CS 710: Complexity Theory 9/8/2011

Lecture 2: Universality

Instructor: Dieter van Melkebeek Scribe: Aaron Gorenstein

This lecture discusses the issues we confront when pursuing our goals in complexity theory, introduces the notion of the universal Turing machine, and examines the repercussions of that Turing machine. In particular, universal Turing machines (UTM) will start to allow us to establish hierarchy theorems.

1 The Goal of Complexity Theory

Recall that our goal is evaluating the complexity measures $t_M(n)$, the most time it takes to solve an input of length n, and $s_M(n)$, the most space used. Our general goal is, given $R \subseteq \Sigma^* \times \Sigma^*$, find the smallest $t_M(n)$ and $s_M(n)$ over M which solve R. This serves as a measure of that problem's complexity.

In finding the "best," however, there are several issues to consider. Underlying all of this is the motivation to find machine-independent results capturing the complexity of computing relations.

1.1 Issue 1: Comparability

In terms of finding the "best" $t_M(n)$ there is the obvious problem: what if $t_{M_1}(n_1) > t_{M_2}(n_1)$, but $t_{M_1}(n_2) < t_{M_2}(n_2)$? In other words, what if between two TMs one is better on one input, and the other machine is better on a different input? In that case it is unclear which is the better machine to have.

Because of this, we instead look at the dual of the problem we originally stated: rather than trying to find minimal $t_M(n)$ for a given relation, we instead try to determine which relations can be computed given a certain time or space bound.

1.2 Issue 2: Hardwired Solutions and Speedup

It is possible to "hard-wire" the solution to a finite number of input instances in the finite control of a TM. This contributes to the comparability issue above. For any finite subset in our relation, there *is* a TM which computes extremely quickly and with little space.

We resolve this issue by considering only the *asymptotic* behavior of the machines.

Speedup is a similar concern. We can "speedup" (reduce) memory usage by increasing the alphabet size: instead of $\{0, 1\}$ we can have $\{00, 01, 10, 11\}$ which would halve the number of cells in use. This significantly reduces the space used. The space usage can be decreased by any constant factor by increasing the size of the tape alphabet to a suitably large constant. Notice that if we had used the alternative definition of space usage that only counts memory cells that are touched during the computation, then this argument would only work for random access machines for algorithms that have a high amount of memory locality.

The number of computation steps needed can also be reduced in a similar, but more complex, way. If a TM spends some time processing over 5 cells, going back and forth, increasing the alphabet size such that those 5 cells are collapsed into 1 would mean all of that computation could be reduced to a single step. This leads to the next question, what if the computation happens to occur between these “super”–cells? You can remember the cell you’re on *and* the left and right cells. Ultimately, we can reliably reduce the number of steps in the computation.

Because of these constant factor speedups, we ignore constant factors in running time and space usage. This is formally realized through the “big-O” notation.

1.3 Issue 3: Robustness

Recall last time we made some design decisions concerning our model of computation. We have random access instead of sequential, but also our index tape is erased after each use, and other such details. Our decisions about these details are not important, and choosing one way or another should not affect our pursuit of complexity goals.

We would like our results to be *robust*. We would like our results to be independent of the particular choices we have made in defining the Turing machine. Recall the *Church-Turing thesis* - that any relation computable on a physically realizable computing device can also be computed on a Turing machine. This belief underscores the use of the Turing machine in computability theory as the computing device that is studied. Note that the Church-Turing thesis is not violated by any of the known models of computation.

Robustness with respect to model While sufficient for computability, the above Church-Turing thesis makes no mention of resource usage. Enter the *Strong Church-Turing thesis*, which states that any relation computable on a physically realizable computing device can be computed by our model of a Turing machine with a polynomial overhead in time and a constant overhead in space. Namely, if some machine uses $t(n)$ time to compute the relation, there is a Turing machine using $\text{poly}(n, t(n))$ time; and if there is a machine using $s(n)$ space, there is a Turing machine using $O(\log n + s(n))$ space. Notice that the Strong Church-Turing thesis is a much bolder statement than the Church-Turing thesis. In particular, it is widely believed to be violated by quantum machines (although it is debatable if these are physically realizable), and may even be violated by randomized machines (although this is believed not to be the case).

Following from the Strong Church-Turing thesis, if we have our complexity measures accurate “within a polynomial,” then our measures are robust with respect to any computational model we choose.

Robustness with respect to input encoding Consider the shortest path problem. The input to the problem includes a graph, which must be represented somehow on the input tape. Standard methods of representing a graph include an adjacency matrix and an adjacency list. Note there may be a linear factor difference in the size of these. This effects the running time and space usage because these are functions of the size of the input. If an input is made to be artificially large (say, by padding with unnecessary 0’s) then the running time and space usage are artificially decreased as functions of the size of the input.

