14

NP, NP completeness, and the Cook-Levin Theorem

“In this paper we give theorems that suggest, but do not imply, that these problems, as well as many others, will remain intractable perpetually”, Richard Karp, 1972

“Sad to say, but it will be many more years, if ever before we really understand the Mystical Power of Twoness… 2-SAT is easy, 3-SAT is hard, 2-dimensional matching is easy, 3-dimensional matching is hard. Why? oh, Why?” Eugene Lawler

14.1 THE CLASS NP

So far we have shown that 3SAT is no harder than Quadratic Equations, Independent Set, Maximum Cut, and Longest Path. But to show that these problems are computationally equivalent we need to give reductions in the other direction, reducing each one of these problems to 3SAT as well. It turns out we can reduce all three problems to 3SAT in one fell swoop.

In fact, this result extends far beyond these particular problems. All of the problems we discussed in Chapter 13, and a great many other problems, share the same commonality: they are all search problems, where the goal is to decide, given an instance $x$, whether there exists a solution $y$ that satisfies some condition that can be verified in polynomial time. For example, in 3SAT, the instance is a formula and the solution is an assignment to the variable; in Max-Cut the instance is a graph and the solution is a cut in the graph; and so on and so forth. It turns out that every such search problem can be reduced to 3SAT.

To make this precise, we make the following mathematical definition: we define the class NP to contain all Boolean functions that correspond to a search problem of the form above— that is, functions that output 1 on $x$ if and only if there exists a solution $w$ such that
the pair \((x, w)\) satisfies some polynomial-time checkable condition.

Formally, \(\text{NP}\) is defined as follows:

**Definition 14.1 — NP.** We say that \(F: \{0, 1\}^* \rightarrow \{0, 1\}\) is in \(\text{NP}\) if there exists some constants \(a, b \in \mathbb{N}\) and \(V: \{0, 1\}^* \rightarrow \{0, 1\}\) such that \(V \in \text{P}\) and for every \(x \in \{0, 1\}^n\),

\[
F(x) = 1 \iff \exists w \in \{0, 1\}^{anb} \text{ s.t. } V(xw) = 1.
\]

In other words, for \(F\) to be in \(\text{NP}\), there needs to exist some polynomial-time computable verification function \(V\), such that if \(F(x) = 1\) then there must exist \(w\) (of length polynomial in \(|x|\)) such that \(V(xw) = 1\), and if \(F(x) = 0\) then for every such \(w\), \(V(xw) = 0\).

Since the existence of this string \(w\) certifies that \(F(x) = 1\), \(w\) is often referred to as a certificate, witness, or proof that \(F(x) = 1\).

See also Fig. 14.1 for an illustration of Definition 14.1. The name \(\text{NP}\) stands for “nondeterministic polynomial time” and is used for historical reasons; see the bibliographical notes. The string \(w\) in (??) is sometimes known as a solution, certificate, or witness for the instance \(x\).

**Remark 14.2 — NP and proof systems.** The definition of \(\text{NP}\) means that for every \(F \in \text{NP}\) and string \(x \in \{0, 1\}^*\), \(F(x) = 1\) if and only if there is a short and efficiently verifiable proof of this fact. That is, we can think of the function \(V\) in Definition 14.1 as a verifier algorithm, similar to what we’ve seen in Section 10.1. The verifier checks whether a given string \(w \in \{0, 1\}^*\) is a valid proof for the statement “\(F(x) = 1\)”. Essentially all proof systems considered in mathematics involve line-by-line checks that can be carried out in polynomial time. Thus the heart of \(\text{NP}\) is asking for statements that have short (i.e., polynomial in the size of the statements) proof. Indeed, as we will see in Chapter 15, Kurt Gödel phrased the question of whether \(\text{NP} = \text{P}\) as asking whether “the mental work of a mathematician [in proving theorems] could be completely replaced by a machine”.

**Definition 14.1 is asymmetric** in the sense that there is a difference between an output of 1 and an output of 0. You should make sure you understand why this definition does not guarantee that if \(F \in \text{NP}\) then the function \(1 - F\) (i.e., the map \(x \mapsto 1 - F(x)\)) is in \(\text{NP}\) as well. In fact, it is believed that there do exist functions \(F\) satisfying \(F \in \text{NP}\) but \(1 - F \notin \text{NP}\). \(^1\) This is in
14.1.1 Examples of NP functions

Example 14.3 — 3SAT ∈ NP. 3SAT is in NP since for every $\ell$-variable formula $\varphi$, $3\text{SAT}(\varphi) = 1$ if and only if there exists a satisfying assignment $x \in \{0,1\}^\ell$ such that $\varphi(x) = 1$, and we can check this condition in polynomial time.

The above reasoning explains why 3SAT is in NP, but since this is our first example, we will now belabor the point and expand out in full formality what is the precise representation of the witness $w$ and the algorithm $V$ that demonstrate that 3SAT is in NP.

Specifically, we can represent a 3CNF formula $\varphi$ with $k$ variables and $m$ clauses as a string of length $n = O(m \log k)$, since every one of the $m$ clauses involves three variables and their negation, and the identity of each variable can be represented using $\lceil \log_2 k \rceil$. We assume that every variable participates in some clause (otherwise it can be ignored) and hence that $m \geq k$, which in particular means that $n$ is larger than both $m$ and $k$.

