# Chapter 22

# NP Completeness and Cook-Levin Theorem

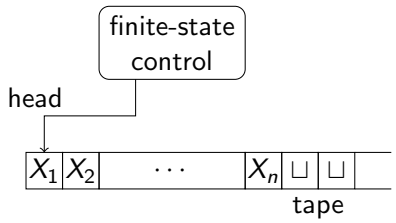
**CS 473: Fundamental Algorithms, Spring 2011**
April 19, 2011

## 22.1 P, NP and other complexity classes

(A) **P**: set of decision problems that have polynomial time algorithms
(B) **NP**: set of decision problems that have polynomial time non-deterministic algorithms

The first natural question is what an algorithm? Informally, one can think about an algorithm as a mechanism that gets an input, applies to it a long sequence of operations, and then stops and outputs. This is still too imprecise. Indeed, the answer largely depends on our computation model (i.e., what operations are allowed, etc). In particular, if we want to argue formally about algorithms we need a precise definition of an algorithm (and a computer).

Formally speaking our model of computation is going to be a ***Turing Machines***. Here is a quick reminder of Turing Machines. It has:
(A) Infinite tape
(B) Finite state control
(C) Input at beginning of tape
(D) Special tape letter "blank" $\sqcup$
(E) Head can move only one cell to left or right




**Definition 22.1.1** *A Turing Machine (TM) is 7-tuple* $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, *where*
*(A)* $Q$ *is set of states in finite control*
*(B)* $q_0$ *start state,* $q_{accept}$ *is accept state,* $q_{reject}$ *is reject state*
*(C)* $\Sigma$ *is input alphabet,* $\Gamma$ *is tape alphabet (includes* $\sqcup$*)*

*(D)* $\delta : Q \times \Gamma \to \{L, R\} \times \Gamma \times Q$ *is transition function.*
*Here* $\delta(q, a) = (q', b, L)$ *means that* $M$ *in state* $q$ *and head seeing* $a$ *on tape will move to state* $q'$ *while replacing* $a$ *on tape with* $b$ *and head moves left.*

**Definition 22.1.2** *The* **language** $L(M)$ *accepted by a Turing Machine* $M$ *is set of all input strings* $s$ *on which* $M$ *accepts; that is:*
*(A)* TM *is started in state* $q_0$.
*(B) Initially, the tape head is located at the first cell.*
*(C) The tape contain* $s$ *on the tape followed by blanks.*
*(D) The* TM *halts in the state* $q_{accept}$.

**Definition 22.1.3** *A* TM $M$ *is a* **polynomial time** TM *if there is some polynomial* $p(\cdot)$ *such that on all inputs* $w$, $M$ *halts in* $p(|w|)$ *steps.*

**Definition 22.1.4** *A language* $L$ *in* **P** *if there is a polynomial time* TM $M$ *such that* $L = L(M)$. *That is,* $L$ *is solvable* **polynomial time**.

**Definition 22.1.5** *A language* $L$ *is an* **NP language** *if and only if there is a* non-deterministic *polynomial time* TM $M$ *such that* $L = L(M)$.

A non-deterministic TM at each step has a choice of moves. Formally, the transition function maps the current state to a set of states

$$\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\}).$$

At every step, the TM moves to one of these possible moves. For example, $\delta(q, a) = \{(q_1, b, L), (q_2, c, R), (q_3, a, R)\}$ means that $M$ can non-deterministically choose one of the three possible moves from $(q, a)$.

**Definition 22.1.6** *The language of a non-deterministic* TM $M$, *denoted by* $L(M)$, *is the set of all strings* $s$ *on which there* **exists** *some sequence of valid choices at each step that lead from* $q_0$ *to* $q_{accept}$.

There are two definition of **NP**:
(A) $L$ is in **NP** if and only if $L$ has a polynomial time certifier $C(\cdot, \cdot)$.
(B) $L$ is in **NP** if and only if $L$ is decided by a non-deterministic polynomial time TM $M$.

**Claim 22.1.7** *The two definitions above of* **NP** *are equivalent.*

Informal proof:
(A) The certificate $t$ for $C$ corresponds to non-deterministic choices of $M$ and vice-versa. In other words $L$ is in **NP** if and only if $L$ is accepted by a NTM which first guesses a proof $t$ of length poly in input $|s|$ and then acts as a *deterministic* TM.
(B) A non-deterministic machine has choices at each step and accepts a string if there *exists* a set of choices which lead to a final state.

