NP-Completeness and Reduction

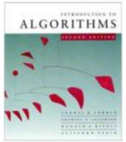
Classes of problems

P (Polynomial-time)

- Most of the algorithms studied are polynomial time.
- On the input size of *n*, the running time is $O(n^k)$ *for some constant k*.
- All the problems can be solved in P? Answer is no.

NP (Nondeterministic polynomial time)

- Decision problems that are verifiable in polynomial time.
- Super-polynomial time for solving problems.
- Any problem in P is also in NP.

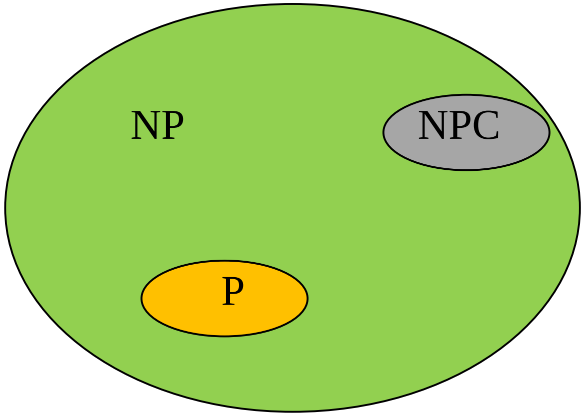


Classes of problems

NPC (NP-complete)

- A problem is in NP.
- It is as hard as any problem in NP.
- Can NPC problems be solved in polynomial time?

  Not found yet.



Current view
$P \cap NPC = \emptyset$

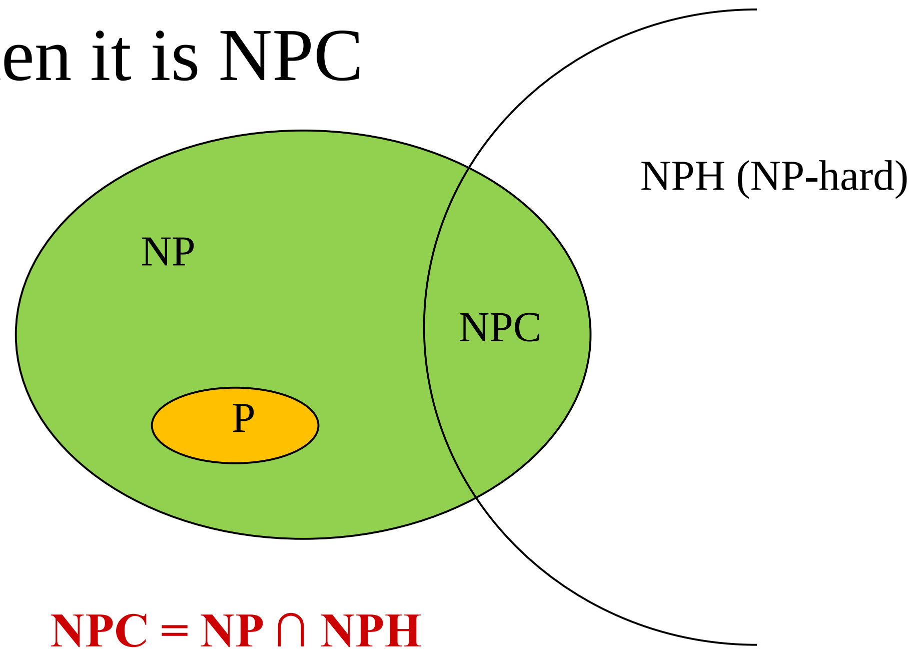
Classes of problems

NPH (NP-hard) and NPC

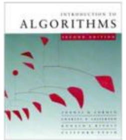
- A problem X is in NPH if every problem in NP reduces to the problem X.

(X is NP-hard if every problem Y∈NP reduces to X => (this implies that) X∉P unless P=NP)

- If X is NP then it is NPC



NPC = NP ∩ NPH

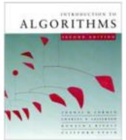


Overview of showing NP-complete

- In showing a problem is NP-complete, make a statement **of how hard it is**, not how easy it is.

(We are not trying to prove the existence of an efficient algorithm, but rather that no efficient algorithm is likely to exist.)

- Three key concepts in showing a problem to be NP-complete
  1) Optimization problems vs. Decision problems
  2) Reductions
  3) A first NP-complete problem



Overview of showing NP-complete

Optimization problems vs. Decision problems

Optimization problems

- Finding the best value (optimal value).
- Example: The shortest path in a graph.
  The fastest time in a scheduling.