We can represent an assignment to the $k$ variables using a $k$-length string, which, since $n > k$, can be “padded” to a string $w \in \{0,1\}^n$ in some standard way. (For example, if $y \in \{0,1\}^k$ is the assignment, we can let $w = y10^{n-k-1}$; given the string $w$ we can “read off” $y$, by chopping off all the zeroes at the end of $w$ until we encounter the first 1, which we remove as well.)

Now checking whether a given assignment $y \in \{0,1\}^k$ satisfies a given $k$-variable 3CNF $\varphi$ can be done in polynomial time through the following algorithm $V$:

\[ V(w) = \begin{cases} 1 & \text{if } w \text{ satisfies } \varphi \\ 0 & \text{otherwise} \end{cases} \]
Algorithm $V$:

Input:
1. 3CNF formula $\varphi$ with $k$ variables and $m$ clauses (encoded as a string of length $n = O(m \log k)$)
2. Assignment $y \in \{0, 1\}^k$ to the variables of $\varphi$ (encoded using padding as a string $w \in \{0, 1\}^n$)

Output: 1 if and only if $y$ satisfies $\varphi$.

Operation:
1. For every clause $C = (\ell_1 \lor \ell_2 \lor \ell_3)$ of $\varphi$ (where $\ell_1, \ell_2, \ell_3$ are literals), if all three literals evaluate to false under the assignment $y$ then halt and output 0.
2. Output 1.

Algorithm $V$ runs in time polynomial in the length $n$ of $\varphi$’s description as a string. Indeed there are $m$ clauses, and checking the evaluation of a literal of the form $y_i$ or $\neg y_j$ can be done by scanning the $k$-length string $y$, and hence the running time of Algorithm $V$ is at most $O(mk) = O(n^2)$, as both $k$ and $m$ are smaller than $n$.

By its definition the algorithm outputs 1 if and only if the assignment $y$ satisfies all the clauses of the 3CNF formula $\varphi$, which means that $3\text{SAT}(\varphi) = 1$ if and only if there exists some $w \in \{0, 1\}^n$ such that $V(\varphi w) = 1$ which is precisely the condition needed to show that $3\text{SAT} \in \text{NP}$ per Definition 14.1.

**Remark 14.4 — Padding a witness.** The “padding trick” we used in Example 14.3 can always be used to expand a witness of length smaller than $an^b$ to a witness of exactly that length. Therefore one can think of the condition (??) in Definition 14.1 as simply stipulating that the “solution” $w$ to the problem $x$ is of length at most polynomial in $|x|$.

Here are some more examples for problems in $\text{NP}$. For each one of these problems we merely sketch how the witness is represented and why it is efficiently checkable, but working out the details can be a good way to get more comfortable with Definition 14.1:
• **QUADEQ** is in **NP** since for every ℓ-variable instance of quadratic equations \( E \), \( \text{QUADEQ}(E) = 1 \) if and only if there exists an assignment \( x \in \{0, 1\}^\ell \) that satisfies \( E \). We can check the condition that \( x \) satisfies \( E \) in polynomial time by enumerating over all the equations in \( E \), and for each such equation \( e \), plug in the values of \( x \) and verify that \( e \) is satisfied.

• **ISET** is in **NP** since for every graph \( G \) and integer \( k \), \( \text{ISET}(G, k) = 1 \) if and only if there exists a set \( S \) of \( k \) vertices that contains no pair of neighbors in \( G \). We can check the condition that \( S \) is an independent set of size \( \geq k \) in polynomial time by first checking that \( |S| \geq k \) and then enumerating over all edges \( \{u, v\} \) in \( G \), and for each such edge verify that either \( u \notin S \) or \( v \notin S \).

• **LONGPATH** is in **NP** since for every graph \( G \) and integer \( k \), \( \text{LONGPATH}(G, k) = 1 \) if and only if there exists a simple path \( P \) in \( G \) that is of length at least \( k \). We can check the condition that \( P \) is a simple path of length \( k \) in polynomial time by checking that it has the form \((v_0, v_1, \ldots, v_k)\) where each \( v_i \) is a vertex in \( G \), no \( v_i \) is repeated, and for every \( i \in [k] \), the edge \( \{v_i, v_{i+1}\} \) is present in the graph.

• **MAXCUT** is in **NP** since for every graph \( G \) and integer \( k \), \( \text{MAXCUT}(G, k) = 1 \) if and only if there exists a cut \((S, \overline{S})\) in \( G \) that cuts at least \( k \) edges. We can check that condition that \((S, \overline{S})\) is a cut of value at least \( k \) in polynomial time by checking that \( S \) is a subset of \( G \)'s vertices and enumerating over all the edges \( \{u, v\} \) of \( G \), counting those edges such that \( u \in S \) and \( v \notin S \) or vice versa.

### 14.1.2 Basic facts about NP

The definition of **NP** is one of the most important definitions of this book, and is worth while taking the time to digest and internalize. The following solved exercises establish some basic properties of this class. As usual, I highly recommend that you try to work out the solutions yourself.

**Solved Exercise 14.1 — Verifying is no harder than solving.** Prove that \( \mathbf{P} \subseteq \mathbf{NP} \).

**Solution:**

Suppose that \( F \in \mathbf{P} \). Define the following function \( V : V(x0^n) = 1 \) iff \( n = |x| \) and \( F(x) = 1 \). (\( V \) outputs 0 on all other inputs.) Since \( F \in \mathbf{P} \) we can clearly compute \( V \) in polynomial time as well.

Let \( x \in \{0, 1\}^n \) be some string. If \( F(x) = 1 \) then \( V(x0^n) = 1 \). On the other hand, if \( F(x) = 0 \) then for every \( w \in \{0, 1\}^n \), \( V(xw) = 0 \).
Therefore, setting \( a = b = 1 \), we see that \( V \) satisfies (??), and establishes that \( F \in \text{NP} \).

