15. HARDNESS AND COMPLETENESS

Mihai-Valentin DUMITRU
mihai.dumitru2201@upb.ro

November 2024

In the last three lectures we have looked at a collection of problems that share some interesting properties.

First, all these problems have an *easily-verifiable solution*. We have formalized this property by employing the notion of *polynomial certificates* and defined a complexity class, **NP**, to consist of exactly those problems which have a polynomial certificate. We can say now that `SAT`, `VERTEX COVER`, `SUBSET SUM` etc. are in **NP**.

Another shared characteristic of these problems is that, while obviously solvable by trying all possible solutions, there is no known *tractable* solution, i.e. one that runs in polynomial time. However, there is also no known proof that such a solution doesn't exist. This gives rise to the "**P** versus **NP** problem": *are all easily-verifiable problems, easily solvable?*

In a previous lecture we saw how we can *reduce* some of these problems to each other in polynomial time and hopefully gained the intuition that any of them can be reduced to any other. In this sense, they are "equally difficult".

Now we are ready to expand on the structure of our complexity classes and discuss the class of "the hardest problems in **NP**".

1 Hardness and Completeness

We pointed out that, intuitively, if a problem f reduces to another problem g ($f \leq_P g$), that means that g is *harder* (or of equal "difficulty" to f). This makes sense because the existence of a reduction says that, with some preprocessing, we can turn any input instance of f into an input instance of g and feed it into our g solver to get the correct answer. In particular, note that the "g solver" can be much more costly than the preprocessing: a solution for g may have exponential time complexity, but the transformation involved in the reduction has, by definition, polynomial time complexity.

Now we can move on to think of hardness in relation to a *class of problems*. A "hard problem" in this sense would be one that is harder (or as-hard-as) than any problem in that complexity class.

Let us focus on the class **NP** and polynomial-time many-one reductions:

Definition 15.1.

$$f \text{ is } \mathbf{NP}\text{-Hard} \stackrel{\text{def}}{=} \forall g \in \mathbf{NP}, g \leq_P f$$

$$\mathbf{NPH} \stackrel{\text{def}}{=} \{f : \Sigma^* \to \{\texttt{FALSE}, \texttt{TRUE}\} \mid f \text{ is } \mathbf{NP}\text{-hard } \}$$

The concept of hardness can be extended to other classes (e.g. we can talk about "**P**-Hardness") and other types of reductions; but we shall only consider **NP**-hardness.

The concept of "hardness" is closely related to that of "completeness".

Definition 15.2.

$$f \text{ is } \mathbf{NP}\text{-complete} \stackrel{\text{def}}{=} \begin{cases} f \in \mathbf{NP} \\ f \in \mathbf{NPH} \end{cases}$$

$$\mathbf{NPC} \stackrel{\text{def}}{=} \{f : \Sigma^* \to \{\texttt{FALSE}, \texttt{TRUE}\} \mid f \text{ is } \mathbf{NP}\text{-complete } \}$$

It is not immediately obvious that such problems exist. In the next section, we shall prove that `SAT` is just such a problem; then we will show that **all** problems introduced under the description "hard?" fit in this class. In fact, **NP**-completeness is the formalization of the notion "*easy-to-check, possibly hard-to-find* solution".

2 SAT is NP-complete

When we introduced SAT, we also presented a deterministic polynomial algorithm for verifying a solution; later we explicitly identified a truth assignment as a *certificate*, polynomial in the size of the input.

This shows that $\texttt{SAT} \in \mathbf{NP}$. To prove completeness, all that is left to do is to prove hardness.

Theorem 15.1.

$$\forall f \in \mathbf{NP}, f \leq_P \texttt{SAT}$$

Any problem $f \in \mathbf{NP}$ has, by definition, some Nondeterministic Turing Machine N_f that decides it. The idea of our proof is to take the encoding of N_f's computational history on some word w and create a formula ϕ that is satisfiable if and only if $N_f[w] \to \texttt{TRUE}$. We must also take care that this transformation can be done in polynomial time.

Because we're talking about **NP** problems, it means that there is some $k \in \mathbb{N}$, such that $N_f[w]$ takes less than $|w|^k - 3$[†] transitions. Moreover, remember that extending the tape contents by overwriting a blank symbol with some non-blank one requires an extra transition, so after $|w|^k - 3$ transitions the tape contents are at most $|w|^k - 3$ symbols long. We can summarize an entire *computation path* of $N_f[w]$ as a $|w|^k \times |w|^k$ **tableau** of symbols.

