14 NP-Completeness

14.1 “Easy” and “hard” problems

In Section 10, we only considered two levels of difficulty of problems: computable problems and uncomputable problems. Starting with Section 12, we have begun by considering only just two levels of difficulty, where we think of a problem as “easy” if it is in P and as “hard” if it is not in P. Let’s refer to the class of decision problems that are not in P (the “hard” problems) as $\overline{\mathrm{P}}$.

The previous section introduced a third level of difficulty, NP. A problem that is in P is also in NP and every problem that is in NP is computable.

A common misconception is that $\mathrm{NP} = \overline{\mathrm{P}}$. That is not the case at all. Can you think of a problem that is in $\overline{\mathrm{P}}$ but not in NP? How about the Halting Problem (HLT)? Since every problem in NP is computable (Theorem 13.4), HLT is not in NP. HLT is also not in P, so it is in $\overline{\mathrm{P}}$.

Our ultimate goal is to find problems that are in NP but not in P. That is, they are not in P, but are only slightly outside of P since they are in NP. This section identifies “hardest” problems in NP, which are the best candidates for languages that are in NP but not in P. In overview:

1. We define polynomial-time mapping reductions and relation $A \le_p B$ where, if $A \le_p B$ and $B \in \mathrm{P}$ then $A \in \mathrm{P}$. Intuitively, you can think of $A \le_p B$ as saying that B is at least as hard as A (in our new two levels of difficulty P and $\overline{\mathrm{P}}$).

2. We define a problem to be NP-complete if it is among the hardest problems in NP. That is, it must be in NP and it must be at least as hard as every other problem in NP.

3. Section 15 identifies some NP-complete problems. In this section, we look at the consequences of a problem being NP-complete.

14.2 Polynomial-time mapping reductions

Definition 14.1. Suppose that A and B are languages (decision problems). A *polynomial-time mapping reduction from A to B* is a function f where

(a) f is computable in polynomial time.

(b) For every string x, $x \in A \leftrightarrow f(x) \in B$.

The only difference between a polynomial-time mapping reduction and the mapping reductions defined in Section 10 is the requirement that f must not merely be computable, but must be computable in polynomial time. It should come as no surprise that polynomial-time mapping reductions have properties that are similar to unrestricted mapping reductions, but with respect to P and $\overline{\text{P}}$ rather than with respect to computable and uncomputable problems.

Theorem 14.1. If $A \leq_p B$ and $B \in \text{P}$ then $A \in \text{P}$.

Proof.

1. Ask someone else to choose arbitrary decision problems A and B, and suppose that $A \leq_p B$ and $B \in \text{P}$.

Known variables:	A, B
Know (1):	$A \leq_p B$.
Know (2):	$B \in \text{P}$.
Goal:	$A \in \text{P}$.

2. Since $A \leq_p B$, there exists a polynomial-time mapping reduction from A to B. Ask someone else to provide one, and call it f.

Known variables:	A, B, f
Know (1):	$A \leq_p B$.
Know (2):	$B \in \mathrm{P}$.
Know (3):	f is a polynomial-time mapping reduction from A to B.
Goal:	$A \in \mathrm{P}$.

3. By the definition of a polynomial-time mapping reduction, we know two things.

   (a) $f(x)$ is computable in polynomial time. That means there exists a polynomial-time algorithm $F(x)$ that computes $f(x)$. But a polynomial-time algorithm is required to run in time $O(n^k)$ for some particular positive integer k, where n is the length of the input. Since $F(x)$ and k exist, let's get them from someone else.

   (b) $x \in A \leftrightarrow f(x) \in B$ for every x.

Known variables:	A, B, f, F, k
Know (1):	$A \leq_p B$.
Know (2):	$B \in \mathrm{P}$.
Know (3):	f is a polynomial-time mapping reduction from A to B.
Know (4):	Algorithm $F(x)$ computes $f(x)$ in time $O(n^k)$ where $n = \|x\|$.
Know (5):	$\forall x(x \in A \leftrightarrow f(x) \in B)$
Goal:	$A \in \mathrm{P}$.

4. Since $B \in P$, there must exist a polynomial-time algorithm $b(y)$ that tells whether $y \in B$. By the definition of a polynomial-time algorithm, $b(y)$ takes time $O(m^j)$ for a particular positive integer j, where $m = |y|$. Ask someone else to provide algorithm $b(y)$ and integer j.

Known variables:	A, B, f, F, k, b, j
Know (1):	$A \leq_p B$.
Know (2):	$B \in \mathrm{P}$.
Know (3):	f is a polynomial-time mapping reduction from A to B.
Know (4):	Algorithm $F(x)$ computes $f(x)$ in time $O(n^k)$ where $n = \lvert x \rvert$.
Know (5):	$\forall x (x \in A \leftrightarrow f(x) \in B)$
Know (6):	Algorithm $b(y)$ tells whether $y \in B$ in time $O(m^j)$ where $m = \lvert y \rvert$.
Goal:	$A \in \mathrm{P}$.

