Polynomial Church-Turing thesis

A decision problem can be solved in polynomial time by using a reasonable sequential model of computation if and only if it can be solved in polynomial time by a Turing machine.

P is the class of these decision problems

Search Problems: **NP**

L is in **NP** iff there is a language *L*' in **P** and a polynomial *p* so that:

$$\forall x : \quad x \in L \Leftrightarrow [\exists y \in \{0,1\}^* : |y| \leq p(|x|) \wedge \langle x, y \rangle \in L']$$

Intuition

$$\forall x : \quad x \in L \Leftrightarrow [\exists y \in \{0,1\}^* : |y| \leq p(|x|) \wedge \langle x, y \rangle \in L']$$

- The *y*-strings are the ***possible solutions*** to the instance *x*.
- We require that solutions are not too long and that it can be checked efficiently if a given *y* is indeed a solution or ***witness*** (we have a “simple” search problem)

Reductions

- A ***reduction*** *r* of L_1 to L_2 is a polynomial time computable map so that

$$\forall x: \quad x \in L_1 \text{ iff } r(x) \in L_2$$

- We write $L_1 \leq L_2$ if L_1 reduces to L_2.

- **Intuition:** Efficient software for L_2 can also be used to efficiently solve L_1.

NP-hardness

- A language L is called **NP**-hard iff

  $\forall L' \in \mathbf{NP}: \quad L' \leq L$

- **Intuition:** Software for L is strong enough to be used to solve any simple search problem.

- **Proposition**: If some **NP**-hard language is in **P**, then **P**=**NP**.

NPC

- A language L $\in$ **NP** that is **NP**-hard is called **NP**-complete.

- **NPC** := the class of **NP**-complete problems.

- **Proposition**:

  $L \in$ **NPC** $\Rightarrow$ [$L \in$ **P** iff **P**=**NP**].

- *L* is in **NP** means:

  There is a language *L'* in **P** and a polynomial *p* so that

$$\forall x: \quad x \in L \Leftrightarrow [\exists y \in \{0,1\}^* : |y| \leq p(|x|) \wedge \langle x, y \rangle \in L']$$

- $L_1 \leq L_2$ means:

  For some polynomial time computable map *r* :

  $\forall x: \quad x \in L_1$ iff $r(x) \in L_2$

- *L* is **NP**-hard means:

