Lecture 3: Hierarchy Theorems

Anup Rao

October 11, 2023

In the last lecture, we showed that there are natural functions that cannot be computed by Turing machines. To do this, we used the technique of diagonalization. In this lecture, we shall combine diagonalization with the universal simulation ability of Turing machines to show that Turing machines with more time/space are strictly more powerful than Turing machines with less time/space.

We are going to use diagonalization to show that Turing Machines that have more time can compute things that are not computable by Turing Machines with less time. Such a result is called a *hierarchy theorem*, it shoes that there is a hierarchy of power that comes with increasing computational 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. To formally prove the hierarchy theorems, we need some more concepts:

Definition 1 (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 $O(t(|x|))$.*

Almost every running time or space bound you can think of like $n^5, 2^n, 2^{2^n}$ is time constructible and space constructible. (But not all functions are time constructible, since not all functions can be computed by turing machines). We shall also need a result about simulating turing machines by Turing Machines, that we discussed in the third lecture:

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 $T \geq 1$ steps to compute an output for x, then M computes the same output in $O(CT \log T)$ steps, where here C is a number that depends only on α and not on x.*

We are now ready to prove our first hierarchy theorem:

Theorem 3 (Time Hierarchy)**.** *If r, t are time-constructible functions satisfying $r(n) \log r(n) = o(t(n))$, then $\mathsf{DTIME}(r(n)) \subsetneq \mathsf{DTIME}(t(n))$.*

Proof Recall that M_α denotes the Turing Machine whose code is α. The key idea is to use a function very similar to the one we defined

in the last lecture for our diagonalization proofs:

$$f(\alpha) = \begin{cases} 1 & \text{if } M_\alpha(\alpha) \text{ halts and outputs 0 after } t(|\alpha|) \text{ steps of the simulator,} \\ 0 & \text{else.} \end{cases}$$

We claim:

Claim 4. *f can be computed in time $O(t(n))$.*

To compute f, we first compute $t(|\alpha|)$ and then apply Theorem 2 to simulate $M_\alpha(\alpha)$ for $t(|\alpha|)$ steps of the simulator. So, f can be computed in time $O(t(n))$.

On the other hand, we shall show:

Claim 5. *f cannot be computed in time $O(r(n))$.*

If β is the code of a machine that computes f in time $c \cdot r(n)$. Let C_β be such that the execution of r steps of the machine M_β can be simulated in $C_\beta r \log r$ steps by the universal machine. Then there must be some binary string β' that represents the same machine as β, but is long enough so that

$$t(|\beta'|) > C_\beta \cdot c \cdot r(|\beta'|) \log r(|\beta'|)$$

This is because by assumption $r(n) \log r(n) = o(t(n))$, and so for large enough n,

$$t(n) > 2C_\beta \cdot c \cdot r(n) \log(c \cdot r(n)) > C_\beta \cdot c \cdot r(n) \log(c \cdot r(n)).$$

Moreover, we can always add redundant lines to the code in β, until the code becomes long enough for $t(|\beta'|) > 2C_\beta \cdot c \cdot r(|\beta'|) \log r(|\beta'|)$.

If $M_\beta(\beta') = 0$, then $M_{\beta'}(\beta') = 0$ and so $f(\beta') = 1$ by the guarantee of Theorem 2. If $M_\beta(\beta') = 1$, $M_{\beta'}(\beta') = 1$, and so $f(\beta') = 0$, which proves that M_β does not compute f. ■

Similarly, one can prove a Space hierarchy theorem:

Definition 6 (Space Constructible Functions)**.** *We say that the map $s : \mathbb{N} \to \mathbb{N}$ is* space constructible *if $s(n) \geq \log n$ and on input x there is a Turing Machine that computes $s(|x|)$ in space $O(s(|x|))$.*

We saw in lecture 3 that:

Theorem 7. *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.*

One can prove the following space hierarchy theorem:

Theorem 8 (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, since they are exactly the same as the previous result.

As a consequence of these hierarchy theorems we get:

Corollary 9. $\mathbf{P} \neq \text{Exp}$.

