**Theorem 7.38** (Cook–Levin Theorem)

SAT is NP-complete.

**Proof Idea** Showing that SAT is in NP is easy, and we do so shortly. The hard part of the proof is showing that any language in NP is polynomial time reducible to SAT.

To do so we construct a polynomial time reduction for each language $A$ in NP to SAT. The reduction for $A$ takes a string $w$ and produces a Boolean formula $\phi$ that simulates the NP machine for $A$ on input $w$. If the machine accepts, $\phi$ has a satisfying assignment that corresponds to the accepting computation. If the machine doesn’t accept, no assignment satisfies $\phi$. Therefore $w$ is in $A$ if and only if $\phi$ is satisfiable.

Actually constructing the reduction to work in this way is a conceptually simple task, though we must cope with many details. A Boolean formula may contain the Boolean operations AND, OR, and NOT, and these operations form the basis for the circuitry used in electronic computers. Hence the fact that we can design a Boolean formula to simulate a Turing machine isn’t surprising. The details are in the implementation of this idea.

**Proof** First, we show that SAT is in NP. A nondeterministic polynomial time machine can guess an assignment to a given formula $\phi$ and accept if the assignment satisfies $\phi$.

Next, we take any language $A$ in NP and show that $A$ is polynomial time reducible to SAT. Let $N$ be a nondeterministic Turing machine that decides $A$ in $n^k$ time for some constant $k$. (For convenience we actually assume that $N$ runs in time $n^k - 3$, but only those readers interested in details should worry about this minor point.) The following notion helps to describe the reduction.

A **tableau** for $N$ on $w$ is an $n^k \times n^k$ table whose rows are the configurations of a branch of the computation of $N$ on input $w$, as shown in the following figure.

![Diagram of a tableau](image)

**Figure 7.38**
A tableau is an $n^k \times n^k$ table of configurations
For convenience later we assume that each configuration starts and ends with a # symbol, so the first and last columns of a tableau are all #s. The first row of the tableau is the starting configuration of \( N \) on \( w \), and each row follows the previous one according to \( N \)'s transition function. A tableau is accepting if any row of the tableau is an accepting configuration.

Every accepting tableau for \( N \) on \( w \) corresponds to an accepting computation branch of \( N \) on \( w \). Thus the problem of determining whether \( N \) accepts \( w \) is equivalent to the problem of determining whether an accepting tableau for \( N \) on \( w \) exists.

Now we get to the description of the polynomial time reduction \( f \) from \( A \) to \( SAT \). On input \( w \), the reduction produces a formula \( \phi \). We begin by describing the variables of \( \phi \). Say that \( Q \) and \( \Gamma \) are the state set and tape alphabet of \( N \). Let \( C = Q \cup \Gamma \cup \{\#\} \). For each \( i \) and \( j \) between 1 and \( n^k \) and for each \( s \) in \( C \) we have a variable, \( x_{i,j,s} \).

Each of the \((n^k)^2\) entries of a tableau is called a cell. The cell in row \( i \) and column \( j \) is called cell\([i,j]\] and contains a symbol from \( C \). We represent the contents of the cells with the variables of \( \phi \). If \( x_{i,j,s} \) takes on the value 1, it means that cell\([i,j]\] contains an \( s \).

Now we design \( \phi \) so that a satisfying assignment to the variables does correspond to an accepting tableau for \( N \) on \( w \). The formula \( \phi \) is the AND of four parts \( \phi_{cell} \land \phi_{start} \land \phi_{move} \land \phi_{accept} \). We describe each part in turn.

As we mentioned previously, turning variable \( x_{i,j,s} \) on corresponds to placing symbol \( s \) in cell\([i,j]\]. The first thing we must guarantee in order to obtain a correspondence between an assignment and a tableau is that the assignment turns on exactly one variable for each cell. Formula \( \phi_{cell} \) ensures this requirement by expressing it in terms of Boolean operations:

\[
\phi_{cell} = \bigwedge_{1 \leq i, j \leq n^k} \left[ \left( \bigvee_{s \in C} x_{i,j,s} \right) \land \left( \bigwedge_{s, t \in C, s \neq t} (\overline{x_{i,j,s}} \lor \overline{x_{i,j,t}}) \right) \right].
\]