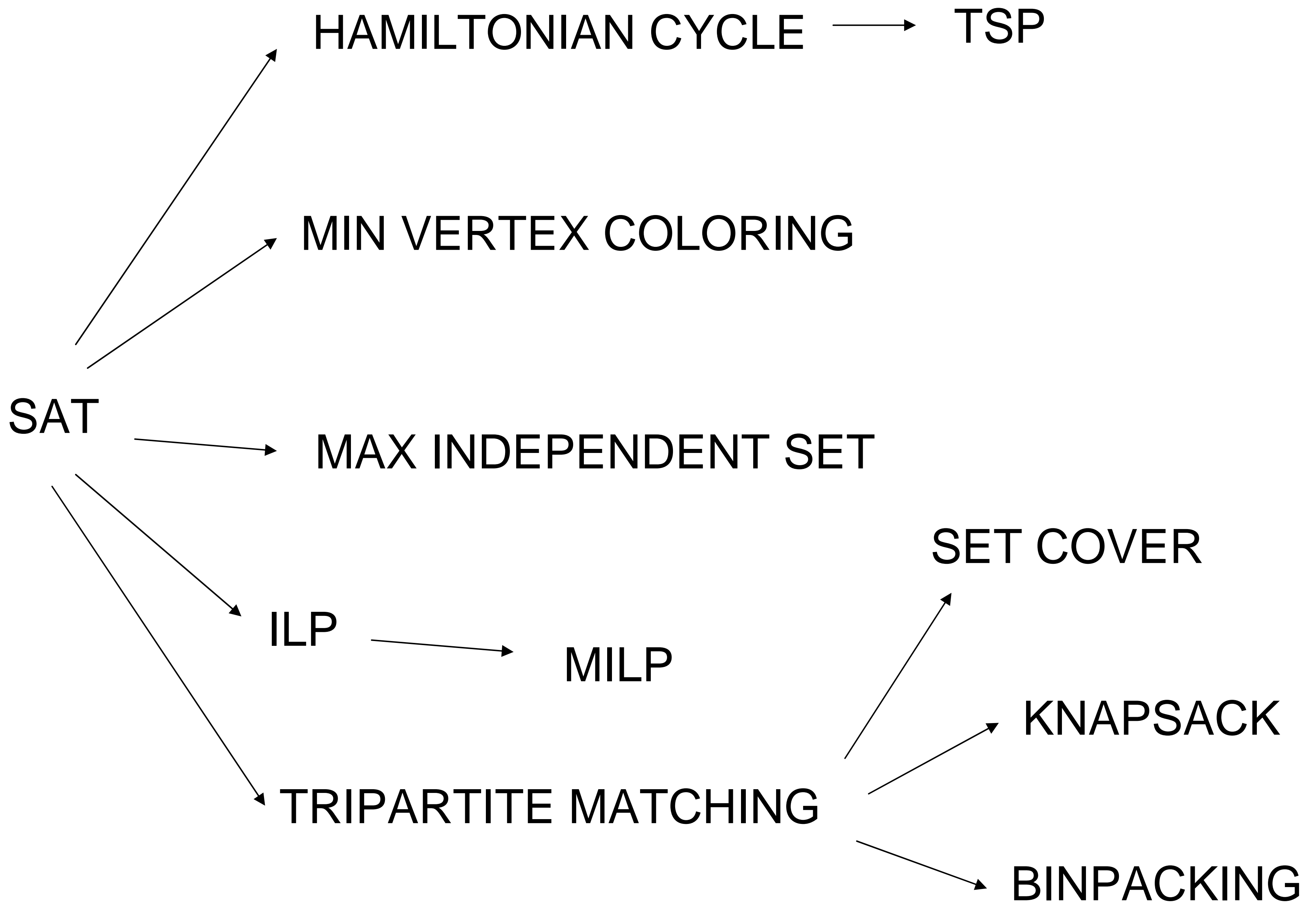
  $\forall L' \in$ **NP**: $L' \leq L$

- *L* is in **NPC** means:

  $L \in$ **NP** and *L* is **NP**-hard

How to establish **NP**-hardness

- Thousands of natural problems are **NP**-complete:
- **Empirical fact**: Most natural problems in **NP** are either in **P** or **NP**-hard.
- **Lemma**: If L_1 is **NP**-hard and $L_1 \leq L_2$ then L_2 is **NP**-hard.
- We need to establish ***one*** problem to be **NP**-hard, the rest follows using chains of reductions. Cook (1972) established SAT to be **NP**-hard.


HAMILTONIAN CYCLE
TSP
MIN VERTEX COLORING
SAT
MAX INDEPENDENT SET
SET COVER
ILP
MILP
KNAPSACK
TRIPARTITE MATCHING
BINPACKING

Boolean functions

- f: {false, true}^{n} $\rightarrow$ {false,true}
- Example: XOR(x_1, x_2) = $x_1 \oplus x_2$
- XOR(true , true) = false
- In this course 0=false, 1=true.
- XOR(1,1)=0

How to represent Boolean functions on a computer

- Tables.
- Formulae.

Tables

A	B	Z
0	0	0
0	1	1
1	0	1
1	1	0

- More compact representation as Boolean String: “0110”.
- A function f: $\{0,1\}^n \rightarrow \{0,1\}$ can be represented as a table using …

……..2^n bits

Boolean Formulae

- X_1, X_2, …, X_n are formulae.
- If f is a formula then $\neg$ f is a formula.
- If f_1 and f_2 are formulae then $(f_1) \wedge (f_2)$ and $(f_1) \vee (f_2)$ are formulae.
- Sometimes we leave out parentheses….

Formulae represent Boolean functions

- "$(x_1 \wedge \neg x_2) \vee (x_2 \wedge \neg x_1)$" represents the function XOR.
- Sometimes formula-representation is much more compact than table representation. Sometimes not.
- It's never much *less* compact.

