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 \( \mathsf{P} \) and as “hard” if it is not in \( \mathsf{P} \). Let’s refer to the class of decision problems that are not in \( \mathsf{P} \) (the “hard” problems) as \( \overline{\mathsf{P}} \).

The previous section introduced a third level of difficulty, \( \mathsf{NP} \). A problem that is in \( \mathsf{P} \) is also in \( \mathsf{NP} \) and every problem that is in \( \mathsf{NP} \) is computable.

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

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

1. We define polynomial-time mapping reductions and relation \( A \leq_p B \) where, if \( A \leq_p B \) and \( B \in \mathsf{P} \) then \( A \in \mathsf{P} \). 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 \( \mathsf{P} \) and \( \overline{\mathsf{P}} \)).

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

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 $x$, $x \in A \iff f(x) \in B$.

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 $\overline{P}$ rather than with respect to computable and uncomputable problems.

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

Proof.

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

<table>
<thead>
<tr>
<th>Known variables:</th>
<th>$A$, $B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Know (1):</td>
<td>$A \leq_p B$</td>
</tr>
<tr>
<td>Know (2):</td>
<td>$B \in P$</td>
</tr>
<tr>
<td>Goal:</td>
<td>$A \in P$</td>
</tr>
</tbody>
</table>

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 $f$. 

2
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 $x$.

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 $j$. 

---

<table>
<thead>
<tr>
<th>Known variables:</th>
<th>$A, B, f$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Know (1):</strong></td>
<td>$A \leq_p B.$</td>
</tr>
<tr>
<td><strong>Know (2):</strong></td>
<td>$B \in P.$</td>
</tr>
<tr>
<td><strong>Know (3):</strong></td>
<td>$f$ is a polynomial-time mapping reduction from $A$ to $B.$</td>
</tr>
<tr>
<td><strong>Goal:</strong></td>
<td>$A \in P.$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Known variables:</th>
<th>$A, B, f, F, k$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Know (1):</strong></td>
<td>$A \leq_p B.$</td>
</tr>
<tr>
<td><strong>Know (2):</strong></td>
<td>$B \in P.$</td>
</tr>
<tr>
<td><strong>Know (3):</strong></td>
<td>$f$ is a polynomial-time mapping reduction from $A$ to $B.$</td>
</tr>
<tr>
<td><strong>Know (4):</strong></td>
<td>Algorithm $F(x)$ computes $f(x)$ in time $O(n^k)$ where $n =</td>
</tr>
<tr>
<td><strong>Know (5):</strong></td>
<td>$\forall x (x \in A \leftrightarrow f(x) \in B)$</td>
</tr>
<tr>
<td><strong>Goal:</strong></td>
<td>$A \in P.$</td>
</tr>
</tbody>
</table>
**Known variables:** $A, B, f, F, k, b, j$

**Know (1):** $A \leq_p B$.

**Know (2):** $B \in P$.

**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):** $\forall x (x \in A \leftrightarrow f(x) \in B)$

**Know (6):** Algorithm $b(y)$ tells whether $y \in B$ in time $O(m')$ where $m = |y|$.

**Goal:** $A \in P$.

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

\[
\begin{align*}
   a(x) = 1 & \iff b(F(x)) = 1 \quad \text{by inspection of } a \\
   & \iff b(f(x)) = 1 \quad \text{by fact (4)} \\
   & \iff f(x) \in B \quad \text{by fact (6)} \\
   & \iff x \in A \quad \text{by fact (5)}
\end{align*}
\]

We can also show that $a(x)$ runs in polynomial time. To see that, suppose that $|x| = n$. 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 \( m = |y| \leq c_1 n^k \). Running \( b(y) \) takes \( c_3 m^j \leq c_2 (c_1 n^k)^j = c_3 n^{jk} \) steps, where \( c_3 = c_2 c_1^j \). The total time is \( O(n^{jk}) \).

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

\[ \text{Corollary 14.2.} \ \text{If} \ A \leq_p B \ \text{and} \ A \notin \mathbf{P} \ \text{then} \ B \notin \mathbf{P}. \]

\textbf{Proof.} Use Theorem 14.1 and tautology

\[ (P \land Q) \rightarrow R \equiv (P \land \neg R) \rightarrow \neg Q. \]

\[ \text{Theorem 14.3.} \ \text{If} \ A \leq_p B \ \text{and} \ B \leq_p C \ \text{then} \ A \leq_p C. \ \text{That is, relation} \leq_p \ \text{is transitive.} \]

\textbf{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 \( y \),

\[ x \in A \iff f(x) \in B \quad \text{(1)} \]
\[ y \in B \iff g(y) \in C \quad \text{(2)} \]

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

\[ x \in A \iff f(x) \in B \quad \text{by (1)} \]
\[ \iff g(f(x)) \in C \quad \text{by (2)} \]
\[ \iff h(x) \in C \quad \text{by the definition of} \ h(x) \]

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 \emph{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 \emph{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 \( D \not\in P \), by Corollary 14.2, \( E \not\in P \). We have just proved the following.

Theorem 14.4. If \( P \neq NP \) and \( E \) is NP-complete then \( E \not\in P \).

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 \( P \neq \text{NP} \). But nobody knows if the conjecture is true.

**Conjecture 14.1** \( P \neq \text{NP} \).

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 \( P = \text{NP} \).

**Proof.** Suppose \( X \) is an arbitrary problem in \( \text{NP} \). 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 \( \text{NP} \) is also in \( P \). That is \( \text{NP} \subseteq P \). Since \( P \subseteq \text{NP} \) (Theorem 13.5), \( P = \text{NP} \).

\[ \Diamond \]

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.