For this reason, we always assume a “reasonable” encoding of the input. We leave this notion qualitative, although it can be quantified. As long as we choose a reasonable encoding of the input, the running time and space usage do not depend too much on the input encoding.

2 Complexity Classes

We choose our complexity classes in context of the above robustness issues. The complexity class a problem falls into remains the same regardless of the particular model or reasonable input encoding chosen.

With the sets DTIME and DSPACE (which are *not* robust classifications!) defined below, we can build robust classifications.

Definition 1. *For a given $t : \mathbb{N} \to \mathbb{N}$ and $s : \mathbb{N} \to \mathbb{N}$, we define:*
DTIME($t(n)$) = { *Decision problems solvable in $O(t(n))$ time by some random access Turing machine* },
DSPACE($s(n)$) = { *Decision problems solvable in $O(s(n))$ space by some random access Turing machine* }.

What follows are the definitions of the standard (deterministic) complexity classes. Note that almost all of them are considered a union over all polynomials, either for space or time.

Definition 2. *The definitions of standard complexity classes.*

$$\begin{array}{lcl} \mathrm{P} & = & \cup_{c>0}\,\mathrm{DTIME}(n^c), \\ \mathrm{EXP} & = & \cup_{c>0}\,\mathrm{DTIME}(2^{n^c}), \\ \mathrm{E} & = & \cup_{c>0}\,\mathrm{DTIME}(2^{c\cdot n}), \\ \mathrm{L} & = & \mathrm{DSPACE}(\log n), \\ \mathrm{PSPACE} & = & \cup_{c>0}\,\mathrm{DSPACE}(n^c). \end{array}$$

The only exception to the polynomial is E, in which we have to take care about the *input encoding*. E is still robust with respect to the model. All of the classes are robust with respect to model, and all but E for input.

It may seem that restricting the space usage to logarithmic in the case of the complexity class L is too great. For graph problems, logarithmic space is only enough to remember a constant number of vertexes within the graph. However, it turns out that many interesting problems can be solved within L. For example, a recent result showed that the problem of determining connectivity in an undirected graph can be achieved using logarithmic space on deterministic machines.

We are quite interested in relationships between these standard complexity classes. We will later show the following relationships between complexity classes:

$$\mathrm{L} \subseteq \mathrm{P} \subseteq \mathrm{PSPACE} \subseteq \mathrm{EXP}.$$

Note that $P \subseteq$ PSPACE is pretty straightforward, but the other inclusions may require some thought. It is open whether any of these containments is proper.

3 Universality

We now understand that Turing machines in general are robust for measuring complexity, but ideally we would like just one Turing machine to work with. As TMs can be described as strings, why not use a universal TM (UTM) which takes a machine description and input pair as its input, and simulates the machine on the input?

The basic definition of a universal TM (UTM) is simply the following:

$$U(\langle M, x\rangle) = M(x)$$

On an input string specifying the description of the TM M and its input x, have the same result as M would on x.

For computability such simulation is sufficient, but in complexity we need to take into account the resource-usage overhead of our simulation. Are there UTMs that can simulate all TM descriptions efficiently with respect to space or time? Fortunately, such UTMs exist.

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

$$t_{U_{\text{DTIME}}}(\langle M, x\rangle) = O(|M| \cdot t_M(x) \cdot \log(t_M(x))) \tag{1}$$
$$s_{U_{\text{DSPACE}}}(\langle M, x\rangle) = O(\log(|M|) \cdot s_M(x)), \tag{2}$$

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

Note that for time the only real overhead is a logarithm. For space it is even better—constant factor overhead.

Not only does there exist those TMs, but they are in fact one in the same. There is one TM which is both. Given any description of a TM which can compute a problem with such-and-such resources, we can have (the same, always) precisely-specified UTM that can simulate it with the above, known, overhead. Let us consider a sketch of the proof for this, this UTM.

Sketch of Universal TM. Our input TM can have any number of work tapes. How can we keep up? We have a single *storage* tape. If the input TM has k tapes, then the storage tape stores the contents of the first cell in the first work tape, then the first cell in the second work tape, and so on for all k work tapes. Then it proceeds to store the second cell of the first work tape, 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.

While this "interleaving" stores the general contents well enough, each tape can have a head in a very different position. Hence our second tape, the *locator* tape, on which we store the location of the heads. We can copy one of the addresses on the locator tape into our index tape, and so instantly have our work-tape head be at the location of the current work-tape head it is trying to emulate.

Lastly, as we are using the random access model, our universal TM must itself have its own index tape!

Note that the the finite control of the TM the UTM is emulating may either be easily accessible as part of the input string on the storage tape, or can just prepend room for it on the front of the storage tape.

The extra logarithmic factor primarily comes in if our universal TM must emulate a sequential access machine. For every access the sequential machine makes, we have to make an index request, which is an additional log overhead. [1]

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

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).

