Computationally Hard Problems

$\mathcal{NP}$-Completeness, Cook-Levin Theorem and Satisfiability

Carsten Witt

Institut for Matematik og Computer Science
Danmarks Tekniske Universitet

Fall 2020

Motivation

- Some problems in $\mathcal{NP}$ are hard, others are efficiently solvable.
- Membership in $\mathcal{NP}$ is not an indicator for hardness.
- The aim is to classify the hard problems in $\mathcal{NP}$.
- Idea: A problem that can be used to solve others is at least as hard than those.
- More precise: If the solution to problem P_2 can be converted into one of problem P_1 with little extra work then P_2 is at least as hard as P_1.
- The hardest problems are those that can be used to solve all others (in a certain class).

Formalizing Things

Steps to show that problem P_2 is at least as hard as problem P_1:

- Let $\boldsymbol{X_1}$ be an instance of problem P_1.
- Transform $\boldsymbol{X_1}$ into an instance $\boldsymbol{X_2}$ of problem P_2.
- The transformation has to be performed in time polynomial in $\|\boldsymbol{X_1}\|$.
- Then show that the answer to $\boldsymbol{X_1}$ is YES
  if and only if the answer to $\boldsymbol{X_2}$ is YES
  (Equivalent formulation: if answer to $\boldsymbol{X_1}$ is YES then answer to $\boldsymbol{X_2}$ is YES and if answer to $\boldsymbol{X_1}$ is NO then answer to $\boldsymbol{X_2}$ NO.)

Informally this means: If one had a solution method for P_2 then this can be used as a *Black Box* for solving P_1. The challenge is to find a transformation with the stated properties.

Formalizing Things

Definition

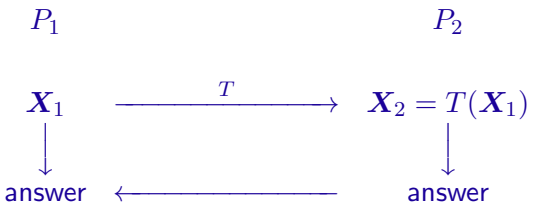
Let P_1 and P_2 be yes-no-problems. Let Σ_1 and Σ_2 be the respective input alphabets. We identify the problems with the languages, i.e., $P_i \subseteq \Sigma_i^*$. We say that P_1 is *polynomial-time reducible* to P_2 (*reducible*, for short) if there is a polynomially bounded **deterministic** algorithm $T\colon \Sigma_1^* \to \Sigma_2^*$ such that

$$\forall \boldsymbol{X}_1 \in \Sigma_1^*\colon \boldsymbol{X}_1 \in P_1 \iff T(\boldsymbol{X}_1) \in P_2 \ .$$

We write $P_1 \leq_p P_2$ if P_1 is reducible to P_2.

Formalizing Things

Let $\boldsymbol{X}_1 \in \Sigma_1^*$ and $\boldsymbol{X}_2 \in \Sigma_2^*$.



Consequences

Claim

If $P_1 \leq_p P_2$ and P_2 is efficiently solvable then so is P_1.

An equivalent formulation is:

Claim

If $P_1 \leq_p P_2$ and P_1 is not efficiently solvable then P_2 is not efficiently solvable either.

Proof of the First Claim

Claim

If $P_1 \leq_p P_2$ and P_2 is efficiently solvable then so is P_1.

- Let A_2 be a p-bounded deterministic algorithm for solving P_2, where p is a polynomial.
- Let T be a q-bounded transformation (q also polynomial).
- Let $\boldsymbol{X}_1$ be an instance of P_1.
- Then $\boldsymbol{X}_2 = T(\boldsymbol{X}_1)$ is an instance of P_2 of size at most $q(\|\boldsymbol{X}_1\|)$.
- A_2 solves $\boldsymbol{X}_2$ in time $p(\|\boldsymbol{X}_2\|)$, and its answer is the correct answer for X_1.
- Note that: $p(\|\boldsymbol{X}_2\|) = p(q(\|\boldsymbol{X}_1\|))$
- The total running time is thus bounded by the polynomial $r = p \circ q$.