**Remark 14.5 — NP does not mean non-polynomial!**  
People sometimes think that NP stands for “non-polynomial time”. As Solved Exercise 14.1 shows, this is far from the truth, and in fact every polynomial-time computable function is in NP as well.

If \( F \) is in NP it certainly does not mean that \( F \) is hard to compute (though it does not, as far as we know, necessarily mean that it’s easy to compute either). Rather, it means that \( F \) is easy to verify, in the technical sense of Definition 14.1.

**Solved Exercise 14.2 — NP is in exponential time.** Prove that \( \text{NP} \subseteq \text{EXP} \).

**Solution:**
Suppose that \( F \in \text{NP} \) and let \( V \) be the polynomial-time computable function that satisfies (??) and \( a, b \) the corresponding constants. Then the following is an exponential-time algorithm \( A \) to compute \( F \):

**Algorithm \( A \):**
**Input:** \( x \in \{0, 1\}^* \), let \( n = |x| \)
**Operation:**
1. For every \( w \in \{0, 1\}^{anb} \), if \( V(xw) = 1 \) then halt and output 1.
2. Output 0.

Since \( V \in \text{P} \), for every \( x \in \{0, 1\}^n \), Algorithm \( A \) runs in time \( \text{poly}(n)2^{anb} \). Moreover by (??), \( A \) will output 1 on \( x \) if and only if \( F(x) = 1 \).

Solved Exercise 14.1 and Solved Exercise 14.2 together imply that

\[
\text{P} \subseteq \text{NP} \subseteq \text{EXP} .
\]  \hspace{1cm} (14.2)

The time hierarchy theorem (Theorem 12.8) implies that \( \text{P} \subsetneq \text{EXP} \) and hence at least one of the two inclusions \( \text{P} \subseteq \text{NP} \) or \( \text{NP} \subseteq \text{EXP} \) is strict. It is believed that both of them are in fact strict inclusions. That is, it is believed that there are functions in NP that cannot be
computed in polynomial time (this is the $\mathbf{P} \neq \mathbf{NP}$ conjecture) and that there are functions $F$ in $\mathbf{EXP}$ for which we cannot even efficiently certify that $F(x) = 1$ for a given input $x$.

We have previously informally equated the notion of $F \leq_p G$ with $F$ being "no harder than $G" and in particular have seen in Solved Exercise 13.1 that if $G \in \mathbf{P}$ and $F \leq_p G$, then $F \in \mathbf{P}$ as well. The following exercise shows that if $F \leq_p G$ then it is also "no harder to verify" than $G$. That is, regardless of whether or not it is in $\mathbf{P}$, if $G$ has the property that solutions to it can be efficiently verified, then so does $F$.

Solved Exercise 14.3 — Reductions and $\mathbf{NP}$. Let $F, G : \{0, 1\}^* \to \{0, 1\}$.
Show that if $F \leq_p G$ and $G \in \mathbf{NP}$ then $F \in \mathbf{NP}$.

\begin{align*}
F(x) = 1 & \iff G(R(x)) = 1 \\
& \iff \exists w\in\{0,1\}^{|R(x)|} V'(R(x)w) = 1.
\end{align*}

Since there are some constants $a', b'$ such that $|R(x)| \leq a'|x|^b$ for every $x \in \{0, 1\}^*$, by simple padding we can modify $V'$ to an algorithm that certifies that $F \in \mathbf{NP}$.

14.2 FROM $\mathbf{NP}$ TO 3SAT: THE COOK-LEVIN THEOREM

We have seen everal example of problems for which we do not know if their best algorithm is polynomial or exponential, but we can show that they are in $\mathbf{NP}$. That is, we don’t know if they are easy to solve, but we do know that it is easy to verify a given solution. There are many, many, many, more examples of interesting functions we would like to compute that are easily shown to be in $\mathbf{NP}$. What is quite amazing is that if we can solve 3SAT then we can solve all of them!

The following is one of the most fundamental theorems in Computer Science:

Theorem 14.6 — Cook-Levin Theorem. For every $F \in \mathbf{NP}$, $F \leq_p 3\mathbf{SAT}$.
reduce to 3SAT. Combining it with the reductions we’ve seen in Chapter 13, it implies that all these problems are equivalent! For example, to reduce QUADEQ to LONGPATH, we can first reduce QUADEQ to 3SAT using Theorem 14.6 and use the reduction we’ve seen in Theorem 13.7 from 3SAT to LONGPATH. That is, since QUADEQ ∈ NP, Theorem 14.6 implies that QUADEQ ≤ₚ 3SAT, and Theorem 13.7 implies that 3SAT ≤ₚ LONGPATH, which by the transitivity of reductions (Lemma 13.2) means that QUADEQ ≤ₚ LONGPATH. Similarly, since LONGPATH ∈ NP, we can use Theorem 14.6 and Theorem 13.4 to show that LONGPATH ≤ₚ 3SAT ≤ₚ QUADEQ, concluding that LONGPATH and QUADEQ are computationally equivalent.

There is of course nothing special about QUADEQ and LONGPATH here: by combining (14.6) with the reductions we saw, we see that just like 3SAT, every F ∈ NP reduces to LONGPATH, and the same is true for QUADEQ and MAXCUT. All these problems are in some sense “the hardest in NP” since an efficient algorithm for any one of them would imply an efficient algorithm for all the problems in NP. This motivates the following definition:

**Definition 14.7 — NP-hardness and NP-completeness.** We say that $G : \{0, 1\}^* \rightarrow \{0, 1\}$ is NP hard if for every $F \in$ NP, $F \leq_p G$.

