INFR11102: Computational Complexity 24/09/2019

Lecture 3: Hierarchy Theorems

Lecturer: Heng Guo

1 Resource-bounded computation

The branch of mathematical logic called recursion theory (also known as computability theory) studies what problems are computable and what are not, as well as their hardness relative to each other. On the other hand, computational complexity studies how efficiently can we solve problems, based on the premise that real-world computation is usually resource-bounded.

It all started with a paper by Hartmanis and Sterns [HS65], where they coined the term "computational complexity" as well as initiated this whole subject. They also won the Turing award because of this work in 1993.

Our model of computation is multi-tape TMs. For definiteness, we fix the alphabet to be $\{0, 1, \sqcup\}$. The most natural resource bound is time. The time complexity of a deterministic TM M is

$$time_M(n) := \max_{x:|x|=n} \{\text{number of transitions taken in } M(x)\},$$

where $x \in \{0,1\}^*$ and $|x|$ denotes the length of x. For a TM M, let $L(M)$ be the language (or decision problem) that M computes. Namely

$$L(M) := \{x \in \{0,1\}^* \mid M(x) = 1\}.$$

For a function $f(n) \geq n$, define the complexity class

$$\texttt{DTime}[f] = \{L \mid \exists M \text{ and } c, \text{ s.t. } L(M) = L \text{ and } time_M(n) \leq cf(n)\}.$$

In other words, $\texttt{DTime}[f]$ is the class of languages that can be computed within time at most $O(f(n))$.

The main reason to ignore the constant factor is that it usually depends on the model of computation. We have seen in the last lecture that we can simulate an arbitrary finite alphabet Σ using $\{0, 1, \sqcup\}$ with $O(\lceil \log |\Sigma| \rceil)$ slowdown. This reduction actually goes both ways. If we use a larger alphabet to simulate a smaller one, then we can gain a constant speedup by simulating multiple steps in one. This technique is known as "linear speedup", and details can be found in [Pap94, Chapter 2.4]. Since constants are sensitive to the model of computation, we define complexity classes up to an arbitrary constant factor. For example, $\texttt{DTime}[f]$ is the same as $\texttt{DTime}[3f]$ or $\texttt{DTime}[C \cdot f]$ for any constant $C > 0$.

Also, we will only consider "nice" (*time-constructible*) functions f due to technical reasons. A function f is called *time-constructible* if there exists a TM M such that for any x

with $|x| = n$, $M(x)$ runs in exactly $f(n)$ steps. This is just to avoid some pathological situations. Although there exist functions that are not time-constructible, I claim that all natural functions you will ever encounter are time-constructible. Typical complexity functions are n, n^c, $n^{c \log n}$, $2^{\sqrt{n}}$, 2^n, $2^{2^{\cdot^{\cdot^{\cdot^{2^n}}}}}$, etc.

Completely analogously, we can define space complexity by the number of cells a TM uses, and space-constructible functions. For a function f, $\texttt{DSpace}[f]$ denotes the class of decision problems that can be solved within space at most $O(f(|x|))$ for an input x.

The only complicacy for space complexity is that sometimes the "working" space required is sometimes much smaller than the input size. To get around the issue, for space complexity, we can assume that the input is in a spacial "input" tape, and then we have several "working" tapes, and then we have a special "output" tape. We can scan multiple times for the input, but not write on the input tape, and the output tape is only allowed to be written once. This allows us to talk about $\texttt{DSpace}[\log n]$, for example.

As an example, what is the complexity of the following problem?

Name: PARITY

Input: A string $x \in \{0,1\}^*$.

Output: The parity of the Hamming weight of x.

It is easy to see that PARITY $\in \texttt{DTime}[n]$ and PARITY $\in \texttt{DSpace}[1]$. Both can be achieved even simultaneously. In fact, any language recognizable by a finite automata can be computed in linear time and constant space.

2 Hierarchy theorems

A natural question is, whether more time buys us more computational power? This is morally correct, known as Hierarchy theorems. Roughly speaking, for two functions T_1 and T_2, if T_2 grows (sufficiently) faster than T_1, then we can solve more problems in time T_2 than in T_1.