(C) Equivalently the choices can be thought of as *guessing* a solution and then *verifying* that solution. In this view all the choices are made a priori and hence the verification can be deterministic. The "guess" is the "proof" and the "verifier" is the "certifier".
(D) We reemphasize the asymmetry inherent in the definition of non-determinism. Strings in the language can be easily verified. No easy way to verify that a string is not in the language.

**Algorithms: TMs vs RAM Model.** Why do we use TMs some times and RAM Model other times?
(A) TMs are very simple: no complicated instruction set, no jumps/pointers, no explicit loops etc.
  (i) Simplicity is useful in proofs.
  (ii) The "right" formal bare-bones model when dealing with subtleties.
(B) RAM model is a closer approximation to the running time/space usage of realistic computers for reasonable problem sizes
  (i) Not appropriate for certain kinds of formal proofs when algorithms can take super-polynomial time and space.

## 22.2 Cook-Levin Theorem

### 22.2.1 Completeness and hardness

**Question 22.2.1** *What is the* **hardest** *problem in* **NP***? How do we define it?*

**Towards a definition**:
(A) Hardest problem must be in **NP**.
(B) Hardest problem must be at least as "difficult" as every other problem in **NP**.

**Definition 22.2.2** *A problem $X$ is said to be* **NP-Complete** *if*
*(A)* $X \in \mathbf{NP}$.
*(B) (Hardness) For any* $Y \in \mathbf{NP}$, $\mathbf{Y} \leq_P \mathbf{X}$.

The key observation is that **NP** problems all become easy if one of them is solvable in polynomial time, as testified by the following.

**Proposition 22.2.3** *Suppose $X$ is* **NP-Complete**. *Then $X$ can be solved in polynomial time if and only if* $\mathbf{P} = \mathbf{NP}$.

*Proof*: $\Rightarrow$ Suppose $X$ can be solved in polynomial time. Then, for any $Y \in \mathbf{NP}$, we know that $\mathbf{Y} \leq_P \mathbf{X}$. However, we showed that if $\mathbf{Y} \leq_P \mathbf{X}$ and $X$ can be solved in polynomial time, then $Y$ can be solved in polynomial time. Thus, every problem $Y \in \mathbf{NP}$, it holds that $Y \in P$. This implies that $NP \subseteq P$. Now, since $\mathbf{P} \subseteq \mathbf{NP}$, we have $\mathbf{P} = \mathbf{NP}$.
$\Leftarrow$ Since $\mathbf{P} = \mathbf{NP}$, and $X \in \mathbf{NP}$, we have a polynomial time algorithm for $X$. ■

**Definition 22.2.4** *A problem $X$ is said to be* **NP-Hard** *if For any $Y \in$ NP, it holds that* $Y \leq_P X$.

*Namely, any problem in* **NP** *can be reduced in polynomial time to $X$. A somewhat more general (and intuitive) definition is that a problem is* **NP-Hard** *if solving it in polynomial time leads to polynomial time algorithm for all the problems in* **NP**.

(Another way to think about it: You are given a black-box for $X$ that works in polynomial time, and you can solve some **NP-Complete** problem in polynomial time using this black box.)

An **NP-Hard** problem need not be in **NP**! As a concrete example, it is not hard to see that the **Halting** problem (which is not even decidable!) is **NP-Hard** (why?) but not **NP-Complete**.

**Remark 22.2.5** The **Halting** problem is **NP-Hard** because of the following. We are given a program $X \in$ **NP**, and input $I$ for it. As we already discussed, any **NP** problem can be solved in exponential time, so consider an exponential time algorithm $M$ that solves $X$ in exponential time. Now. if $M$ reject its input, we modify it so that it goes into an infinite loop. Clearly, deciding if the modified TM $M'$ halts on $I$, is equivalent to deciding if $I \in L(M)$. Furthermore, this reduction is polynomial (the modification of $M$ to get $M'$ is quite minor).