Proof On the one hand, $\mathbf{P} \subseteq \mathsf{DTIME}(n^{\log n})$. On the other hand, by Theorem 3, $\mathsf{DTIME}(n^{\log n}) \neq \mathsf{DTIME}(2^n)$, since $n^{\log n} = o(2^n)$. ■

Hierarchy Theorem for Circuits

We define the class $\mathsf{SIZE}(s(n))$ to be the set of functions $f : \{0,1\}^* \to \{0,1\}$ that can be computed by circuit families of size $s(n)$.

We have proved the following theorems:

Theorem 10. *Every function $f : \{0,1\}^* \to \{0,1\}$ is in $\mathsf{SIZE}(O(2^n/n))$.*

Theorem 11. *For every large enough n, there is a function $f : \{0,1\}^n \to \{0,1\}$ that cannot be computed by a circuit of size $2^n/3n$.*

We can use this theorem to prove a hierarchy bound for size.

Theorem 12. *There is a constant c such that for every functions $s(n), s'(n)$ satisfying $2^n/n > s'(n) > cs(n) > n$, we have that $\mathsf{SIZE}(s(n)) \subsetneq \mathsf{SIZE}(s'(n))$.*

Proof Suppose every function on n bits can be computed using a circuit of size $k2^n/n$. Set $c = 3k$. Let ℓ be such that $k2^\ell/\ell = s'(n)$. Then every function on ℓ bits can be computed by a circuit of size $s'(n)$. On the other hand, there is some function on ℓ bits that cannot be computed using a circuit of size $2^\ell/3\ell = s'(n)/c$, as required. ■

NP

In the last class, we introduced the concept of *complexity classes*. We saw the classes P, L, E, EXP and $PSPACE$. These classes were obtained by considering functions that can be computed with limited time or limited space. Today, we explore a different kind of class, the class NP.

Recall: $L \subseteq P \subseteq PSPACE \subseteq EXP$.

NP is interesting chiefly because many problems that we would like to solve efficiently with a computer, but cannot solve, belong to NP. The list of such problems includes essentially all problems

solved today with machine learning, and many other practically important problems. Before giving the definition of NP, let us see some examples of problems in NP.

Independent Set Given a graph G and a number k, does the graph have an independent set of size k? Let $ISet(G,k) = 1$ if the graph has an independent set, and 0 otherwise.

Recall that an independent set is a set of nodes that does not contain any edges.

Subset sum : Given a list of numbers $a_1, \ldots, a_\ell, t$, is there some subset of the numbers $a_1, \ldots, a_\ell$ that sums to t? Let $SubSum(a_1, \ldots, a_\ell, t) = 1$ if there is such a subset, and 0 otherwise.

Composite numbers : Given a number N, decide if it is composite or not. Let $Comp(N) = 1$ if N is composite, and 0 otherwise.

Matching : Given a graph G and a number k, are there k disjoint edges in the graph? Let $Match(G,k)$ be 1 if there are k such edges, and 0 otherwise.

All of these problems have something in common: although it may be hard to efficiently compute the functions they define, it is very easy to *check* a solution if one is given to us! For example, if $ISet(G,k) = 1$, then there is a an independent set S of size k, and given G, S, k, one can check that S is an independent set of size k in polynomial time. Similarly, if $SubSum(a_1, \ldots, a_\ell, t) = 1$, then there is a subset of the numbers $S \subseteq \{a_1, \ldots, a_\ell\}$, that if given as input can be verified to have the sum t.

NP is the class of all functions f that have the above property, where if $f(x) = 1$, then this can be checked efficiently by an efficient verifier:

Definition 13. *$f : \{0,1\}^* \to \{0,1\}$ is in* **NP** *if there exists a polynomial p and a polynomial time machine V such that for every $x \in \{0,1\}^*$,*

$$f(x) = 1 \Leftrightarrow \exists w \in \{0,1\}^{p(|x|)}, V(x,w) = 1$$

The witness w is restricted to being of polynomial length to ensure that the running time of V is actually polynomial in the length of x. If we allowed the witness to be arbitrarily long, then V would be allowed to run very long computations on x.

V is usually called the *verifier* and w is usually called the witness or certificate or proof. For example, in the independent set problem above, the witness w would correspond to an independent set, and the verifier V would be the program that checks that w is in fact an independent set of size k in the input graph.

Many important combinatorial optimization problems can be cast as problems in **NP**.

P, NP and EXP

Fact 14. *$P \subseteq NP \subseteq EXP$.*

To see the first containment, observe that if $f \in P$, there is a polynomial time Turing machine M with $M(x) = f(x)$. But M itself is a verifier for f (with a witness of length 0) proving that $f \in NP$.

