, we impose the restriction that that we may read and write on the oracle tape only from left to right, so that the oracle tape cannot be used as procedural memory. We do not count the size of the oracle tape against our space efficiency.

Proposition 1. *Let $\tau \in \{m, o\}$ and let $r \in \{\mathrm{P}, \log\}$. The following are true:*

- *Reducibility is transitive. If $A \leq_\tau^r B$ and $B \leq_\tau^r C$, then $A \leq_\tau^r C$.*
- *If $A \leq_\tau^{\mathrm{P}} B$ and $B \in \mathrm{P}$, then $A \in \mathrm{P}$.*
- *If $A \leq_\tau^{\log} B$ and $B \in \mathrm{L}$, then $B \in \mathrm{L}$.*

Definition 4. *Given a reduction relation $\leq$ and complexity class $\mathcal{C}$, B is* hard *for $\mathcal{C}$ under $\leq$ if, for every problem A in $\mathcal{C}$, $A \leq B$. We say that B is* complete *for $\mathcal{C}$ under $\leq$ if B is hard for $\mathcal{C}$ under $\leq$ and $B \in \mathcal{C}$.*

The choice of reduction depends on context. For example, it is known that $\mathrm{L} \subseteq \mathrm{P}$, but is P equal to L? In this case, we use log-space reductions for the following reason: given a problem B that is complete for P under $\leq_\tau^{\log}$, $\mathrm{P} \subseteq \mathrm{L}$ if and only if B is in L.

Given the connection between complete problems and complexity class collapses, it is useful to have a variety of complete problems to make use of. As a first start, we can construct complete languages out of efficient universal Turing machines.

Proposition 2. *Let K_D be the language of tuples $\langle M, x, 0^t \rangle$ such that M halts and accepts input x in no more than t steps. K_D is complete for* P *under $\leq_m^{\log}$.*

Proof. The language K_D is in P because U_{DTIME} runs with polynomial time overhead.

Suppose A is a language in P. We demonstrate a function f demonstrating that $A \leq_m^{\log} K_D$. If we know the Turing machine N that computes A in polynomial time, then we can define the mapping f that takes x to $\langle N, x, 0^{|x|^c} \rangle$ such that $f(x) \in K_D$ iff $x \in A$. We can hard-code N into f, and it is possible to output both x and $0^{|x|^c}$ in log space with c hard-coded into f. Thus $f(x)$ is a log-space mapping reduction from A to K_D. □

3 Nondeterminism

Note well: the nondeterministic model of computation is not intuitive, and for good reason. *Nondeterminism is not a physically realizable model of computation.* The utility of the model lies not in modeling actual computation, but in exactly capturing the complexity of an important class of problems.

3.1 Model

The standard model for nondeterministic computation is the nondeterministic Turing machine (NTM). An NTM has a definition very similar to a normal, deterministic Turing machine, except that the transition function δ in a TM may be a relation in an NTM. Thus, for any combination of internal state and read-head state, δ may provide multiple next-state instructions.

The language of an NTM M, $L(M)$ is the set of strings x such that there exists a valid computation of M on input x that halts and accepts. That is, we say that M accepts x if some possible computation path of M accepts x.

There are several fruitful interpretations of NTMs. We can view an NTM as a machine allowed to make guesses, and these guesses are controlled by some force that wants the machine to accept. For a particular input x, if any permissible series of guesses will lead the machine to its accepting state then the machine will make those guesses and accept x. If there is no such series of guesses, then the machine cannot accept x and is forced to reject x.

We can also think of an NTM as a massively parallel computer. Every time it makes a choice, it spawns a copy of itself for every possible branch of that choice, and all of these copies continue processing. If any child machine accepts an input, then the machine as a whole accepts that input.

3.2 Time and Space

NTM efficiency is stated in terms of worst-case behavior. So, "M runs in time t" means that *every* branch of the execution of M halts within t steps. Similarly, "M runs in space s" means that every branch of the execution of M uses only s cells.