Decision problems

- Finding an answer that is simply yes or no (1 or 0).
- Solving a decision problem is referred to as **verification**.
- An optimization problem can be casted to a **related** decision problem.
  e.g.) Find a shortest path. (optimal problem)
  Is a path with k-length the shortest? (yes or no decision problem)

A decision problem is easier ,or no harder, than the related optimization problem.
Thus, a decision problem is hard → a related optimization problems is also hard.

Overview of showing NP-complete

Reductions

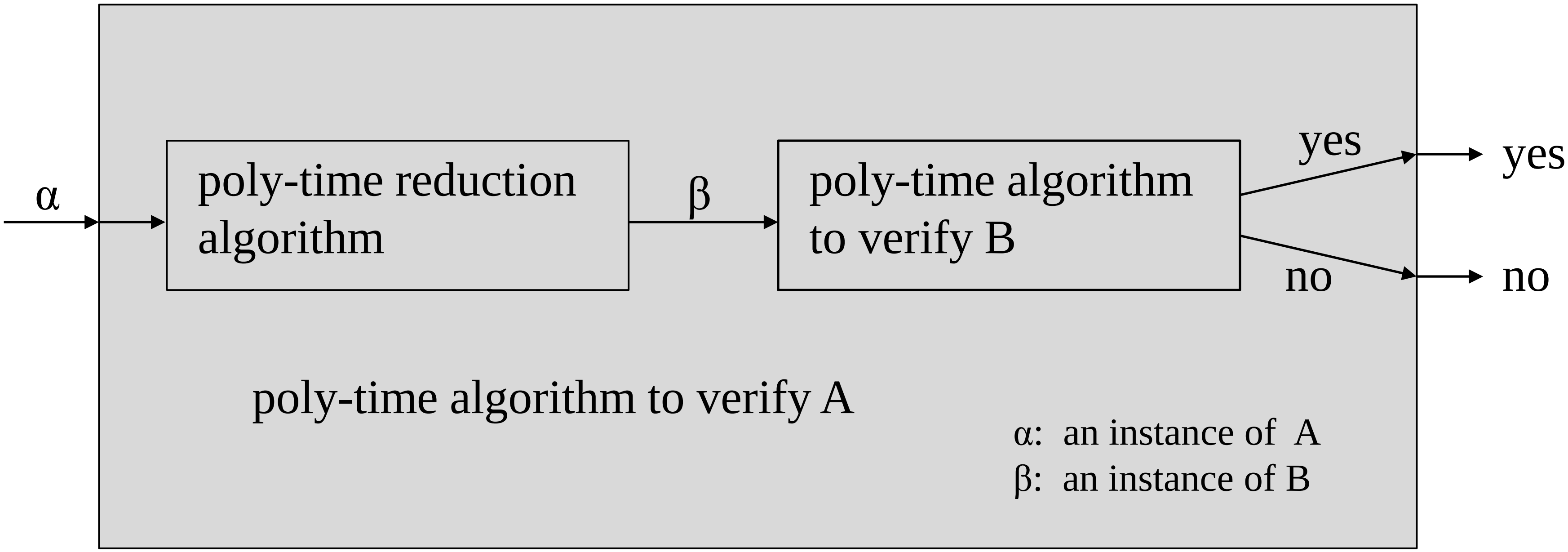
• Reduction from problem A to problem B
= (means there is) poly-time algorithm converting
A inputs to equivalent B inputs.
Yes/No answer

• Which means the followings:
If B∈NP then A∈NP

• A and B are both decision problems with an identical answer.

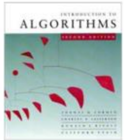
Overview of showing NP-complete

Reductions



- A way to verify A in polynomial time (reducing verifying A to verifying B).

1. Given α of A, use the polynomial time reduction algorithm to transform it to β of B.
2. Run the polynomial time decision algorithm for B on β.
3. Use the answer for β as the answer for α.



Overview of showing NP-complete

Reductions

- Consider **two decision problems, A and B.**

- Want to verify A in polynomial time.

- Input to a problem is an instance of that problem.
  e.g.) Instance in PATH problem: G (graph), *u* and *v* (1^{st} and last vertices),
  k (the length of a path from *u* to *v*)
- How to verify B in polynomial time is already known.