For the second containment, if $f \in NP$, then f has a verifier $V(x, w)$. Consider the algorithm that on input x runs over all possible w and checks if $V(x, w) = 1$. If any witness makes $V(x, w) = 1$, the algorithm outputs 1, otherwise it outputs 0. This algorithm computes f and runs in exponential time, so $f \in EXP$.

Nondeterministic Machines, and a Hierarchy Theorem

The original definition of **NP** was by considering Turing machines that are allowed to make non-deterministic choices: namely after each step, the machine is allowed to make a guess about which state to transition to in the next step. The machine computes 1 if there is a single accepting computational path, and 0 otherwise.

We can define $\mathsf{NTIME}(t(n))$ in the same way as $\mathsf{DTIME}(t(n))$, it is the set of functions computable by non-deterministic machines in time $O(t(n))$, and then you can check that $\mathbf{NP} = \bigcup_c \mathsf{NTIME}(n^c)$. Just as for deterministic time, there is a non-deterministic time hierarchy theorem:

Theorem 15. *If r, t are time-constructible functions satisfying $r(n+1) = o(t(n))$, then*

$$\mathsf{NTIME}(r(n)) \subsetneq \mathsf{NTIME}(t(n)).$$

Polynomial time Reductions

One of the central questions in complexity theory is whether or not $\mathbf{P} = \mathbf{NP}$. Although we don't know the answer to this question, we can prove a lot about the class **NP**, via the concept of polynomial time reductions:

Definition 16. *A function f is polynomial time reducible to a function g if there is a polynomial time computable function h such that $f(x) = g(h(x))$. We write $f \leq_P g$.*

Note that the above definition is not the only one that makes sense. In general it makes sense to allow our reductions to make multiple calls to the problem being reduced to. However, we will be able to prove many of our results using the stronger notion above, so that is what we shall use.

Definition 17. *We say f is* **NP***-hard if $g \leq_P f$ for every $g \in$* **NP***. We say f is* **NP***-complete if f is* **NP***-hard and $f \in$* **NP***.*

Theorem 18. *Here are some easy facts that one can prove about reductions:*

- *If $f \leq_P g$ and $g \leq_P h$, then $f \leq_P h$.*
- *If f is* **NP***-hard and $f \in \mathbf{P}$, then $P = NP$.*
- *If f is* **NP***-complete, then* $\mathbf{P} = \mathbf{NP}$ *if and only if $f \in P$.*

NP-*complete problems*

The above definitions make sense because we do know of examples of **NP**-complete problems.

Circuit-Sat

Definition 19. *CircuitSat* $: \{0,1\}^* \to \{0,1\}$ *is the function that views its input as a circuit C and outputs 1 iff $\exists x$ such that $C(x) = 1$.*

I have claimed in class that circuits can simulate Turing Machines. Here is what you can actually prove in this regard:

Theorem 20. *If a function $f : \{0,1\}^* \to \{0,1\}$ can be computed in time $t(n)$ by a Turing machine, then for every n there is a circuit of size $O(t(n) \log t(n))$ that computes f restricted to the inputs of size n.*

Although we did not prove this theorem in class, we sketched how you could find a circuit of size $O(t(n)^2)$ that computes f. The idea was to add a layer of gates that maintains the entire state of the Turing machine—contents of all tapes, pointers, and the line of code being executed. Then we add a new layer that computes this configuration after one execution step of the Turing machine, using the earlier configuration as input. A single configuration can be written down using $O(t(n))$ gates since we only need to write down the values of the tapes up to $O(t(n))$ coordinates. The new configuration can be computed from the old one with $O(t(n))$ gates as well. After repeating this $O(t(n))$ times, we obtain the final configuration of the Turing machine, which must include the value of $f(x)$.

