CS 710: Complexity Theory 9/13/2011

Lecture 3: Reductions and Completeness

Instructor: Dieter van Melkebeek Scribe: Brian Nixon

Last lecture we introduced the notion of a universal Turing machine for deterministic computations, presented an efficient construction, and sketched one implication, namely the existence of hierarchies in time and space complexity classes. In this lecture we give a formal proof for the theorem that establishes this implication, the Time Hierarchy Theorem, and develop another implication of the existence of efficient universal Turing machines, namely the existence of complete problems for the robust classes we introduced earlier. We also start the discussion of the nondeterministic model of computation.

1 Review and Time Hierarchy Theorem

Recall from last time that the existence of the universal Turing machine allows us to view the set of Turing machines as a set of input strings to the universal machine. This lets one use a diagonalization argument similar to the Cantor proof of the uncountability of the real numbers to demonstrate the separation of Turing machines into a time hierarchy. Also central to the proof is the efficiency in the universal Turing machine's simulation, without which it's use would not provide as meaningful a separation.

The implication is that a Turing machine allowed to use slightly more resources (either time or space) can com- pute relations that cannot be computed on a Turing machine restricted to slightly less resources. Thus is makes sense to talk of separate complexity classes of problem and impose a strict-subset relation between certain complexity classes.

Theorem 1 (Time Hierarchy Theorem for Deterministic Turing Machines)**.** *Let $t, t' : \mathbb{N} \to \mathbb{N}$ be functions such that t' is time-constructable and $t'(n) = \omega(t(n)\log(t(n)))$. Then* $\mathrm{DTIME}(t(N)) \subsetneq \mathrm{DTIME}(t'(n))$.

Proof. Let μ be a mapping from Σ^* to the set of Turing machines such that

- μ is linear time computable
- Every Turing machine is mapped from infinitely many elements of Σ^*

These allow us to compute a Turing machine from a string without imposing an additional initial overhead for resource use. We only want to prove separation so we only need to construct a machine that runs in time $t'(n)$ that doesn't run in $t(n)$.

Consider the Turing machine M' where $M'(x)$ executes the following steps.

1. Calculate machine $M = \mu(x)$.
2. Using the universal Turing machine U, run the simulation $U(\langle M, x\rangle)$ for $t'(|x|)$ steps. This step is where we use the time constructible part so we don't have to worry about halts.
3. If the simulation halts or rejects then accept, else reject.

In the above simulation, M must be able to keep track of its time usage because it must run in a bounded number of steps. The property we require is that M can in time $O(t'(|x|))$ write down $t'(|x|)$ 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. The clocking ensures that $L(M') \in \text{DTIME}(t'(n))$.

Now we note by construction we cannot have $L(M') \in \text{DTIME}(t(n))$. Assuming otherwise means $\exists M$ such that $L(M) \in \text{DTIME}(t(n))$ and $L(M) = L(M')$. As the running time of M is asymptotically bounded by that of M', there is an input length ℓ after which for every input M always halts before M'. Since μ maps to $L(M)$ infinitely many times $\exists x$ such that $|x| > \ell$ and $\mu(x) = L(M)$. However, M' cannot fail to halt on input x and must return a different answer than M on x. This contradicts our assumption that the two languages were equal. □

The argument for space is similar except we don't need the additional log factor in our theorem statement because the universal Turing machine doesn't need it. The theorem statement is stated: let $s, s' : \mathbb{N} \to \mathbb{N}$ be functions such that s' is space-constructable and $s'(n) = \omega(s(n)))$. Asymptotic bounds ignore constants so the separation in space bounds are as tight as we could hope.

To be thorough, we include the formal definition of time-constructable. 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 (Time-constructable)**.** *A function $t : N \to N$ 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 $s : N \to N$ is* space- constructable *if the function that maps 0^n to $0^{s(n)}$ can be computed in space $O(s(n))$.*

2 Reductions

Completeness is a quality produced by a ranking of the difficulty of problems. Before we can address it we need to develop a tool to produce this ordering.

Definition 2 (Reduction)**.** *We say $A \leq B$ or A* reduces *to B if we can solve an instance of problem A when we have a solution to problem B. Equivalently, we say there $A \leq B$ if there exists a* reduction *from A to B.*

A reduction directly compares the difficulty of two problems. With $A \leq B$ we can say A is not harder than B as the complexity of A cannot be greater than that of B (modulo the complexity of the reduction). When $A \leq B$ and $B \leq A$, we establish an equivalence in difficulty between the two problems.

