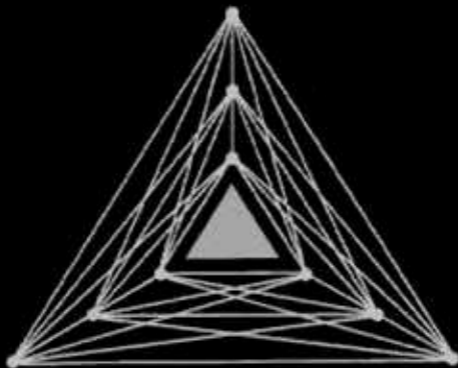# COMPUTERS AND INTRACTABILITY
A Guide to the Theory of NP-Completeness

Michael R. Garey / David S. Johnson

requirements we have just described would appear to be rather demanding. One must show that *every* problem in NP transforms to our prospective NP-complete problem Π. It is not at all obvious how one might go about doing this. *A priori*, it is not even apparent that any NP-complete problems need exist.

The following lemma, which is an immediate consequence of our definitions and the transitivity of ∝, shows that matters would be simplified considerably if we possessed just one problem that we knew to be NP-complete.

Lemma 2.3 If L_1 and L_2 belong to NP, L_1 is NP-complete, and $L_1 \propto L_2$, then L_2 is NP-complete.
Proof: Since $L_2 \in$ NP, all we need to do is show that, for every $L' \in$ NP, $L' \propto L_2$. Consider any $L' \in$ NP. Since L_1 is NP-complete, it must be the case that $L' \propto L_1$. The transitivity of ∝ and the fact that $L_1 \propto L_2$ then imply that $L' \propto L_2$. ■

Translated to the decision problem level, this lemma gives us a straightforward approach for proving new problems NP-complete, once we have at least one known NP-complete problem available. To prove that Π is NP-complete, we merely show that

1. Π ∈ NP, and

2. some known NP-complete problem Π′ transforms to Π.

Before we can use this approach, however, we still need some first NP-complete problem. Such a problem is provided by Cook's fundamental theorem, which we state and prove in the next section.

2.6 Cook's Theorem

The honor of being the "first" NP-complete problem goes to a decision problem from Boolean logic, which is usually referred to as the SATISFIABILITY problem (SAT, for short). The terms we shall use in describing it are defined as follows:

Let $U = \{u_1, u_2, \ldots, u_m\}$ be a set of Boolean *variables*. A *truth assignment* for U is a function $t: U \to \{T, F\}$. If $t(u) = T$ we say that u is "true" under t; if $t(u) = F$ we say that u is "false." If u is a variable in U, then u and $\bar{u}$ are *literals* over U. The literal u is true under t if and only if the variable u is true under t; the literal $\bar{u}$ is true if and only if the variable u is false.

A *clause* over U is a set of literals over U, such as $\{u_1, \bar{u}_3, u_8\}$. It represents the disjunction of those literals and is *satisfied* by a truth assignment if and only if at least one of its members is true under that assignment. The clause above will be satisfied by t *unless* $t(u_1) = F$, $t(u_3) = T$,

and $t(u_8) = F$. A collection C of clauses over U is *satisfiable* if and only if there exists some truth assignment for U that simultaneously satisfies all the clauses in C. Such a truth assignment is called a *satisfying truth assignment* for C. The SATISFIABILITY problem is specified as follows:

SATISFIABILITY
INSTANCE: A set U of variables and a collection C of clauses over U.
QUESTION: Is there a satisfying truth assignment for C?

For example, $U = \{u_1, u_2\}$ and $C = \{\{u_1, \bar{u}_2\}, \{\bar{u}_1, u_2\}\}$ provide an instance of SAT for which the answer is "yes." A satisfying truth assignment is given by $t(u_1) = t(u_2) = T$. On the other hand, replacing C by $C' = \{\{u_1, u_2\}, \{u_1, \bar{u}_2\}, \{\bar{u}_1\}\}$ yields an instance for which the answer is "no"; C' is not satisfiable.

The seminal theorem of Cook [1971] can now be stated:

Theorem 2.1 (Cook's Theorem) SATISFIABILITY is NP-complete.
Proof: SAT is easily seen to be in NP. A nondeterministic algorithm for it need only guess a truth assignment for the given variables and check to see whether that assignment satisfies all the clauses in the given collection C. This is easy to do in (nondeterministic) polynomial time. Thus the first of the two requirements for NP-completeness is met.

