# NP – Completeness

## 1. Satisfiable boolean formula

Stephen Cook and Leonid Levin discovered certain problems in NP whoose individual complexity is related to that of the entire class. If a polynomial time algorithm exists for any of these problems, all problems in NP would be polynomial time solvable. These problems are called NP-complete.

The first NP-complete problem that we present is called the satisfiability problem. We represent True by 1 and False by 0. The Boolean operations AND, OR, and NOT, represented by the symbols $\wedge$, $\vee$, and $\neg$ are described below. Notice that $\overline{x}$ means $\neg x$.

| | | |
|---|---|---|
| $0 \wedge 0 = 0$ | $0 \vee 0 = 0$ | $\overline{0} = 1$ |
| $0 \wedge 1 = 0$ | $0 \vee 1 = 1$ | $\overline{1} = 0$ |
| $1 \wedge 0 = 0$ | $1 \vee 0 = 1$ | |
| $1 \wedge 1 = 1$ | $1 \vee 1 = 1$ | |

A Boolean formula is an expression involving Boolean variables and operations. For example, $\phi = (\overline{x} \wedge y) \vee (x \wedge \overline{z})$. is a Boolean formula.

A Boolean formula is satisfiable if some assignment of 0s and 1s to the variables makes the formula evaluate to 1. The previous formula is satisfiable for $x=0$, $y=1$ and $z=0$.

The satisfiability problem is to test whether a Boolean formula is satisfiable.

Let $SAT = \{\langle \phi \rangle \mid \phi$ is a satisfiable Boolean formula$\}$.

Theorem: $\boxed{SAT \in P \text{ iff } P = NP}$

We state a theorem that links the complexity of the SAT problem to the complexities of all problems in NP.

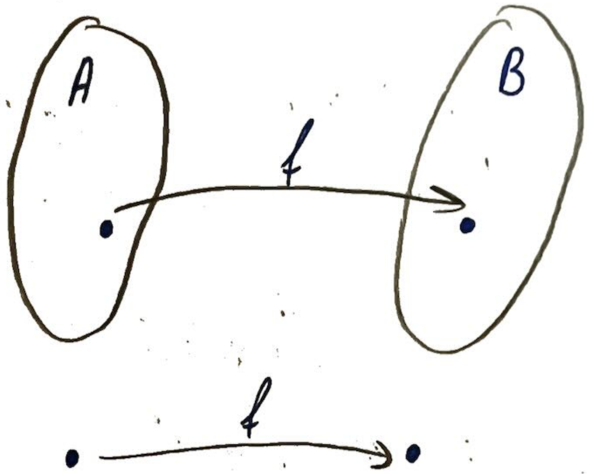
2. Polinomial time reducibility. CNF - formula

When problem A reduces to problem B, a solution to B can be used to solve A. When problem A is efficiently reducible to problem B, an efficient solution to B can be used to solve A efficiently.

Definition: A function $f : \Sigma^* \to \Sigma^*$ is a polynomial time computable function if some polynomial time Turing machine M exists that halts with just $f(w)$ on its tape, when started on any input $w$.

Definition: Language $A$ is polynomial time mapping reducible, or simply polynomial time reducible, to language $B$, written $A \leq_P B$, if a polynomial time computable function $f : \Sigma^* \longrightarrow \Sigma^*$ exists, where for every $w$, $$w \in A \iff f(w) \in B$$

The function $f$ is called the polynomial time reduction of $A$ to $B$.




Polynomial time function $f$ reducing $A$ to $B$

A polynomial time reduction of $A$ to $B$ provides a way to convert membership testing in $A$ to membershi testing in $B$ - but now the conversation is done efficiently. To test whether $w \in A$, we use the reduction $f$ to map $w$ to $f(w)$ and test whether $f(w) \in B$.

