NP and NP Completeness

Lecture 24

April 24, 2025

Part I

Efficient Computation: P and NP

What is Efficiency?

Last lecture, we discussed what problems can't have *any* algorithm—what changes if we care about *efficient* algorithms?

What is Efficiency?

Last lecture, we discussed what problems can't have *any* algorithm—what changes if we care about *efficient* algorithms?

Definition

We say that a language L is in P if there exists an algorithm M that decides L, where for some constant c, $M(x)$ runs in time $O(|x|^c)$.

(Informally: P is the set of languages with polynomial-time algorithms.)

What is Efficiency?

Last lecture, we discussed what problems can't have *any* algorithm—what changes if we care about *efficient* algorithms?

Definition

We say that a language L is in P if there exists an algorithm M that decides L, where for some constant c, $M(x)$ runs in time $O(|x|^c)$.

(Informally: P is the set of languages with polynomial-time algorithms.)

Why do we allow for *any* polynomial run time?

What is Efficiency?

Last lecture, we discussed what problems can't have *any* algorithm—what changes if we care about *efficient* algorithms?

Definition

We say that a language L is in P if there exists an algorithm M that decides L, where for some constant c, $M(x)$ runs in time $O(|x|^c)$.

(Informally: P is the set of languages with polynomial-time algorithms.)

Why do we allow for *any* polynomial run time?

- Makes it simpler to describe algorithms.
- Polynomials have helpful closure properties: if $p(n)$ and $q(n)$ are polynomials, so are $p(n) + q(n)$, $p(n) \cdot q(n)$, and $p(q(n))$.
- We are interested in finding problems that *can't* be solved efficiently, so having a lax definition is more meaningful!

NP: Efficient Verification

Definition

We say that a language L is in NP if there is a polynomial $p(\cdot)$ and a machine M (running in time $O(|x|^c)$) such that:

- For every $x \in L$, there is a $w \in \{0,1\}^{p(|x|)}$ such that $M(x, w)$ accepts.
- For every $x \notin L$ and every $w \in \{0,1\}^{p(|x|)}$, $M(x, w)$ rejects.

NP: Efficient Verification

Definition

We say that a language L is in NP if there is a polynomial $p(\cdot)$ and a machine M (running in time $O(|x|^c)$) such that:

- For every $x \in L$, there is a $w \in \{0,1\}^{p(|x|)}$ such that $M(x,w)$ accepts.
- For every $x \notin L$ and every $w \in \{0,1\}^{p(|x|)}$, $M(x,w)$ rejects.

Intuitively, L is in NP if it is easy to verify a proof that $x \in L$.

NP: Efficient Verification

Definition

We say that a language L is in NP if there is a polynomial $p(\cdot)$ and a machine M (running in time $O(|x|^c)$) such that:

- For every $x \in L$, there is a $w \in \{0,1\}^{p(|x|)}$ such that $M(x,w)$ accepts.
- For every $x \notin L$ and every $w \in \{0,1\}^{p(|x|)}$, $M(x,w)$ rejects.

Intuitively, L is in NP if it is easy to verify a proof that $x \in L$.

Examples we've already seen:

- Independent Set: w describes an IS of size k.
- Clique: w describes a clique of size k.

Why NP?

NP captures “most” problems we run into in the wild.

(Notable exception: the halting problem is *not* in ***NP***!)

Why NP?

NP captures "most" problems we run into in the wild.

(Notable exception: the halting problem is *not* in ***NP***!)

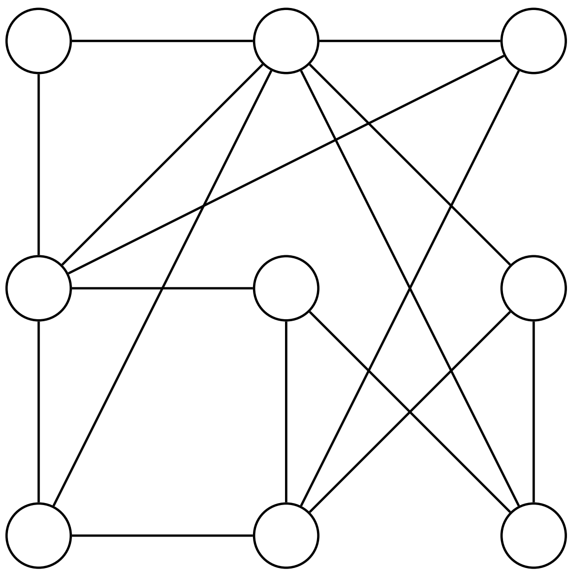
We *think* that not all problems in ***NP*** can be solved efficiently: verifying answers seems easier than coming up with them!

Why NP?

NP captures "most" problems we run into in the wild.

(Notable exception: the halting problem is *not* in ***NP***!)