When we studied the proof of undecidability for Post's Correspondence Problem, we introduced a succinct way of representing each configuration as a string, by using separate symbols for each state and writing the "current state symbol" to the left of the current symbol under the machine head. So $(100, 10, q_1)$ gets encoded as the string: $\texttt{100}\mathbf{q_1}\texttt{10}$

We *expand* each of these configuration strings with additional blanks, until they are all $|w|^k - 2$ symbols in length (note that we need an extra "tape symbol", such as q_1). The *rows* of the tableau will represent these configurations; each row begins and ends with a special delimiter symbol: $\#$.

Thus, the first row of the table will be the encoding of the initial configuration, the second row will be the encoding of the configuration resulting after one transition and so on. The last row of the table will contain a final configuration (accepting or rejecting).

We have thus a tableau of $|w|^{2k}$ symbols, drawn from the alphabet: $\Pi = Q \cup \Gamma \cup \{\#\}$. The goal is to form a boolean formula that is equivalent to telling whether the tableau is an accepting one or not.

The formula will have around[‡] $m = |\Pi||w|^{2k}$ variables. Each variable $x_{i,j,s}$ tells us if on line i, column j, the value of the cell is the symbol s.

Structurally, the formula consists of four parts:

$$\phi = \phi_{cell} \wedge \phi_{init} \wedge \phi_{transition} \wedge \phi_{accept}$$

The first three serve to guarantee that our tableau is well-formed. ϕ_{cell} expresses the fact that each cell should contain exactly one symbol:

$$\phi_{\text{cell}} = \bigwedge_{1 \leq i,j \leq |w|^k} \left[\left(\bigvee_{s \in \Pi} x_{i,j,s} \right) \wedge \left(\bigwedge_{\substack{s,t \in \Pi \\ s \neq t}} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}}) \right) \right]$$

Do not be dismayed by how intimidating the formal notation looks! What is expressed here is actually quite simple. For each cell $T_{i,j}$ of the tableau T, we must ensure the following two conditions simultaneously:

- there is some symbol from Π in the cell
- for each pair of distinct symbols of Π we choose, at least one must not be in the cell

[†] Read this footnote after you understand the described tableau; we want it to be $|w|^k \times |w|^k$; say the machine does $t(|w|)$ transitions. Each configuration's tape contents are shorter than, but can be extended to, $t(|w|) + 1$. We also want to add two columns of #s at the beginning and the end, so the table has $t(|w|) + 3$ columns, so it must have $t(|w|) + 3$ lines, thus $t(|w|) = |w|^k - 3$.

[‡] We allow ourselves to be sloppy, because we don't aim to provide *exact* answers. Our goal here is to show that this can be done in polynomial time; this keeps our analysis clean and readable.

These two properties together express that in each cell there is **exactly one** symbol. It may seem strange at first, but it is a common technique of expressing such constraints using boolean formulas.

ϕ_{init} ensures that the first line of the tableau represents the encoding of the initial configuration of N_f on w.

$$\phi_{init} = x_{1,1,\#} \wedge x_{1,2,q_1} \wedge \left(\bigwedge_{1 \leq i \leq |w|} x_{1,2+i,w_i} \right) \wedge \left(\bigwedge_{|w|+3 \leq j < |w|^k} x_{1,j,\square} \right) \wedge x_{1,|w|^k,\#}$$

$\phi_{transition}$ is the most complex formula, which has to ensure that each two consecutive rows represent a valid yielding of one configuration from another. Remember how we built part of the tiles for the proof of PCP's undecidability, to ensure that the bottom part represents the one-step modification of some transition from the top part. What allowed us to do so was the fact that changes to the configuration are *localized to at most three symbols*. We will make use of this idea here as well, and envision 2×3 *windows* whose correctness we must ensure; i.e. given a specific row, we can look at each triplet of symbols[§] and place some conditions on the triplet below, based on N_f's transition rules.

$$\phi_{transition} = \bigwedge_{1 \leq i < |w|^k, 1 \leq j < |w|^k - 1} window_{i,j}$$

Each $window$ formula guarantees that a particular 2×3 windows of symbols is valid, according to the transition function. We skip writing out such a formula explicitly, but it's important to note that it only depends on N_f's description and not on the size of the input $|w|$.

$$window_{i,j} = \bigvee_{s_1,\ldots,s_6 \text{ is a valid window}} (x_{i,j,s_1} \wedge x_{i,j+1,s_2} \wedge x_{i,j+2,s_3} \wedge x_{i+1,j,s_4} \wedge x_{i+1,j+1,s_5} \wedge x_{i+1,j+2,s_6})$$

Note that N_f is **nondeterministic**, so a particular configuration can **yield in a single step two configurations**. But this just means that we have more "valid windows" then we would have for a deterministic machine. The $window$ formulas cover the nondeterministic character of the machine.