Theorem 21. *CircuitSat is* **NP***-complete.*

Proof It is clear that *CircuitSat* is in **NP**. Next we show that for every $f \in \mathbf{NP}$, $f \leq_P$ *CircuitSat*. Let V be a verifier for f. Then to compute $f(x)$, the reduction will build the circuit $C_x(w)$ that computes $V(x,w)$, where here w are the input variables to the circuit and x is the input. Since $f(x) = 1$ if and only if there exists w such that $C_x(w) = 1$, we can determine the value of f by computing *CircuitSat*(C_x). ■

3SAT

A *boolean formula* is an expression of the form

$$(x_1 \wedge \neg x_2) \vee (x_7 \wedge \neg(x_6 \vee \neg x_2)).$$

Formally: it is a circuit where the only allowed gates are $\vee, \wedge, \neg$, and every gate has fan-out at most 1. Input gates are allowed to repeat. As usual, size of the gates is number of gates, and the fan-in is allowed to be at most 2. The formula is said to be in conjunctive normal form (CNF) if it is an AND of OR's. Similarly, it is said to be in disjunctive normal form (DNF) if it is an OR of ANDS. For example

$$(x_1 \vee \neg x_2) \wedge (\neg x_7 \vee x_9 \vee \neg x_1)$$

is a CNF.

We have the following lemma:

Lemma 22. *Every function $f : \{0,1\}^\ell \to \{0,1\}$ can be computed by a CNF (resp. DNF) of size $\ell 2^\ell$.*

Proof For each input z such that $f(z) = 0$, we add the literal x_i to the clause if $z_i = 0$ and $\neg z_i$ otherwise. So for example, if $f(0,1,0) = 0$, we add the clause $(x_1 \vee \neg x_2 \vee x_3)$. Then note that each clause is 0 on exactly one input, and all inputs x for which $f(x) = 0$ make some clause 0. Every other input evaluates to 1. So, the CNF computes f. The resulting formula is of size $\ell 2^\ell$. The case of DNF's is symmetric. ■

We define $\mathsf{SAT} : \{0,1\}^* \to \{0,1\}$ to be the function that takes as input a boolean formula F, and outputs 1 if and only if there is a an x such that $F(x) = 1$. A 3-CNF formula is a CNF where every clause has at most 3 variables. For example:

$$(x_1 \vee \neg x_2 \vee x_3) \wedge (x_3 \vee x_4 \vee \neg x_1) \wedge \cdots .$$

$\mathsf{3SAT} : \{0,1\}^* \to \{0,1\}$ is the function that takes as input 3-CNF and outputs 1 if and only if the formula is satisfiable. Next we show that even this function is **NP**-complete

Theorem 23. 3SAT *is* **NP***-complete.*

Proof $\mathsf{3SAT} \in \mathbf{NP}$ is easy enough to check. The witness is a satisfying assignment to the formula. The verifier simply evaluates the formula on the given witness, and outputs the results of the evaluation.

Since we have already shown that $\mathsf{CKT-SAT}$ is **NP**-hard, it will be enough to show that $\mathsf{CKT-SAT} \leq_P \mathsf{SAT}$.

Is the same true for 2SAT? We do not know. There are polynomial time algorithms for 2SAT, so if you found a reduction to 2SAT, you would prove $\mathbf{P} = \mathbf{NP}$. The algorithm works by viewing every clause $(x \vee y)$ as an implication $\neg x \Rightarrow y$ as well as the implication $\neg y \Rightarrow x$. This defines a directed graph where all the vertices correspond to variables and their negations, and the edges correspond to implications. You can show that the formula is satisfiable if and only if there is no path that leads from a variable to its negation.

Given a circuit, we shall output a CNF formula that is satisfiable if and only if the circuit accepts some input. Introduce a new variable y_g for each internal gate g of the circuit. If the internal gate g has inputs h, q, let F_g be the CNF formula on variables y_g, y_h, y_q that is 1 if and only if $y_g = g(y_q, y_h)$. By Lemma 22, this formula is a 3-CNF of constant size. If the output gate is v, the final formula is

$$y_v \wedge \bigwedge_g F_g,$$

which is satisfied if and only if the circuit has a satisfying assignment.

Every clause of this formula has at most 3 variables. To make sure it has *exactly* 3 variables, we replace each clause with less than 3 variables with a 3-CNF that by adding dummy variables. For example, we can replace y_v by a 3-CNF on the variables y_v, z_1, z_2 that computes the same function as y_v:

$$(y_v \vee z_1 \vee z_2) \wedge (y_v \vee \neg z_1 \vee z_2) \wedge (y_v \vee \neg z_1 \vee \neg z_2) \wedge (y_v \vee z_1 \vee \neg z_2).$$

■