Properties

Claim

The relation $\leq_p$ is transitive, i. e., for all P, R, S we have

$$((P \leq_p R) \wedge (R \leq_p S)) \implies (P \leq_p S)$$

EXERCISE

$\mathcal{NP}$-Completeness

Definition

A problem P is called *$\mathcal{NP}$-complete* if

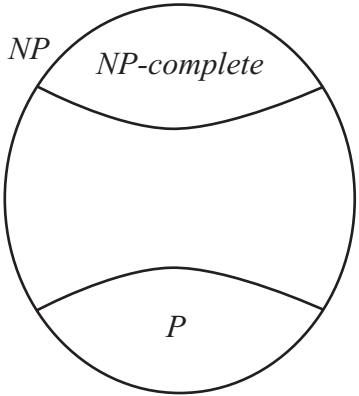
$$P \in \mathcal{NP} \tag{1}$$

$$\forall P' \in \mathcal{NP} \colon P' \leq_p P \tag{2}$$

Condition (2) expresses that problem P is at least as hard as **any** problem in $\mathcal{NP}$.

An algorithm for solving P can be converted into an algorithm for solving **any** other problem in $\mathcal{NP}$ with only a polynomial increase in time.

The $\mathcal{NP}$-world



$\mathcal{NP}$-complete or $\mathcal{NP}$-hard?

People tend to speak of $\mathcal{NP}$-hard problems.

Then they often mean optimization problems with an $\mathcal{NP}$-complete decision variant.

However, there are two slightly conflicting definitions of "$\mathcal{NP}$-hard":

- The first one covers optimization problems with an $\mathcal{NP}$-complete decision variant. Then $\mathcal{NP}$-hard problems are not necessarily in $\mathcal{NP}$.
- The second one uses parts of our definition of $\mathcal{NP}$-completeness ($\forall P' \in \mathcal{NP} \colon P' \leq_p P$), but does not demand that the problem itself is in $\mathcal{NP}$. That definition only covers decision problems.

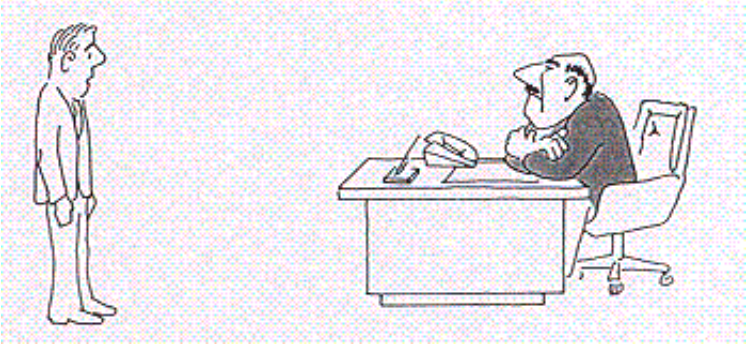
In this course, we focus on decision problems that are in $\mathcal{NP}$:
Then $\mathcal{NP}$-completeness is the appropriate definition to study.

$\mathcal{NP}$-Completeness

- $\mathcal{NP}$-complete problems are **believed** to be computationally hard.
- That means that it is believed that they do not have deterministic polynomial-time algorithms.
- Why is it only believed? No one has been able to prove it.
- BUT no one has come up with a deterministic polynomial algorithm for an $\mathcal{NP}$-complete problem.
- An award of 1 000 000 \$ has been established ...

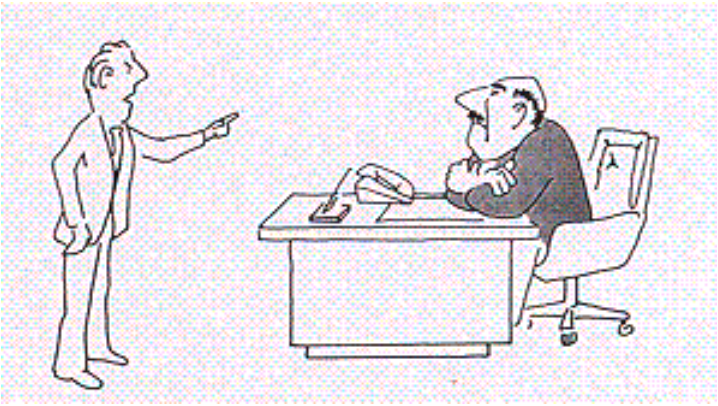
The $\mathcal{P}$ vs. $\mathcal{NP}$ Question (1/3)

Example from the book by Garey and Johnson (1979):