We say that $G : \{0, 1\}^* \rightarrow \{0, 1\}$ is NP complete if $G$ is NP hard and $G$ is in NP.

The Cook-Levin Theorem (Theorem 14.6) can be rephrased as saying that 3SAT is NP hard, and since it is also in NP, this means that 3SAT is NP complete. Together with the reductions of Chapter 13, Theorem 14.6 shows that despite their superficial differences, 3SAT, quadratic equations, longest path, independent set, and maximum cut, are all NP-complete. Many thousands of additional problems have been shown to be NP-complete, arising from all the sciences, mathematics, economics, engineering and many other fields.³

14.2.1 What does this mean?
As we’ve seen in Solved Exercise 14.1, P ⊆ NP. The most famous conjecture in Computer Science is that this containment is strict. That is, it is widely conjectured that P ≠ NP. One way to refute the conjecture that P ≠ NP is to give a polynomial-time algorithm for even a single one of the NP-complete problems such as 3SAT, Max Cut, or the thousands of others that have been studied in all fields of human endeavors. The fact that these problems have been studied by so many people, and yet not a single polynomial-time algorithm for any of them has been found, supports that conjecture that indeed P ≠ NP. In fact, for many of these problems (including all the ones we mentioned above), we don’t even know of a $2^{o(n)}$-time algorithm! However, to the

³ For some partial lists, see this Wikipedia page and this website.
frustration of computer scientists, we have not yet been able to prove that \( P \neq \mathsf{NP} \) or even rule out the existence of an \( O(n) \)-time algorithm for 3SAT. Resolving whether or not \( P = \mathsf{NP} \) is known as the \( \mathsf{P} \ vs \ \mathsf{NP} \) problem. A million-dollar prize has been offered for the solution of this problem, a popular book has been written, and every year a new paper comes out claiming a proof of \( P = \mathsf{NP} \) or \( P \neq \mathsf{NP} \), only to wither under scrutiny.\(^4\) The following 120 page survey of Aaronson, as well as chapter 3 in Wigderson’s upcoming book are excellent sources for summarizing what is known about this problem.

One of the mysteries of computation is that people have observed a certain empirical “zero-one law” or “dichotomy” in the computational complexity of natural problems, in the sense that many natural problems are either in \( \mathsf{P} \) (often in \( \mathit{TIME}(O(n)) \) or \( \mathit{TIME}(O(n^2)) \)), or they are \( \mathsf{NP} \) hard. This is related to the fact that for most natural problems, the best known algorithm is either exponential or polynomial, with not too many examples where the best running time is some strange intermediate complexity such as \( 2^{\sqrt{n}} \). However, it is believed that there exist problems in \( \mathsf{NP} \) that are neither in \( \mathsf{P} \) nor are \( \mathsf{NP} \)-complete, and in fact a result known as “Ladner’s Theorem” shows that if \( P \neq \mathsf{NP} \) then this is indeed the case (see also Exercise 14.1 and Fig. 14.2).\(^5\)

14.2.2 The Cook-Levin Theorem: Proof outline

We will now prove the Cook-Levin Theorem, which is the underpinning to a great web of reductions from 3SAT to thousands of problems across great many fields. Some problems that have been shown to be \( \mathsf{NP} \)-complete include: minimum-energy protein folding, minimum surface-area foam configuration, map coloring, optimal Nash equilibrium, quantum state entanglement, minimum supersequence of a genome, minimum codeword problem, shortest vector in a lattice, minimum genus knots, positive Diophantine equations, integer programming, and many many more. The worst-case complexity of all these problems is (up to polynomial factors) equivalent to that of 3SAT, and through the Cook-Levin Theorem, to all problems in \( \mathsf{NP} \).

To prove Theorem 14.6 we need to show that \( F \leq_p \text{3SAT} \) for every \( F \in \mathsf{NP} \). We will do so in three stages. We define two intermediate problems: \( \text{NANDSAT} \) and \( \text{3NAND} \). We will shortly show the definitions of these two problems, but Theorem 14.6 will follow from combining the following three results:

1. \( \text{NANDSAT} \) is \( \mathsf{NP} \)-hard (Lemma 14.8).
2. \( \text{NANDSAT} \leq_p \text{3NAND} \) (Lemma 14.10).

\(^4\) The following web page keeps a catalog of these failed attempts. At the time of this writing, it lists about 110 papers claiming to resolve the question, of which about 60 claim to prove that \( P = \mathsf{NP} \) and about 50 claim to prove that \( P \neq \mathsf{NP} \).

\(^5\) TODO: maybe add examples of \( \mathsf{NP} \) hard problems as a barrier to understanding - problems from economics, physics, etc. that prevent having a closed-form solutions

\(^6\) TODO: maybe include knots
3. $3NAND \leq_p 3SAT$ (Lemma 14.11).

By the transitivity of reductions, it will follow that for every $F \in \mathbf{NP}$,

$$F \leq_p \text{NANDSAT} \leq_p 3NAND \leq_p 3SAT$$

(14.4)

hence establishing Theorem 14.6.

We will prove these three results Lemma 14.8, Lemma 14.10 and Lemma 14.11 one by one, providing the requisite definitions as we go along.