Two special classes of Boolean formulae

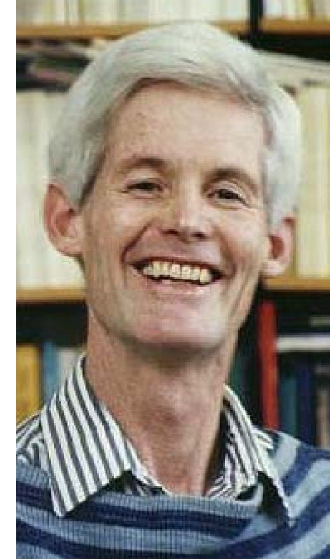
- $f(x_1, x_2) = x_1 \oplus x_2$
- **DNF**: $f(x_1, x_2) = (x_1 \wedge \neg x_2) \vee (x_2 \wedge \neg x_1)$
- **CNF**: $f(x_1, x_2) = (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_2)$
- A CNF is any conjunction of ***clauses*** (disjunctions of literals). A DNF is any disjunction of ***terms*** (conjunctions of literals).
- Any function on *n* variables can be described by a CNF (DNF) formula containing at most 2^n clauses (terms), each containing at most *n* literals.

SAT

- SAT: Given a Boolean function in CNF representation, is there a way to assign truth values to the variables so that the function evaluates to true?
- SAT: Given a CNF, is it true that it does **not** represent the constant-0 function?

- Input: $(\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_2)$
- Output: **Yes.**
- Input: $(\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2)$
- Output: **No.**

SAT

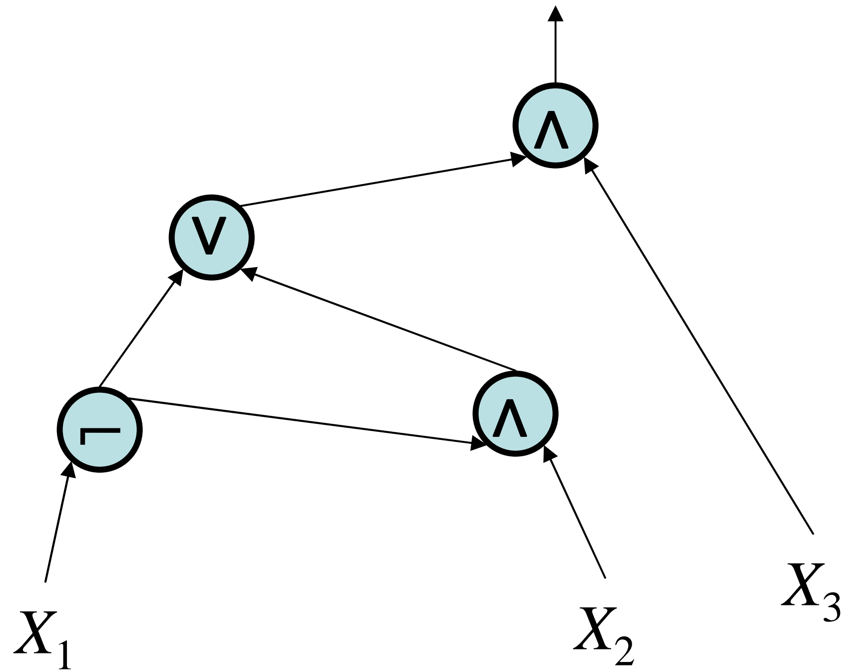


- SAT is in **NP**.
- Cook's theorem (1972): SAT is **NP**-hard.
- Hence, SAT is **NP**-complete. It is a universal search problem and we do ***not*** think it has a polynomial time algorithm. If we find one, we get $1.000.000.

DNF?

- Suppose we ask the same question of DNF formulas, rather than CNF formulas.
- Can you get a million dollars for solving this problem?

Boolean Circuits



Circuits

- A circuit C is a directed ***acyclic*** graph.
- Nodes in C are called ***gates***.