We *think* that not all problems in ***NP*** can be solved efficiently: verifying answers seems easier than coming up with them!

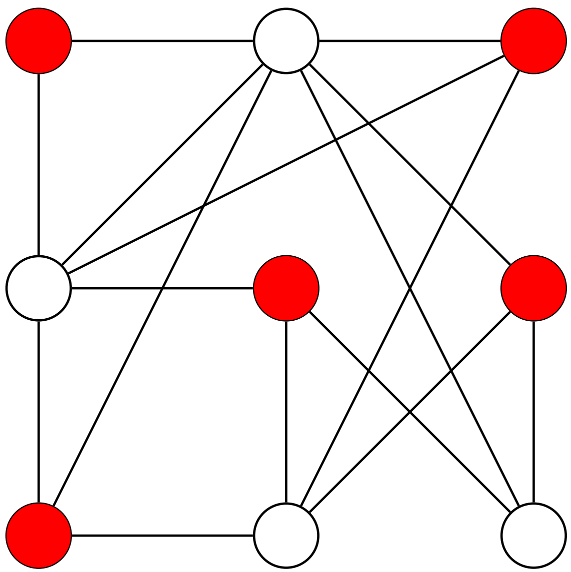


Why NP?

NP captures "most" problems we run into in the wild.

(Notable exception: the halting problem is *not* in ***NP***!)

We *think* that not all problems in ***NP*** can be solved efficiently: verifying answers seems easier than coming up with them!

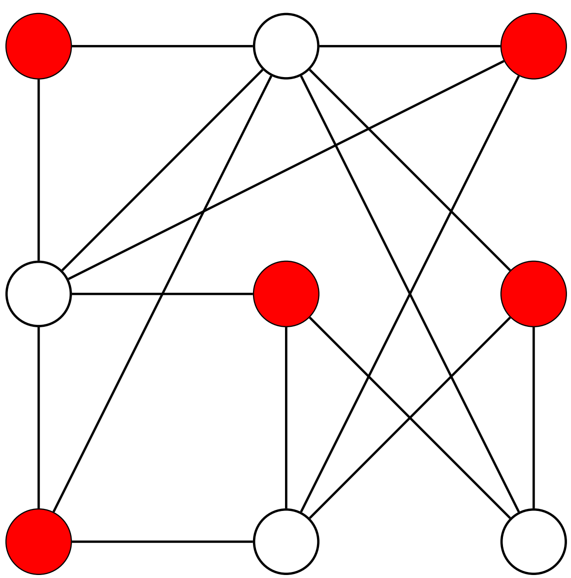


Why NP?

NP captures "most" problems we run into in the wild.

(Notable exception: the halting problem is *not* in ***NP***!)

We *think* that not all problems in ***NP*** can be solved efficiently: verifying answers seems easier than coming up with them!



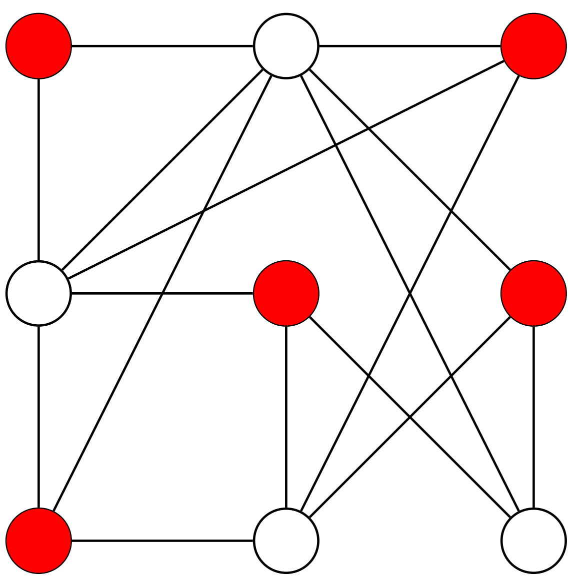
8		6				5		
	9						1	
			6					
			2		7	6	8	
7				4				
							5	9
	7				6			
	3			1			2	5
2			8		9	3		

Why NP?

NP captures “most” problems we run into in the wild.

(Notable exception: the halting problem is *not* in ***NP***!)

We *think* that not all problems in ***NP*** can be solved efficiently: verifying answers seems easier than coming up with them!



8	4	6	9	3	1	5	7	2
5	9	3	4	7	2	8	1	6
1	2	7	6	8	5	4	9	3
3	5	1	2	9	7	6	8	4
7	6	9	5	4	8	2	3	1
4	8	2	1	6	3	7	5	9
9	7	5	3	2	6	1	4	8
6	3	8	7	1	4	9	2	5
2	1	4	8	5	9	3	6	7

P Versus NP

By definition, we have that $\boldsymbol{P} \subseteq \boldsymbol{NP}$.