### 14.3 THE NANDSAT PROBLEM, AND WHY IT IS NP HARD.

We define the NANDSAT problem as follows. On input a string $Q \in \{0, 1\}^*$, we define $\text{NANDSAT}(Q) = 1$ if and only if $Q$ is a valid representation of an $n$-input and single-output NAND program and there exists some $w \in \{0, 1\}^n$ such that $Q(w) = 1$. While we don’t need this to prove Lemma 14.8, note that NANDSAT is in NP since we can verify that $Q(w) = 1$ using the polynomial-time algorithm for evaluating NAND-CIRC programs.\(^7\) We now prove that NANDSAT is NP hard.

**Lemma 14.8** NANDSAT is NP hard.

**Proof Idea:**

To prove Lemma 14.8 we need to show that for every $F \in \mathbf{NP}$, $F \leq_p \text{NANDSAT}$. The high-level idea is that by the definition of NP, there is some NAND-TM program $P^*$ and some polynomial $T(\cdot)$ such that $F(x) = 1$ if and only if there exists some $w \in \{0, 1\}^{|x|}$ such that $P^*(xw)$ outputs 1 within $T(|x|)$ steps. Now by “unrolling the loop” of the NAND-TM program $P^*$ we can convert it into an $O(T(n))$ NAND-CIRC program $Q'$ with $n + anb$ inputs and a single output such that for every $x \in \{0, 1\}^n$ and $w \in \{0, 1\}^{anb}$, $Q'(xw) = P^*(xw)$. on input $x \in \{0, 1\}$ that on input $w$ will simulate $P^*(xw)$ for $T(|x|)$ steps. The next step is to *hardwire* the input $x$ to $Q'$ to obtain an $O(T(n))$ line NAND-CIRC program $Q$ with $m = anb$ inputs such that for every $w \in \{0, 1\}^m$, $Q'(w) = Q(xw)$. By construction it will be the case that for every $x \in \{0, 1\}^n$, $F(x) = 1$ if and only if there exists $w \in \{0, 1\}^{anb}$ such that $Q(w) = 1$, and hence this shows that $F \leq_p \text{NANDSAT}$.

\(^7\) $Q$ is a NAND-CIRC program and not a NAND-TM program, and hence it is only defined on inputs of some particular size $n$. Evaluating $Q$ on any input $w \in \{0, 1\}^n$ can be done in time polynomial in the number of lines of $Q$.

---

\[\text{Tip:}
\]

The proof is a little bit technical but ultimately follows quite directly from the definition of NP, as well as of NAND and NAND-TM programs. If you find it confusing, try to pause here and work out the proof.
yourself from these definitions, using the idea of “unrolling the loop” of a NAND-TM program. It might also be useful for you to think how you would implement in your favorite programming language the function unroll which on input a NAND-TM program $P$ and numbers $T, n$ would output an $n$-input NAND-CIRC program $Q$ of $O(|T|)$ lines such that for every input $z \in \{0, 1\}^n$, if $P$ halts on $z$ within at most $T$ steps and outputs $y$, then $Q(z) = y$.

**Proof of Lemma 14.8.** We now present the details. Let $F \in \text{NP}$. To prove Lemma 14.8 we need to give a polynomial-time computable function that will map every $x^* \in \{0, 1\}^*$ to a NAND-CIRC program $Q$ such that $F(x) = \text{NANDSAT}(Q)$.

Let $x^* \in \{0, 1\}^*$ be such a string and let $n = |x^*|$ be its length. By Definition 14.1 there exists $V \in \text{P}$ and $a, b \in \mathbb{N}$ such that $F(x^*) = 1$ if and only if there exists $w \in \{0, 1\}^{an+b}$ such that $V(x^*w) = 1$.

Let $m = an^b$. Since $V \in \text{P}$ there is some NAND-TM program $P^*$ that computes $V$ on inputs of the form $xw$ with $x \in \{0, 1\}^n$ and $w \in \{0, 1\}^m$ in at most $(n + m)^c$ time for some constant $c$. Using our “unrolling the loop NAND-TM to NAND compiler” of Theorem 12.12, we can obtain a NAND-CIRC program $Q'$ that has $n + m$ inputs and at most $O((n + m)^c)$ lines such that $Q'(xw) = P^*(xw)$ for every $x \in \{0, 1\}^n$ and $w \in \{0, 1\}^m$.

Now we can use the following simple but useful “hardwiring” technique to obtain a program:

**Lemma 14.9 — Hardwiring Lemma.** Given a $T$-line NAND-CIRC program $Q'$ of $n + m$ inputs and $x^* \in \{0, 1\}^n$, we can obtain in polynomial a program $Q$ with $m$ inputs of $T + 3$ lines such that for every $w \in \{0, 1\}^m$, $Q(w) = Q'(x^*w)$.

**Proof.** To compute $Q$, we simply do a “search and replace” for all references in $Q'$ to $X[i]$ for $i \in [n]$, and transform them to either the variable zero or one depending on whether $x^*_i$ is equal to 0 or 1 respectively. By adding three lines to the beginning of $Q'$, we can ensure that the zero and one variables will have the correct value. The only thing that then remains to do another search and replace to transform all references to the variables $X[n], ..., X[n + m - 1]$ to the variables $X[0], ..., X[m - 1]$ so that the $m$ inputs to the new program $Q$ will correspond to last $m$ inputs of the original program $Q'$. See Fig. 14.4 for an implementation of this reduction in Python.

