CS 221 Computational Complexity, Lecture 2 Jan 25, 2018

Time/Space Hierarchy Theorems

Instructor: Madhu Sudan *Scribe: Garrett Tanzer*

This lecture discusses the proof technique of diagonalization, particularly as it relates to time, space, and other resource hierarchies—"more of the same resources → more power". Similar material is covered in Arora and Barak [1] Chapters 3.1 and 3.2.

1 Definitions

1.1 Algorithms

Before we get to content, we have to clarify exactly what we mean when we say "algorithm".

1.1.1 Properties

1. An algorithm is a *finite description*: a constant-sized set of rules that solves a problem on inputs of all lengths. This is a *uniform* model of computation, as opposed to a *nonuniform* model like circuits, where each must solve the problem for one input length n.
2. An algorithm is *interpretable*. This means that a second algorithm can receive the first's description as input and simulate its behavior. This gives rise to the notion of a "universal" Turing machine—a constant-sized machine that can simulate any other Turing machine on any input, with moderate overhead.
3. Algorithms are *enumerable*. This follows from the previous two properties, and is reflected in the convention that we represent algorithms as binary strings. We can therefore conceive of a sequence $A_1, A_2, \ldots, A_i, \ldots$ of all possible algorithms. It is important to note that all items in this enumeration must be valid programs. We can ensure this trivially by specifying in the computational model that, for example, syntactically invalid programs output 0 on all inputs.

1.1.2 Models

It is important to have established a rigorous computational model, like the canonical Turing machine, in order to precisely analyze the complexity of our algorithms. However, we will not focus too much on the details of these models because there is a large body of work showing their equivalence, with at most polynomial overhead. While this might be an important distinction in finer algorithmic analysis, it is negligible when comparing polynomial and exponential complexity, as we will often do.

Turing Machines

The salient features of a Turing machine are a constant-sized machine—often called M—containing a set of rules for state transitions, and a series of memory tapes. We will assume one input tape, one output tape, and two independently addressable working tapes; there are many such variations that may have slight differences in complexity, but they are subsumed in the context of $poly(n)$.

NAND++ Programs

Those who took the most recent iteration of CS 121 will be more familiar with NAND++, which can simulate and be simulated by Turing machines with polynomial overhead.

C Programs

Higher-level languages in the abstract are also Turing equivalent; this can be useful in order to avoid getting bogged down in the details of a particular model.

1.2 Languages

A language is a set of binary strings: $L \subseteq \{0,1\}^* \equiv \bigcup_{n \geq 0} \{0,1\}^n$.

There is a natural decision problem associated with each language: given $x \in \{0,1\}^*$, decide if $x \in L$?

1.3 Computability

Once we have formal definitions, it makes sense to ask what problems we can actually solve on a computer. The first notion of this, developed from the '30s–'50s, was computability.

Definition: L is computable (decidable) if there exists an algorithm A such that:

- $\forall x,\, x \in L \iff A(x) = 1$
- $A(x)$ halts on all inputs

Note that "halting" just means running in finite time for an input, even if that finite time is $O(2^{2^{2^n}})$.

1.4 Tractability

Because a binary yes/no answer seemed to be insufficient to describe the hardness of computational problems, in the '60s there was a movement toward recognizing **P**, defined below, as the set of tractable or efficiently solvable decision problems. See Cobham [2], Edmonds [3], and Peterson [4]. We introduce the notion of *time* and *space complexity* in order to compare the hardness of deciding languages.

Definition: $\mathbf{TIME}(t(n)) \equiv \{L \mid \exists A \text{ solving } L \text{ with running time } O(t(|x|)) \text{ for every } x\}$
Definition: $\mathbf{SPACE}(s(n)) \equiv \{L \mid \exists A \text{ solving } L \text{ using space } O(s(|x|)) \text{ for every } x\}$

Definition: $\mathbf{P} \equiv \bigcup_{c>0} \mathbf{TIME}(n^c)$

2 Diagonalization

Theorem: $\mathbf{TIME}(n^2) \subsetneq \mathbf{TIME}(n^3)$

While it is difficult to prove relationships comparing power of different types of resources, we *are* able to prove that providing appreciably more of a single resource increases power. Today's main theorem, the Time Hierarchy Theorem, can be stated generally as $\mathbf{TIME}(f(n)) \subsetneq \mathbf{TIME}(f(n)\log(f(n)))$, while the Space Hierarchy Theorem is even tighter at $\mathbf{SPACE}(f(n)) \subsetneq \mathbf{SPACE}(\omega(f(n)))$; however, we will focus on a more concrete version to simplify the proof. We will develop the technique of "proof by diagonalization" in order to do so.