As such, this reduction is polynomial time. Furthermore, if one can solve the **Halting** problem in polynomial time (ridiculous assumption, but that is irrelevant) then one could solve $X$ in polynomial time.

### 22.2.2 NP-Complete problems

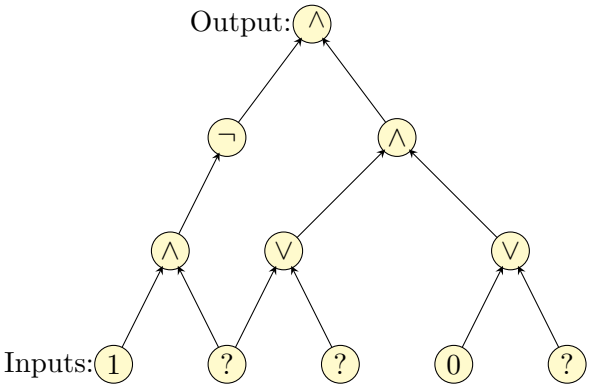
If we know that $X$ is **NP-Complete**, then

(i) since we believe **P** $\neq$ **NP**, and
(ii) solving $X$ implies **P** $=$ **NP**,

this implies that $X$ is *unlikely* to be efficiently solvable.

At the very least, many smart people before you have failed to find an efficient algorithm for $X$. (I.e., if you find a polynomial time algorithm for an **NP-Complete** problem, you should be *extremely* suspicious that something wrong is going on.)

Of course, it is not a priori clear that there any problem is **NP-Complete**; indeed, the requirement that we can reduce any problem in **NP** to it seems a very strong requirement. Surprisingly, there are many, many problems are known to be **NP-Complete**.



**Definition 22.2.6** *A* **circuit** *is a* DAG *(directed acyclic graph) with:*
*(A)* **Input** *vertices (without incoming edges) labeled with* 0, 1 *or a distinct variable*
*(B) Every other vertex is labeled* $\vee$, $\wedge$ *or* $\neg$
*(C) Single node* **output** *vertex with no outgoing edges*
*Such a circuit is depicted on the right.*

**Definition 22.2.7** **CSAT** *(Circuit Satisfaction): Given a circuit as input, is there an assignment to the input variables that causes the output to get value* 1*?*

**Theorem 22.2.8 (Cook-Levin)** **CSAT** *is* **NP-Complete**.

To prove this theorem, we need to show that
(A) **CSAT** is in **NP**.
(B) *every* **NP** problem $X$ reduces to Circuit Satisfaction.

Showing that **CSAT** is in **NP** is quite easy. Indeed, the ***certificate*** proving that a given instance is satisfiable, is an assignment to the input variables for which the given circuit outputs 1. Clearly, the certificate has polynomial size in the size of the given circuit.

The ***certifier*** in this case, evaluate the value of each gate in a topological sort of the DAG and check the output gate value is indeed 1 as required. Clearly, this can be done in polynomial time in the size of the input.

**Idea of proof.** We need to show that *every* **NP** problem $X$ reduces to **CSAT**. Since $X \in$ **NP**, we have that there are polynomials $p(\cdot)$ and $q(\cdot)$ and certifier/verifier program $C$ such that for every string $s$ the following is true:
(A) If $s$ is a YES instance ($s \in X$) then there is a *proof* $t$ of length $p(|s|)$ such that $C(s,t)$ says YES.
(B) If $s$ is a NO instance ($s \notin X$) then for every string $t$ of length at $p(|s|)$, $C(s,t)$ says NO.
(C) $C(s,t)$ runs in time $q(|s| + |t|)$ time (hence polynomial time).

In particular, if $X$ is in **NP**, then we are given $p(\cdot), q(\cdot)$, and the certifier $C(\cdot,\cdot)$. The certifier $C(\cdot,\cdot)$ is a program (or equivalently a Turing Machine).

The polynomials $p(\cdot)$ and $q(\cdot)$ are given as numbers. For example, if 3 is given then $p(n) = n^3$.

Thus an **NP** problem is essentially a three tuple $\langle p, q, C\rangle$ where $C$ is either a program or a TM.

In particular, problem $X$ asks whether given a string $s$, is $s \in L(X)$? This is the same as asking if there is a proof $t$ of length $p(|s|)$ such that $C(s,t)$ says YES.