We can consider the solving of B to be a subroutine of A. In fact, a problem solution may use several such subroutines. As an example of a similar practice in algorithm generation, one can imagine casting a problem as a network flow then using the existing efficient tools to solve it even though the initial problem may seem at first to have little in common with a flow network.

Today we will discuss reductions characterized by two orthogonal properties. First, we have to choose the mechanism being used. The choice here leads to either a mapping reduction or an oracle reduction, which we will define in a moment. Once we've settled on a mechanism, we choose the strength of our reduction by setting a resource restraint of either polytime or logspace. Both choices are independent so we have described four possible reduction classes.

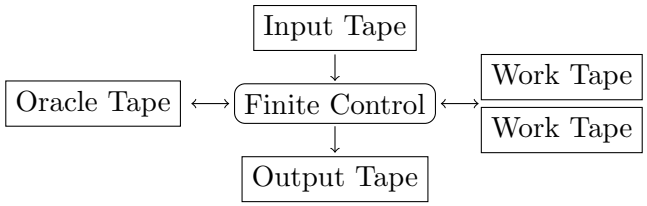

Figure 1: Oracle Turing Machine

Definition 3 (Mapping Reduction)**.** A mapping reduction *is a function that maps instances of one problem to instances of another, preserving their outcome. Formally,* $A \leq_m B$ *if* $\exists f : \Sigma_A^* \to \Sigma_B^*$ *a map from the strings underlying* A *to the strings underlying* B *such that* f *maps a problem of type* A *to an equivalent problem of type* B*. As we view problems as a relation of inputs and outputs this means that* $(f(x), y) \in B \Rightarrow (x, y) \in A$*.*

For the class of decision problems this means $x \in A$ iff $f(x) \in B$ when viewing A and B as languages. Our y term above is dropped as it is used to determine inclusion. This mapping f can be many-to-one, so these are sometimes called Many-to-One reductions. Another term used is Karp reduction after Richard Karp.

A mapping reduction is very restrictive. It is similar to tail recursion as it requires the return value of the subroutine to be the same as the main routine's return value. A much more general idea of reduction is captured by the use of an oracle.

Definition 4 (Oracle Reduction)**.** An oracle reduction *is an algorithm to solve one problem given a solver to a second problem as an instantaneous subroutine. Formally,* $A \leq_o B$ *if there exists an oracle Turing machine* M *such that* M^B *solves* A*.*

These are also called Cook reductions or Turing reductions.

An B-oracle Turing machine is a modification of a Turing machine that allows M to consult an internal oracle that produces instantaneous solutions to queries of type B.

This resembles a standard Turing machine with the addition of a special oracle tape, see Figure 1. The machine is allowed to write on the oracle tape. It can then enter a special phase that consists of one step with respect to the other tapes after which the oracle tape either holds a solution or an indication that no solution exists. Read access is then returned to the Turing machine. The separation of read and write phases prevents the Turing machine from interfering or observing intermediate computation on the oracle tape and so change its computational bounds.

An oracle can make as many queries and combine queries as it wants. A mapping reduction could be carried out with an oracle that is invoked exactly once and whose response is transferred immediately to the Turing machine's output tape. Thus oracle reductions are at least as powerful as mapping reductions as a class of operators. Computability theory worries about whether the oracle is computable or halts and whether f works given the resource bounds of the problem.

Just as mapping and oracle reductions are denoted using subscripts, we specify the resource bound using a superscript. Polynomial time reducibility will be represented with $\leq^p$ and a log space reduction with $\leq^{\log}$. In the case of a log space oracle reduction we don't count the space used in writing to the oracle toward the count from our bounded work tapes.

Proposition 1. *For* $\tau \in \{m, o\}$ *and* $r \in \{p, \log\}$

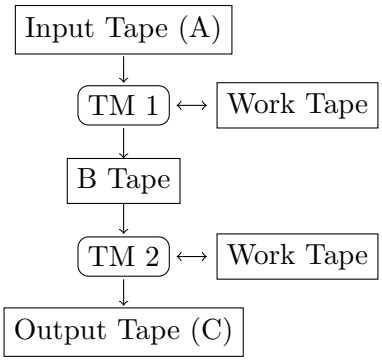

Figure 2: Log Space Composition

1. *(Transitivity)* $A \leq_\tau^r B$ *and* $B \leq_\tau^r C$ *implies* $A \leq_\tau^r C$
2. $A \leq_p B$ *and* $B \in \mathrm{P}$ *implies* $A \in \mathrm{P}$
3. $A \leq_{\log} B$ *and* $B \in \mathrm{L}$ *implies* $A \in \mathrm{L}$

We can approach the first by cases. For $\leq_m^p$ we can compose the maps $A \to B$ and $B \to C$. The resulting map operates within poly time bounds. A similar composition satisfies the relation $\leq_o^p$.