For the second requirement, let us revert to the language level, where SAT is represented by a language $L_{SAT} = L[SAT, e]$ for some reasonable encoding scheme e. We must show that, for all languages $L \in$ NP, $L \propto L_{SAT}$. The languages in NP are a rather diverse lot, and there are infinitely many of them, so we cannot hope to present a separate transformation for each one of them. However, each of the languages in NP can be described in a standard way, simply by giving a polynomial time NDTM program that recognizes it. This allows us to work with a generic polynomial time NDTM program and to derive a generic transformation from the language it recognizes to L_{SAT}. This generic transformation, when specialized to a particular NDTM program M recognizing the language L_M, will give the desired polynomial transformation from L_M to L_{SAT}. Thus, in essence, we will present a simultaneous proof for all $L \in$ NP that $L \propto L_{SAT}$.

To begin, let M denote an arbitrary polynomial time NDTM program, specified by Γ, Σ, b, Q, q_0, q_Y, q_N, and δ, which recognizes the language $L = L_M$. In addition, let $p(n)$ be a polynomial over the integers that bounds the time complexity function $T_M(n)$. (Without loss of generality, we can assume that $p(n) \geqslant n$ for all $n \in Z^+$.) The generic transformation f_L will be derived in terms of M, Γ, Σ, b, Q, q_0, q_Y, q_N, δ, and p.

It will be convenient to describe f_L as if it were a mapping from strings over Σ to instances of SAT, rather than to strings over the alphabet of our encoding scheme for SAT, since the details of the encoding scheme could

be filled in easily. Thus f_L will have the property that for all $x \in \Sigma^*$, $x \in L$ if and only if $f_L(x)$ has a satisfying truth assignment. The key to the construction of f_L is to show how a set of clauses can be used to check whether an input x is accepted by the NDTM program M, that is, whether $x \in L$.

If the input $x \in \Sigma^*$ is accepted by M, then we know that there is an accepting computation for M on x such that both the number of steps in the checking stage and the number of symbols in the guessed string are bounded by $p(n)$, where $n = |x|$. Such a computation cannot involve any tape squares except for those numbered $-p(n)$ through $p(n)+1$, since the read-write head begins at square 1 and moves at most one square in any single step. The status of the checking computation at any one time can be specified completely by giving the contents of these squares, the current state, and the position of the read-write head. Furthermore, since there are no more than $p(n)$ steps in the checking computation, there are at most $p(n)+1$ distinct times that must be considered. This will enable us to describe such a computation completely using only a limited number of Boolean variables and a truth assignment to them.

The variable set U that f_L constructs is intended for just this purpose. Label the elements of Q as $q_0, q_1=q_Y, q_2=q_N, q_3, \ldots, q_r$, where $r = |Q|-1$, and label the elements of Γ as $s_0=b, s_1, s_2, \ldots, s_v$, where $v = |\Gamma|-1$. There will be three types of variables, each of which has an intended meaning as specified in Figure 2.7. By the phrase "at time i" we mean "upon completion of the ith step of the checking computation."

Variable	Range	Intended meaning
$Q[i,k]$	$0 \leqslant i \leqslant p(n)$ $0 \leqslant k \leqslant r$	At time i, M is in state q_k.
$H[i,j]$	$0 \leqslant i \leqslant p(n)$ $-p(n) \leqslant j \leqslant p(n)+1$	At time i, the read-write head is scanning tape square j.
$S[i,j,k]$	$0 \leqslant i \leqslant p(n)$ $-p(n) \leqslant j \leqslant p(n)+1$ $0 \leqslant k \leqslant v$	At time i, the contents of tape square j is symbol s_k.

Figure 2.7 Variables in $f_L(x)$ and their intended meanings.

A computation of M induces a truth assignment on these variables in the obvious way, under the convention that, if the program halts before time $p(n)$, the configuration remains static at all later times, maintaining the same halt-state, head position, and tape contents. The tape contents at

time 0 consists of the input x, written in squares 1 through n, and the guess w, written in squares -1 through $-|w|$, with all other squares blank.

On the other hand, an arbitrary truth assignment for these variables need not correspond at all to a computation, much less to an accepting computation. According to an arbitrary truth assignment, a given tape square might contain many symbols at one time, the machine might be simultaneously in several different states, and the read-write head could be in any subset of the positions $-p(n)$ through $p(n)+1$. The transformation f_L works by constructing a collection of clauses involving these variables such that a truth assignment is a *satisfying* truth assignment if and only if it is the truth assignment induced by an accepting computation for x whose checking stage takes $p(n)$ or fewer steps and whose guessed string has length at most $p(n)$. We thus will have

$x \in L$ $\Longleftrightarrow$ there is an accepting computation of M on x

        $\Longleftrightarrow$ there is an accepting computation of M on x with $p(n)$ or fewer steps in its checking stage and with a guessed string w of length exactly $p(n)$

        $\Longleftrightarrow$ there is a satisfying truth assignment for the collection of clauses in $f_L(x)$.