2.1 Cantor's Theorem

Cantor's Theorem states that the number of real numbers is greater than the number of integers, or that the set of real numbers is uncountably infinite ($|\mathbb{R}| > |\mathbb{Z}|$). We can without loss of generality restrict reals to the interval $[0,1]$ and integers to natural numbers and derive the following:

Theorem: There is no injective function $f : [0,1] \to \mathbb{N}$.

Proof: We prove this by "diagonalization". Assuming for the purpose of contradiction that an injective function f exists, we will enumerate $f^{-1}(\mathbb{N})$ and list for each its real-valued (possibly infinite) binary representation. If we come across a natural number for which f^{-1} is undefined (which can happen because f is

assumed to be injective, not necessarily surjective), we can fill the row with 0s or some other convention.

Now, take the complement of each number on the diagonal and treat the sequence as a new real number. The resulting number isn't equal to any row (because $\forall i$, $f(i)$ differs on the diagonal bit), but is still in $[0,1]$. Therefore, f isn't injective, which is a contradiction. So, there exists no injective function $f : [0,1] \to \mathbb{N}$. ■

	$\mathbb{R}$					
$f^{-1}(1)$	0	1	1	0	1	...
$f^{-1}(2)$	1	0	1	1	0	
$f^{-1}(3)$	1	0	1	1	0	
$f^{-1}(4)$	0	0	0	0	0	
$f^{-1}(5)$	1	1	0	0	1	
$\vdots$	$\vdots$					$\ddots$
	1	1	0	1	0	...

A concern was brought up about the equivalence of 1 and 0111... in binary decimals, but we can avoid this problem by accepting only one such representation or periodically adding a row of all 1s so a 0 is introduced into the diagonal complement.

2.2 Weak Turing's Theorem

We can apply an analogous version of this argument to the weak version of Turing's theorem.

Theorem: $\exists L \subseteq \{0,1\}^* \mid L$ is not decidable.

Proof: We will repeat the same process above, letting the number of languages correspond to reals and the number of algorithms correspond to the number of integers. If we relabel rows as algorithms and columns as inputs, we see that we get the same result. ■

	x_1	x_2	x_3	x_4	x_5	...
A_1	0	1	1	0	1	...
A_2	1	0	1	1	0	
A_3	1	0	1	1	0	
A_4	0	0	0	0	0	
A_5	1	1	0	0	1	
$\vdots$	$\vdots$					$\ddots$
	1	1	0	1	0	...

2.3 Turing's Theorem

Now we will prove the strong version of Turing's Theorem using the weak version.

Theorem: $HALT = \{(A, x) \mid A \text{ halts on input } x\}$ is undecidable.

We also define the diagonal halting problem and its complement.

$D\text{-}HALT = \{A \mid (A, A) \in HALT\}$.
$\overline{D\text{-}HALT} = \{A \mid (A, A) \notin HALT\}$.

	A_1	A_2	A_3	A_4	A_5	...
A_1	0	1	1	0	1	...
A_2	1	0	1	1	0	
A_3	1	0	1	1	0	
A_4	0	0	0	0	0	
A_5	1	1	0	0	1	
$\vdots$	$\vdots$					$\ddots$
	1	1	0	1	0	...

Proof: Assume for the sake of contradiction that $HALT$ is decidable. If $HALT$ is decidable, then $D\text{-}HALT$ is decidable because it is a special case of $HALT$. If $D\text{-}HALT$ is decidable, $\overline{D\text{-}HALT}$ is decidable using the complement. This is a contradiction, since we know by diagonalization that $\overline{D\text{-}HALT}$ is undecidable by any algorithm. Therefore, $HALT$ is undecidable. ■

2.4 Time Hierarchy Theorem

Now we want to modify this argument to prove $TIME(n^2) \neq TIME(n^3)$.

2.4.1 Enumerating $TIME(n^2)$

First, we need to figure out how to enumerate all algorithms that run in time $(O(n^2))$. It is not obvious how to do this, particularly because the language $\{A \mid A \in \mathbf{TIME}(n^2)\}$ is undecidable by Rice's Theorem, but it can be done. We will enumerate over all triplets (A, n_0, c), which represent the algorithm A run for $cn^2 + n_0$ time steps. If $A(x)$ halts in that time, output $A(x)$, else output 0.