Lastly, we need to ensure that the last row contains the accepting state Y. But what if not all computational paths take exactly n^k transitions? Some may be shorter, so the last rows of the tableau, after the one with the encoding of a final configuration are irrelevant. But this means that we must search for the accepting state in any row, not just the last:

$$\phi_{accept} = \bigvee_{1 \leq i,j \leq |w|^k} x_{i,j,Y}$$

Let us now examine the size of our formula and show that it can be built in polynomial time.

ϕ_{cell} consists of a conjunction with $|w|^{2k}$ operands. Each operand is a conjunction between a disjunction of $|\Pi|$ terms and a conjunction of $|\Pi|^2$ terms; note that these depend solely on the machine N_f not on the length of the input w, thus we consider them of *fixed size*. There are $\Theta(|\Pi||w|^{2k})$ variables, so each of them can be represented by a binary string of length $\lceil \log_2(|\Pi||w|^{2k}) \rceil$. So ϕ_{cell} has a size of $\Theta(|w|^{2k} \log_2(|w|^{2k}))$ symbols.

Similarly, $\phi_{transition}$ and ϕ_{accept} also contain a fixed size operation for each cell of the tableau, so they each have a length of $\Theta(|w|^{2k} \log_2(|w|^{2k}))$.

Lastly, ϕ_{init} has an operand for each cell in the top row, so it has a size of $\Theta(|w|^k \log_2(|w|^{2k}))$.

It follows that the size of ϕ is $\Theta(|w|^{2k} \log_2(|w|^{2k}))$ – a polynomial. Building it is straightforward: we only need to imagine nested loops to go through the rows and columns of the table, outputting some part of the formula based on the current cell.

[§]For a row that starts with $\#011q_210...$ we would look at $\#01$, 011, $11q_2$ etc.

3 Useful theorems about Hardness and Completeness

Theorem 15.2.

$$\texttt{SAT} \in \mathbf{P} \Leftrightarrow \mathbf{P} = \mathbf{NP}$$

Proof. The implication to the left is obvious from the fact that $\texttt{SAT} \in \mathbf{NP}$, so let's focus on the "right" direction.

If $\texttt{SAT} \in \mathbf{P}$, then there exists a deterministic machine $D_{\texttt{SAT}}$ that decides SAT in polynomial-time.

The theorem in section 2 tells us that any problem $f \in \mathbf{NP}$ reduces to SAT in polynomial time. So there is a transformation t, computable in polynomial-time by a deterministic machine D_t.

Chaining these two machines (D_t and $D_{\texttt{SAT}}$), we obtain a deterministic polynomial-time machine for any arbitrary **NP** problem f.

□

Note that we can restate this theorem replacing SAT with any other **NPH** problem.

Theorem 15.3.

$$\exists f, \text{ such that } \begin{cases} f \in \mathbf{NPH} \\ f \in \mathbf{P} \end{cases} \Leftrightarrow \mathbf{P} = \mathbf{NP}$$

This is an important result because it shows that "the **P** versus **NP** problem" is tied to many other questions which turn out to be equivalent. For example, a decisive "yes" or "no" answer to the open problem "$\texttt{SAT} \overset{?}{\in} \mathbf{P}$" would also be an answer to $\mathbf{P} \overset{?}{=} \mathbf{NP}$. And this holds true for questions regarding the presence in **P** of any other **NPH** problem.

But what other problems of this kind are there?

One way to show that another problem (e.g. VERTEX COVER) is **NP**-complete is to make a construction similar to the one we did for SAT: a reduction from an arbitrary problem. However this is quite difficult and laborious, but luckily we can take a shortcut by employing reductions.

Theorem 15.4.

$$\texttt{SAT} \leq_P g \Leftrightarrow g \in \mathbf{NPH}$$

Proof. The left implication is easy to see from the definition of **NPH**. So we will focus on the "right" direction.