Using Lemma 14.9, we obtain a program $Q$ of $m$ inputs such that $Q(w) = Q'(x^*w) = P^*(x^*w)$ for every $w \in \{0, 1\}^m$. Since we know that
\( F(x^*) = 1 \) if and only if there exists \( w \in \{0, 1\}^m \) such that \( P^*(x^*w) = 1 \), this means that \( F(x^*) = 1 \) if and only if NANDSAT\((Q) = 1 \), which is what we wanted to prove.

---

14.4 THE 3NAND PROBLEM

The 3NAND problem is defined as follows: the input is a logical formula \( \varphi \) on a set of variables \( z_0, \ldots, z_{r-1} \) which is an AND of constraints of the form \( z_i = \text{NAND}(z_j, z_k) \). For example, the following is a 3NAND formula with 5 variables and 3 constraints:

\[
(z_3 = \text{NAND}(z_0, z_2)) \land (z_1 = \text{NAND}(z_0, z_2)) \land (z_4 = \text{NAND}(z_3, z_1))
\] (14.5)

The output of 3NAND on input \( \varphi \) is 1 if and only if there is an assignment to the variables of \( \varphi \) that makes it evaluate to “true” (that is, there is some assignment \( z \in \{0, 1\}^r \) satisfying all of the constraints of \( \varphi \)). As usual, we can represent \( \varphi \) as a string, and so think of 3NAND as a function mapping \( \{0, 1\}^* \) to \( \{0, 1\} \). We now prove that 3NAND is NP hard:

Lemma 14.10 NANDSAT \( \leq_P \) 3NAND.

Proof Idea:

To prove Lemma 14.10 we need to give a polynomial-time map from every NAND-CIRC program \( Q \) to a 3NAND formula \( \Psi \) such that there exists \( w \) such that \( Q(w) = 1 \) if and only if there exists \( z \) satisfying \( \Psi \). For every line \( i \) of \( Q \), we define a corresponding variable \( z_i \) of \( \Psi \). If the line \( i \) has the form \( \text{foo} = \text{NAND(bar,blah)} \) then we will add the clause \( z_i = \text{NAND}(z_j, z_k) \) where \( j \) and \( k \) are the last lines in which \( \text{bar} \) and \( \text{blah} \) were written to. We will also set variables corresponding to the input variables, as well as add a clause to ensure that the final
output is 1. The resulting reduction can be implemented in about a
dozen lines of Python, see ??.

Figure 14.5: Python code to reduce an instance $Q$ of
$\textit{NANDSAT}$ to an instance $\Psi$ of $3\textit{NAND}$. In the example
above we transform the NAND-CIRC program xor5 which has 5 input variables and 16 lines, into
a $3\textit{NAND}$ formula $\Psi$ that has 24 variables and 20 clauses. Since xor5 outputs 1 on the input 1, 0, 0, 1, 1,
there exists an assignment $z \in \{0, 1\}^{24}$ to $\Psi$ such that
$(z_0, z_1, z_2, z_3, z_4) = (1, 0, 0, 1, 1)$ and $\Psi$ evaluates to
true on $z$.

\begin{verbatim}
# Reduce NANDSAT to 3NAND
# Inputs: NANDSAT formula
# Output: 3NAND formula
# variables: vars is a list of variables
# $Q$ is a NAND-CIRC program
# $\Psi$ is $3\textit{NAND}$ formula
# $x$ is a variable
# $\gamma$ is a literal

vars = defaultdict(lambda: 2) # Line 2 corresponds to zero

for i in range(len(vars) + 1): # Setup $x, r, c, e$
  # $\Psi$ has $m+n$ variables $z_0, \ldots, z_{m+n-1}$
  # The first $n$ variables $z_0, \ldots, z_{n-1}$ correspond to the inputs of $Q$
  # The next $m$ variables $z_n, \ldots, z_{n+m-1}$ correspond to the $m$ lines of $Q$
  # For every $\ell \in \{n, n+1, \ldots, n+m\}$, if the $\ell-n$th line of the program $Q$ is
  # $foo = NAND(bar, blah)$ then we add to $\Psi$ the constraint $z_\ell = NAND(z_j, z_k)$
  # where $j-n$ and $k-n$ correspond to the last lines in which the variables bar and blah (respectively)
  # were written to. If one or both of bar and blah was not written to before then we
  # use $z_{\ell_0}$ instead of the corresponding value $z_j$ or $z_k$ in the constraint,
  # where $\ell_0 - n$ is the line in which zero is assigned a value. If one or
  # both of bar and blah is an input variable $X[i]$ then we use $z_i$ in the
  # constraint.

# Let $\ell^*$ be the last line in which the output $y_0$ is assigned a value.
# Then we add the constraint $z_{\ell^*} = NAND(z_{\ell_0}, z_{\ell_0})$ where $\ell_0 - n$ is as
# above the last line in which zero is assigned a value. Note that this
# is effectively the constraint $z_{\ell^*} = NAND(0, 0) = 1.$

\end{verbatim}
To complete the proof we need to show that there exists \( w \in \{0, 1\}^n \) s.t. \( Q(w) = 1 \) if and only if there exists \( z \in \{0, 1\}^{n+m} \) that satisfies all constraints in \( \Psi \). We now show both sides of this equivalence.

**Part I: Completeness.** Suppose that there is \( w \in \{0, 1\}^n \) s.t. \( Q(w) = 1 \). Let \( z \in \{0, 1\}^{n+m} \) be defined as follows: for \( i \in [n] \), \( z_i = w_i \) and for \( i \in \{n, n+1, \ldots, n+m\} \), \( z_i \) equals the value that is assigned in the \((i-n)\)-th line of \( Q \) when executed on \( w \). Then by construction \( z \) satisfies all of the constraints of \( \Psi \) (including the constraint that \( z_p = \text{NAND}(0,0) = 1 \) since \( Q(w) = 1 \)).