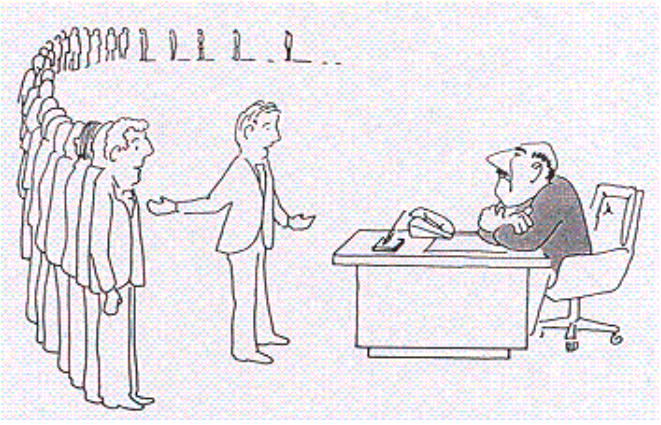
"I can't find an efficient algorithm *[a fast program]*, I guess I'm just too dumb."

The $\mathcal{P}$ vs. $\mathcal{NP}$ Question (2/3)



"I can't find an efficient algorithm, because I can prove that no such algorithm is possible."

The $\mathcal{P}$ vs. $\mathcal{NP}$ Question (3/3)



"I can't find an efficient algorithm, but neither can all these famous people."

About Proving $\mathcal{P} \neq \mathcal{NP}$

- Why is that hard to prove?
- One would have to show that every algorithm takes super-polynomial time on infinitely many problem instances.
- This is a statement about all infinitely many algorithms.
- Neither Computer Science nor Mathematics provides adequate tools for proving such statements.

Proving $\mathcal{NP}$-Completeness

Hence, it seems impossible to prove that a problem P is $\mathcal{NP}$-complete?

- One would have to show that every problem $P' \in \mathcal{NP}$ is reducible to P, i. e.,

$$\forall\, P' \in \mathcal{NP} \colon P' \leq_p P$$

- There are infinitely many $P' \in \mathcal{NP}$.
- How to do that?

Proving $\mathcal{NP}$-Completeness

We can use the transitivity of $\leq_p$ to make things tractable:

- Suppose we know a problem P_c to be $\mathcal{NP}$-complete.
- Then we only show that $P_c \leq_p P$.
- As P_c is $\mathcal{NP}$-complete we know that $\forall\, P' \in \mathcal{NP} \colon P' \leq_p P_c$
- Hence $P' \leq_p P_c \leq_p P$.
- By transitivity it follows $P' \leq_p P$.
- Hence

$$\forall\, P' \in \mathcal{NP} \colon P' \leq_p P$$

and P is $\mathcal{NP}$-complete.

Proving $\mathcal{NP}$-Completeness

Guideline

- Prove that $P \in \mathcal{NP}$.
- Find a suitable problem P_c which is known to be $\mathcal{NP}$-complete.
- Prove $P_c \leq_p P$.

The only difficulty is: One has to have some start problem P_c which is known to be $\mathcal{NP}$-complete.

Satisfiability

The first problem shown to be $\mathcal{NP}$-complete the well-known Satisfiability (or Sat for short):

Problem [Satisfiability]
Input: A set of clauses $C = \{c_1, \ldots, c_k\}$ over n boolean variables $x_1, \ldots, x_n$.
Output: YES is there is a satisfying assignment, i. e., if there is an assignment

$$a\colon \{x_1, \ldots, x_n\} \to \{0, 1\}$$

such that every clause c_j is satisfied, and NO otherwise.

Cook-Levin Theorem

In 1972 Stephen Cook and Leonid Levin independently showed that SATISFIABILITY is $\mathcal{NP}$-complete.

Theorem [Cook-Levin]

Every problem $P \in \mathcal{NP}$ is reducible to SATISFIABILITY:

$$P \leq_p \text{SATISFIABILITY}$$

For a proof, we have to show how to transform an instance $\boldsymbol{X}$ of P into one for SATISFIABILITY and show that the answer carries over.

The difficulty is: We have to do this for every (of the infinitely many) problems in $\mathcal{NP}$.

Idea: Use a "generic" transformation.

The Generic Transformation

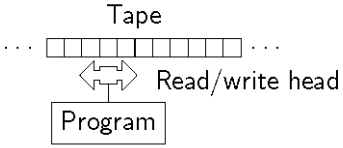
Consider a problem $P \in \mathcal{NP}$.

