Notes on Complexity Theory

Richard Willie

October 27, 2024

Contents

1 The Church-Turing Thesis

2 Diagonalization

A basic goal of complexity theory is to prove that certain complexity classes (e.g. **P** and **NP**) are not the same. To do so, we need to exhibit a machine in one class that differs from every machine in other class in the sense that their answers are different on at least one input. This chapter describes **diagonalization**—essentially the only general technique known for constructing such a machine.

In this chapter, we will use diagonalization in clever ways. We first use it in Section 2.1 and Section 2.2 to prove *hierarchy theorems*, which show that giving Turing machines more computational resources allows them to solve a strictly larger number of problems. We then use diagonalization in Section 2.3 to show a fascinating theorem of Ladner: If $\mathbf{P} \neq \mathbf{NP}$, then there exists a problem that are neither **NP**-complete nor in **P**.

Though diagonalization led to some of these early successes of complexity theory, researchers concluded in the 1970s that diagonalization alone may not resolve **P** vs **NP** and other interesting questions. Section 2.4 describes their reasoning. Ironically, these limits of diagonalization are proved using diagonalization itself.

§2.1 Time Hierarchy Theorem

Common sense suggests that giving a Turing machine more time should increase the class of problems that it can solve. For example, Turing machines should be able to decide more languages in time n^3 than they can in n^2. The **time hierarchy theorem** proves that this intuition is correct, subject to certain conditions described below. We use the term *hierarchy theorem* because this theorem proves that time complexity classes aren't all the same—they form a hierarchy whereby the classes with larger bounds contain more languages than do the classes with smaller bounds.

We begin with the following technical definition, given by Goldreich [Go08].

Definition 2.1.1 (Time-constructible functions)

A function $f : \mathbb{N} \to \mathbb{N}$ is called **time-constructible** if there exists a TM M which, given a string 1^n, outputs the binary representation of $f(n)$ in $O(f(n))$ time.

Example 2.1.1

All the commonly used functions $f(n)$ are time-constructible, as long as $f(n)$ is at least cn for some constant $c > 0$. No function which is $o(n)$ can be time-constructible unless it is eventually constant, since there is insufficient time to read the entire input.

When designing algorithms of arbitrary time complexity $f : \mathbb{N} \to \mathbb{N}$, we need to make sure that the algorithm does not exceed the time bound. Furthermore, when invoked on input w, the algorithm is not given the time bound $f(|w|)$ explicitly. A reasonable design methodology is to have the algorithm compute this bound before doing anything else. This, in turn, requires the algorithm to read the entire input and compute $f(n)$ in $O(f(n))$ steps; otherwise, this preliminary stage already consumes too much time. The latter requirement motivates the notion of time-constructible functions.

Theorem 2.1.2 (Time hierarchy theorem)

For any time-constructible function $f : \mathbb{N} \to \mathbb{N}$, a language A exists that is decidable in $O(f(n))$ time but not decidable in time $o(f(n)/\log f(n))$.

The time hierarchy theorem for deterministic multi-tape Turing machines was first proven by Stearns and Hartmanis [HaSt65]. It was improved some time later when Hennie and Stearns improved the efficiency of the universal Turing machine [HeSt66].

We present the following proof by Sipser [Si13], with slight modification.

Proof Idea. We must demonstrate a language A that has two properties. The first says that A is decidable in $O(f(n))$ time. The second says that A isn't decidable in $o(f(n)/\log f(n))$ time.

We describe A by giving an algorithm D that decides it. Algorithm D runs in $O(f(n))$ time, thereby ensuring the first property. Furthermore, D guarantees that A is different from any language that is decidable in $o(f(n)/\log f(n))$ time, thereby ensuring the second property. Notice that language A is different from other languages in that it lacks a non-algorithmic definition. Therefore, we cannot offer a simple mental picture of A.

In order to ensure that A is not decidable in $o(f(n)/\log f(n))$ time, we design D to implement the diagonalization method. If M is a TM that decides a language in $o(f(n)/\log f(n))$ time, D guarantees that A differs from M's language in at least one place. Which place? The place corresponding to a description of M itself.