**Part II: Soundness.** Suppose that there exists \( z \in \{0, 1\}^{n+m} \) satisfying \( \Psi \). Soundness will follow by showing that \( Q(z_0, \ldots, z_{n-1}) = 1 \) (and hence in particular there exists \( w \in \{0, 1\}^n \), namely \( w = z_0 \cdots z_{n-1} \), such that \( Q(w) = 1 \)). To do this we will prove the following claim (*) for every \( \ell \in [m] \), \( z_{\ell+n} \) equals the value assigned in the \( \ell \)-th step of the execution of the program \( Q \) on \( z_0, \ldots, z_{n-1} \). Note that because \( z \) satisfies the constraints of \( \Psi \), (*) is sufficient to prove the soundness condition since these constraints imply that the last value assigned to the variable \( y_0 \) in the execution of \( Q \) on \( z_0 \cdots w_{n-1} \) is equal to 1. To prove (*) suppose, towards a contradiction, that it is false, and let \( \ell \) be the smallest number such that \( z_{\ell+n} \) is not equal to the value assigned in the \( \ell \)-th step of the execution of \( Q \) on \( z_0, \ldots, z_{n-1} \). But since \( z \) satisfies the constraints of \( \Psi \), we get that \( z_{\ell+n} = \text{NAND}(z_i, z_j) \) where (by the assumption above that \( \ell \) is smallest with this property) these values do correspond to the values last assigned to the variables on the righthand side of the assignment operator in the \( \ell \)-th line of the program. But this means that the value assigned in the \( \ell \)-th step is indeed simply the NAND of \( z_i \) and \( z_j \), contradicting our assumption on the choice of \( \ell \).

**14.5 FROM 3NAND TO 3SAT**

To conclude the proof of Theorem 14.6, we need to show Lemma 14.11 and show that \( 3\text{NAND} \leq_p 3\text{SAT} \):

**Lemma 14.11** \( 3\text{NAND} \leq_p 3\text{SAT} \).

**Proof Idea:**

To prove Lemma 14.11 we need to map a 3NAND formula \( \varphi \) into a 3SAT formula \( \psi \) such that \( \varphi \) is satisfiable if and only if \( \psi \) is. The idea is that we can transform every NAND constraint of the form \( a = \text{NAND}(b,c) \) into the AND of ORs involving the variables \( a, b, c \) and their negations, where each of the ORs contains at most three terms. The construction is fairly straightforward, and the details are given below.
It is a good exercise for you to try to find a 3CNF formula \( \xi \) on three variables \( a, b, c \) such that \( \xi(a, b, c) \) is true if and only if \( a = \text{NAND}(b, c) \). Once you do so, try to see why this implies a reduction from 3NAND to 3SAT, and hence completes the proof of Lemma 14.11.

---

**Proof of Lemma 14.11.** The constraint

\[
\begin{align*}
    z_i &= \text{NAND}(z_j, z_k) \\
    (z_1 &\vee z_2 &\vee z_3) \wedge (z_4 &\vee z_5 &\vee z_6) \wedge (z_7 &\vee z_8 &\vee z_9) \\
    (z_1 &\vee z_5 &\vee z_9) \wedge (z_2 &\vee z_6 &\vee z_7) \wedge (z_3 &\vee z_4 &\vee z_8) \\
    (z_1 &\vee z_9 &\vee z_7) \wedge (z_2 &\vee z_6 &\vee z_4) \wedge (z_3 &\vee z_4 &\vee z_5) \\
    (z_1 &\vee z_4 &\vee z_2) \wedge (z_2 &\vee z_6 &\vee z_5) \wedge (z_3 &\vee z_4 &\vee z_8) \\
    (z_1 &\vee z_5 &\vee z_7) \wedge (z_2 &\vee z_6 &\vee z_3) \wedge (z_3 &\vee z_4 &\vee z_8)
\end{align*}
\]  

(14.6)

is satisfied if \( z_i = 1 \) whenever \( (z_j, z_k) \neq (1, 1) \). By going through all cases, we can verify that (14.6) is equivalent to the constraint

\[
(z_i \lor z_j \lor z_k) \land (z_i \lor z_j) \land (z_i \lor z_k) .
\]  

(14.7)

Indeed if \( z_j = z_k = 1 \) then the first constraint of Eq. (14.7) is only true if \( z_i = 0 \). On the other hand, if either of \( z_j \) or \( z_k \) equals \( 0 \) then unless \( z_i = 1 \) either the second or third constraints will fail. This means that, given any 3NAND formula \( \varphi \) over \( n \) variables \( z_0, \ldots, z_{n-1} \), we can obtain a 3SAT formula \( \psi \) over the same variables by replacing every 3NAND constraint of \( \varphi \) with three 3OR constraints as in Eq. (14.7).\(^8\)

Because of the equivalence of (14.6) and (14.7), the formula \( \psi \) satisfies that \( \psi(z_0, \ldots, z_{n-1}) = \varphi(z_0, \ldots, z_{n-1}) \) for every assignment \( z_0, \ldots, z_{n-1} \in \{0, 1\}^n \) to the variables. In particular \( \psi \) is satisfiable if and only if \( \varphi \) is, thus completing the proof.

\[\Box\]

14.6 WRAPPING UP

We have shown that for every function \( F \) in \( \text{NP} \), \( F \leq_p \text{NANDSAT} \leq_p \text{3NAND} \leq_p \text{3SAT} \), and so \( \text{3SAT} \) is \( \text{NP-hard} \). Since in Chapter 13 we saw that \( \text{3SAT} \leq_p \text{QUADEQ} \), \( \text{3SAT} \leq_p \text{ISET} \), \( \text{3SAT} \leq_p \text{MAXCUT} \).
and 3SAT \textless_p \textsf{LONGPATH}, all these problems are \textbf{NP}-hard as well. Finally, since all the aforementioned problems are in \textbf{NP}, they are all in fact \textbf{NP}-complete and have equivalent complexity. There are thousands of other natural problems that are \textbf{NP}-complete as well. Finding a polynomial-time algorithm for any one of them will imply a polynomial-time algorithm for all of them.