- Then there is an $\mathcal{NP}$-algorithm A for P.
- Let p be the polynomial which bounds the running time of A.
- Let $\boldsymbol{X}$ be an instance of P.
- Algorithm A is deterministic, but gets random numbers as part of the input ($A(\boldsymbol{X}, R)$).
- The computations of A can be described by a set of clauses.
- The set of clauses is satisfiable iff the truth for $\boldsymbol{X}$ is YES.
- The set of clauses depends only on $\boldsymbol{X}$ and is of size polynomial in $\|\boldsymbol{X}\|$.

The Generic Transformation

To simplify things, one restricts the computational model to a very simple – but *Turing-complete* – one: the Turing Machine

- Infinite tape consisting of cells.
- Every cell is blank, contains a 0 or contains a 1
- The machine has a read-write head.
- The machine has a finite program.
- One step consists of
  - Reading the content of the current cell.
  - Writing something into the current cell.
  - Possibly moving the head one cell left/right.



The Generic Transformation

The boolean variables are defined to describe the state of the machine.

The clauses describe the tape content and the steps of the machine.

Boolean variable $M(t, i, c)$ is true iff at time t the i-th memory cell contains value c.

Then $M(0, i, c)$ is used to determine the random string/input when i is a position from random string or input.

$Q(t, j)$ is true iff at time t the j-th instruction is executed.

There are more variables.

The Generic Transformation

Example of clauses

- $c = M(0, i, 1)$; initialize bit i to 1; used to specify $\boldsymbol{X}$
- $c = M(0, i, 1) \vee M(0, i, 0)$;
  initialize bit i to 1 or 0; used to specify R
- $c = Q(t, 1) \vee Q(t, 2) \vee \cdots \vee Q(t, k)$;
  at least one instruction is executed at time t.
- $c = \neg(Q(t, j) \wedge Q(t, j')) = \neg Q(t, j) \vee \neg Q(t, j')$;
  at most one instruction is executed at time t.
- $[M(t, i, c) \wedge Q(t, j)] \implies M(t + 1, i, c')$; the content of memory cell i is is changed from c to c' when executing operation j. (Of course, j has to be the appropriate type of operation.)
- many more clauses.

The Generic Transformation

The size of the set of clauses is polynomial in $\|\boldsymbol{X}\|$.

A satisfying assignment of the clauses guarantees that

- the problem instance is $\boldsymbol{X}$,
- some random string is on the tape,
- the computations of A are properly executed.

Algorithm A has a statement $\mathrm{return(YES)}$.

If there is a time t such that the variable $Q(t, \mathrm{return(YES)})$ is true then the algorithm will answer YES.

Altogether: A answers YES for some R iff all clauses can be satisfied: $P \leq_p \textsc{Satisfiability}$

Steps for proving $\mathcal{NP}$-completeness (rep.)

We will now show for a number of problems that they are $\mathcal{NP}$-complete, i. e., hard to solve.

- Prove that $P \in \mathcal{NP}$.
- Find a suitable problem P_c which is known to be $\mathcal{NP}$-complete.
- Prove $P_c \leq_p P$.

Right now, the only problem we can use as P_c is SATISFIABILITY.

3-Satisfiability

We want to show that a restricted version of Satisfiability is also $\mathcal{NP}$-complete.

Problem [3-Satisfiability]

Input: A set of clauses $C = \{c_1, \ldots, c_k\}$ over n boolean variables $x_1, \ldots, x_n$, where every clause contains exactly three literals (negated or unnegated variables).

Output: YES is there is a satisfying assignment, i. e., if there is an assignment

$$a\colon \{x_1, \ldots, x_n\} \to \{0, 1\}$$

such that every clause c_j is satisfied, and NO otherwise.

We write 3-Sat for short.

3-SAT is in $\mathcal{NP}$

Theorem

The problem 3-SAT is $\mathcal{NP}$-complete.

Sketch of the proof: Given an instance (a set of clauses) for SAT, transform it into an instance (a set of clauses) for 3-SAT such that the original instance has a satisfying assignment if and only if the transformed one has.

We skip proving that 3-SAT is in $\mathcal{NP}$; this is shown by the usual arguments.

