1 Reductions

One of the most important techniques in complexity theory is reductions. We have seen log-space reductions $\leq_l$ last time. Another natural one is called Turing reductions.

In fact, in Turing’s original paper [Tur37], he defined a notion called oracle TMs. We equip a TM $M$ with an oracle $O$, denoted $M^O$, if $M$ has a special tape, and during the execution of $M$, it can write $x$ on the special tape, and get the answer $O(x)$ in one step. Then we say $A$ is Turing reducible to $B$, denoted $A \leq_t B$, if there exists a TM $M$ with oracle $B$ that computes $A$. Again, the intuition of this notation is that $A$ is easier than $B$. The oracle should be thought of as some subroutine that we can invoke. This allows us to talk about relative hardness of problems.

For problems in $\text{NP}$, we will consider the following notion, called Karp reductions or polynomial time many-one reductions.

**Definition 1** (Karp reductions). A language $A$ is Karp reducible to another language $B$, denoted $A \leq_p B$, if there is a function $f : \Sigma^* \rightarrow \Sigma^*$ such that,

- $x \in A \Leftrightarrow f(x) \in B$;
- $f$ can be computed in polynomial time.

By Definition 1, if $A \leq_p B$ and $B \in \text{P}$, then $A \in \text{P}$. Also, if $A \leq_p B$ and $A \leq_p C$, then $A \leq_p C$.

We now proceed to define $\text{NP}$-completeness.

**Definition 2** ($\text{NP}$-completeness). A language $A$ is $\text{NP}$-hard (under Karp reductions) if for any $B \in \text{NP}$, $B \leq_p A$.

$A$ is $\text{NP}$-complete if $A \in \text{NP}$ and $A$ is $\text{NP}$-hard.

$\text{NP}$-complete problems are the most difficult ones among all problems in $\text{NP}$. By Definition 2, if there exists a $\text{NP}$-complete language $A \in \text{P}$, then $\text{P} = \text{NP}$!

Definition 2 sounds like very demanding, but $\text{NP}$-complete problems do exist. The canonical one is the following:
Name: \( L_{\text{NP}} \)

Input: A non-deterministic TM \( N \), an input \( x \), and a unary string \( 1^t \).

Output: Does \( N \) accept \( x \) within \( t \) steps?

**Proposition 1.** \( L_{\text{NP}} \) is \( \text{NP-complete} \).

*Proof.* It is easy to see that \( L_{\text{NP}} \in \text{NP} \). Given \( M \), \( x \) and \( 1^t \), we non-deterministically simulate \( M \) on \( x \) for \( t \) steps and output the same bit.

Next we show \( L_{\text{NP}} \) is \( \text{NP-hard} \). Let \( A \in \text{NP} \) be a language computed by a NTM \( N \) in time \( c_1 n^{c_2} \) for constants \( c_1 \) and \( c_2 \). Our reduction algorithm, given \( x \), simply outputs \( (N, x, 1^{c_1 n^{c_2}}) \). Then let us check Definition 1:

- \( x \in A \) if and only if \( (N, x, 1^{c_1 n^{c_2}}) \in L_{\text{NP}} \) by the definition of \( L_{\text{NP}} \);
- the reduction takes time \( O(n^{c_2}) \) (mostly to write down \( 1^{c_1 n^{c_2}} \)).

The \( 1^t \) part of the input is to make sure that \( L_{\text{NP}} \) is in \( \text{NP} \). If the input \( t \) is in binary, namely its size is \( \log t \), then the simulation would take some polynomial time in \( t \), which is exponential in \( \log t \), the input size.

It is somewhat straightforward to design complete languages like this for other complexity classes, such as \( \text{PSpace} \) or \( \text{L} \), as well.

## 2 Cook-Levin Theorem

\( L_{\text{NP}} \) is not a very useful \( \text{NP-complete} \) problem. The surprising discovery in the 70s, by Stephen Cook \([\text{Coo71}]\) and Leonid Levin \([\text{Lev73}]\), independently, is that the following natural problem is \( \text{NP-complete} \).

Name: \( \text{SAT} \)

Input: A CNF formula \( \varphi \)

Output: Is \( \varphi \) satisfiable?

Recall that a CNF (Conjunction Normal Form) formula is a conjunction of a number of disjunction clauses, like, \((x_1 \lor x_2) \land (\overline{x_1} \lor x_3 \lor x_4) \land \ldots \). To satisfy a CNF formula, we need to find an assignment so that all clauses are satisfied.

Given an assignment \( \sigma : X \rightarrow \{0, 1\} \), where \( X \) is the variable set, it is straightforward to check whether \( \sigma \) satisfies \( \varphi \). This means \( \text{SAT} \in \text{NP} \). (Recall the verification characterization of \( \text{NP} \) last time.)

**Theorem 2** (Cook-Levin). \( \text{SAT} \) is \( \text{NP-complete} \).
**Proof sketch.** The basic goal of the proof is that, given a polynomial time NTM $N$ and an input $x$, the computation of $N$ on $x$ can be encoded into a Boolean formula $\varphi$ so that $M$ accepts $x$ if and only if $\varphi$ is satisfiable. Additionally, the length of the formula is polynomial if the machine runs in polynomial time.

We may assume that $N$ is single-tape, since it can simulate $k$-tapes NTMs with at most quadratic slowdown. We may also assume that the tape is one-sided, since we can always “fold” the tape by enlarging the alphabet size. Moreover, we assume that $N$ always has 2 choices at every step. This is okay since we can always add $t - 2$ new states to mimic a $t$ choices non-deterministic step. If there is only one choice, then we consider the two coincide. Now the non-deterministic choices are simply a 0, 1-string: $c = c_0, c_1, \ldots, c_T$ where $T$ is the running time.

We form a $T$-by-$T$ “computational table” as follows. Rows are time indices, and each row is the encoding of the configuration at the corresponding time. So the $i$th row encodes the configuration at time $i$. If we fix the choices $c$, then the computation of $N$ on $x$ is completely deterministic and this table can be constructed. Equivalently, we may add an additional column of the table to reflect the choices $c$.

We introduce one variable $x$ for each cell of this conceptual table. Thus, we have $T^2$ variables. We then introduce one subformula for each two consecutive rows, and encode it to be true if and only if it is a valid transition in $N$ from the first row to the next. We also need to introduce clauses to validate the initial row, the final row is accepting, etc.

The crucial part of the construction of $\varphi$ is how to encode the transition function. It is possible only because the computation is local. Basically, to determine whether two such rows are “compatible”, we only need to look at constantly many cells. We know that any Boolean function can be encoded as a (possibly exponential size) CNF. The saving grace is that exponential of a constant is still a constant.

As of the total size of $\varphi$, notice that $T$ is a polynomial in $n$, and thus $T^2$ is still a polynomial. The number of clauses, as explained above, is also bounded by a polynomial. \(\square\)

Full proof details can be found in [AB09, Theorem 2.10] or [Pap94, Theorem 8.2], as well as many other books.

**Remark (Bibliographic).** Relevant chapters are [AB09, Chapter 2] and [Pap94, Chapter 8 and 9].

**References**