Theorem 1. *Let $T_1(n)$ and $T_2(n)$ be two functions such that T_2 is time-constructible. If $\lim_{n\to\infty} \frac{T_1(n)^2}{T_2(n)} = 0$ (or equivalently, $T_1(n)^2 = o(T_2(n))$), then there exists a language $L \in \texttt{DTime}[T_2(n)] - \texttt{DTime}[T_1(n)]$.*

Again, the proof is a diagonalisation argument. We will construct the language L by giving a $T_2(n)$-time algorithm, and then show that it does not belong to $\texttt{DTime}[T_1(n)]$.

Before going to the proof, let us recall the Universal Turing Machine (UTM) from the last lecture. Given an encoding of a TM M, and an input x, the UTM $U(M,x)$ simulates M on x, yielding the same output. This is to say, that if M accepts x, then $U(M,x)$ accepts, and if M rejects x, then $U(M,x)$ rejects. If M does not halt on x, $U(M,x)$ also does not halt. Suppose that M takes at most $T(n)$ time and $S(n)$ space for an input of length n, then the UTM U uses at most $C_M T(n)^2$ time and $S(n) + C'_M$ space, where C_M and C'_M are two constants depending only on M. This is because we do roughly $C_M T(n)$ operations per

each operation of M, and essentially do not use any extra space. See [AB09, Claim 1.6] if you are curious about the details.

The rough idea to prove Theorem 1 is the following: we list all TMs that run in time $T_1(n)$ (again, up to constant factors). This can be done, since this is a subset of all TMs, and there are only countably many TMs. Suppose this list is $M_1, M_2, \cdots$. Also list all possible inputs as $x_1, x_2, \cdots$. Our diagonalisation TM A is defined by the following simple specification:

Name: TM A

Input: x_i

Output: $1 - M_i(x_i)$

Note that A can be implemented by simulating M_i on input x_i via the Universal Turing Machine.

We claim that $L(A) \notin \texttt{DTime}[T_1(n)]$. Suppose otherwise. Then $L(A)$ can be computed by a TM in our list with some index k. (Note that M_k is not necessarily A itself.) What is the value of $A(x_k)$? By construction, it must be that $A(x_k) = 1 - M_k(x_k) = 1 - A(x_k)$, which is impossible as $A(x_k) \in \{0, 1\}$.

To finish the proof, we still want to show that $L(A) \in \texttt{DTime}[T_2(n)]$. Namely, we need to show that there exists a constant c, such that A runs within time $cT_2(n)$. However, this is not necessarily the case. The trouble is that although any M_i running in time $T_1(n)$ can be simulated in $C_{M_i}T_1^2(n)$ time via the UTM, the latter is not necessarily smaller than than $T_2(n)$. There is no guarantee on a uniform upper bound of C_{M_i} over all i. From the assumption $\lim_{n\to\infty} \frac{T_1(n)^2}{T_2(n)} = 0$, we only know that for any $C > 0$, $\exists N_C$ such that $\forall n > N_C$, $CT_1^2(n) \leq T_2(n)$.

To circumvent this issue, we use a *delayed diagonalisation* argument. The goal is to diagonalize against every M_i on infinitely many inputs, so that the "sufficiently large" part in our assumptions kicks in. Recall that every TM can be encoded by infinitely many strings by padding. If $L(A)$ can be computed in T_1 time, then our proof (not the algorithm below) will find a sufficiently large index k for it.

We construct a TM A' in Algorithm 1. With a little abuse of notation, we write M_x for M_i where i is the index of x.

Algorithm 1 A'

Input: $x \in \{0, 1\}^*$.
Simulate $M_x(x)$, where M_x is the TM encoded by x.
On a separate tape, run the clock of $T_2(|x|)$.
if $T_2(|x|)$ time is reached, **then**
  return 1.
end if
return $1 - M_x(x)$.

From the description above, it is easy to see that $L(A') \in \texttt{DTime}[T_2(n)]$ because of the clock. Now the main task is to show that $L = L(A') \notin \texttt{DTime}[T_1(n)]$. We will leave that to the next time.

Remark (Bibliographic). Hierarchy theorems can be found in [AB09, Chapter 3] and [Pap94, Chapter 7].

References

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity - A Modern Approach*. Cambridge University Press, 2009.

[HS65] Juris Hartmanis and Richard E. Stearns. On the computational complexity of algorithms. *Trans. Amer. Math. Soc.*, 117:285–306, 1965.

[Pap94] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.