CS 710: Complexity Theory 1/21/2010

Lecture 2: Universality

Instructor: Dieter van Melkebeek Scribe: Tyson Williams

In this lecture, we introduce the notion of a universal machine, develop efficient universal Turing machines for deterministic computation, and discuss two consequences: time and space hierarchy theorems for deterministic computation, and completeness results.

1 Universal Turing Machine

A Turing machine (TM) U is called a *universal Turing machine* (UTM) if it is able to simulate all other TMs: for all TMs M and inputs x, $U(\langle M, x\rangle) = M(x)$. A key property that leads to a number of results is that there exist UTMs that incur a small overhead in time and space.

Theorem 1. *There are UTMs U_{DTIME} and U_{DSPACE} such that for all TMs M and inputs x,*

$$t_{U_{\mathrm{DTIME}}}(\langle M, x\rangle) = O(|M| \cdot t_M(x) \cdot \log(t_M(x))) \tag{1}$$

$$s_{U_{\mathrm{DSPACE}}}(\langle M, x\rangle) = O(\log(|M|) \cdot s_M(x)), \tag{2}$$

where $|M|$ denotes the length of the description of M.

In fact, we will show that a single machine can be used for both U_{DTIME} and U_{DSPACE}.

Proof sketch. The main difficulty in designing U_{DTIME} and U_{DSPACE} is that each must have a fixed number of work tapes while simulating machines with any number of work tapes.

First consider U_{DSPACE}. Suppose M has k work tapes. We would like to keep the contents of these on a single tape for U_{DSPACE}. We call this tape the storage tape. This is done by first storing the first cell from each of M's k tapes in the first k cells of U_{DSPACE}'s storage tape, then storing the second cell from each of M's k tapes in the next k cells, and so on. In general, the contents of the ith cell from the jth tape of M will be at location $k \cdot i + j$ on our storage tape. Recall that in a single step, M reads the contents of each work tape, and the locations of the tape head can be different for each of these. So, U_{DSPACE} must know where each of M's tape heads is. U_{DSPACE} stores this information on an additional work tape that we will call the lookup tape. U_{DSPACE} also must have an index tape in order to perform tape head jumps. We leave it as an exercise to verify that U_{DSPACE} can simulate M using these three tapes (storage, lookup, and index), and that the simulation is as efficient as claimed.

The log factor in (1) comes from working index calculations.[1] Just like the number of tapes, the tape alphabet of our UTM may be smaller than the tape alphabet Γ_M of M. However, we can simulate Γ_M using $\log|\Gamma_M| \in O(\log|M|)$ space, which explains where the log factor comes from in (2).

Consider the time required to run the above simulation. For each step of M's execution, U_{DSPACE} must read the contents of the sequential access work tapes and must remember the

[1] Even if our UTM was of the sequential access type, the best known and highly difficult construction still produces a log factor.

contents of the current cell on each of the random access work tapes. As the sequential access tapes access locations that are at most $t_M(x)$, the address for each sequential access tape head takes $O(\log t_M(x))$ space. We leave it as an exercise to verify that each transition function step of M takes $O(|M| \cdot \log t_M(x))$ steps for U_{DSPACE} to simulate. Together with the fact that U_{DSPACE} performs tape head jump operations in constant time just as M does, we conclude that $U_{\text{DTIME}} = U_{\text{DSPACE}}$ is as efficient with respect to time as claimed. Notice that this analysis takes into account that U_{DTIME} has a random access input tape although M may have a sequential access input tape. For simulating M that have a sequential access input tape, U_{DTIME} incurs a log factor overhead in time to simulate sequential access on its input tape. □

A key component of the analysis of the time efficiency in the above proof is that each work tape is only either sequential access or random access. If a single work tape were allowed to be both sequential and random access, the analysis would fail (the counterexample in this case is a machine M that jumps to location 2^r and then performs r local operations near this address - then M runs in $O(r)$ time while the simulation would take time $\Theta(|M| \cdot r^2)$). Even in this situation, a universal machine U_{DTIME} with similar overhead in time can be constructed by ensuring a simulation of M where U_{DTIME} only ever accesses tape cells with small addresses - namely $O(\log t_M(x))$. This is achieved by keeping track of random access tape operations as (tape cell address, tape cell contents) pairs and storing these in an efficient data structure such as a balanced binary tree.

The same trick of keeping track of random access tape operations in a data structure can be used to convert any machine M using time t into another machine M' that computes the same relation and uses space $O(t)$. Because of how we have defined space complexity, notice that this transformation is not trivial - a machine M can use space that is exponential in t by writing down a large address and accessing that location on its random access tape.

The existence of an efficient UTM is a key component to a number of important results. Two of these important results are hierarchy theorems and complete problems.