P Versus NP

By definition, we have that $\boldsymbol{P} \subseteq \boldsymbol{NP}$.

Major open question: does $\boldsymbol{P} = \boldsymbol{NP}$?

P Versus NP

By definition, we have that $\boldsymbol{P} \subseteq \boldsymbol{NP}$.

Major open question: does $\boldsymbol{P} = \boldsymbol{NP}$?

- "Most" computer scientists conjecture no, but so far we can only prove that certain proof techniques aren't enough to show this!

P Versus NP

By definition, we have that $\boldsymbol{P} \subseteq \boldsymbol{NP}$.

Major open question: does $\boldsymbol{P} = \boldsymbol{NP}$?

- "Most" computer scientists conjecture no, but so far we can only prove that certain proof techniques aren't enough to show this!
- For 374, we will assume $\boldsymbol{P} \neq \boldsymbol{NP}$ unless otherwise stated, so *some* problem in $\boldsymbol{NP}$ cannot be solved efficiently.

P Versus NP

By definition, we have that $\boldsymbol{P} \subseteq \boldsymbol{NP}$.

Major open question: does $\boldsymbol{P} = \boldsymbol{NP}$?

- "Most" computer scientists conjecture no, but so far we can only prove that certain proof techniques aren't enough to show this!
- For 374, we will assume $\boldsymbol{P} \neq \boldsymbol{NP}$ unless otherwise stated, so *some* problem in $\boldsymbol{NP}$ cannot be solved efficiently.

Can we come up with a *specific* language that isn't in $\boldsymbol{P}$?

P Versus NP

By definition, we have that $\boldsymbol{P} \subseteq \boldsymbol{NP}$.

Major open question: does $\boldsymbol{P} = \boldsymbol{NP}$?

- "Most" computer scientists conjecture no, but so far we can only prove that certain proof techniques aren't enough to show this!
- For 374, we will assume $\boldsymbol{P} \neq \boldsymbol{NP}$ unless otherwise stated, so *some* problem in $\boldsymbol{NP}$ cannot be solved efficiently.

Can we come up with a *specific* language that isn't in $\boldsymbol{P}$?

Idea: if we can reduce *every* language in $\boldsymbol{NP}$ to some specific language $\boldsymbol{L}$, then we know $\boldsymbol{L}$ in particular is not in $\boldsymbol{P}$!

NP-Hard Languages

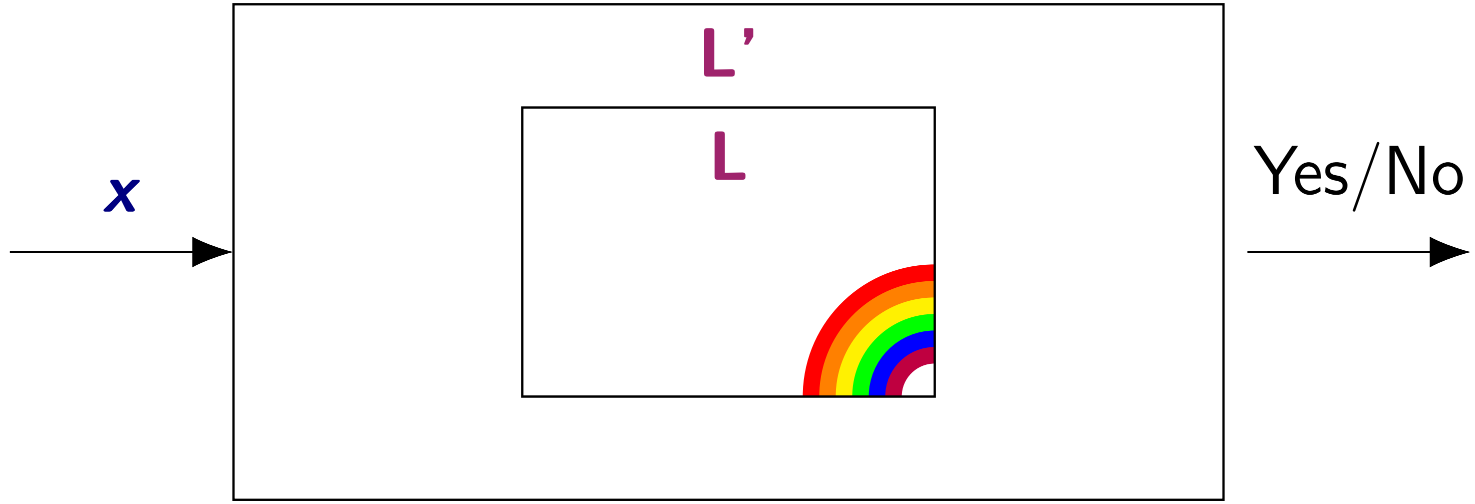
Definition

We say that $\boldsymbol{L}$ is $\boldsymbol{NP}$-Hard if for every $\boldsymbol{L'} \in \boldsymbol{NP}$, there is a *polynomial-time* reduction from $\boldsymbol{L'}$ to $\boldsymbol{L}$.