- Polynomial time **reduction algorithm** is a procedure that transforms any instance α of A into some instance β of B with the following characteristics:

1. The transformation takes polynomial time.
2. The answers are the same, that is the answer for α is "yes" if and only if the answer for β is also "yes."

Overview of showing NP-complete

Polynomial time verification

- **Hamiltonian cycles problem**

  "Does a graph G have a hamiltonian cycle?"

- **Hamiltonian cycle:**

  The hamiltonian cycle of an undirected graph G = (V, E) is a simple cycle that contains each vertex in V.

  A graph that contains a hamiltonian cycle is hamiltonian.

Overview of showing NP-complete

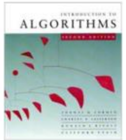
Polynomial time verification

- **How to decide HAM-CYCLE?**
  (How to verify HAM-CYCLE?)

Given a problem instance <G>. G: undirected graph

1) **Naïve algorithm:**
   List all permutations of G and check each permutation to see if it is a hamiltonian cycle.
   Running time is $\Omega(n!)$, where n is number of vertices in G.
   $\Omega(n!)$ grows faster than polynomial time.

2) **Polynomial time verification algorithm:**
   Provide a G and a hamiltonian cycle and verify if G is a hamiltonian.
   Check if the cycle is the permutation of the vertices of G.
   Check if the consecutive edges along the cycle actually exists in G.
   The running time is $O(m^2)$, m is the number edges in G.
   $O(m^2)$ is polynomial time.



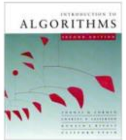
Showing first NP-completeness

First NP-complete problem usage

- **Use polynomial time reductions from A to B to show that no polynomial time algorithm can exist for a problem B.**

- Suppose for a problem A no polynomial time algorithm exists (A is hard).
- Between A and B the difference is only the polynomial time (Since B is reduced from A).
- So, no polynomial time algorithm for B (B is also hard problem).
- **If problem A is NPC then problem B is at least as hard as A.**

- **There should be a first problem that is NP-complete to show that certain problems are NP-complete.**



Showing first NP-completeness

- A problem is NP-complete if
  1. It is NP
  2. It is NPH (Every problem in NP reduces to it in poly-time.)

- A problem Y is NP-complete if
  1. Y ∈ NP
  2. X ≤p Y for every X ∈ NP
     (Where, ≤p is the polynomial time reduction.)

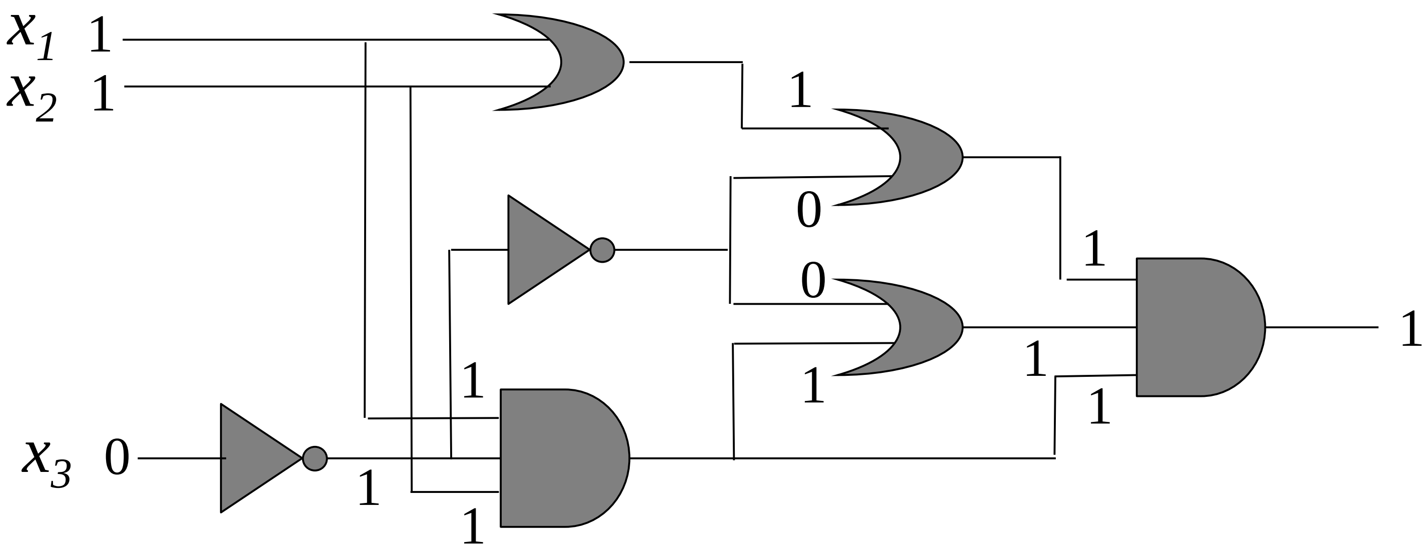
- Circuit satisfiability problem: The first NP-complete problem.