The nondeterministic complexity classes NTIME(t) and NSPACE(s) are precisely analogous to their deterministic counterparts DTIME(t) and DSPACE(s): a problem is in NTIME(t) if there exists an NTM M that solves that problem and runs in time $O(t(n))$ for inputs of size n. A problem is in NSPACE(s) if there exists an NTM M that solves that problem and runs in space $O(s(n))$ for inputs of size n.

The following complexity classes are analogous to their deterministic counterparts:

- $\mathrm{NP} = \cup_{c>0} \mathrm{NTIME}(n^c)$
- $\mathrm{NE} = \cup_{c>0} \mathrm{NTIME}(2^{cn})$
- $\mathrm{NEXP} = \cup_{c>0} \mathrm{NTIME}(2^{n^c})$
- $\mathrm{NL} = \mathrm{NSPACE}(\log(n))$
- $\mathrm{NPSPACE} = \cup_{c>0} \mathrm{NSPACE}(n^c)$

We won't much talk about NPSPACE, because it turns out that PSPACE = NPSPACE, a result we'll prove later.

3.3 Universal Machines

Just as we constructed universal deterministic Turing machines, we can construct universal nondeterministic Turing machines. First notice that we can use the deterministic universal machine construction given in the first lecture here as well. This gives a single universal nondeterministic machine with logarithmic overhead in time and constant overhead in space.

For the task of simulating nondeterministic machines time-efficiently, we can do better than in the deterministic setting. If given as input the running time of the machine we wish to simulate, we demonstrate a machine that simulates the given machine with only a constant factor overhead in time.

Theorem 4. *There exists a NTM U_{NTIME} such that $t_{U_{\text{NTIME}}}(\langle M, x, 0^t \rangle) = O(\text{poly}(|M|) \cdot t)$ and for all $t \geq t_M(x)$, $U_{\text{NTIME}}(\langle M, x, 0^t \rangle) = M(x)$.*

Proof. U_{NTIME} assumes that $M(x)$ runs in time at most t and begins by guessing a computation record for $M(x)$ and writing it down on one of its work tape. For each time step, the computation record includes a guess for each of the following: motion (either L or R) for each of M's tape heads, new cell contents under each of M's tape heads, and new internal state. The length of this computation record is $O(\text{poly}(|M|) \cdot t)$.

After writing down the guessed computation record, U_{NTIME} verifies that the guessed computation record corresponds to a valid computation according to M's transition function, and that the computation ends in an accepting state. This is achieved by checking the validity of the computation for each of M's work tapes in turn. The important point is that if there is an accepting computation path for $M(x)$ of length at most t, then at least one of U_{NTIME}'s guessed computation records causes it to accept as well. We leave it to the reader to verify the time efficiency of the simulation. $\square$

It is critical in the above proof that U_{NTIME} takes as input the amount of time to simulate M for. The construction can be modified if this information cannot be given. Namely, we run the construction above by first passing in 1 as the maximum time. If no accepting computation is found, the construction is repeated with double the amount of maximum time passed in. This is continued until an accepting computation is found. Note that if $M(x) = 1$, then this construction is correct and runs in time $O(\text{poly}(|M|)t_M(x))$. However, if there is no accepting computation path for $M(x)$, this construction never halts.

As for deterministic machines, we use the universal machines for nondeterministic machines to define a complete language - now for NP. The proof of the following is similar to the proof of Proposition 2

Proposition 3. *Let K_N be the language of tuples $\langle M, x, 0^t \rangle$ such that the NTM M accepts on input x in time t. K_N is complete for NP under $\leq_m^{\log}$.*

Hierarchy results in the nondeterministic model are more difficult to achieve than in the deterministic model, as complementation is more difficult. To perform the same diagonalization for NTMs that we did on DTMs, we would need a simple way to accept an input when a simulated machine rejects that input, and vice versa. However, an NTM accepts when any of its possible computation branches accept, and rejects only when all of its possible computation branches reject. This asymmetry seems to make complementation inefficient. Whether complementation on NTMs can be efficiently computed is unknown; this is the NP vs. coNP problem.

However, hierarchy results are still possible, as we will show in a later lecture.

4 Next Lecture

Next time, we will start by discussing the relationship between the class NP and deterministic verification.