5. Consider the following program.

```
"{a(x):
   y = F(x)
   if b(y) == 1
      return 1
   else
      return 0
}"
```

It is clear that $a(x)$ correctly answers the question "is $x \in A$," since

$$\begin{aligned}
a(x) = 1 \quad &\leftrightarrow \quad b(F(x)) = 1 && \text{by inspection of } a \\
&\leftrightarrow \quad b(f(x)) = 1 && \text{by fact (4)} \\
&\leftrightarrow \quad f(x) \in B && \text{by fact (6)} \\
&\leftrightarrow \quad x \in A && \text{by fact (5)}
\end{aligned}$$

We can also show that $a(x)$ runs in polynomial time. To see that, suppose that $\lvert x \rvert = n$. Computing $y = F(x)$ takes time at most $c_1 n^k$ for some constant c_1. But in t steps, F cannot write down a string that is more than t symbols long. So $m = |y| \le c_1 n^k$. Running $b(y)$ takes $c_2 m^j \le c_2(c_1 n^k)^j = c_3 n^{jk}$ steps, where $c_3 = c_2 c_1^j$. The total time is $O(n^{jk})$.

So $A \in \mathrm{P}$ since $a(x)$ is a polynomial-time algorithm that solves A.

$\diamond$————————————————————————$\diamond$

Corollary 14.2. If $A \le_p B$ and $A \notin \mathrm{P}$ then $B \notin \mathrm{P}$.

Proof. Use Theorem 14.1 and tautology

$$(P \wedge Q) \rightarrow R \;\equiv\; (P \wedge \neg R) \rightarrow \neg Q.$$

$\diamond$————————————————————————$\diamond$

Theorem 14.3. If $A \le_p B$ and $B \le_p C$ then $A \le_p C$. That is, relation $\le_p$ is *transitive*.

Proof. Suppose that $f(x)$ is a polynomial-time mapping reduction from A to B and $g(y)$ is a polynomial-time mapping reduction from B to C. By the definition of a mapping reduction, for every x and y,

$$x \in A \quad \leftrightarrow \quad f(x) \in B \qquad (1)$$
$$y \in B \quad \leftrightarrow \quad g(y) \in C \qquad (2)$$

Define $h(x) = g(f(x))$. Notice that

$$\begin{aligned} x \in A \quad &\leftrightarrow \quad f(x) \in B && \text{by (1)} \\ &\leftrightarrow \quad g(f(x)) \in C && \text{by (2)} \\ &\leftrightarrow \quad h(x) \in C && \text{by the definition of } h(x) \end{aligned}$$

Also, $h(x)$ can be computed in polynomial time. If $f(x)$ is computable in time $O(n^k)$ and $g(y)$ is computable in time $O(n^j)$ then $h(x) = g(f(x))$ can be computed in time $O(n^{jk})$ by an argument similar to the one in step 5 of the proof of Theorem 14.1.

14.3 Definition of an NP-complete problem

Suppose that you want to find the tallest person t in a room. The first requirement, of course, is that person t must be in the room. The second is that person t must be at least as tall as every other person in the room.

Similarly, a decision problem A is a hardest problem in NP if A is in NP and A is at least as hard as every other problem in NP. A hardest problem in NP is called an NP-complete problem.

Definition 14.2. Suppose that A is a language. Say that A is *NP-complete* if

(a) $A \in NP$

(b) For every language $X \in NP$, $X \leq_p A$.

Since A is in NP, the second condition says that A is as hard as *every* problem in NP, including A itself. That is okay: $A \leq_p A$ is clearly true. A is at least as hard as itself.

14.4 Consequences of NP-completeness

It is not obvious that there exists an NP-complete problem. In Section 15, we will see some problems that are provably NP-complete. But right now, let's ask what NP-completeness tells us about a problem.

Our goal is to identify problems that are in NP but not in P. But nobody knows whether there exist *any* problems that are in NP that are not in P! Clearly, NP-completeness does not take us to our goal.

But suppose, for the sake of argument, that it turns out that P $\neq$ NP, and there is at least one language D in NP $-$ P. Also, suppose that problem E is NP-complete. Since $D \in$ NP, it must be the case that $D \leq_p E$. (*All* languages in NP polynomial-time reduce to an NP-complete problem.) Since $D \notin$ P, by Corollary 14.2, $E \notin P$. We have just proved the following.

Theorem 14.4. If P $\neq$ NP and E is NP-complete then $E \notin P$.

On the other hand, what if P = NP? By definition, an NP-complete problem is in NP, so if P = NP, then an NP-complete problem is also in P. That does not mean the problem is not NP-complete. It just means that NP-completeness is not interesting.

It is widely conjectured that P $\neq$ NP. But nobody knows if the conjecture is true.

Conjecture 14.1 P $\neq$ NP.

What would happen if someone finds a polynomial-time algorithm for an NP-complete problem? The following theorem tells you.

Theorem 14.5. If E is NP-complete and $E \in$ P then P = NP.

Proof. Suppose X is an arbitary problem in NP. Since E is NP-complete, $X \leq_p E$. By Theorem 14.1, since X polynomial-time reduces to a problem that is in P, X is also in P.

So every problem in NP is also in P. That is NP $\subseteq$ P. Since P $\subseteq$ NP (Theorem 13.5), P = NP.

◇————◇

So, if you are so inclined, you know how to prove that Conjecture 14.1 is wrong. Just find a polynomial-time algorithm for a problem that is known to be NP-complete. But be careful. Every year, a few people have tried to do exactly that. But their algorithms either do not run in polynomial time or do not work.

prev **next**