Circuit satisfiability

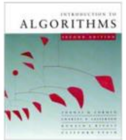
Circuit satisfiablility problem (CIRCUIT-SAT),

- Circuit satisfiablility problem: given a boolean combinational circuit composed of AND, OR, and NOT gates, is it satisfiable?
- Satisfying assignment : a truth assignment that causes the output to be 1.
- **Satisfiable :** an one-output boolean combinational circuit has a satisfying assignment.

Circuit satisfiability



The circuit is satifiable.



Circuit satisfiability

A CIRCUIT-SAT is NP-complete if,

1. It is NP
2. It is NP-hard

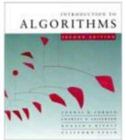
First, show the CIRCUIT-SAT is NP

• We have to show : given a boolean combinational circuit and a truth assignment to the input variables, it takes polynomial time to verify that the output is 1.

• The verification algorithm runs in poly-time because,

Input length to verifier : number of gates (number of input variables).
Truth assignment to input variables: polynomial in the size of the input.

So, the evaluation of all the gates, given a truth assignment, takes polynomial time. That is, the verification algorithm takes poly-time.



Circuit satisfiability

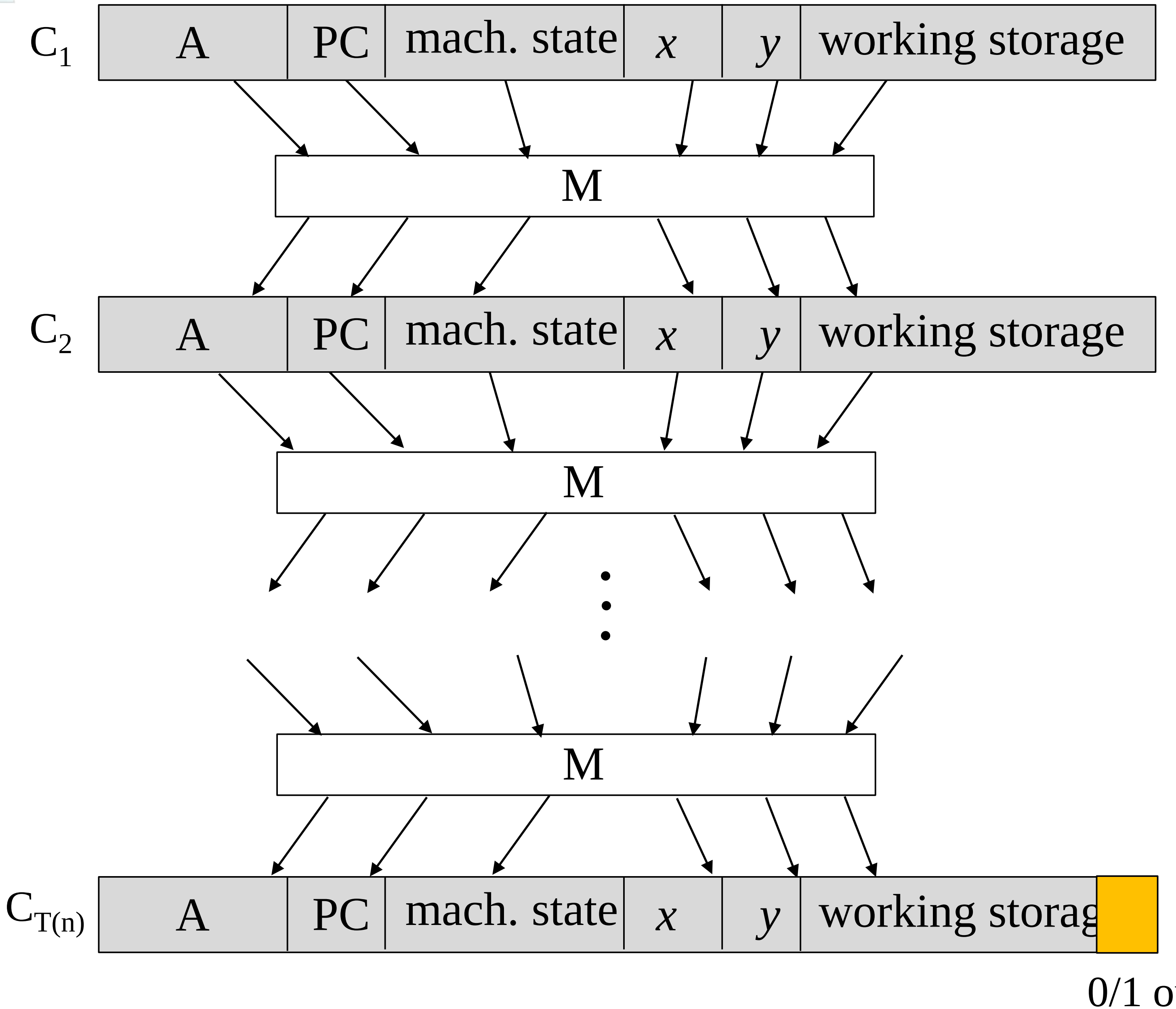
Second, show CIRCUIT-SAT is NP-hard

We have to show : every problem in NP reduces to CIRCUIT-SAT.

- A verification algorithm A of every problem in NP consists of a set of instructions.
- The number of instructions in A is polynomial.