This will mean that f_L satisfies one of the two conditions required of a polynomial transformation. The other condition, that f_L can be computed in polynomial time, will be verified easily once we have completed our description of f_L.

The clauses in $f_L(x)$ can be divided into six groups, each imposing a separate type of restriction on any satisfying truth assignment as given in Figure 2.8.

It is straightforward to observe that if all six clause groups perform their intended missions, then a satisfying truth assignment will have to correspond to the desired accepting computation for x. Thus all we need to show is how clause groups performing these missions can be constructed.

Group G_1 consists of the following clauses:

$$\{Q[i,0], Q[i,1], \ldots, Q[i,r]\}, \; 0 \leqslant i \leqslant p(n)$$
$$\{\overline{Q[i,j]}, \overline{Q[i,j']}\}, \; 0 \leqslant i \leqslant p(n), 0 \leqslant j < j' \leqslant r$$

The first $p(n)+1$ of these clauses can be simultaneously satisfied if and only if, for each time i, M is in *at least* one state. The remaining $(p(n)+1)(r+1)(r/2)$ clauses can be simultaneously satisfied if and only if at no time i is M in *more than one* state. Thus G_1 performs its mission.

Groups G_2 and G_3 are constructed similarly, and groups G_4 and G_5 are both quite simple, each consisting only of one-literal clauses. Figure 2.9 gives a complete specification of the first five groups. Note that the number

Clause group	Restriction imposed
G_1	At each time i, M is in exactly one state.
G_2	At each time i, the read-write head is scanning exactly one tape square.
G_3	At each time i, each tape square contains exactly one symbol from Γ.
G_4	At time 0, the computation is in the initial configuration of its checking stage for input x.
G_5	By time $p(n)$, M has entered state q_Y and hence has accepted x.
G_6	For each time i, $0 \leqslant i < p(n)$, the configuration of M at time $i+1$ follows by a single application of the transition function δ from the configuration at time i.

Figure 2.8 Clause groups in $f_L(x)$ and the restrictions they impose on satisfying truth assignments.

Clause group	Clauses in group
G_1	$\{Q[i,0],Q[i,1],\ldots,Q[i,r]\}$, $0 \leqslant i \leqslant p(n)$ $\{\overline{Q[i,j]},\overline{Q[i,j']}\}$, $0 \leqslant i \leqslant p(n), 0 \leqslant j < j' \leqslant r$
G_2	$\{H[i,-p(n)],H[i,-p(n)+1],\ldots,H[i,p(n)+1]\}$, $0 \leqslant i \leqslant p(n)$ $\{\overline{H[i,j]},\overline{H[i,j']}\}$, $0 \leqslant i \leqslant p(n), -p(n) \leqslant j < j' \leqslant p(n)+1$
G_3	$\{S[i,j,0],S[i,j,1],\ldots,S[i,j,v]\}$, $0 \leqslant i \leqslant p(n), -p(n) \leqslant j \leqslant p(n)+1$ $\{\overline{S[i,j,k]},\overline{S[i,j,k']}\}$, $0 \leqslant i \leqslant p(n), -p(n) \leqslant j \leqslant p(n)+1, 0 \leqslant k < k' \leqslant v$
G_4	$\{Q[0,0]\},\{H[0,1]\},\{S[0,0,0]\}$, $\{S[0,1,k_1]\},\{S[0,2,k_2]\},\cdots,\{S[0,n,k_n]\}$, $\{S[0,n+1,0]\},\{S[0,n+2,0]\},\ldots,\{S[0,p(n)+1,0]\}$, where $x = s_{k_1}s_{k_2}\cdots s_{k_n}$
G_5	$\{Q[p(n),1]\}$

Figure 2.9 The first five clause groups in $f_L(x)$.

of clauses in these groups, and the maximum number of literals occurring in each clause, are both bounded by a polynomial function of n (since r and v are *constants* determined by M and hence by L).

The final clause group G_6, which ensures that each successive configuration in the computation follows from the previous one by a single step of program M, is a bit more complicated. It consists of two subgroups of clauses.

The first subgroup guarantees that if the read-write head is *not* scanning tape square j at time i, then the symbol in square j does not change between times i and $i+1$. The clauses in this subgroup are as follows:

$$\{\overline{S[i,j,l]}, H[i,j], S[i+1,j,l]\}, \ 0 \leqslant i < p(n), -p(n) \leqslant j \leqslant p(n)+1, 0 \leqslant l \leqslant v$$

For any time i, tape square j, and symbol s_l, if the read-write head is not scanning square j at time i, and square j contains s_l at time i but not at time $i+1$, then the above clause based on i, j, and l will fail to be satisfied (otherwise it *will* be satisfied). Thus the $2(p(n)+1)^2(v+1)$ clauses in this subgroup perform their mission.