Outline of the Transformation

The input to the transformation is an instance of SAT, i. e., a set $\{c_1, \ldots, c_k\}$ of clauses over boolean variables $\{x_1, \ldots, x_n\}$.

The transformation treats each clause separately.

Clause c_j is replaced by a number of clause $c'_{j,1}, c'_{j,2}, \ldots$ of length 3.

Introduce new variables $y_{j,i}$ (hereinafter often called *helper variables*) used only in clauses $c'_{j,\cdot}$.

Satisfiability is maintained: A satisfying assignment for c_j
satisfies all $c'_{j,\cdot}$ and vice versa.

Let $\boldsymbol{X} = (X, C)$ be the SAT instance, and $\boldsymbol{X'} = (X', C')$ the resulting 3-SAT instance.

The underlying principle of this approach is called *local replacement*.

3-clauses

Let $c_j = z_1 \vee z_2 \vee z_3$, where z_i, $1 \leq i \leq 3$, is a literal x_{k_i} or $\overline{x_{k_i}}$.

Then we copy that clause, i.e., $c'_{j1} = c_j$.

Consider a truth assignment to X which makes c_j true. Then **all** extensions of it to X' will make **all** $c'_{j,\cdot}$ (actually there is only one) true.

Every truth assignment to X' making all $c'_{j,\cdot}$ true makes c_j true.

2-clauses

Let $c_j = z_1 \vee z_2$, , where z_i, $1 \leq i \leq 2$, is a literal $x_{x_{k_i}}$ or $\overline{x_{k_i}}$. We introduce one new variable $y_{j,1}$ and define two new clauses:

$$\begin{aligned} c'_{j,1} &= z_1 \vee z_2 \vee y_{j,1} \\ c'_{j,2} &= z_1 \vee z_2 \vee \overline{y}_{j,1} \end{aligned}$$

Consider a truth assignment to X which makes c_j true. Then **all** extensions of it to X' will make **all** $c'_{j,\cdot}$ true.

Every truth assignment to X' making all $c'_{j,\cdot}$ true makes c_j true.

1-clauses

Let $c_j = z_1$, where z_1 is a literal x_{k_1} or $\overline{x_{k_1}}$. We introduce two new variables $y_{j,1}$ and $y_{j,2}$ and define four new clauses:

$$\begin{aligned} c'_{j,1} &= z_1 \vee y_{j,1} \vee y_{j,2} \\ c'_{j,2} &= z_1 \vee \overline{y}_{j,1} \vee y_{j,2} \\ c'_{j,3} &= z_1 \vee y_{j,1} \vee \overline{y}_{j,2} \\ c'_{j,4} &= z_1 \vee \overline{y}_{j,1} \vee \overline{y}_{j,2} \end{aligned}$$

Consider a truth assignment to X which makes c_j true. Then **all** extensions of it to X' will make **all** $c'_{j,\cdot}$ true.

Every truth assignment to X' making all $c'_{j,\cdot}$ true makes c_j true.

(> 3)-clauses

Let $c_j = z_1 \vee z_2 \vee \cdots \vee z_m$, $m > 3$. We introduce $m - 3$ new variables $y_{j,1}, y_{j,2}, \ldots, y_{j,m-3}$, and define $m - 2$ clauses:

$$
\begin{array}{lcllllll}
c'_{j,1} & = & z_1 & \vee & z_2 & \vee & y_{j,1} \\
c'_{j,2} & = & \overline{y}_{j,1} & \vee & z_3 & \vee & y_{j,2} \\
c'_{j,3} & = & \overline{y}_{j,2} & \vee & z_4 & \vee & y_{j,3} \\
c'_{j,4} & = & \overline{y}_{j,3} & \vee & z_5 & \vee & y_{j,4} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
c'_{j,i} & = & \overline{y}_{j,i-1} & \vee & z_{i+1} & \vee & y_{j,i} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
c'_{j,m-3} & = & \overline{y}_{j,m-4} & \vee & z_{m-2} & \vee & y_{j,m-3} \\
c'_{j,m-2} & = & \overline{y}_{j,m-3} & \vee & z_{m-1} & \vee & z_m
\end{array}
$$

(> 3)-clauses

Consider a truth assignment to X which satisfies

$$c_j = z_1 \vee z_2 \vee \cdots \vee z_m$$

We show that there is a truth assignment to X' satisfying all $c'_{j,\cdot}$.