- Transformation of the instructions in A into a boolean combinational circuit M takes polynomial time.
- Since the logic in A and M are identical, their outputs are identical.
- Thus, the reduction of A into M takes polynomial time.

The every problem (verification problem of A) in NP is reduced to the problem of CIRCUIT-SAT of M.

Circuit satisfiability



NP-completeness proofs

Prove that a problem Y is NP-complete without directly reducing every problem in NP to Y.

1. **If a problem X in NPC reduces to a problem Y then Y is NPH.**
2. **If Y is NP then Y is NPC.**

Formula satisfiability

Formula satisfiability (SAT)

Naïve algorithm for determining the boolean formula statisfiability:

- **There are 2^n possible assignments in a formula ϕ with n variables.**
- **If the length of ϕ is polynomial in n, then checking every assignment requires $\Omega(2^n)$ which is superpolynimial. A polynomial algorithm is unlikely to exist.**

To prove that satisfiability of boolean formulas (SAT) is NP-complete,

1. show that SAT is NP,

2. show that SAT is NPH by showing that CIRCUIT-SAT can be reduced to SAT in polynomial time.

Formula satisfiability

1. SAT is NP

- **Given a satisfying assignment for the variables in a formula it takes polynomial time to evaluate the variables.**

- **Simply replace each variable in the formula with its corresponding value and then evaluate the expression.**