2 Hierarchy Results

A *hierarchy theorem* states that a TM with slightly more resources (either time or space) can compute relations that cannot be computed with slightly less resources. More specifically, a hierarchy theorem imposes a strict-subset relation between two complexity classes. That is, for 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 $\text{DTIME}(t') \subsetneq \text{DTIME}(t)$. We prove this using the technique of *diagonalization*, which originated in Cantor's proof that $[0, 1]$ is uncountable.

Theorem 2 (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 from $[0, 1]$. 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.

Now, we consider the "diagonal" elements $b_{i,i}$. 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 in $[0, 1]$ but is not enumerated as some 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 the complement of the (bold) diagonal elements.

Actually, this isn't *quite* true because every rational number has two infinite binary representations: one terminating with $000\ldots$ and one terminating with $111\ldots$. 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. Therefore $[0, 1]$ is not countable. □

The proofs of the deterministic time and space hierarchies emulate Cantor's diagonalization proof. The theorem requires that one of the time bounds is time-constructable, which we describe as it appears in the proof.

Theorem 3. *If t and t' are functions from $\mathbb{N}$ to $\mathbb{N}$, t or t' is time-constructable, and $t(n) = \omega\left(t'(n) \log t'(n)\right)$, then* $\text{DTIME}(t') \subsetneq \text{DTIME}(t)$.

Proof. We will use a UTM to construct a contrary machine M. We build M so that for each TM 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 TMs with the following properties: (i) μ is computable in linear time and (ii) each TM 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 TMs. Let x_i denote the i^{th} possible input, and let $\mu(x_i) = \langle M_i \rangle$. 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 UTM, 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.
(5) Otherwise reject x_i.

In the above simulation, M must be able to keep track of its time usage because it cannot run for more than $t(|x_i|)$ steps. This is where we use the time-constructability of t or 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.

Let's assume that t is time-constructable. The proof when t' is time-constructable is similar. Since t is time-constructable, 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 UTM from theorem 1, M has enough time to perform the simulation for each M' running in t' time for all but finitely many inputs. Because each 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 of M' given that M' runs in t' time.

Thus, $L(M) \notin \text{DTIME}(t')$ whereas $L(M) \in \text{DTIME}(t)$. Therefore, $\text{DTIME}(t') \subsetneq \text{DTIME}(t)$. □

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, exponential, logarithms, etc.) that are at least linear are time-constructable and those that are at least logarithmic are space-constructable.

Definition 1. *A function $t : \mathbb{N} \to \mathbb{N}$ is* time-constructable *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-constructable *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 TMs, giving the following theorem. Due to the fact that the overhead in space for a UTM is smaller than the overhead for time, the space hierarchy theorem is tighter than the time hierarchy theorem.

Theorem 4. *If s and s' are functions from $\mathbb{N}$ to $\mathbb{N}$, s or s' is space-constructable, and $s(n) = \omega(s'(n))$, then* $\text{DSPACE}(s') \subsetneq \text{DSPACE}(s)$.

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 $\text{P} \subsetneq \text{E} \subsetneq \text{EXP}$, and that $\text{L} \subsetneq \text{PSPACE}$.

3 Reductions

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 $A \leq B$ denotes that A reduces to B. 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 $A \leq B$ is sensible in this context.

We consider two types of reductions, mapping reductions 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 $A \leq_m B$ iff 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, x is in the language of A iff $f(x)$ is in the language of B.

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 TM with a special oracle tape, and a specific query state. When the OTM 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 $A \leq_o B$, 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:

(1) Read input x.
(2) Compute $f(x)$ and write it to the oracle tape.
(3) Query the oracle, and return its output.

Since any mapping reduction is also 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 $\leq^{\log}$.

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 $A \in \mathrm{L}$.*

Proof Sketch. The polynomial-time propositions follow from the fact that a composition of polynomials is again a polynomial.

The log-space propositions require an insight. The naive approach is to compute the entire input for the next machine and then run that machine, but this could use a linear amount of space. Remember that when measuring the amount of space that is used, we only count the work tapes and not the input or output tapes. Thus, computing the output of one machine before passing it along to another effectively changes an output tape into a work tape.

To avoid this problem, we exploit the fact that we have no time restriction. We begin by running the second machine. When the second machine needs the ith bit of its input, we run the first machine without writing down its output until we know the ith bit of the output. Then this bit is given to the second machine and it resumes its work. □

4 Completeness

Another consequence of the existence of efficient UTMs is complete problems for deterministic computations under efficient reductions. Intuitively, a problem is complete for a complexity class if it is both within the class and at least as difficult as all other problems within the class. We formalize the notion in the following strong sense.

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 the 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 $\leq_\tau^{\log}$, $\mathrm{P} \subseteq \mathrm{L}$ iff B is in L.

Given the connection between complete problems and complexity class collapses, it is useful to have a variety of complete problems. For starters, we can construct complete languages out of efficient UTMs.

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. Let N be the TM that decides A in polynomial time. Then we can define the mapping f that takes x to $\langle N, x, 0^{|x|^c} \rangle$ such that $x \in A \iff f(x) \in K_D$. 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. □