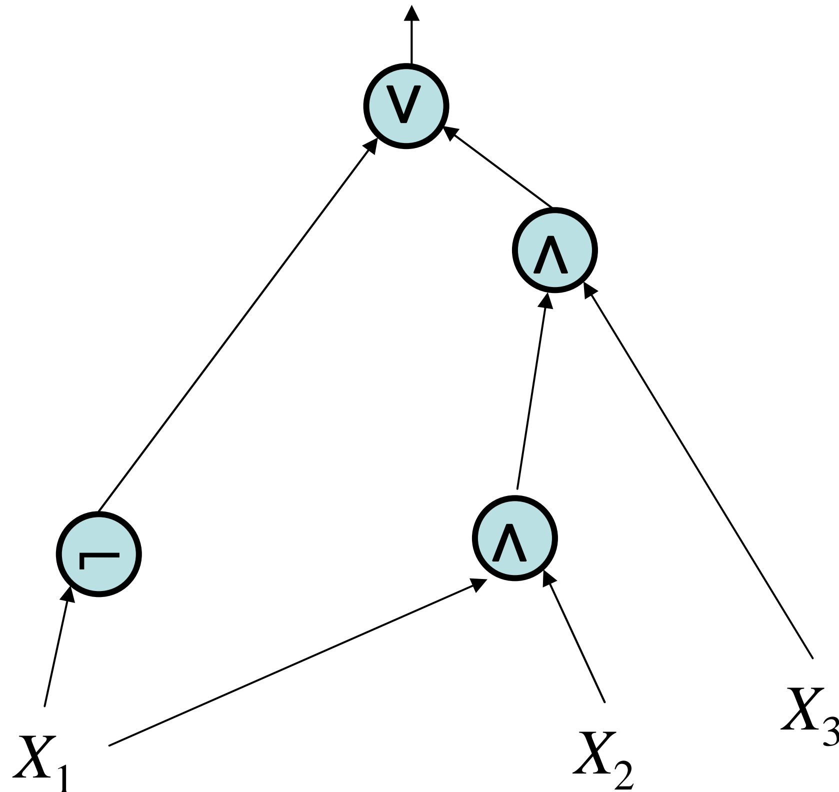
Types of gates:

  - Variable (or ***input***) gates of indegree 0, labeled X_1, X_2, …X_n
  - **Constant** gates of indegree 0, labeled 0,1
  - **AND**-, **OR**-gates of indegree 2,
  - **NOT**-, **COPY**-gates of indegree 1.
  - m distinguished gates are ***output*** gates.

- The circuit C computes a Boolean function $C:\{0,1\}^n \rightarrow \{0,1\}^m$

A formula can be viewed as a special kind of circuit

$$f(X_1, X_2, X_3) = \neg X_1 \vee (X_1 \wedge X_2 \wedge X_3)$$



Are formulas and circuits in fact the same thing?

- **Not quite!**
- Given a circuit, we can write down an equivalent formula, but in may become ***much*** bigger.
- A circuit is allowed to ***reuse*** the result of a sub-computation without doing the computation again!

Multi-output circuits

- Any function f: $\{0,1\}^n \to \{0,1\}^m$ is represented by some circuit.

Proof:

- Any function f: $\{0,1\}^n \to \{0,1\}$ can be described by a Boolean formula.
- A formula can also be viewed as a circuit.
- Combine m such circuits.

Tables – Formulae - Circuits

- For a function f: $\{0,1\}^n \rightarrow \{0,1\}$, let $s_t = 2^n$ be the size (in bits) of its representation as a table.
- Let s_f be the size (in bits) of its smallest representation as a formula. Then,
  - $|s_f| \leq 10\,|s_t|^2$ (formulae are not much less compact than tables).
  - For all n, there is a function so that $|s_f| \leq 10\,(\log |s_t|)^2$ (for some functions, formulae are ***much*** more compact)
- Let s_c be the size (in bits) of its smallest representation as a circuit. Then,
  - $|s_c| \leq 10\,|s_f|^2$ (circuits are not much less compact than formulae).
  - Is it true than there for all n is a function so that circuits are ***much*** more compact (e.g., $|s_c| \leq 10\,(\log |s_f|)^2$)? **This is open!**
- **You won't get $1.000.000 for solving this, but you'll still get pretty famous ….**

Circuits vs. Turing Machines

- Circuits can be given as ***inputs*** to algorithms but they can also be seen as computational ***devices*** themselves!
- Like Turing Machines, circuits C: $\{0,1\}^n \rightarrow \{0,1\}$ solve ***decision*** problems on $\{0,1\}^n$.
- Unlike Turing machines, circuits takes inputs of a ***fixed*** input length n only.

Theorem

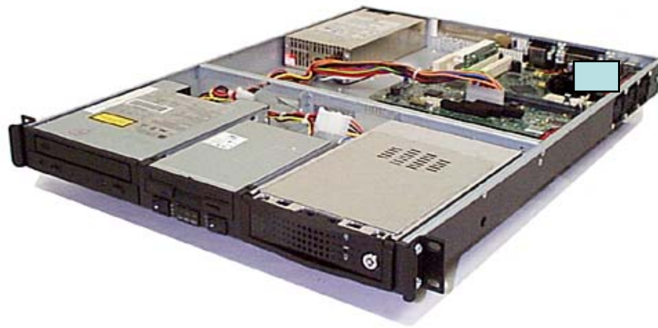
- Given Turing Machine M running in ***time*** at most $p(n)$ on inputs of length n, where p is a polynomial.

