CSE531: Computational Complexity Theory January 9, 2012

Lecture 2 Hierarchy Theorems

Lecturer: Anup Rao

Last time, we claimed the existence of a universal turing machine:

Theorem 1. *There is a turing machine M such that given the code of any Turing machine α and an input x as input to M, if α takes $T \geq 1$ steps to compute an output for x, then M computes the same output in $O(CT)$ steps, where here C is a number that depends only on α and not on x.*

A similar theorem holds for space bounded computation:

Theorem 2. *There is a turing machine M such that given the code of any Turing machine α and an input x as input to M, if α takes $S \geq \log |x|$ space to compute an output for x, then M computes the same output in $O(CS)$ space, where here C is a number that depends only on α and not on x.*

However, proving the space bounded version requires some care. We need the concept of a configuration graph of a turing machine. A configuration of a Turing Machine is the contents of all of its tapes, head locations and current state (or line of code). Given a fixed input x to the machine M, the configuration graph $G_{M,x}$ is the directed graph whose vertices are configurations of the machine during its execution, and there is an edge (u, v) if and only if the configuration u changes to v after one step.

Then we have the following by counting all possible configurations:

Lemma 3. *If a turing machine uses space $s(n) \geq \log n$, the number of configurations on input x is at most $c = 2^{O(s(|x|))}$.*

Observe that if the Turing Machine is going to halt, it must halt in c steps, since after so many steps it will repeat a configuration (by the pigeonhole principle). and so enter an infinite loop. Thus the simulation needs to simulate the Turing machine for c steps. All of this can be done in space $O(s(|x|))$.

1 Diagonalization

Our main weapon to show that Turing Machines have trouble computing things is the technique of diagonalization. This is a technique that has its roots in classical mathematics. Let us use it to prove a classical theorem:

Theorem 4. *There is no way to list every real number in the interval* $[0, 1]$.

Proof Suppose not, and consider any enumeration of these real numbers. Let x be the real number of the form $0\ldots$, where the i'th digit of x is not equal to the i'th digit of the i'th real number in the enumeration.

Then x cannot appear in the list of real numbers, for if it did occur in position j, its j'th digit would disagree with x's j'th digit. ■

A very similar idea gives an impossibility result for Turing Machines. We shall need some notation. Given a string α, we write M_α to denote the Turing Machine whose code is is α.

Theorem 5. *There is a function that is not computed by any Turing Machine.*

Before we see the the simple proof, let us point out that this is philosophically a very powerful fact. A consequence of it is that assuming the Church-Turing Thesis is true, there are some ways to manipulate information that can never occur in the universe. It seems hard to imagine a physical process that violates the Church-Turing thesis, and it also seems hard to stomach the fact that the universe cannot manipulate information in a particular way, yet one of those two (admittedly wishy washy) strange things must happen.

Proof

Consider the function $f : \{0,1\}^* \to \{0,1\}$ defined as follows:

$$f(\alpha) = \begin{cases} 1 & \text{if } M_\alpha(\alpha) = 0 \\ 0 & \text{else.} \end{cases}$$

No Turing Machine can compute this function, for if there was some machine that could, then let γ denote the binary encoding of its code. Then we have that $M_\gamma(\gamma) = f(\gamma)$, but this contradicts the definition of f. ■

You may object that the uncomputable f that we found above is very unnatural, but actually it is not hard to come up with natural examples that are also impossible to compute using Turing Machines.

For example, we can define the function $\mathsf{HALT} : \{0,1\}^* \to \{0,1\}$ that takes as input two strings α, x, and then decides whether $M_\alpha(x)$ halts or runs forever. This seems like a very useful function to compute, but it is also uncomputable.

Theorem 6. HALT *is not computable by a Turing Machine.*

Proof Suppose it was. Then consider the machine M that on input α first simulates $\mathsf{HALT}(\alpha, \alpha)$. If the answer is that $M_\alpha(\alpha)$ halts, then M simulates $M_\alpha(\alpha)$ and outputs the opposite of its output. If $M_\alpha(\alpha)$ does not halt, then M outputs 0. Then M computes the uncomputable function f above. ■

2 Hierarchy Theorems

In this section, we shall show that Turing Machines are strictly more powerful when given more resources.

Let us start by formalizing the notion of the class of functions computable in a certain time.

Definition 7. *We say that a function $f : \{0,1\}^* \to \{0,1\}$ is in $\mathsf{DTIME}(t(n))$ if there is a Turing Machine computing f that halts in time $O(t(n))$.*

Question: Does giving a Turing Machine more time actually allow it to compute things that it cannot compute without the extra time?