The symbols \( \land \) and \( \lor \) stand for iterated AND and OR. For example, the expression in the preceding formula

\[
\bigvee_{s \in C} x_{i,j,s}
\]

is shorthand for

\[
x_{i,j,s_1} \lor x_{i,j,s_2} \lor \cdots \lor x_{i,j,s_l}
\]

where \( C = \{s_1, s_2, \ldots, s_l\} \). Hence \( \phi_{cell} \) is actually a large expression that contains a fragment for each cell in the tableau because \( i \) and \( j \) range from 1 to \( n^k \). The first part of each fragment says that at least one variable is turned on in the corresponding cell. The second part of each fragment says that no more than one variable is turned on (literally, it says that in each pair of variables, at least one is turned off) in the corresponding cell. These fragments are connected by \( \land \) operations.
The first part of $\phi_{\text{cell}}$ inside the brackets stipulates that at least one variable that is associated to each cell is on, whereas the second part stipulates that no more than one variable is on for each cell. Any assignment to the variables that satisfies $\phi$ (and therefore $\phi_{\text{cell}}$) must have exactly one variable on for every cell. Thus any satisfying assignment specifies one symbol in each cell of the table. Parts $\phi_{\text{start}}$, $\phi_{\text{move}}$, and $\phi_{\text{accept}}$ ensure that these symbols actually correspond to an accepting tableau as follows.

Formula $\phi_{\text{start}}$ ensures that the first row of the table is the starting configuration of $N$ on $w$ by explicitly stipulating that the corresponding variables are on:

$$
\phi_{\text{start}} = x_{1,1,1} \wedge x_{1,2,0} \wedge x_{1,3,1} \wedge x_{1,4,0} \wedge \ldots \wedge x_{1,n+3,1} \wedge \ldots \wedge x_{1,n+k-1,1} \wedge x_{1,n+k,0}.
$$

Formula $\phi_{\text{accept}}$ guarantees that an accepting configuration occurs in the tableau. It ensures that $q_{\text{accept}}$, the symbol for the accept state, appears in one of the cells of the tableau, by stipulating that one of the corresponding variables is on:

$$
\phi_{\text{accept}} = \bigvee_{1 \leq i, j \leq n^k} x_{i,j,q_{\text{accept}}}.
$$

Finally, formula $\phi_{\text{move}}$ guarantees that each row of the table corresponds to a configuration that legally follows the preceding row’s configuration according to $N$’s rules. It does so by ensuring that each $2 \times 3$ window of cells is legal. We say that a $2 \times 3$ window is legal if that window does not violate the actions specified by $N$’s transition function. In other words, a window is legal if it might appear when one configuration correctly follows another.\(^3\)

For example, say that $a$, $b$, and $c$ are members of the tape alphabet and $q_1$ and $q_2$ are states of $N$. Assume that, when in state $q_1$ with the head reading an $a$, $N$ writes a $b$, stays in state $q_1$ and moves right, and that when in state $q_1$ with the head reading a $b$, $N$ nondeterministically either

1. writes a $c$, enters $q_2$ and moves to the left, or
2. writes an $a$, enters $q_2$ and moves to the right.

Expressed formally, $\delta(q_1, a) = \{(q_1, b, R)\}$ and $\delta(q_1, b) = \{(q_2, c, L), (q_2, a, R)\}$.

Examples of legal windows for this machine are shown in Figure 7.39.

---

\(^3\)We could give a precise definition of legal window here, in terms of the transition function. But doing so is quite tedious and would distract us from the main thrust of the proof argument. Anyone desiring more precision should refer to the related analysis in the proof of Theorem 5.15, the undecidability of the Post Correspondence Problem.
FIGURE 7.39
Examples of legal windows

In Figure 7.39, windows (a) and (b) are legal because the transition function allows $N$ to move in the indicated way. Window (c) is legal because, with $q_1$ appearing on the right side of the top row, we don’t know what symbol the head is over. That symbol could be an $a$, and $q_1$ might change it to a $b$ and move to the right. That possibility would give rise to this window, so it doesn’t violate $N$’s rules. Window (d) is obviously legal because the top and bottom are identical, which would occur if the head weren’t adjacent to the location of the window. Note that # may appear on the left or right of both the top and bottom rows in a legal window. Window (e) is legal because state $q_1$ reading a $b$ might have been immediately to the right of the top row, and it would then have moved to the left in state $q_2$ to appear on the right-hand end of the bottom row. Finally, window (f) is legal because state $q_1$ might have been immediately to the left of the top row and it might have changed the $b$ to a $c$ and moved to the left.