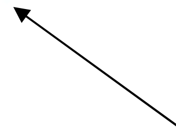
- For every n, there is a circuit C_n with at most $O(p(n)^2)$ ***gates*** so that

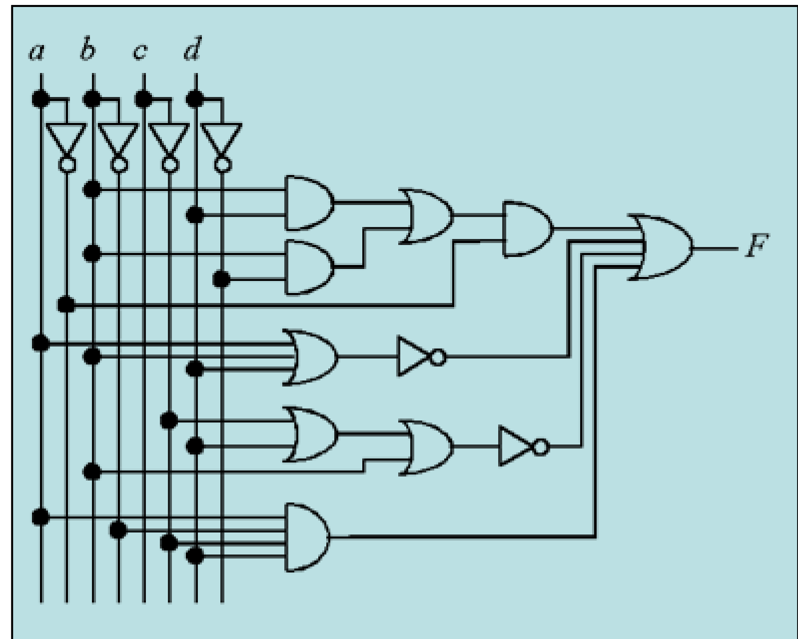
  $\forall x \in \{0,1\}^n$: $C_n(x)=1$ iff M accepts x.

- The map $1^n \to C_n$ is polynomial time computable.

Intuition behind proof







Problem: Cycles!

D

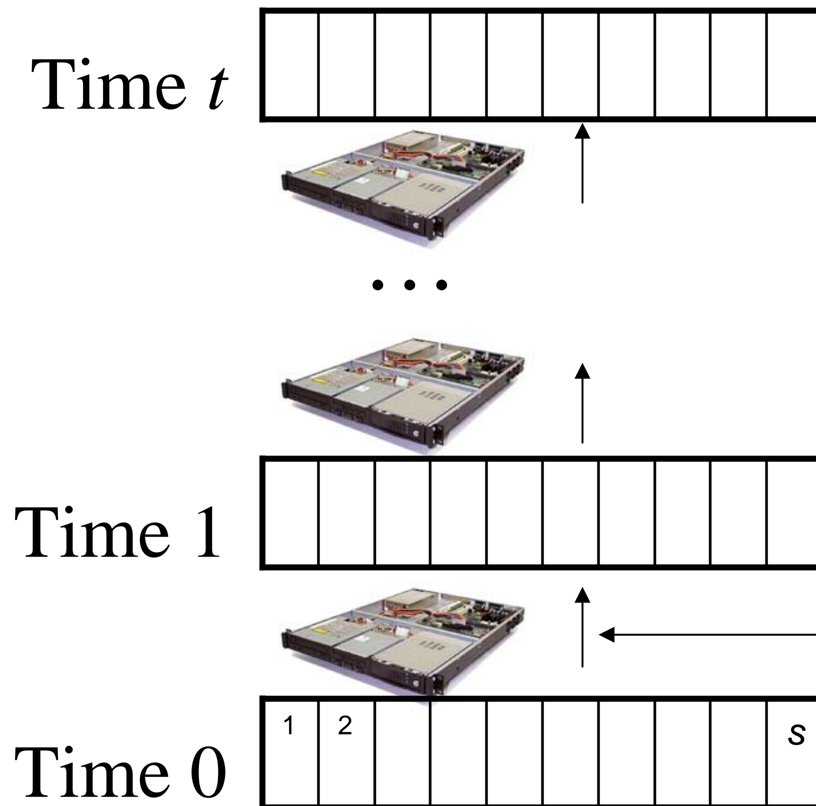
CLK

Q

$\overline{Q}$

Flip-Flop, ***stores*** one bit.