Let's look at the way D operates. Roughly speaking, D takes its input to be the description of a TM M. (If the input isn't the description of any TM, then D's action is inconsequential on this input, so we arbitrarily make D reject.) Then, D runs M on the same input—namely, $\langle M\rangle$—within the time bound $\lceil f(n)/\log f(n)\rceil$. If M halts within that much time, D accepts iff M rejects. If M doesn't halt, D just rejects. So if M runs within time $\lceil f(n)/\log f(n)\rceil$, D has enough time to ensure that its language is different from M's. If not, D doesn't have enough time to figure out what M does, but fortunately D has no requirement to act differently from machines that don't run in $o(f(n)/\log f(n))$ time, so D's action on this input is inconsequential.

This description captures the essence of the proof but omits several important details. If M runs in $o(f(n)/\log f(n))$ time, D must guarantee that its language is different from M's language. But even when M runs in $o(f(n)/\log f(n))$ time, it may use more than $f(n)$ time for small n, when the asymptotic behavior hasn't "kicked in" yet. Possibly, D might not have enough time to run M to completion on input $\langle M\rangle$, and hence D will miss its opportunity to avoid M's language. So, if we aren't careful, D might end up deciding the same language M decides, and the theorem wouldn't be proved.

We can fix this problem by modifying D to give it additional opportunities to avoid M's language. Instead of running M only when D receives input $\langle M\rangle$, it runs M whenever it receives an input of the form $\langle M\rangle\,\texttt{10}^*$, that is, an input of the form $\langle M\rangle$ followed by a `1` and some number of `0`s. In other words, we get to simulate M on infinitely many

w. Then, if M really is running in $o(f(n)/\log f(n))$, we will eventually encounter a sufficiently large w, where $w = \langle M \rangle\, \mathtt{10}^k$ for some large value of k, and D will have enough time to run it to completion because the asymptotic must eventually kick in.

One last technical point. The simulation of M by D introduces a logarithmic factor overhead. This overhead is the reason for the appearance of the $1/\log f(n)$ factor in the statement of this theorem. If we had a way of simulating a single-tape TM by another single-tape TM for a prespecified number of steps, using only a constant factor overhead in time, we would be able to strengthen this theorem by changing $o(f(n)/\log f(n))$ to $o(f(n))$. No such efficient simulation is known. □

We now describe the proof formally.

Proof. The following $O(f(n))$ time algorithm D decides a language A that is not decidable in $o(f(n)/\log f(n))$ time.

$D =$ "On input w:

1. Let n be the length of w.
2. Compute $f(n)$ using time constructibility, and store the value $\lceil f(n)/\log f(n) \rceil$ in a binary counter. Decrement this counter before each step used to carry out stages 3, 4, and 5. If the counter ever hits 0, *reject.*
3. If w is not of the form $\langle M \rangle\, \mathtt{10}^*$ for some TM M, *reject.*
4. Simulate M on w.
5. If M accepts, then *reject.* If M rejects, then *accept.*"

We examine each of the stages of this algorithm to determine the running time. Obviously, stages 1, 2, and 3 can be performed within $O(f(n))$ time. In stage 4, every time D simulates one step of M, it takes M's current state together with the tape symbol under M's tape head and looks up M's next action in its transition function so that it can update M's tape appropriately. All three of these objects (state, tape symbol, and transition function) are stored on D's tape somewhere. If they are stored far from each other, D will need many steps to gather this information each time it simulates one of M's steps. Instead, D always keep this information close together.

We can think of D's single tape as organized into *tracks.* One way to get two tracks is by storing one track in the odd positions and the other in the even positions. Alternatively, the two-track effect may be obtained by enlarging D's tape alphabet to include each pair of symbols, one from the top track and the second from the bottom track. We can get the effect of additional tracks similarly. Note that multiple tracks introduce only a constant factor overhead in time, provided that only a fixed number of tracks are used. Here, D has three tracks.

One of the tracks contains the information on M's tape, and a second contains its current state and a copy of M's transition function. During the simulation, D keeps the information on the second track near the current position of M's head on the first track. Every time M's head position moves, D shifts all the information on the second track to keep it near the head. Because the size of the information on the second track depends only on M and not on the length of the input to M, the shifting adds only a constant factor to the simulation time. Furthermore, because the required information is kept close together, the cost of looking up M's next action in its transition function and updating its tape is only a constant. Hence, if M runs in $g(n)$ time, D can simulate it in $O(g(n))$ time.