The windows shown in the following figure aren’t legal for machine $N$.

FIGURE 7.40
Examples of illegal windows

In window (a) the central symbol in the top row can’t change because a state wasn’t adjacent to it. Window (b) isn’t legal because the transition function specifies that the $b$ gets changed to a $c$ but not to an $a$. Window (c) isn’t legal because two states appear in the bottom row.

CLAIM 7.41
If the top row of the table is the start configuration and every window in the table is legal, each row of the table is a configuration that legally follows the preceding one.
We prove this claim by considering any two adjacent configurations in the table, called the upper configuration and the lower configuration. In the upper configuration, every cell that isn’t adjacent to a state symbol and that doesn’t contain the boundary symbol #, is the center top cell in a window whose top row contains no states. Therefore that symbol must appear unchanged in the center bottom of the window. Hence it appears in the same position in the bottom configuration.

The window containing the state symbol in the center top cell guarantees that the corresponding three positions are updated consistently with the transition function. Therefore, if the upper configuration is a legal configuration, so is the lower configuration, and the lower one follows the upper one according to $N$'s rules. Note that this proof, though straightforward, depends crucially on our choice of a $2 \times 3$ window size, as Exercise 7.39 shows.

Now we return to the construction of $\phi_{\text{move}}$. It stipulates that all the windows in the tableau are legal. Each window contains six cells, which may be set in a fixed number of ways to yield a legal window. Formula $\phi_{\text{move}}$ says that the settings of those six cells must be one of these ways, or

$$\phi_{\text{move}} = \bigwedge_{1<i\leq n^k, \ 1<j<n^k} \text{(the } (i, j) \text{ window is legal)}$$

We replace the text “the $(i, j)$ window is legal” in this formula with the following formula. We write the contents of six cells of a window as $a_1, \ldots, a_6$.

$$\bigvee_{a_1, \ldots, a_6} (x_{i,j-1,a_1} \land x_{i,j,a_2} \land x_{i,j+1,a_3} \land x_{i+1,j-1,a_4} \land x_{i+1,j,a_5} \land x_{i+1,j+1,a_6})$$

is a legal window

Next we analyze the complexity of the reduction to show that it operates in polynomial time. To do so we examine the size of $\phi$. First, we estimate the number of variables it has. Recall that the tableau is an $n^k \times n^k$ table, so it contains $n^{2k}$ cells. Each cell has $l$ variables associated with it, where $l$ is the number of symbols in $C$. Because $l$ depends only on the TM $N$ and not on the length of the input $n$, the total number of variables is $O(n^{2k})$.

We estimate the size of each of the parts of $\phi$. Formula $\phi_{\text{cell}}$ contains a fixed-size fragment of the formula for each cell of the tableau, so its size is $O(n^{2k})$. Formula $\phi_{\text{start}}$ has a fragment for each cell in the top row, so its size is $O(n^k)$. Formulas $\phi_{\text{move}}$ and $\phi_{\text{accept}}$ each contain a fixed-size fragment of the formula for each cell of the tableau, so their size is $O(n^{2k})$. Thus $\phi$'s total size is $O(n^{2k})$. That bound is sufficient for our purposes because it shows that the the size of $\phi$ is polynomial in $n$. If it were more than polynomial, the reduction wouldn’t have any chance of generating it in polynomial time. (Actually our estimates are low by a factor of $O(n \log n)$ because each variable has indices that can range up to $n^k$ and so may require $O(n \log n)$ symbols to write into the formula, but this additional factor doesn’t change the polynomiality of the result.)

To see that we can generate the formula in polynomial time, observe its highly repetitive nature. Each component of the formula is composed of many nearly
identical fragments, which differ only at the indices in a simple way. Therefore we may easily construct a reduction that produces $\phi$ in polynomial time from the input $w$.

Thus we have concluded the proof of the Cook–Levin theorem, showing that $SAT$ is NP-complete. Showing the NP-completeness of other languages generally doesn’t require such a lengthy proof. Instead NP-completeness can be proved with a polynomial time reduction from a language that is already known to be NP-complete. We can use $SAT$ for this purpose, but using $3SAT$, the special case of $SAT$ that we defined on page 278, is usually easier. Recall that the formulas in $3SAT$ are in conjunctive normal form (cnf) with three literals per clause. First, we must show that $3SAT$ itself is NP-complete. We prove this assertion as a corollary to Theorem 7.37.