How do we reduce $X$ to **CSAT**? Need an algorithm $\mathcal{A}$ that
(A) takes $s$ (and $\langle p, q, C\rangle$) and creates a circuit $G$ in polynomial time in $|s|$ (note that $\langle p, q, C\rangle$ are fixed).
(B) $G$ is satisfiable if and only if there is a proof $t$ such that $C(s,t)$ says YES

**Simple but Deep Insight:** Programs are essentially the same as Circuits!

(A) Convert $C(s,t)$ into a circuit $G$ with $t$ as unknown inputs (rest is known including $s$)

(B) We know that $|t| = p(|s|)$ so express boolean string $t$ as $p(|s|)$ variables $t_1, t_2, \ldots, t_k$ where $k = p(|s|)$.

(C) Asking if there is a proof $t$ that makes $C(s,t)$ say YES is same as whether there is an assignment of values to "unknown" variables $t_1, t_2, \ldots, t_k$ that will make $G$ evaluate to true/YES.

#### 22.2.2.1 Example: Reducing Independent Set to a circuit

In the **Independent Set** problem, given a graph $G = (V, E)$ and an integer $k$, one needs to decide if $G = (V, E)$ have an **Independent Set** of size $\geq k$. In this case, the ***certificate*** is a subset $S \subseteq V$. The ***certifier*** verifies that $|S| \geq k$ and no pair of vertices in $S$ is connected by an edge in $G$.

Even more formally, we have

(A) The input for **Independent Set** is

$$\langle n, y_{1,1}, y_{1,2}, \ldots, y_{1,n}, y_{2,1}, \ldots, y_{2,n}, \ldots, y_{n,1}, \ldots, y_{n,n}, k \rangle \quad \text{encoding } \langle G, k \rangle .$$

- (A) $n$ is number of vertices in $G$, encoded say in unary (i.e., it is a sequence of 1s followed by a zero).
- (B) $y_{i,j}$ is a bit which is 1 if edge $(i,j)$ is in $G$ and 0 otherwise (adjacency matrix representation)
- (C) $k$ is the size of the required independent set (also, say, encoded in unary).

(B) The certificate is a sequence of $n$ bits: $t = t_1 t_2 \ldots t_n$.
Here $t_i$ is 1 if vertex $i$ is in the independent set, 0 otherwise.

The certifier $C(s,t)$ for **Independent Set** using this input is now pretty simple:

```
if (t_1 + t_2 + ... + t_n < k) then
    return NO
else
    for each (i,j) do
        if (t_i ∧ t_j ∧ y_{i,j}) then
            return NO

return YES
```

It is now straightforward to interpret this certifier as circuit. An example of that is show in Figure 22.1.

#### 22.2.2.2 Back to the reduction of an NP problem to CSAT

Consider a "program" $A$ that takes $f(|s|)$ steps on input string $s$.