- **If it evaluates to 1 the formula is statisfiable.**

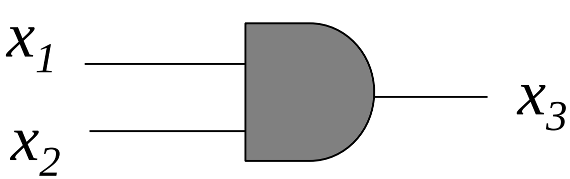
Formula satisfiability

2. SAT is NPH

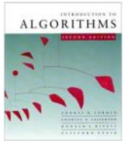
- Show that CIRCUIT-SAT $\leq p$ SAT

- **Reduction:**

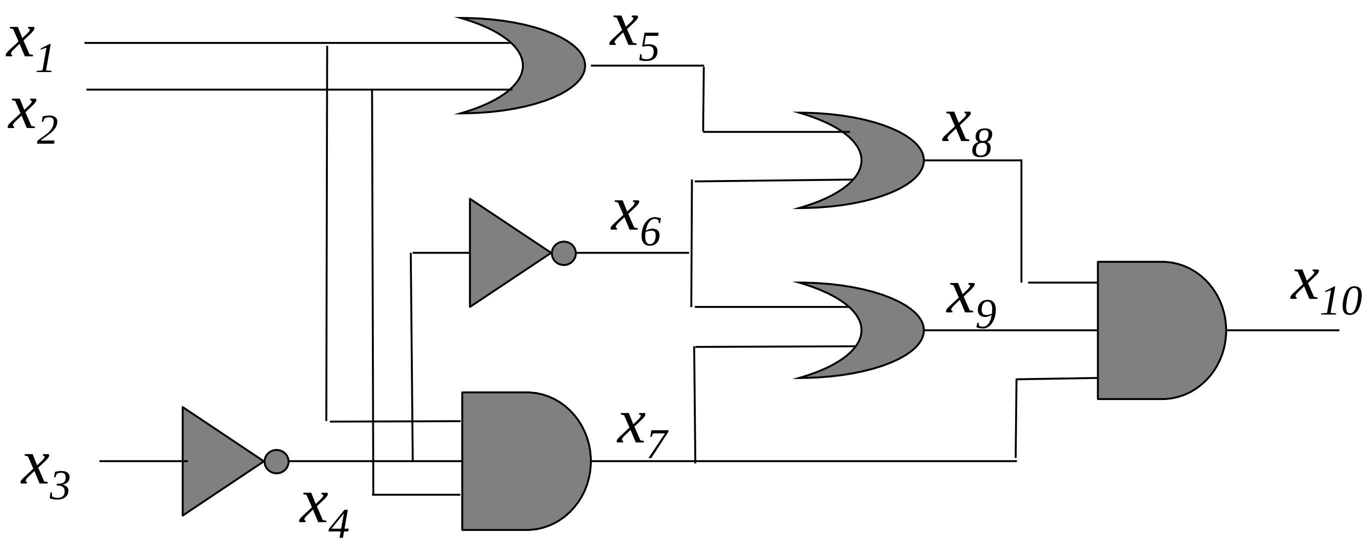
  - If the circuit below has a satisfiying assignment, the ouput is 1. Thus, $x_1=1, x_2= 1, x_3=1$
  - The assignment of above wire values to the variables of the formula below evaluates the formula to 1 also.
  - That is, the formula is satisfiable when the circuit is satisfiable, and vice versa.
  - The transformation takes polynomial time.



$$(x_3 \leftrightarrow (x_1 \text{ and } x_2))$$



Formula satisfiability



$$
\begin{aligned}
\phi = x_{10} &\text{ and } (x_4 \leftrightarrow (\text{not } x_3)) \\
&\text{ and } (x_5 \leftrightarrow (x_1 \text{ or } x_2)) \\
&\text{ and } (x_6 \leftrightarrow (\text{not } x_4)) \\
&\text{ and } (x_7 \leftrightarrow (x_1 \text{ and } x_2 \text{ and } x_4)) \\
&\text{ and } (x_8 \leftrightarrow (x_5 \text{ and } x_6)) \\
&\text{ and } (x_9 \leftrightarrow (x_6 \text{ or } x_7)) \\
&\text{ and } (x_{10} \leftrightarrow (x_7 \text{ and } x_8 \text{ and } x_9)
\end{aligned}
$$

The circuit and the formula have the equivalent structure. Therefore, the outputs are identical.

3-CNF satisfiability

Example of a boolean formula in 3-CNF (conjunctive normal form)

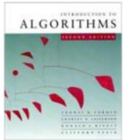
(x_1 or x_2 or (not x_2)) and (x_3 or x_2 or x_4) and …
There are three literals in every clause.

3-CNF-SAT (3 conjunctive normal form formula satifiability)