**COROLLARY 7.42**

$3SAT$ is NP-complete.

**PROOF** Obviously $3SAT$ is in NP, so we only need to prove that all languages in NP reduce to $3SAT$ in polynomial time. One way to do so is by showing that $SAT$ polynomial time reduces to $3SAT$. Instead, we modify the proof of Theorem 7.37 so that it directly produces a formula in conjunctive normal form with three literals per clause.

Theorem 7.37 produces a formula that is already almost in conjunctive normal form. Formula $\phi_{cell}$ is a big AND of subformulas, each of which contains a big OR and a big AND of ORs. Thus $\phi_{cell}$ is an AND of clauses and so is already in cnf. Formula $\phi_{start}$ is a big AND of variables. Taking each of these variables to be a clause of size 1 we see that $\phi_{start}$ is in cnf. Formula $\phi_{accept}$ is a big OR of variables and is thus a single clause. Formula $\phi_{move}$ is the only one that isn’t already in cnf, but we may easily convert it into a formula that is in cnf as follows.

Recall that $\phi_{move}$ is a big AND of subformulas, each of which is an OR of ANDs that describes all possible legal windows. The distributive laws, as described in Chapter 0, state that we can replace any OR of ANDs with an equivalent AND of ORs. Doing so may significantly increase the size of each subformula, but it can only increase the total size of $\phi_{move}$ by a constant factor because the size of each subformula depends only on $N$. The result is a formula that is in conjunctive normal form.

Now that we have written the formula in cnf, we convert it to one with three literals per clause. In each clause that currently has one or two literals, we replicate one of the literals until the total number is three. In each clause that has more than three literals, we split it into several clauses and add additional variables to preserve the satisfiability or nonsatisfiability of the original.

For example, we replace clause $(a_1 \lor a_2 \lor a_3 \lor a_4)$, wherein each $a_i$ is a literal, with the two-clause expression $(a_1 \lor a_2 \lor z) \land (\neg z \lor a_3 \lor a_4)$, wherein $z$ is a new
variable. If some setting of the \( a_i \)'s satisfies the original clause, we can find some setting of \( z \) so that the two new clauses are satisfied. In general, if the clause contains \( l \) literals,

\[(a_1 \lor a_2 \lor \cdots \lor a_l),\]

we can replace it with the \( l - 2 \) clauses

\[(a_1 \lor a_2 \lor z_1) \land (\overline{z_1} \lor a_3 \lor z_2) \land (\overline{z_2} \lor a_4 \lor z_3) \land \cdots \land (\overline{z_{l-3}} \lor a_{l-1} \lor a_l).\]

We may easily verify that the new formula is satisfiable iff the original formula was, so the proof is complete.

---

### 7.5 ADDITIONAL NP-COMPLETE PROBLEMS

The phenomenon of NP-completeness is widespread. NP-complete problems appear in many fields. For reasons that are not well understood, most naturally occurring NP-problems are known either to be in P or to be NP-complete. If you seek a polynomial time algorithm for a new NP-problem, spending part of your effort attempting to prove it NP-complete is sensible because doing so may prevent you from working to find a polynomial time algorithm that doesn’t exist.

In this section we present additional theorems showing that various languages are NP-complete. These theorems provide examples of the techniques that are used in proofs of this kind. Our general strategy is to exhibit a polynomial time reduction from 3SAT to the language in question, though we sometimes reduce from other NP-complete languages when that is more convenient.

When constructing a polynomial time reduction from 3SAT to a language, we look for structures in that language that can simulate the variables and clauses in Boolean formulas. Such structures are sometimes called gadgets. For example, in the reduction from 3SAT to CLIQUE presented in Theorem 7.32, individual nodes simulate variables and triples of nodes simulate clauses. An individual node may or may not be a member of the clique, which corresponds to a variable that may or may not be true in a satisfying assignment. Each clause must contain a literal that is assigned \textsc{true} and that corresponds to the way each triple must contain a node in the clique if the target size is to be reached. The following corollary to Theorem 7.32 states that CLIQUE is NP-complete.

---

**Corollary 7.43**

CLIQUE is NP-complete.