If one language is polynomial time reducible to a language already known to have a polynomial time solution, we obtain a polynomial time solution to the original language, as in the following theorem:

$$\boxed{\text{If } A \leq_P B \text{ and } B \in P \text{, then } A \in P}$$

Proof: Let $M$ be the polynomial time algorithm deciding $B$ and $f$ be the polynomial time reduction from $A$ to $B$. We describe a polynomial time algorithm $N$ deciding $A$ as follows.

$N =$ "On input $w$:

1. Compute $f(w)$
2. Run $M$ on input $f(w)$ and output whatever $M$ outputs."

We have $w \in A$ whenever $f(w) \in B$ because $f$ is a reduction from $A$ to $B$. Thus, $M$ accepts $f(w)$ whenever $w \in A$.

3SAT is a special case of the satisfiability problem whereby all formulas are in a special form. A literal is a Boolean variable or a negated Boolean variable, as in $x$ or $\overline{x}$. A clause is several literals connected with $\vee$s, as in $(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4)$. A Boolean formula is in conjunctive normal form, called a cnf-formula, if it comprises several clauses connected with $\wedge$s, as in

$$(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6})$$

It is a 3CNF-formula if all the clauses have three literals, as in $(x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6} \vee x_4)$
$\wedge (x_4 \vee x_5 \vee x_6)$

Let $3SAT = \{\langle \phi \rangle \mid \phi$ is a satisfiable 3CNF-formula$\}$

## 3. NP-complete. Cook-Levin theorem

A language B is NP-complete if it satisfies two conditions:

1. B is in NP
2. every A in NP is polynomial time reducible to B.

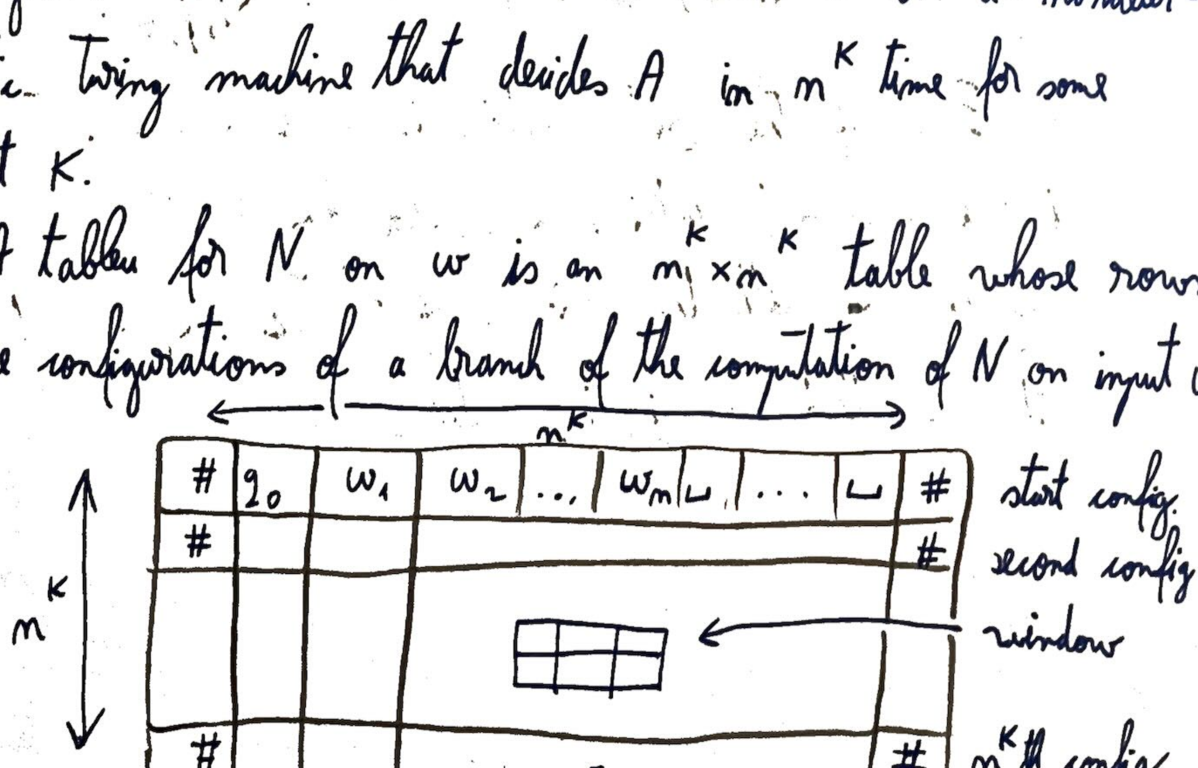
Theorem: $\boxed{\text{SAT is NP-complete}}$