That we emulate in proportional work space is pretty straightforward. $\square$

4 Hierarchies

So we have our universal TM. How can we use it to reason about computational complexity? We can confirm our intuition that increasing the amount of time or space a TM has access to increases the languages it can compute. A *hierarchy* is an intuitive sequence of classes with inclusions that are known or suspected to be strict. We have hierarchies over time and space resources. We can show that these inclusions for resources are strict with *diagonalization*.

Let us review Cantor's theorem:

Theorem 2 (Cantor's Diagonalization Theorem)**.** *The set of* $\{0,1\}^\infty$ *is not countable.*

The set of infinite sequences is closely related to the reals in $[0,1]$, just put a decimal point before the infinite sequence. Recall that "countable" means we can label each element with an integer.

Proof. We can prove this by contradiction. Suppose that the set of sequences is countable. Then there exists an enumeration of them, we can label each one the first, the second, and so on. Then we can make a table of them (figure 1). Each row is an entry on the table, and each column is the j^{th} bit.

	b_1	b_2	b_3	$\cdots$
s_1	**0**	1	1	$\cdots$
s_2	1	**0**	0	$\cdots$
s_3	1	0	**1**	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$
s'	1	1	0	$\cdots$

Figure 1: s' is the complement of the (bold) diagonal elements.

Now we can compose the new sequence s' such that its bit b_j "disagrees" with the the b_j bit of s_j. The start of this new s', given the above table, would be 110, as you see those values are the "opposite" of the values on the diagonal. Thus the sequence cannot be s_j, for *all such s*. So we have a sequence that is not on the table of all sequences. Contradiction. $\square$

4.1 Establishing Hierarchies

This diagonalization technique is similar to the one we will use for hierarchy results. The idea is to use our universal Turing machine to emulate each other Turing machine that runs in time $t(n)$. Our UTM simulates the target running on its own string description, and when the target either accepts or rejects, our UTM has enough time to "disagree." This is similar to the diagonalization technique above. Our UTM will simulate the j^{th} TM on the j^{th} lexical input, and then disagree, just as the binary sequence looks at the j^{th} sequence's j^{th} bit.

In doing so, we define a *language* that cannot be recognized by a TM running in time $O(t(n))$. We can establish analogous results for space as well. This is the separation result, demonstrating that more of a resource really does allow for more languages to be recognized (equivalently, more relations to be computed).

More formally, the statement that we will show is

Theorem 3.

$$\mathrm{DTIME}(t(n)) \subsetneq \mathrm{DTIME}(t'(n))$$

Where $t : \mathbb{N} \to \mathbb{N}$, $t' : \mathbb{N} \to \mathbb{N}$, and $t'(n) = \omega(t(n) \log t(n))$. We will also need one more minor condition which we will discuss later.

And more formally, we can set up our diagonalization table like so:

Sketch of Time Hierarchy Separation. We want to list all of the $t(n)$-time-bounded TMs. However, we cannot compute such a list, because that is uncomputable. So we simply include *all* Turing machines.

Our Turing machine takes as input $\langle M, x\rangle$, a Turing machine description and an input for that machine to run on. Recall our machine runs in time $\omega(t(n)\log(n))$. That is enough time to run the simulation for the necessary $t(n)$ steps, and then "disagree" with the result. □

There are a few final formal points to consider.

When proceeding through the list of all Turing machines, what if our simulation doesn't end in time? In that case, there are two possibilities.

- The target is not a $t(n)$ machine, so it does not matter. We can arbitrarily choose to always reject.
- The target *is* a $t(n)$ machine, but it has not yet reached its asymptotic behavior, so is running "slower." Due to this consideration, we construct our initial TM enumeration such that all TMs are enumerated infinitely often. Thus, we'll run into this machine again with a longer input, and then be able to properly disagree—or if not that time, the next. In the meantime, we can arbitrarily decide to reject.

Why is our separation particularly for time $\omega(t(n)\log n)$? The logarithmic factor is simply an effect of the time our simulation takes. If we improve the simulation time, we can tighten the hierarchy. The analogous hierarchy for space has only constant-factor separation.

What is the mechanism we use to "time" the TMs to determine if they may be one with $t(n)$ asymptotic behavior? We can easily keep track of whether we're within $t(n)$ time so long as that value is *time constructable.*

Definition 3. *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 $O(s(n))$.*

Time-constructable means that $\exists TMM_t$ such that on input 0^n (any input of length n, really) M outputs $0^{t(n)}$ in time $O(t(n))$. So if M_t is our clock, every "tick" is when we run it until it produces another zero.

We can choose to either time the $t(n)$ TM we're simulating, or our own $t(n)\log(n)$ time consumption. Many functions are time-constructable.

5 Coming up next

Finish formally establishing the hierarchy theorems and introduce completeness and nondeterminism.

Acknowledgments

In writing the notes for this lecture, I made use of the notes by Jeff Kinne for lecture 1 in Spring 2010 as well as the notes by Tyson Williams for lecture 2 again in Spring 2010.