**Question:** What computer is the program running on and what does a *step* mean?

Real computers difficult to reason with mathematically because

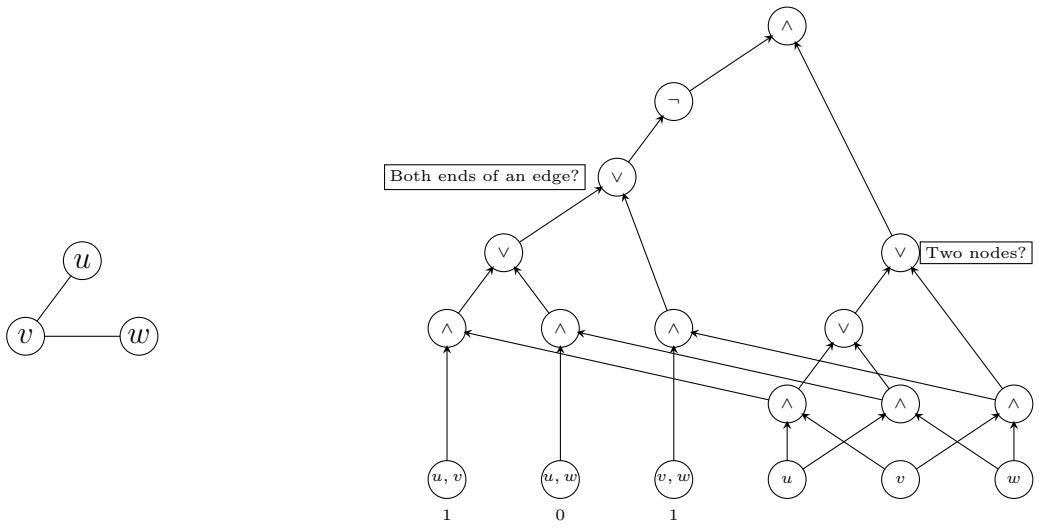



Figure 22.1: A graph and its certifier for **Independent Set**.

(i) instruction set is too rich,
(ii) pointers and control flow jumps in one step,
(iii) assumption that pointer to code fits in one word.

Turing Machines on the other hand are

(i) simpler model of computation to reason with,
(ii) can simulate real computers with *polynomial* slow down
(iii) all moves are *local* (head moves only one cell).

Assume $C(\cdot, \cdot)$ is a (deterministic) Turing Machine $M$.

Problem: Given $M$, input $s$, $p$, $q$ decide if there is a proof $t$ of length $p(|s|)$ such that $M$ on $s, t$ will halt in $q(|s|)$ time and say YES.

There is an algorithm $\mathcal{A}$ that can reduce above problem to **CSAT** mechanically as follows.

(A) $\mathcal{A}$ first computes $p(|s|)$ and $q(|s|)$.
(B) Knows that $M$ can use at most $q(|s|)$ memory/tape cells
(C) Knows that $M$ can run for at most $q(|s|)$ time
(D) Simulates the evolution of the state of $M$ and memory over time using a big circuit.

The idea is now to simulate the computation of the certifier via a circuit:

(A) Think of $M$'s state at time $\ell$ as a string $x^\ell = x_1 x_2 \ldots x_k$ where each $x_i \in \{0, 1, B\} \times Q \cup \{q_{-1}\}$.
(B) At time 0 the state of $M$ consists of input string $s$ a guess $t$ (unknown variables) of length $p(|s|)$ and rest $q(|s|)$ blank symbols.
(C) At time $q(|s|)$ we wish to know if $M$ stops in $q_{accept}$ with say all blanks on the tape.
(D) We write a circuit $C_\ell$ which captures the transition of $M$ from time $\ell$ to time $\ell + 1$.
(E) Composition of the circuits for all times 0 to $q(|s|)$ gives a big (still poly) sized circuit $\mathcal{C}$

(F) The final output of $\mathcal{C}$ should be true if and only if the entire state of $M$ at the end leads to an accept state.

The key Ideas in the reduction of an NP problem $X$ to **CSAT** are:
(A) Use TMs as the code for certifier for simplicity
(B) Since $p()$ and $q()$ are known to $\mathcal{A}$, it can set up all required memory and time steps in advance
(C) Simulate computation of the TM from one time to the next as a circuit that only looks at three adjacent cells at a time

Note: Above reduction can be done to **SAT** as well. Reduction to **SAT** was the original proof of Steve Cook.

## 22.3 Other NP-Complete Problems

### 22.3.1 SAT

**SAT** is **NP-Complete**:
(A) We have seen that **SAT** $\in$ **NP**
(B) To show **NP-Hardness**, we will reduce Circuit Satisfiability (**CSAT**) to **SAT**

So, we are given an instance of **CSAT** (i.e., a boolean circuit $C$) and we need to convert it into an instance $I$ of **SAT** (i.e., a CNF formula).

**Reduction**:
(A) For each gate (vertex) $v$ in the circuit $C$, create a variable $x_v$.
(B) Write down a sub-formula that is correct if and only if the gate is being computed correctly. The conjunction of all these sub-formulas is true if and only if the original circuit is satisfiable.
(C) Convert every sub-formula into an equivalent CNF sub-formula.
(D) Let $I$ be the conjunction of all these CNF sub-formulas.