To prove that 3-CNF-SAT is NP-complete,

1. show that 3-CNF-SAT is NP

2. show that 3-CNF-SAT is NPH by showing SAT $\leq_p$ 3-CNF-SAT



3-CNF satisfiability

1. 3-CNF-SAT is NP

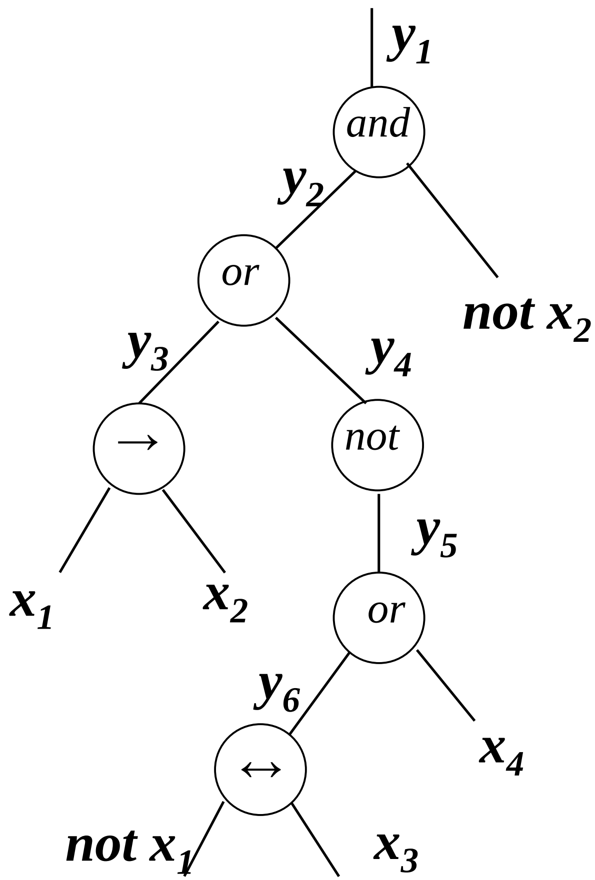
- **The proof of this is similar to SAT.**

Given a satisfying assignment for the variables in a formula it takes polynomial time to evaluate the variables.

2. 3-CNF-SAT is NPH, i.e., show SAT $\leq_p$ 3-CNF-SAT

Given a formula ϕ below,
$\phi = ((x_1 \rightarrow x_2)$ *or not* $((\textit{not } x_1 \leftrightarrow x_3)$ *or* $x_4))$ *and not* x_2
a binary parse tree can be constructed.

3-CNF satisfiability



$$\phi' = y_1 \text{ and } (y_1 \leftrightarrow (y_2 \text{ and } (\text{not } x_2)))$$
$$\text{and } (y_2 \leftrightarrow (y_3 \text{ or } y_4))$$
$$\text{and } (y_3 \leftrightarrow (y_1 \rightarrow x_2))$$
$$\text{and } (y_4 \leftrightarrow (\text{not } y_5))$$
$$\text{and } (y_5 \leftrightarrow (y_6 \text{ or } x_4))$$
$$\text{and } (y_6 \leftrightarrow ((\text{not } x_1) \leftrightarrow x_3))$$

From the left tree, ϕ' can be constructed.

Given ϕ, a binary parse tree can be constructed.

3-CNF satisfiability

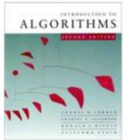
$\phi_1' = (y_1 \leftrightarrow (y_2 \text{ and } (\text{not } x_2)))$

Truth table for ϕ_1'

y_1	y_2	x_2	$(y_1 \leftrightarrow (y_2 \text{ and } (\text{not } x_2)))$
1	1	1	0
1	1	0	1
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	1
0	0	0	1

From the truth table, (not ϕ') can be obtained.

$(\text{not } \phi') = (y_1 \text{ and } y_2 \text{ and } x_2)$ or
$(y_1 \text{ and } (\text{not } y_2) \text{ and } x_2)$ or
$(y_1 \text{ and } (\text{not } y_2) \text{ and } (\text{not } x_2))$ or
$((\text{not } y_1) \text{ and } y_2 \text{ and } (\text{not } x_2))$



3-CNF satisfiability

Applying DeMorgan's laws to (not ϕ'), we get the CNF formula

$\phi'' =$ ***((not y_1) or (not y_2) or (not x_2)) and ((not y_1) or y_2 or (not x_2))***
((not y_1) or y_2 or x_2) and (y_1 or (not y_2) or x_2)

As the final step of the reduction, transform the formula so that each clause has 3 distinct literals.

For each clause of ϕ'', include the following clauses in ϕ'''.

- If a clause has 3 distinct literals, then simply include it as a clause of **ϕ'''.**
- If a clause has 2 distinct literals, then do as shown below.

(l_1 or l_2) => (l_1 or l_2 or p) and (l_1 or l_2 or (not p)) : included as a clause of ϕ'''.
These two clauses are equivalent**.**

- If a clause has 1 distinct literal, then do as shown below.

(l) => (l or p or q) and (l or p or (not q))
and (l or (not p) or q) and (l or (not p) or (not q)
: included as a clause of ϕ'''.
These two clauses are equivalent.

3-CNF satisfiability

The 3-CNF formula ϕ''' is satisfiable iff ϕ is satisfiable.

Also observe that each transformation step takes polynomial time.

Therefore, the reduction can be computed in polynomial time.