At every step in stages 3 and 4, D must decrement the step counter originally set in stage 2. Here, D can do so without adding excessively to the simulation time by keeping the counter in binary on a third track and moving it to keep it near the present head position. This counter has a magnitude of about $f(n)/\log f(n)$, so its length is $\log(f(n)/\log f(n))$, which is $O(\log f(n))$. Hence, the cost of updating and moving at each step adds a $\log f(n)$ factor to the simulation time, thus brining the total running time to $O(f(n))$. Therefore, A is decidable in time $O(f(n))$.

Now, we show that A is not decidable in $o(f(n)/\log f(n))$ time. Assume the contrary that some TM M decides A in time $g(n)$, where $g(n)$ is $o(f(n)/\log f(n))$. Here, D can simulate M, using time $d \cdot g(n)$ for some constant d. If the total simulation time (not counting the time to update the step counter) is at most $f(n)/\log f(n)$, the simulation will run to completion. Because $g(n)$ is $o(f(n)/\log f(n))$, some constant n_0 exists where $d \cdot g(n) < f(n)/\log f(n)$ for all $n \geq n_0$. Therefore, D's simulation of M will run to completion as long as the input has length n_0 or more. Consider what happens when we run D on input $\langle M \rangle \, \mathtt{10}^{n_0}$. This input is longer than n_0 so the simulation in stage 4 will complete. Therefore, D will do the opposite of M on the same input. Hence, M doesn't decide A, which contradicts our assumption. Therefore, A is not decidable in $o(f(n)/\log f(n))$ time. □

Corollary 2.1.3

For any two time-constructible functions $f, g : \mathbb{N} \to \mathbb{N}$ where $f(n) = o(g(n)/\log g(n))$,

$$\mathbf{DTIME}(f(n)) \subsetneq \mathbf{DTIME}(g(n)).$$

This corollary allows us to separate various deterministic time complexity classes. For example, we can show that the function n^c is time-constructible for any natural number c. Hence, for any two natural numbers $c_1 < c_2$ we can prove that $\mathbf{DTIME}(n^{c_1}) \subsetneq \mathbf{DTIME}(n^{c_2})$. With a bit more work we can show that n^c is time-constructible for any rational number $c > 1$ and thereby extend the preceding containment to hold for any rational numbers $1 \leq c_1 < c_2$. Observing that two rational numbers c_1 and c_2 always exist between any two real numbers $\epsilon_1 < \epsilon_2$ such that $\epsilon_1 < c_1 < c_2 < \epsilon_2$ we obtain the following additional corollary demonstrating a fine hierarchy within the class $\mathbf{P}$.

Corollary 2.1.4

For any two real numbers $1 \leq \epsilon_1 < \epsilon_2$,

$$\mathbf{DTIME}(n^{\epsilon_1}) \subsetneq \mathbf{DTIME}(n^{\epsilon_2}).$$

We can also use the time hierarchy theorem to separate the classes $\mathbf{P}$ and $\mathbf{EXPTIME}$.

Corollary 2.1.5

$\mathbf{P} \subsetneq \mathbf{EXPTIME}$.

Proof. We have

$$\mathbf{DTIME}(2^n) \subseteq \mathbf{DTIME}\left(o\left(\frac{2^{2n}}{2n}\right)\right) \subsetneq \mathbf{DTIME}(2^{2n})$$

by the time hierarchy theorem. It follows that

$$\mathbf{P} \subseteq \mathbf{DTIME}(2^n) \subsetneq \mathbf{DTIME}(2^{2n}) \subseteq \mathbf{EXPTIME}.$$

□

This corollary establishes the existence of decidable problems that are decidable in principle but not in practice—that is, problems that are decidable but intractable.

§2.2 Nondeterministic Time Hierarchy Theorem

The following time hierarchy theorem for nondeterministic Turing machines is due to Seiferas, Fischer, and Meyer [SeFiMe78].

Theorem 2.2.1 (Nondeterministic time hierarchy theorem)
For any two time-constructible functions $f, g : \mathbb{N} \to \mathbb{N}$ where $f(n+1) = o(g(n))$,

$$\mathbf{NTIME}(f(n)) \subsetneq \mathbf{NTIME}(g(n)).$$

Our first instinct is to duplicate the proof of Theorem 2.1.2, since there is a universal TM for nondeterministic computation as well. (In fact, the simulation of an NTM by another NTM is very efficient, incurring only a constant factor overhead in time [ArBa09].) The problem, however, is that we don't know how such a construction would work on our new machine D. In particular, it remains unclear how we could "flip the answer" of M, i.e. to efficiently compute, given the description of an NTM M and an input w, the value $1 - M(w)$. We give the following hypothetical (erroneous) construction of D to illustrate our point.