**Reduction in more detail**:
(A) For each gate (vertex) $v$ in the circuit $C$, create a variable $x_v$.
(B) Write down a sub-formula that is correct if and only if the gate is being computed correctly.
(C) Convert these sub-formula into CNF sub-formula. If the gate is
  (I) Negation gate ($\neg$): $v$ is labeled $\neg$ and has one incoming edge from $u$ (so $x_v = \neg x_u$). Create the CNF sub-formula $(x_u \vee x_v) \wedge (\neg x_u \vee \neg x_v)$.
  It is easy to verify that

$$x_v = \neg x_u \text{ is true} \iff \begin{matrix}(x_u \vee x_v)\\ (\neg x_u \vee \neg x_v)\end{matrix} \text{ are both true.}$$

  (II) Or gate ($\vee$): So $x_v = x_u \vee x_w$. Create the CNF sub-formula $(x_v \vee \neg x_u) \wedge (x_v \vee$

$\neg x_w) \wedge (\neg x_v \vee x_u \vee x_w)$. It is easy to verify that

$$x_v = x_u \vee x_w \text{ is true} \iff \begin{array}{l}(x_v \vee \neg x_u),\\ (x_v \vee \neg x_w),\\ (\neg x_v \vee x_u \vee x_w)\end{array} \quad \text{are all true.}$$

(III) And gate ($\wedge$): So $x_v = x_u \wedge x_w$. Create the CNF sub-formula $(\neg x_v \vee x_u) \wedge (\neg x_v \vee x_w) \wedge (x_v \vee \neg x_u \vee \neg x_w)$. Again. one need to verify that

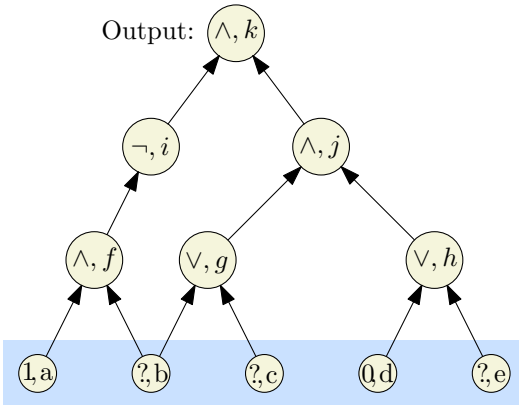
$$x_v = x_u \wedge x_w \text{ is true} \iff \begin{array}{l}(\neg x_v \vee x_u),\\ (\neg x_v \vee x_w),\\ (x_v \vee \neg x_u \vee \neg x_w)\end{array} \quad \text{all true.}$$

(IV) Input gate with a fixed value: If $v$ is an input gate with a fixed value. If $x_v = 1$ then we create the CNF sub-formula $x_v$. If $x_v = 0$ then we we create the CNF sub-formula $\neg x_v$.

(D) Add the clause $x_v$ where $v$ is the variable for the output gate, as a sub-formula by itself. This CNF snippet will be satisfied if and only if the assignment satisfies the original circuit.

(E) Take the conjunction of all these CNF sub-formulas to get a large CNF formula $I$ that is satisfiable if and only if the original circuit is satisfiable.

**Example.** The idea is to introduce variables $x_a, x_b, \ldots x_k$, one for each gate in the circuit. Relationship between variables given by gates and their input/outputs. For example:




Sub-formulas from circuit on the left.

$$\begin{array}{lll} x_a = 1 & x_d = 0 & \\ x_f = x_a \wedge x_b & x_g = x_b \vee x_c & x_h = x_d \vee x_e \\ x_i = \neg x_f & x_j = x_g \wedge x_h & \\ x_k = x_i \wedge x_j & & \end{array}$$

We now convert the above formula, by expressing equalities using CNF clauses.

$$\begin{array}{lll} x_a & \neg x_d & \\ (\neg x_f \vee x_a) & (\neg x_f \vee x_b) & (x_f \vee \neg x_a \vee \neg x_b) \\ (x_g \vee \neg x_b) & (x_g \vee \neg x_c) & (\neg x_g \vee x_b \vee x_c) \\ (x_h \vee \neg x_d) & (x_h \vee \neg x_e) & (\neg x_h \vee x_d \vee x_e) \\ (x_i \vee x_f) & (\neg x_i \vee x_f) & \\ (\neg x_j \vee x_g) & (\neg x_j \vee x_h) & (x_j \vee \neg x_g \vee \neg x_h) \\ (\neg x_k \vee x_i) & (\neg x_k \vee x_j) & (x_k \vee \neg x_i \vee \neg x_j) \end{array}$$