The Tableau Method



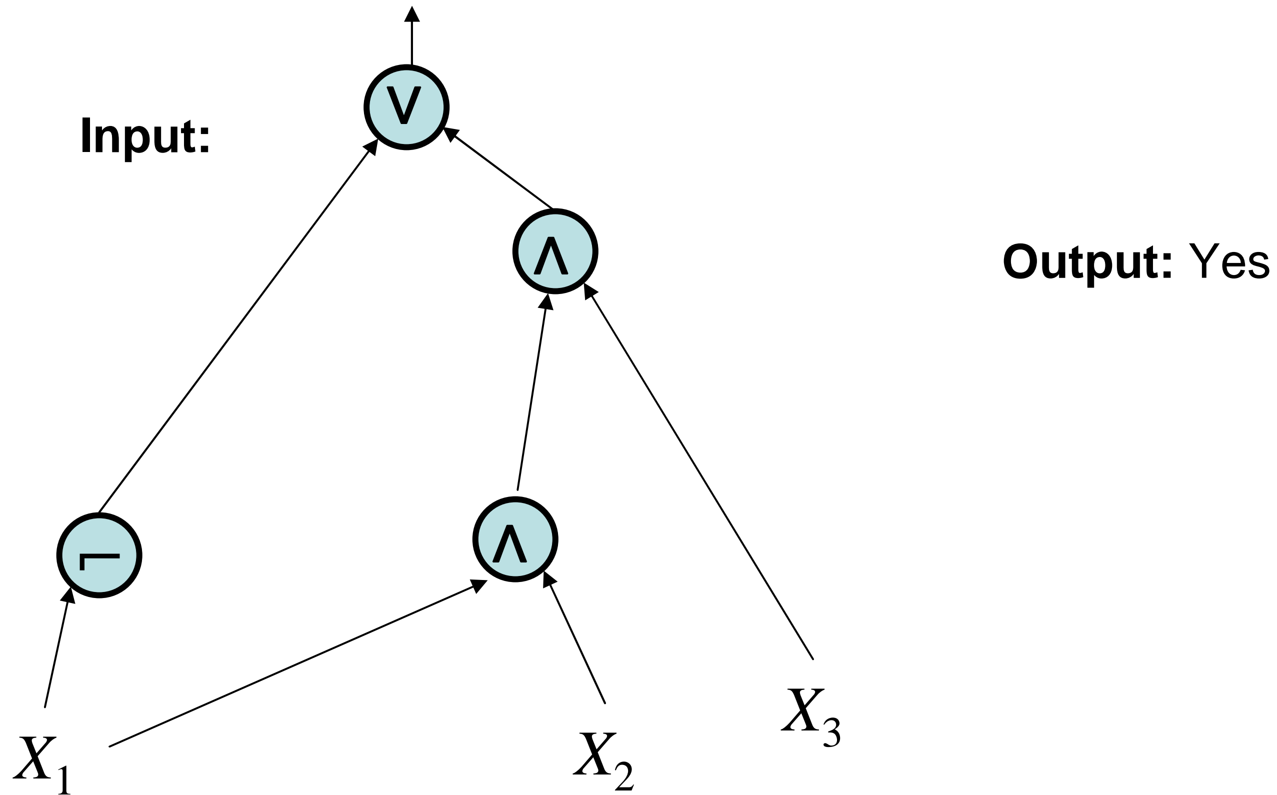
Can be replaced by *acyclic* Boolean circuit of size ≈ s

CIRCUIT SAT

- CIRCUIT SAT: Given a Boolean circuit, is there a way to assign truth values to the input gates variables, so that the output gate evaluates to true?

- Generalizes SAT, as CNFs are formulas and formulas are circuits.