Figure 14.8: An instance of the independent set problem obtained by applying the reductions NANDSAT \textless_p 3NAND \textless_p 3SAT \textless_p ISAT starting with the xor5 NAND-CIRC program.

Lecture Recap

- Many of the problems for which we don’t know polynomial-time algorithms are \textbf{NP}-complete, which means that finding a polynomial-time algorithm for one of them would imply a polynomial-time algorithm for all of them.
- It is conjectured that \textbf{NP} \neq \textbf{P} which means that we believe that polynomial-time algorithms for these problems are not merely unknown but are nonexistent.
- While an \textbf{NP}-hardness result means for example that a full-fledged “textbook” solution to a problem such as \textsf{MAX-CUT} that is as clean and general as the algorithm for \textsf{MIN-CUT} probably does not exist, it does not mean that we need to give up whenever we see a \textsf{MAX-CUT} instance. Later in this course we will discuss several strategies to deal with \textbf{NP}-hardness, including average-case complexity and approximation algorithms.

14.7 Exercises

Exercise 14.1 — Poor man’s Ladner’s Theorem. Prove that if there is no \( n^{O(\log^2 n)} \) time algorithm for 3SAT then there is some \( F \in \textbf{NP} \) such that \( F \notin \textbf{P} \) and \( F \) is not \textbf{NP} complete.\(^9\)

\(^9\) Hint: Use the function \( F \) that on input a formula \( \varphi \) and a string of the form \( 1^t \), outputs 1 if and only if \( \varphi \) is satisfiable and \( t = |\varphi|^\log |\varphi| \).
Exercise 14.2 Define WSAT to be the following function: the input is a CNF formula \( \varphi \) where each clause is the OR of one to three variables \((without\ negations)\), and a number \( k \in \mathbb{N} \). For example, the following formula can be used for a valid input to WSAT: \( \varphi = (x_5 \lor x_2 \lor x_1) \land (x_1 \lor x_3 \lor x_0) \land (x_2 \lor x_4 \lor x_0) \). The output \( WSAT(\varphi, k) = 1 \) if and only if there exists a satisfying assignment to \( \varphi \) in which exactly \( k \) of the variables get the value 1. For example for the formula above \( WSAT(\varphi, 2) = 1 \) since the assignment \((1, 1, 0, 0, 0, 0)\) satisfies all the clauses. However \( WSAT(\varphi, 1) = 0 \) since there is no single variable appearing in all clauses.

Prove that WSAT is NP-complete.

Exercise 14.3 In the employee recruiting problem we are given a list of potential employees, each of which has some subset of \( m \) potential skills, and a number \( k \). We need to assemble a team of \( k \) employees such that for every skill there would be one member of the team with this skill.

For example, if Alice has the skills “C programming”, “NAND programming” and “Solving Differential Equations”, Bob has the skills “C programming” and “Solving Differential Equations”, and Charlie has the skills “NAND programming” and “Coffee Brewing”, then if we want a team of two people that covers all the four skills, we would hire Alice and Charlie.

Define the function EMP s.t. on input the skills \( L \) of all potential employees (in the form of a sequence \( L \) of \( n \) lists \( L_1, \ldots, L_n \), each containing distinct numbers between 0 and \( m \)), and a number \( k \), \( EMP(L, k) = 1 \) if and only if there is a subset \( S \) of \( k \) potential employees such that for every skill \( j \) in \([m]\), there is an employee in \( S \) that has the skill \( j \).

Prove that EMP is NP complete.

Exercise 14.4 — Balanced max cut. Prove that the “balanced variant” of the maximum cut problem is NP-complete, where this is defined as \( BMC : \{0, 1\}^* \rightarrow \{0, 1\} \) where for every graph \( G = (V, E) \) and \( k \in \mathbb{N} \), \( BMC(G, k) = 1 \) if and only if there exists a cut \( S \) in \( G \) cutting at least \( k \) edges such that \( |S| = |V|/2 \).

Exercise 14.5 — Regular expression intersection. Let MANYREGS be the following function: On input a list of regular expressions \( exp_0, \ldots, exp_m \) (represented as strings in some standard way), output 1 if and only if
there is a single string $x \in \{0, 1\}^*$ that matches all of them. Prove that \textit{MANYREGS} is \textbf{NP}-hard.

14.8 BIBLIOGRAPHICAL NOTES

Eugene Lawler’s quote on the “mystical power of twoness” was taken from the wonderful book “The Nature of Computation” by Moore and Mertens. See also this memorial essay on Lawler by Lenstra.

14.9 FURTHER EXPLORATIONS

Some topics related to this chapter that might be accessible to advanced students include: (to be completed)

14.10 ACKNOWLEDGEMENTS

"TODO: credit surveys of Avi, Madhu"