For $\leq_m^{\log}$ we think of constructing a Turing machine that takes input as A, transforms A into B onto an intermediate tape then transforms B into C and solves it, writing to the output tape. This is diagrammed in Figure 2 with machine TM 1 performing the transformation of A to B and TM 2 transforming B to C. We are guaranteed to use log space on every tape except the intermediate one as the output of type B by TM 1 is allowed to be greater than log of the input on the A tape. This can be solved by only providing one bit of the output on the intermediate tape to the second half of the process at a time. Thus we only need to track an index which will fit in log space. This does increase our time complexity by imposing a polynomial overhead factor. A similar composition satisfies the relation $\leq_o^{\log}$.

The second fact follows from the composition of poly-time algorithms being poly-time.

The third fact uses that same log-space memory trick that proved transitivity for $\leq_\tau^{\log}$.

Which reduction we use depends on our setting. In L we would use a log reduction as a polytime reduction would be trivial. The choice must be the appropriate tool to prove a separation for the reduction to have significance.

3 Completeness

A consequence of the existence of universal Turing machines is ability to separate out certain problems of a class as "complete" for that class. This notion distinguishes the hardest problems of a class.

Definition 5 (Complete)**.** *Given a class of* C *of computation problems, we call* B complete *for* C *under* $\leq$ *if* $B \in C$ *and* $\forall A \in C$, $A \leq B$.

It is not obvious that such a B exists for a given C. This is a very strong notion of hardness. We can use universality to show such B exist for our standard classes.

Theorem 2. P *has a complete problem under* $\leq_m^{\log}$

Proof. Take $K_D = \{\langle M, x, 0^t\rangle |\ M$ is a Turing machine that accepts x in at most t steps$\}$. 0^t is a unary time measure, a string of all zeroes. The name's origin is K from 'komplett', the German word for complete and D for deterministic.

$K_D \in$ P because of our universal Turing machine runs with only a polynomial time overhead so our first condition for completeness is fulfilled.

To prove our second condition holds, we can show $\forall A \in$ P, $A \leq_m^{\log} K_D$. Let M_A be a polytime Turing machine such that $C(M_A) = A$. $A \in$ P implies $\exists c$ such that DTIME$(M_A) = O(n^c)$. Consider $f_A : x \mapsto \langle M_A, x, 0^{|x|^c}\rangle$. M_A is fixed so can be hard wired. x is merely copied. $0^{|x|^c}$ can be computed in log space by hardcoding c into the map f. Thus, this function is efficient. The separation between entries in the three-tuple can be done easily with special characters. □

Note that we must ensure the reduction tool used isn't granted too much power. In the case of P we think that $\leq^{\log}$ is meaningful as a tool to determine whether we can collapse P to L or whether the inclusion L $\subset$ P is strict.

There is a similar complete problem for the non-deterministic model that we will introduce later and talk more about next lecture.

4 Non-Determinism

Non-determinism is not a physically realizable model of computation. We use it because it captures an important class of problems. Recall that the difference between a deterministic Turing machine and a non-deterministic Turing machine (NTM) is a relaxation of the transition function to allow a set of valid output states, transforming it from a function to a relation. In this model there are multiple valid computation paths and we accept if any one of them reaches an accepting state. Note that we allow the case of no valid output states in our transition relation.

Non-determinism is mainly used when considering decision problems. Thus we may speak of the language of a machine M as $L(M) = \{x \in \Sigma^* | \exists$ a valid computation on x that leads to acceptance $\}$.

There are several ways of interpreting this model. We can think computational steps in this setting as spawning many child threads on each of the possible output states of a transition which go on to evaluate in parallel. Another view would be a computation that magically always picks the right path toward an accepting state if such a state exists. A third interpretation uses graphs, we will talk about this later when we discuss space bounded determinism.