Final **SAT** formula $\varphi_C$: conjunction of all of above clauses and the clause $x_k$, the output of the final gate. The resulting CNF formula in this case is

$$\begin{aligned} I \equiv \; & x_a \bigwedge \neg x_d \bigwedge (\neg x_f \vee x_a) \bigwedge (\neg x_f \vee x_b) \bigwedge (x_f \vee \neg x_a \vee \neg x_b) \bigwedge (x_g \vee \neg x_b) \bigwedge (x_g \vee \neg x_c) \\ & \bigwedge (\neg x_g \vee x_b \vee x_c) \bigwedge (x_h \vee \neg x_d) \bigwedge (x_h \vee \neg x_e) \bigwedge (\neg x_h \vee x_d \vee x_e) \bigwedge (x_i \vee x_f) \\ & \bigwedge (\neg x_i \vee x_f) \bigwedge (\neg x_j \vee x_g) \bigwedge (\neg x_j \vee x_h) \bigwedge (x_j \vee \neg x_g \vee \neg x_h) \bigwedge (\neg x_k \vee x_i) \\ & \bigwedge (\neg x_k \vee x_j) \bigwedge (x_k \vee \neg x_i \vee \neg x_j) \\ & \bigwedge x_k. \end{aligned}$$

**Correctness of Reduction.** Need to show circuit $C$ is satisfiable if and only if $\varphi_C$ is satisfiable:

$\Rightarrow$ Consider a satisfying assignment $a$ for $C$
- (A) Find values of all gates in $C$ under $a$
- (B) Give value of gate $v$ to variable $x_v$; call this assignment $a'$
- (C) $a'$ satisfies $\varphi_C$ (exercise)

$\Leftarrow$ Consider a satisfying assignment $a$ for $\varphi_C$
- (A) Let $a'$ be the restriction of $a$ to only the input variables
- (B) Value of gate $v$ under $a'$ is the same as value of $x_v$ in $a$
- (C) Thus, $a'$ satisfies $C$

### 22.3.2 How to show that a problem $X$ is NP-Complete?

To prove $X$ is **NP-Complete**, show

(A) Show $X$ is in **NP**. That is show that there is a
- (A) certificate/proof of polynomial size in input.
- (B) And there is a polynomial time certifier $C(s, t)$.

(B) Reduction from a known **NP-Complete** problem such as **CSAT** or **SAT** to $X$.

SAT $\leq_P$ X implies that every **NP** problem $Y \leq_P X$. Why?

Transitivity of reductions:

$Y \leq_P$ **SAT** and **SAT** $\leq_P X$ and hence $Y \leq_P X$.

### 22.3.3 Problems we now know are NP-Complete

(A) **CSAT** is **NP-Complete**
(B) **CSAT** $\leq_P$ **SAT** and **SAT** is in **NP** and hence **SAT** is **NP-Complete**
(C) **SAT** $\leq_P$ **3SAT** and hence **3SAT** is **NP-Complete**
(D) **3SAT** $\leq_P$ Independent Set (which is in **NP**) and hence Independent Set is **NP-Complete**
(E) Vertex Cover is **NP-Complete**
(F) Clique is **NP-Complete**

Hundreds and thousands of different problems from many areas of science and engineering have been shown to be **NP-Complete**.

A surprisingly frequent phenomenon!

## 22.4 A sketch of the proof that CSAT is NP-Complete

We are given:

(A) a problem $X$ in **NP**,

(B) a polynomial time certifier for $X$ (given as a non-deterministic TM $M$),

(C) and an input $I$ of size $n$.

The next step is modify $M$ such that it stores as much information on its state on the tape. We assume the following:

(A) The TM $M$ writes the state (of the control) that it is in on the tape. This can be done by extending the alphabet of the TM $M$ such that the new alphabet is $\Sigma \times Q$ (i.e., every cell in the tape stores a pair of symbols - since the alphabet is still finite, this is completely legal).