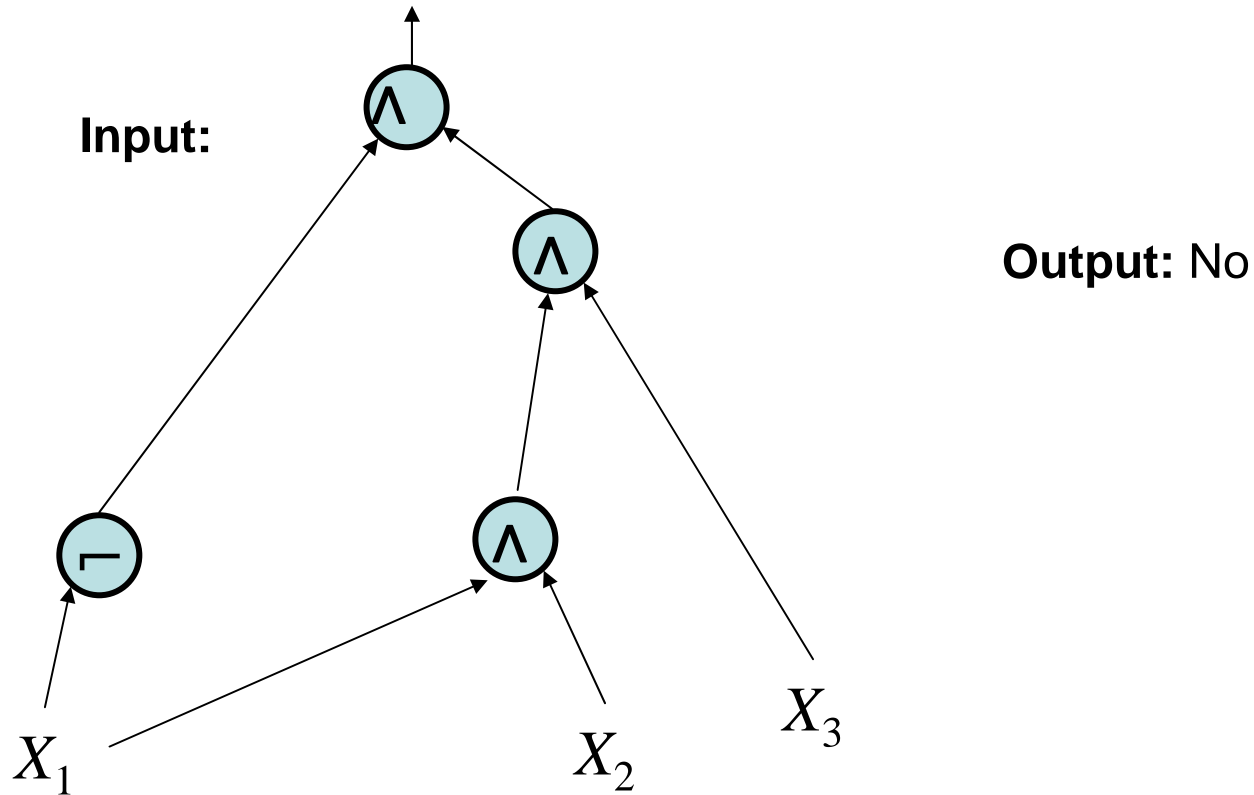
Example



Input:

Output: Yes

Example



Output: No

CIRCUIT SAT

- CIRCUIT SAT is in **NP**.
- Proof of Cook's theorem:
  - CIRCUIT SAT reduces to SAT.
  - CIRCUIT SAT is **NP**-hard.
  - Hence, SAT is **NP**-hard.

CIRCUIT SAT $\longrightarrow$ SAT

CIRCUIT SAT is **NP**-hard

- Given an arbitrary language L in NP we must show that L reduces to CIRCUIT SAT.

- This means: We must construct a polynomial time computable map r mapping instances of L to circuits, so that

  $\forall$ x: x $\in$ L $\Leftrightarrow$ r(x) $\in$ CIRCUIT SAT

- The only thing we know about L is that there is a language L' in P and a polynomial p, so that:

$$\forall x : \quad x \in L \Leftrightarrow [\exists y \in \{0,1\}^* : |y| \leq p(|x|) \wedge \langle x, y \rangle \in L']$$

Theorem

- Given Turing Machine M running in ***time*** at most $p(n)$ on inputs of length n, where p is a polynomial.

- For every n, there is a circuit C_n of ***size*** at most $O(p(n)^2)$ so that

  $\forall x \in \{0,1\}^n$: $C_n(x)=1$ iff M accepts x.

- The map $1^n \to C_n$ is polynomial time computable.