Given that many computation paths will now return the answer how do we define the time and space functions? With an NTM, it is possible that for some inputs x, some acceptance/rejection paths are very short and use little space, while other paths are much longer. Here we take a worst case measure by considering the maximum over all valid computational paths of the number of steps taken. Formally, time is measured as $t_M(x) =$ max number of steps needed by M to compute x. Thus we have a consistent definition of NTIME$(t(n)) = \{$ set of decision problems that can be solved by a NTM in time $O(t(n))\}$ and NSPACE$(t(n)) = \{$ set of decision problems that can be solved by a NTM in space $O(t(n))\}$.

Of course, we are most interested in robust time and space bounds so define classes to respect that. The following complexity classes are analogs to their deterministic namesakes.

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

In the coming lectures we will prove that PSPACE = NPSPACE. We will note that, just like the deterministic model, NL $\subset$ NP $\subset$ NPSPACE $\subset$ NEXP and that NL $\neq$ NPSPACE and NP $\neq$ NEXP.

4.1 Universal NTM

Just as there exists a universal deterministic Turing machine there exists a universal Turing machine in the non-deterministic model, an more efficient machine than its counterpart as here we can get rid of the extra log factor of time cost. Unfortunately, with the new machine our corresponding hierachy theorem becomes much more complicated. We can't simply reverse the answer for easy contradiction in a diagonalization argument. This relates to the NP, coNP problem.

Trivially, we can get the same resource bounds from our universal NTM by using the same constructions as the deterministic model. To eliminate the extra log factor, we have to be more clever.

Theorem 3. *There exists a NTM U_{NTIME} such that $t_{U_{\text{NTIME}}}(\langle M, x, 0^t\rangle) = O(\text{poly}(|M|)t)$ and for all $t \geq t_M(x)$, $U_{\text{NTIME}}(\langle M, x, 0^t\rangle) = M(x)$.*

Proof. Our machine uses two phases. In phase 1, 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 Ms tape heads, new cell contents under each of Ms tape heads, and new internal state. The length of this computation record is $O(\text{poly}(|M|)t)$.

In phase 2, we check the validity of that sequence against memory. Thus if δ_j is the first transition that affects tape location a after transition δ_i writes to it, the content that δ_j reads is the same that δ_i writes. Figure 3 demonstrates the situation where $j = i + 4$ is the first transition after δ_i wrote to location x_i to read that location.

We can check each tape individually and assume the transitions on the other tapes are valid. Thus we can work on a fixed number of tapes no matter how many tapes we're simulating. This lets us avoid the extra overhead incurred by the universal deterministic Turing machine.

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. The guess and check phases both run in time linear in t so the time bound is $t_{U_{\text{NTIME}}}(\langle M, x, 0^t\rangle) = O(\text{poly}(|M|)t)$.

□

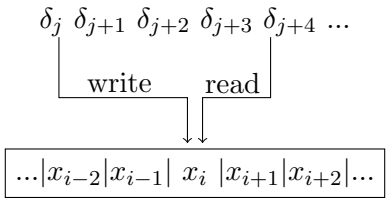

Figure 3: Transition Validity

With respect to completeness, the problem $K_N = \{\langle M, x, 0^t\rangle | M$ is a NTM that accepts in less than t steps$\}$ is complete for NP under $\leq_m^{\log}$ by the same reasoning that K_D was complete. However, this is not the most intuitive problem to consider when working with NP-completeness.

The additional argument 0^t tells us how many transition steps to guess. If we don't know this running time estimate we can get it efficiently by first setting $t = 2$ and doubling our estimate of t after each failure by the simulation to halt within the current time bounds.

5 Coming Up

Next lecture we will discuss NP-completeness more, discuss more natural problems in NP such as SAT and VERTEX-COVER, introduce verifiers, and talk about coNP. Eventually, we will also look at the hierarchy theorem for non-deterministic time.

Acknowledgements

In writing the notes for this lecture, I sampled from the notes by Tyson Williams for lecture 2 from the Spring 2010 offering of CS 710 to revise and expand the section on the Time Hierarchy Theorem for deterministic Turing machines, the section on reductions, and the section on Completeness. I similarly used the notes by Andrew Bolanowski for lecture 3 from the Spring 2010 offering of CS 710 to improve my section on non-determinism.

The material in the lecture notes by Matt Elder for lecture 2 from the Spring 2007 offering of CS 810 overlap with the material presented here and offer an alternate guide.