If $\texttt{SAT} \leq_P g$, then there exists a polynomial-time-computable transformation t_1 that maps inputs of SAT to inputs of g such that they have the same truth-value.

We know now that SAT is **NP**-complete, so that for each problem $f \in \mathbf{NP}$ there exists a polynomial-time-computable transformation t_f that maps inputs of f to inputs of SAT such that they have the same truth-value.

The composition of these two transformations, $t = t_1 \circ t_f$ is a polynomial-time-computable transformation that maps inputs of f to inputs of g such that they have the same truth-value.

□

Now we can show **NP**-hardness simply by performing a reduction from SAT. Once we show a new problem to be **NP**-Hard, we can then use it as a point of further reference. This brings us to the generalization:

Theorem 15.5.

$$\left.\begin{matrix} f \in \mathbf{NPH} \\ f \leq_P g \end{matrix}\right\} \Rightarrow g \in \mathbf{NPH}$$

The proof of this involves, as before, the composition of two polynomial-time reductions.

Any theorem in this section regarding **NPH** can be transformed into one about **NPC** if we just add a condition that the relevant problem is also in **NP**.

We have suggested in the introduction that the class **NPC** consists of "the hardest problems in **NP**". But does this concept even make sense? We do not know whether $\mathbf{P} = \mathbf{NP}$; if the answer is "yes", aren't all problems as hard? The answer is "almost".

Theorem 15.6.

$$\mathbf{P} = \mathbf{NP} \Leftrightarrow \mathbf{NPC} = \mathbf{NP} \setminus \{f_{\texttt{FALSE}}, f_{\texttt{TRUE}}\}$$

$f_{\texttt{FALSE}}$ and $f_{\texttt{TRUE}}$ are the constant functions that map every input to the values `FALSE` and `TRUE` respectively.

Proof. If $\mathbf{P} = \mathbf{NP}$, each problem in **NP** has a deterministic-polynomial-time solution. So any problem can be reduced to almost any problem: $f \leq_P g$. The transformation needed by the polynomial-time reduction can simply solve f in polynomial-time and inspect the answer; if it is `TRUE`, then it produces a word x such that $g(x) = \texttt{TRUE}$, otherwise it produces y such that $g(y) = \texttt{FALSE}$.

But for this to be possible, such an x and y must exist; which is the case for all functions, except for the two constant ones.

For the reverse implication, we can reduce `SAT` to some problem that we know is in **P** (and must be **NPC** under the assumption), thus proving that `SAT` itself is in **P**. This result combined with Theorem 3 gives us the left hand side.

□

4 References and further reading

The theorem in section 2 is generally known as "the Cook-Levin theorem".

In his 1971 paper "The Complexity of Theorem-Proving Procedures" [1], Stephen Cook proved that all problems in **NP** are polynomially reducible to the problem of *tautologies* (this is similar to `SAT`, but we have to determine if **all** truth assignments satisfy a given formula).

Independently, Leonid Levin showed in 1973 [2] that a set of six problems are **NPC** and that if any of them are in **P**, then they all are and vice-versa. An English translation of this two-page paper can be found in Boris Trakhtenbrot's 1984 survey "A survey of Russian approaches to perebor (brute-force searches) algorithms" [3].

The proof presented here is adapted from [4], Chapter 7, proof of Theorem 7.37.

Interest for **NP**-completeness and **NP**-complete problems was greatly increased by Richard Karp's 1972 article "Reducibility Among Combinatorial Problems" [5]. In this paper, Karp addresses 21 problems of practical interests and shows through various reductions that they are **NPC**.

Bibliography

[1] Stephen A. Cook. "The Complexity of Theorem-Proving Procedures". In: *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*. ACM, 1971, pp. 151–158.

[2] Левин, Леонид Анатольевич. "Универсальные задачи перебора". In: Проблемы передачи информации 9.3 (1973), pp. 115–116.

[3] Boris A Trakhtenbrot. "A survey of Russian approaches to perebor (brute-force searches) algorithms". In: *Annals of the History of Computing* 6.4 (1984), pp. 384–400.

[4] Michael Sipser. *Introduction to the Theory of Computation, Third Edition*. CENGAGE Learning, 2012.

[5] Richard M. Karp. "Reducibility Among Combinatorial Problems". In: *Proceedings of a symposium on the Complexity of Computer Computations*. The IBM Research Symposia Series. Plenum Press, New York, 1972, pp. 85–103.