16 NP-Completeness

16.1 "Easy" and "Hard" Problems

In Section 11, we only considered two levels of difficulty of problems: computable problems and uncomputable problems. Starting with Section 14, 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 $\overline{\text{P}}$.

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 NP = $\overline{\text{P}}$. That is not the case at all. Can you think of a problem that is in $\overline{\text{P}}$ but not in NP? How about the Halting Problem (HLT)? Since every problem in NP is computable (Theorem 15.12), HLT is not in NP. HLT is also not in P, so it is in $\overline{\text{P}}$.

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 $A \leq_p B$ where, if $A \leq_p B$ and $B \in \text{P}$ then $A \in \text{P}$. Intuitively, you can think of $A \leq_p B$ as saying that B is at least as hard as A (in our new two levels of difficulty P and $\overline{\text{P}}$).

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

16.2 Polynomial-Time Mapping Reductions

BIG IDEA: Polynomial-time mapping reductions are a special case of mapping reductions that give a closer relationship between problems than the mapping reductions that we saw earlier.

Definition 16.1. 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, $x \in A \leftrightarrow f(x) \in B$.

Definition 16.2. We say that $A \leq_p B$ if there exists a polynomial-time mapping reduction from A to B.

The only difference between a polynomial-time mapping reduction and the mapping reductions defined in Section 11 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 16.3. If $A \leq_p B$ and $B \in \text{P}$ then $A \in \text{P}$.

Proof. Suppose that $A \leq_p B$ and $B \in P$. Let $f(x)$ be a mapping reduction from A to B. The following program $a(x)$ tells whether $x \in A$ in polynomial time.

```
"{a(x):
   y = f(x)
   if y ∈ B
      return 1
   else
      return 0
}"
```

1. The program correctly tells whether $x \in A$ because $x \in A \leftrightarrow f(x) \in B$.

2. Because $f(x)$ is a polynomial-time mapping reduction, step

$$y = f(x)$$

can be done in polynomial time (say, n^k) in the length n of x. Since $B \in P$, test

$$y \in B$$

can be done in polynomial time (say, m^j) in the length m of y. But we need to know what that is in terms of n. How long can y be? Since $f(x)$ runs in time n^k, it only has time to write down n^k symbols. So $m \leq n^k$ and testing whether $y \in B$ can be done in time $(n^k)^j = n^{kj}$. That is polynomial time in the length n of x.

Corollary 16.4. If $A \leq_p B$ and $A \notin \mathrm{P}$ then $B \notin \mathrm{P}$.

Proof. Use Theorem 16.3 and tautology

$$(p \wedge q) \rightarrow r \;\leftrightarrow\; (p \wedge \neg r) \rightarrow \neg q.$$

$\diamondsuit$

Theorem 16.5. If $A \leq_p B$ and $B \leq_p C$ then $A \leq_p C$. That is, relation $\leq_p$ 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,

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

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 the proof of Theorem 16.3.

16.3 Definition of an NP-Complete Problem

BIG IDEA: We can identify problems that are among the most difficult problems that are in NP.

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 16.6. 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.

16.4 Consequences of NP-Completeness

It is not obvious that there exists an NP-complete problem. In Section 17, 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 16.4, $E \notin P$. We have just proved the following.

Theorem 16.7. 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 16.8. P $\neq$ NP.

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

Theorem 16.8. 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 16.3, since X polynomial-time reduces to a problem that is in P, X is also in P.

So, if the conditions stated in Theorem 16.8 are true, then every problem in NP is also in P. That is NP $\subseteq$ P. Since P $\subseteq$ NP (Theorem 15.13), P = NP.

$\diamondsuit$

So, if you are so inclined, you know what would be involved in proving that Conjecture 16.8 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**