The remaining subgroup of G_6 guarantees that the *changes* from one configuration to the next are in accord with the transition function δ for M. For each quadruple (i,j,k,l), $0 \leqslant i < p(n)$, $-p(n) \leqslant j \leqslant p(n)+1$, $0 \leqslant k \leqslant r$, and $0 \leqslant l \leqslant v$, this subgroup contains the following three clauses:

$$\{\overline{H[i,j]}, \overline{Q[i,k]}, \overline{S[i,j,l]}, H[i+1,j+\Delta]\}$$
$$\{\overline{H[i,j]}, \overline{Q[i,k]}, \overline{S[i,j,l]}, Q[i+1,k']\}$$
$$\{\overline{H[i,j]}, \overline{Q[i,k]}, \overline{S[i,j,l]}, S[i+1,j,l']\}$$

where if $q_k \in Q-\{q_Y,q_N\}$, then the values of Δ, k', and l' are such that $\delta(q_k,s_l) = (q_{k'},s_{l'},\Delta)$, and if $q_k \in \{q_Y,q_N\}$, then $\Delta = 0$, $k' = k$, and $l' = l$.

Although it may require a few minutes of thought, it is not difficult to see that these $6(p(n))(p(n)+1)(r+1)(v+1)$ clauses impose the desired restriction on satisfying truth assignments.

Thus we have shown how to construct clause groups G_1 through G_6 performing the previously stated missions. If $x \in L$, then there is an accepting computation of M on x of length $p(n)$ or less, and this computation, given the interpretation of the variables, imposes a truth assignment that satisfies all the clauses in $C = G_1 \cup G_2 \cup G_3 \cup G_4 \cup G_5 \cup G_6$.

Conversely, the construction of C is such that any satisfying truth assignment for C must correspond to an accepting computation of M on x. It follows that $f_L(x)$ has a satisfying truth assignment if and only if $x \in L$.

All that remains to be shown is that, for any fixed language L, $f_L(x)$ can be constructed from x in time bounded by a polynomial function of $n = |x|$. Given L, we choose a particular NDTM M that recognizes L in time bounded by a polynomial p (we need not find this NDTM itself in polynomial time, since we are only proving that the desired transformation f_L *exists*). Once we have a specific NDTM M and a specific polynomial p, the construction of the set U of variables and collection C of clauses amounts to little more than filling in the blanks in a standard (though complicated) formula. The polynomial boundedness of this computation will follow immediately once we show that Length $[f_L(x)]$ is bounded above by a polynomial function of n, where Length $[I]$ reflects the length of a string encoding the instance I under a reasonable encoding scheme, as discussed in Section 2.1. Such a "reasonable" Length function for SAT is given, for example, by $|U| \cdot |C|$. No clause can contain more than $2 \cdot |U|$ literals (that's all the literals there are), and the number of symbols required to describe an individual literal need only add an additional $\log |U|$ factor, which can be ignored when all that is at issue is polynomial boundedness. Since r and v are fixed in advance and can contribute only constant factors to $|U|$ and $|C|$, we have $|U| = O(p(n)^2)$ and $|C| = O(p(n)^2)$. Hence Length $[f_L(x)] = |U| \cdot |C| = O(p(n)^4)$, and is bounded by a polynomial function of n as desired.

Thus the transformation f_L can be computed by a polynomial time algorithm (although the particular polynomial bound it obeys will depend on L and on our choices for M and p), and we conclude that, for every $L \in NP$, f_L is a polynomial transformation from L to SAT (technically, of course, from L to L_{SAT}). It follows, as claimed, that SAT is NP-complete. ∎

3

Proving NP-Completeness Results

If every NP-completeness proof had to be as complicated as that for SATISFIABILITY, it is doubtful that the class of known NP-complete problems would have grown as fast as it has. However, as discussed in Section 2.4, once we have proved a single problem NP-complete, the procedure for proving additional problems NP-complete is greatly simplified. Given a problem $\Pi \in NP$, all we need do is show that some already known NP-complete problem Π' can be transformed to Π. Thus, from now on, the process of devising an NP-completeness proof for a decision problem Π will consist of the following four steps:

(1)   showing that Π is in NP,

(2)   selecting a known NP-complete problem Π',

(3)   constructing a transformation f from Π' to Π, and

(4)   proving that f is a (polynomial) transformation.

In this chapter, we intend not only to acquaint readers with the end results of this process (the finished NP-completeness proofs) but also to prepare them for the task of constructing such proofs on their own. In Section 3.1 we present six problems that are commonly used as the "known NP-complete problem" in proofs of NP-completeness, and we prove that