We know that this enumeration scheme is exhaustive by the definition of Big O notation:

Claim: $\forall A, n_0, c,\ (A, n_0, c) \in \mathbf{TIME}(n^2)$
Claim: $\forall A \in \mathbf{TIME}(n^2),\ \exists n_0, c \mid A \equiv (A, n_0, c)$

	x_1	x_2	x_3	x_4	x_5	...
$(A_1, 3, 0)$	0	1	1	0	1	...
$\vdots$	$\vdots$					
$(A_1, 5, 10)$	1	0	1	1	0	
$\vdots$	$\vdots$					
$(A_2, 7, 6)$	1	0	1	1	0	
$\vdots$	$\vdots$					$\ddots$
	1	1	0 ...			

Therefore, we have proven that there exists a language that *cannot* be decided in time $O(n^2)$. But we still need to prove that this language *can* be decided in time $O(n^3)$.

2.4.2 Simulating in n^3 Time

Our strategy to compute this $L \notin \mathbf{TIME}(n^2)$ we constructed by diagonalization is to simulate the algorithm A_i on a universal interpreter, then negate the input. There are established results showing existence of a universal interpreter that takes $t(n)\log(t(n))$ time and $s(n)$ space to run a program that takes $t(n)$ time and $s(n)$ space, for "nice" values of $t(n)$ and $s(n)$. Therefore, we can interpret a machine in $\mathbf{TIME}(n^2)$ in $2n^2 \log n = o(n^3)$ time.

The problem is that while this A_i will take time $O(n^2)$, the hidden constant behind the O may be 2^i, 2^{2^i}, etc., and $n = |i|$—so we can't necessarily simulate $A_i(x_i)$ in time n^3. We can resolve this issue by using an enumeration $B_1, \ldots, B_j, \ldots$ where each algorithm A_i appears infinitely often, and where there exists an efficient function $i(j)$ that transforms an index in the series of algorithms B_j into the index for the equivalent A_i. That is to say, $B_j = A_{i(j)}$. This way, once n is arbitrarily large, the constant factor of the original, repeated algorithm is arbitrarily small compared to input length.

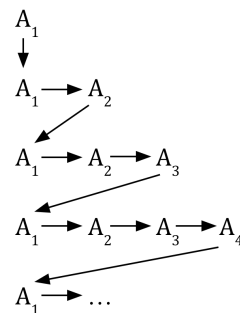

Figure 1: An example of an enumeration where each algorithm appears infinitely often.

Claim: $L \notin \mathbf{TIME}(n^2), L \in \mathbf{TIME}(n^3)$

Proof: We culminate in the following algorithm describing L:

- Find $i(j)$, and get the original (unpadded) algorithm $A_i = B_j$ in time $O(n^3)$.
- Try to compute $A_i(B_j)$ in time $O(n^3)$.
- If $A_i(B_j)$ halts within that time, output $\overline{A_i(B_j)}$).
- Else output 0.

We see that this algorithm runs in $O(n^3)$ time, since we explicitly bound computation. However, we also see that for sufficiently large inputs B_j, the constant factor in A_i is arbitrarily small, so we can definitely complement the output in $O(n^3)$ time. Since we repeat the algorithms (all of which are in $\mathbf{TIME}(n^2)$) an infinite number of times, this proves by diagonalization that $L \notin \mathbf{TIME}(n^2)$. Therefore, $\mathbf{TIME}(n^2) \subsetneq \mathbf{TIME}(n^3)$. ■

2.4.3 Looking Ahead

There are some other interesting results related to the hierarchy theorems that we either aren't yet equipped to engage with or will not cover in this course.

Ladner's Theorem, often invoked in the context of the class **NPI** (**NP**-intermediate), can be used to show that if $\mathbf{P} \neq \mathbf{NP}$, then problems exist that are neither in **P** nor are **NP**-complete. More generally, it states that given two complexity classes $C_1 \subsetneq C_2$, then $\exists C_{1.5} \mid C_1 \subsetneq C_{1.5} \subsetneq C_2$. We can use this theorem repeatedly to find as many strictly contained classes as desired.

Another problem amenable to diagonalization is $\mathbf{NTIME}(n) \stackrel{?}{=} \mathbf{NTIME}(n^2)$. Because the complement becomes more complicated once we add nondeterminism, we can't invert the diagonal as easily as in the proofs for deterministic algorithms. However, using a technique called "lazy diagonalization", it has been proven that:

$$\mathbf{NTIME}(t(n)) \not\subseteq \text{co-}\mathbf{NTIME}(t(n)^2)$$
$$\mathbf{NTIME}(t(n)) \subsetneq \mathbf{NTIME}(\omega(t(n+1)))$$

References

[1] Arora and Barak, *Computational Complexity: A Modern Approach*

[2] Alan Cobham, "The intrinsic computational difficulty of functions" (1965)

[3] Jack Edmonds, "Paths, trees, and flowers" (1965)

[4] Peterson ???