D = "On input w:

1. Let n be the length of w.
2. Compute $f(n)$ using time constructibility, and store the value $f(n)$ in a binary counter. Decrement this counter before each step used to carry out stages 3, 4, and 5. If the counter ever hits 0, *reject*.
3. If w is not of the form $\langle M \rangle \mathtt{1}\mathtt{0}^*$ for some TM M, *reject*.
4. Nondeterministically simulate M on w.
5. If M accepts, then *reject*. If M rejects, then *accept*."

Ignoring other potential issues with the above construction, our main concern is that it's unclear how we could determine efficiently, whether or not M rejects the input w. A nondeterministic Turing machine accepts if at least one path accepts; it rejects only when all paths reject. This asymmetry makes it hard to "flip answers" efficiently. For instance, suppose we have an NTM M that has two paths for input w where one accepts and the other rejects. M has at least one accepting path for w, so it accepts. Suppose we want to produce a machine that accepts exactly the input that M rejects. The obvious attempt is to take M and make its accepting state reject, and its rejecting states accept. However, this new machine M' has one rejecting path and one accepting path. So it still accepts w, which it was supposed to reject.

One limitation of the nondeterministic Turing machine is that it can't look at all its paths simultaneously and take action based on what all of those paths do. When simulating an NTM M with another machine M', each path M' can only simulate one path of M and has no knowledge of the other paths. It cannot decide to accept/reject based on the outcomes of paths it cannot see. Therefore, each path of M' can only follow simple rules based on its own outcome. When M' discovers that M rejects the input w on a certain path, M' still can't verify whether w is in the language of M or not (because of the asymmetric accept/reject condition of the NTM). Therefore, it seems that the only reliable way to determine if M rejects w is to exhaustively check every possible path, which is exponentially more time-consuming than checking a single path.

What now? There are two options here:

1. We admit defeat. Accept the exponential-time slowdown, but then we won't get a fine hierarchy at all.
2. Or, we try to come up with a better solution. Consider the following idea. Now, our new simulator is in no hurry to diagonalize, it will not try to "flip the answer" of a subroutine NTM on every input, but only on a crucial input out of a sufficiently large (i.e. exponentially large) interval. This technique is known as **lazy diagonalization**. We show that it does suffice to establish our hierarchy theorem.

We present the following proof due to Žák [**Žá83**].

Proof. □

We also present a different proof by Fortnow and Santhanam [**FoSa11**].

Proof. □

§2.3 Ladner's Theorem: Existence of NP-Intermediate Problems

§2.4 Oracle Machines and the Limits of Diagonalization

Bibliography

[**ArBa09**] Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach*. Cambridge University Press, 2009 (cited p. 8)

[**FoSa11**] Lance Fortnow and Rahul Santhanam. “Robust simulations and significant separations”. In: *International Colloquium on Automata, Languages, and Programming*. Springer. 2011, pp. 569–580 (cited p. 9)

[**Go08**] Oded Goldreich. Computational complexity: a conceptual perspective. In: (2008) (cited p. 4)

[**HaSt65**] Juris Hartmanis and Richard E Stearns. On the computational complexity of algorithms. In: *Transactions of the American Mathematical Society*, **117**: (1965), pp. 285–306 (cited p. 5)

[**HeSt66**] Fred C Hennie and Richard Edwin Stearns. Two-tape simulation of multitape Turing machines. In: *Journal of the ACM (JACM)*, **13**:4 (1966), pp. 533–546 (cited p. 5)

[**SeFiMe78**] Joel I Seiferas, Michael J Fischer, and Albert R Meyer. Separating nondeterministic time complexity classes. In: *Journal of the ACM (JACM)*, **25**:1 (1978), pp. 146–167 (cited p. 8)

[**Si13**] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 2013 (cited p. 5)

[**Žá83**] Stanislav Žák. A Turing machine time hierarchy. In: *Theoretical Computer Science*, **26**:3 (1983), pp. 327–333 (cited p. 9)