At least one of the literals in c_j is true, say z_{i+1}.

Then the clause $c'_{j,i}$ of C' is satisfied:

$$c'_{j,i} = \overline{y}_{j,i-1} \vee z_{i+1} \vee y_{j,i}$$

(> 3)-clauses

Now consider a clause $c'_{j,k}$ with $k < i$.

$$c'_{j,k} = \overline{y}_{j,k-1} \vee z_{k+1} \vee y_{j,k}$$

If we set $y_{j,k} = \text{true}$ then $c'_{j,k}$ is satisfied.

Next consider a clause $c'_{j,k}$ with $k > i$.

$$c'_{j,k} = \overline{y}_{j,k-1} \vee z_{k+1} \vee y_{j,k}$$

If we set $\overline{y}_{j,k-1} = \text{true}$ then $c'_{j,k}$ is satisfied.

(> 3)-clauses

$$c_j = z_1 \vee \ldots \vee z_{i+1} \vee \cdots \vee z_m$$

$$\begin{array}{lcllcl}
c'_{j,1} & = & z_1 & \vee\ z_2 & \vee & y_{j,1} \\
c'_{j,2} & = & \overline{y}_{j,1} & \vee\ z_3 & \vee & y_{j,2} \\
\vdots & \vdots & \vdots & \vdots\ \vdots & \vdots & \vdots \\
c'_{j,i-1} & = & \overline{y}_{j,i-2} & \vee\ z_i & \vee & y_{j,i-1} \\
c'_{j,i} & = & \overline{y}_{j,i-1} & \vee\ z_{i+1} & \vee & y_{j,i} \\
c'_{j,i+1} & = & \overline{y}_{j,i} & \vee\ z_{i+2} & \vee & y_{j,i+1} \\
\vdots & \vdots & \vdots & \vdots\ \vdots & \vdots & \vdots \\
c'_{j,m-3} & = & \overline{y}_{j,m-4} & \vee\ z_{m-2} & \vee & y_{j,m-3} \\
c'_{j,m-2} & = & \overline{y}_{j,m-3} & \vee\ z_{m-1} & \vee & z_m
\end{array}$$

Green literal true by assumption, red literals set true.

(> 3)-clauses

Consider a truth assignment to X' satisfying all $c'_{j,\cdot}$.

We claim that this implies that at least one literal z_k has to be true, whence c_j is satisfied.

For a contradiction assume that

$$z_1 = z_2 = \cdots = z_m = \text{false}.$$

(> 3)-clauses

Look at the clauses:

$$
\begin{array}{lcllllll}
c'_{j,1} & = & z_1 & \vee & z_2 & \vee & y_{j,1} \\
c'_{j,2} & = & \overline{y}_{j,1} & \vee & z_3 & \vee & y_{j,2} \\
c'_{j,3} & = & \overline{y}_{j,2} & \vee & z_4 & \vee & y_{j,3} \\
c'_{j,4} & = & \overline{y}_{j,3} & \vee & z_5 & \vee & y_{j,4} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
c'_{j,m-3} & = & \overline{y}_{j,m-4} & \vee & z_{m-2} & \vee & y_{j,m-3} \\
c'_{j,m-2} & = & \overline{y}_{j,m-3} & \vee & z_{m-1} & \vee & z_m
\end{array}
$$

Literals shown in red are false.

(> 3)-clauses

Clause $c'_{j,1}$ is true, thus we know $y_{j,1} = \text{true}$.

$$
\begin{array}{lcllllll}
c'_{j,1} & = & z_1 & \vee & z_2 & \vee & y_{j,1} \\
c'_{j,2} & = & \overline{y}_{j,1} & \vee & z_3 & \vee & y_{j,2} \\
c'_{j,3} & = & \overline{y}_{j,2} & \vee & z_4 & \vee & y_{j,3} \\
c'_{j,4} & = & \overline{y}_{j,3} & \vee & z_5 & \vee & y_{j,4} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
c'_{j,m-3} & = & \overline{y}_{j,m-4} & \vee & z_{m-2} & \vee & y_{j,m-3} \\
c'_{j,m-2} & = & \overline{y}_{j,m-3} & \vee & z_{m-1} & \vee & z_m
\end{array}
$$

Literals shown in red are false those in green are true.