We are going to use diagonalization to show that Turing Machines that have more resources can compute things that are not computable by Turing Machines with fewer resources. The basic idea is that a Turing Machine with more resources can simulate every machine that requires fewer resources and do the opposite of what it does on *some* input.

However, we shall need a few concepts to formalize this.

Definition 8 (Time Constructible Functions)**.** *We say that the map $t : \mathbb{N} \to \mathbb{N}$ is time constructible if $t(n) \geq n$ and on input x there is a Turing Machine that computes $t(|x|)$ in time $O(t(|x|))$.*

Definition 9 (Space Constructible Functions)**.** *We say that the map $s : \mathbb{N} \to \mathbb{N}$ is time constructible if $s(n) \geq \log n$ and on input x there is a Turing Machine that computes $s(|x|)$ in space $O(s(|x|))$.*

To simplify the proof, we shall restrict the way Turing Machines are encoded into binary strings. We shall assume that our encoding satisfies the properties that:

- Every machine is encoded by an infinite number of different binary strings. This can be achieved, for example, by adding irrelevant bits of code (like comments) to our programming language
- Every binary string encodes some Turing Machine.

Given an encoding satisfying these assumptions, we are ready to prove the hierarchs theorem:

Theorem 10 (Time Hierarchy)**.** *If r, t are time-constructible functions satisfying $r(n) \log r(n) = o(t(n))$, then $\mathsf{DTIME}(r(n)) \subsetneq \mathsf{DTIME}(t(n))$.*

In order to avoid distracting details, we shall prove the easier case when $r(n) \log r(n) = o(t(n))$.
Proof Let $\ell(n)$ be the function $r(n) \log(r(n))$. Then consider the machine M that on input x, simulates the computation of $M_x(x)$. We shall ensure that M runs for $\ell(|x|)$ steps (if it runs for longer, M will terminate itself). If the simulation terminates with output 0, M outputs 1, else it outputs 0. By Theorem 1, this simulation takes $O(\ell(|x|))$ steps.

Note that M computes a function $f : \{0,1\}^* \to \{0,1\}$ such that

$$f \in \mathsf{DTIME}(\ell(n)) \subseteq \mathsf{DTIME}(r(n)\log(r(n))) \subseteq \mathsf{DTIME}(t(n)).$$

On the other hand, if α is the code of a machine that computes a function in $\mathsf{DTIME}(r(n))$, then if n is large enough, α is guaranteed to halt in $c.r(n)$ steps for some constant c. Since the same machine is represented by an infinite number of strings, there is some long enough representation α' such that $\ell(|\alpha'|) > cr(|\alpha'|)$, then the simulation of $M_{\alpha'}$ must compute the opposite, so we must have that $M_{\alpha'}(\alpha') \neq f(\alpha')$. ■

For the general case, we just need to pick a function $\ell(n)$ that grows faster than $O(r(n))$, but slower than $t(n)$. The rest of the proof is the same. Similarly, by appealing to the simulation promised by Theorem 2, we can prove a space hierarchy theorem:

Theorem 11 (Space Hierarchy)**.** *If q, s are space-constructible functions satisfying $q(n) = o(s(n))$, then $\mathsf{DSPACE}(q(n)) \subsetneq \mathsf{DSPACE}(s(n))$.*

We leave out the details.

3 Gődel's Incompleteness Theorem

We would be remiss not to mention one of the most important results in mathematics at this point, namely Gődel's incompleteness theorem. The theorem concerns the problem of trying to find an axiomatic basis for mathematics. Ideally one would like to find a constant set of axioms that define the base truth in mathematics, in such a way that every true statement is provable using the axioms. Informally, Gődel showed that this project is doomed to failure.

For example, given a set of axioms about the integers like

$$\begin{aligned} a(b+c) &= ab+ac \\ a+b &= b+a \end{aligned}$$

We would like to be able to prove every true statement about the integers. For example the equation $(a+b)^2 = a^2 + 2ab + b^2$ can be derived using the above axioms.

We say that such a set of axioms is *consistent* if the axioms do not contradict themselves. We say that the axioms are *complete* if every equation that is true can be proven to be true using the axioms.

He proved that:

Theorem 12 (Incompleteness Theorem (informal statement))**.** *For every set of axioms about the integers, either the axioms are inconsistent, or they are incomplete.*