Proof

First we show that SAT is in NP.

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.

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



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 tablеu for $N$ on $w$ corresponds to an accepting computation branch of $N$ on $w$. The problem of determining whether $N$ accepts $w$ is equivalent to the problem of determining whether an accepting tableu for $N$ on $w$ exists.

On input $w$, the reduction produces a formula $\phi$. We begin by describing the variables of $\phi$.

$Q$ is the state set
$\Gamma$ is the 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 is called a cell. The cell in row $i$ and column $j$ 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 assigment to the variables does correspond to an accepting tableu of $N$ on $w$.

$$\phi = \phi_{cell} \wedge \phi_{start} \wedge \phi_{move} \wedge \phi_{accept}$$

where :

- $\phi_{cell} = \bigwedge_{1 \leq i,j \leq n^k} \left[ \left( \bigvee_{s \in C} x_{i,j,s} \right) \wedge \left( \bigwedge_{\substack{s,t \in C \\ s \neq t}} \overline{x_{i,j,s}} \vee \overline{x_{i,j,t}} \right) \right]$

- $\phi_{start} = x_{1,1,\#} \wedge x_{1,2,q_0} \wedge x_{1,3,w_1} \wedge x_{1,4,w_2} \wedge \ldots$

  $\ldots \wedge x_{1,n+2,w_n} \wedge x_{1,n+3,\sqcup} \wedge \ldots \wedge x_{1,n^k-1,\sqcup} \wedge$

  $x_{1,n^k,\#}$

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

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

Next, we analyze the complexity of the reduction to show that it operates in polynomial time. We examine the size of $\phi$.

The tablen has $n^{2K}$ cells and each cell has $l$ variables where $l$ is the number of symols in $C$. So the total number of variables is $O(n^{2K})$.

We estimate the size of each of the parts of $\phi$. $\phi$ cell has the size of $O(n^{2K})$. Size of $\phi_{start}$ is $O(n^K)$. $\phi_{move}$ and $\phi_{accept}$ have the same size, $O(n^{2K})$. Thus, $\phi$'s total size is $O(n^{2K})$. That bound is sufficient and it shows that the size of $\phi$ is polynomial in $n$. Thus, we concluded that SAT is NP-complete.

4. 3SAT is NP - complete

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.

Formula $\phi_{cell}$ is a big AND of subformulas that contain a big OR and a big AND of ORs. 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 a CNF. Formula $\phi_{accept}$ is a big OR of variables and thus is a single clause. Formula $\phi_{move}$ is the only one that isn't a CNF, but we may easily convert it into a formula that is CNF.

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 $(a_1 \vee a_2 \vee a_3 \vee a_4)$ where $a_i$ is a literal, with the two-clause expression $(a_1 \vee a_2 \vee z) \wedge (\bar{z} \vee a_3 \vee a_4)$ where $z$ is a new variable.

In general, if the clause contains $l$ literals,

$$(a_1 \vee a_2 \vee a_3 \dots \vee a_l)$$

we can replace it with $l-2$ clauses

$$(a_1 \vee a_2 \vee z_1) \wedge (\bar{z}_1 \vee a_3 \vee z_2) \wedge (\bar{z}_2 \vee a_4 \vee z_3) \wedge \dots$$

$$\dots \wedge (\overline{z_{l-3}} \vee a_{l-1} \vee a_l).$$