(> 3)-clauses

Clause $c'_{j,2}$ is true, thus we know $y_{j,2} = \text{true}$.

$$
\begin{array}{lcllllll}
c'_{j,1} & = & z_1 & \vee & z_2 & \vee & y_{j,1} \\
c'_{j,2} & = & \overline{y}_{j,1} & \vee & z_3 & \vee & y_{j,2} \\
c'_{j,3} & = & \overline{y}_{j,2} & \vee & z_4 & \vee & y_{j,3} \\
c'_{j,4} & = & \overline{y}_{j,3} & \vee & z_5 & \vee & y_{j,4} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
c'_{j,m-3} & = & \overline{y}_{j,m-4} & \vee & z_{m-2} & \vee & y_{j,m-3} \\
c'_{j,m-2} & = & \overline{y}_{j,m-3} & \vee & z_{m-1} & \vee & z_m
\end{array}
$$

(> 3)-clauses

General $c'_{j,i}$ is true, thus we know $y_{j,i} = \text{true}$.

$$
\begin{array}{lcllllll}
c'_{j,1} & = & z_1 & \vee & z_2 & \vee & y_{j,1} \\
c'_{j,2} & = & \overline{y}_{j,1} & \vee & z_3 & \vee & y_{j,2} \\
c'_{j,3} & = & \overline{y}_{j,2} & \vee & z_4 & \vee & y_{j,3} \\
c'_{j,4} & = & \overline{y}_{j,3} & \vee & z_5 & \vee & y_{j,4} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
c'_{j,m-3} & = & \overline{y}_{j,m-4} & \vee & z_{m-2} & \vee & y_{j,m-3} \\
c'_{j,m-2} & = & \overline{y}_{j,m-3} & \vee & z_{m-1} & \vee & z_m
\end{array}
$$

Then the last clause $c'_{j,m-2}$ is not satisfied,
contrary to our assumption.

Hence the assumption that all z_i are false is not correct.

(> 3)-clauses, Summary

We have shown:

- An assignment satisfying c_j can be extended to one satisfying all $c'_{j,\cdot}$.
- An assignment satisfying all $c'_{j,\cdot}$ satisfies c_j.

Note that a new variable $y_{j,\cdot}$ only appears in clauses $c'_{j,\cdot}$.

Time Analysis

The transformation can be performed in polynomial time:

EXERCISE

Iff

We show that:
The answer to $\boldsymbol{X}$ is YES if and only if the answer to $\boldsymbol{X'}$ is YES.

We prove this in two steps

The "only if" ($\Rightarrow$) part: The answer to $\boldsymbol{X}$ is YES $\Rightarrow$ The answer to $\boldsymbol{X'}$ is YES.

The " if" ($\Leftarrow$) part: The answer to $\boldsymbol{X'}$ is YES $\Rightarrow$ The answer to $\boldsymbol{X}$ is YES.

Only if ($\Rightarrow$)

- The answer to $\boldsymbol{X}$ is YES.
- Then there is a truth assignment to X which satisfies all clauses in C.
- We have just shown that for every clause c_j in C such an assignment can be extended to one to X' which satisfies all clauses $c'_{j,\cdot}$ in C'.
- The extension is by appropriately setting the truth values of the helper variables $y_{j,\cdot}$.
- For different clauses c_j and c_s the sets of helper variables are distinct.
- Hence a satisfying assignment for X can be extended to one for X', i. e., one that satisfies all clauses in C'.
- Hence, the answer to $\boldsymbol{X'}$ is YES.

If ($\Leftarrow$)

- The answer to $\boldsymbol{X'}$ is YES.
- Then there is a truth assignment to X' which satisfies all clauses in C'.
- We have just shown that if (for fixed j) all 3-clauses $c'_{j,\cdot}$ are satisfied then so is c_j.
- Hence the assignment satisfies all clauses in C.
- Hence, the answer to $\boldsymbol{X}$ is YES.

Summarizing

We have

- constructed a transformation which converts SAT instances into 3-SAT instances
- shown that the transformation runs in polynomial time.
- shown that the SAT instance has answer YES
  if and only if the transformed 3-SAT instance has.
- This establishes

$$\text{SAT} \leq_p \text{3-SAT} .$$

- It is easy to show that $\text{3-SAT} \in \mathcal{NP}$.
- Altogether this establishes that 3-SAT is $\mathcal{NP}$-complete.