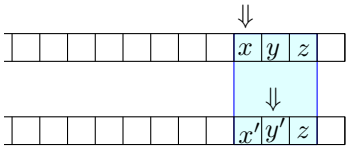
(B) Similarly, we store for every cell on the tape of the TM a bit that tells us whether the head of the TM is ta this location.

(C) As such, the content of a tape cell can be written as a sequence of $k$ bits. Here $k$ is a some fixed small constant. Formally $k = \lceil \lg |\Sigma| + \lg |Q| + 2 \rceil$.

(Our description here of the TM is slightly non-standard: As the head moves to the new location, it is immediately write the state of the control into the tape in the new location, and only then ends this step. It is easy to verify that this can be simulation by a regular TM- it is intended only to make our proof easier.)

Now a ***configuration*** of the TM is fully defined by the content of its tape. Given such a tape of length $m$, it is easy to write a formula of length $O(m^2)$ that verifies that exactly one location on the tape has the active head indicator on.

Now, consider two consecutive configurations of the Turing machine, both of them using only the first $m$ locations on the tape. Clearly, the two tapes are identical except maybe for two locations (the old location, where we wrote a new content, and the new location we wrote the control state the TM is in, and the bit indicating that the head of the TM is in this location):




In particular, the content of every cell of the new configuration is a function of three cells in the old configuration (i.e., cell before it, above it, and to its right). Of course, of almost all cells it is just a copy of the old content. So, consider such a cell on the new configuration. We can write down a (potentially long) boolean formula that would be one if and only if the cell content is computed correctly from the previous configuration. (Note, that since the TM is non-deterministic there might be several values that might be legal to be stored in

this cell.) This formula might be quite big (as a function of $k$, the TM control size, etc), but it is a constant per cell. In particular, verifying that indeed given two configurations $C_1$ and $C_2$ that $C_2$ is indeed the configuration following $C_1$ can be done by a circuit of size $m \cdot O(1) = O(m)$.

Let us go back to our problem. We are given a certifier $M$ and an input $I$ that is written in the beginning of the tape. Let $C_0$ denote this initial configuration of the tape. We know that since $M$ is a polynomial time certifier, it is going to use at most $m = p(n)$ cells of the tape. And the sequence of configurations it is going to go through is polynomially large. Say it is going to make at most $t = q(n)$ steps. Here both $p$ and $q$ are polynomials that are associated with the given problem $X$.

If $M$ accepts $I$, then there must be a sequence of configurations $C_0 \implies C_1 \implies C_2 \implies \ldots \implies C_t$, such that:

(A) The configuration $C_{i+1}$ is one possible configuration the TM might be in after performing exactly one step from the configuration $C_i$, for all i.

(B) $C_t$ is an accepting configuration.

Except for $C_0$, which is known to us, all the other configurations are unknown to us. So, for every configuration, we encode it as a collection of variables (every cell will use $k$ new variables) and every configuration would be encoded by $O(km)$ boolean variables. Now, let us write a boolean formula $F_i$ that is correct if and only if $C_i \implies C_{i+1}$ is a valid transition, for all $i$. By the above, for every step, such a formula can be constructed in polynomial time, and it is of size $O(m)$. The formula verifying all the transitions (i.e., $F_1 \wedge F_2 \wedge \ldots \wedge F_t$) as such is of size $O(mt)$ (which is polynomial in the input size).

We can similarly write a formula that verifies that $C_0$ is valid (i.e., it indeed have $I$ in the first $n$ cells, the head is at the first cell, and the initial state is stored at this cell).

Putting these formulas together, we get a huge formula $F$ that is satisfiable if and only if there is a valid execution of $M$ such that it accepts $I$. Furthermore, this formula can be computed in polynomial time, and it has a polynomial size.

While we described in the above the construction in terms of formulas, the same description works verbatim if we construct a circuit instead – the two are the same thing.

Thus, if this formula (or if you want, the equivalent circuit) is satisfiable then the answer of $X$ for the input $I$ is yes. Putting it differently, given a polynomial time solver to **CSAT** we can solve $X$ in polynomial time.

## 22.5 Bibliographical notes

The classical text on **NP-Complete**ness is the work by Garey and Johnson [Garey and Johnson, 1990]. An accessible description of the Cook-Levin theorem (with more details than we provided) is given by Sipser [Sipser, 2005].