NP-Hard Languages

Definition

We say that L is NP-Hard if for every $L' \in NP$, there is a *polynomial-time* reduction from L' to L.



Requirement: as long as "magic box" for L runs in polynomial time, so does the "box" we build for L'.

NP-Hard Example

Claim

$L_{HNI} = \{\langle M \rangle \mid M \text{ halts given no input}\}$ *is* ***NP****-Hard.*

NP-Hard Example

Claim

$L_{HNI} = \{\langle M \rangle \mid M \textit{ halts given no input}\}$ *is* $\boldsymbol{NP}$*-Hard.*

Let L be a language in NP, with M as the machine that verifies solutions and $p(\cdot)$ the polynomial such that $w \in \{0, 1\}^{p(|x|)}$.

NP-Hard Example

Claim

$L_{HNI} = \{\langle M \rangle \mid M \text{ halts given no input}\}$ is NP-Hard.

Let L be a language in NP, with M as the machine that verifies solutions and $p(\cdot)$ the polynomial such that $w \in \{0,1\}^{p(|x|)}$.

```
from magic import TestHNI
DecideL(x):
    Construct a program (machine) P() that:
        For each w ∈ {0,1}^p(|x|), runs M(x, w)
        If any iteration accepts, halt; else infinite loop
    return TestHNI(⟨P⟩)
```

NP-Hard Example

Claim

$L_{HNI} = \{\langle M \rangle \mid M$ halts given no input$\}$ is NP-Hard.

Let L be a language in NP, with M as the machine that verifies solutions and $p(\cdot)$ the polynomial such that $w \in \{0,1\}^{p(|x|)}$.

```
from magic import TestHNI
DecideL(x):
    Construct a program (machine) P() that:
        For each w ∈ {0,1}^{p(|x|)}, runs M(x, w)
        If any iteration accepts, halt; else infinite loop
    return TestHNI(⟨P⟩)
```

Better question: is there an NP-hard problem *that is also in NP*? (We call such problems NP-complete.)

Part II

SAT

Boolean Satisfiablility (SAT)

Consider a boolean formula using AND, OR, and NOT:

$$((a \vee b \vee \overline{c}) \wedge \overline{(b \vee c)}) \vee d$$

Boolean Satisfiablility (SAT)

Consider a boolean formula using AND, OR, and NOT:

$$((a \vee b \vee \overline{c}) \wedge \overline{(b \vee c)}) \vee d$$

Is there an assignment of True/False to a, b, c, and d such that this formula evaluates to True?

all variables to True except d is False

$$\left(\underbrace{\underbrace{(T \vee T \vee \overline{F})}_{T} \wedge \underbrace{\overline{(T \vee T)}}_{F}}_{F}\right) \vee \overset{F}{\cancel{\overline{T}}}$$

$\cancel{\overline{T}}\ F$

Boolean Satisfiablility (SAT)

Consider a boolean formula using AND, OR, and NOT:

$$((a \vee b \vee \overline{c}) \wedge \overline{(b \vee c)}) \vee d$$

Is there an assignment of True/False to a, b, c, and d such that this formula evaluates to True?

What about $a \wedge (\overline{a} \vee b) \wedge (\overline{a} \vee \overline{b})$?

Boolean Satisfiablility (SAT)

Consider a boolean formula using AND, OR, and NOT:

$$((a \vee b \vee \overline{c}) \wedge \overline{(b \vee c)}) \vee d$$

Is there an assignment of True/False to a, b, c, and d such that this formula evaluates to True?

What about $a \wedge (\overline{a} \vee b) \wedge (\overline{a} \vee \overline{b})$?

Definition

Let $SAT = \{\varphi \mid \varphi \text{ is satisfiable}\}$

Note $SAT \in NP$: we can take w to be an assignment of True/False to each variable!

Using SAT

SAT turns out to be very powerful for modeling other problems!

Example: does $G = (V, E)$ have a *path* that visits every vertex?

"proof": sequence of vertices that the path visits.

model as SAT: create variables u_i for each $u \in V$ and $i \in \{1, \dots, n\}$ to represent "is vertex u the ith vertex in the path?"

① every vertex has some index: $(u_1 \vee u_2 \vee \dots \vee u_n)$

② no vertex appears twice: $\forall i \neq j \; \overline{(u_i \wedge u_j)}$

③ always using an edge: $\forall (u,v) \notin E \; \forall i \in \{1, \dots n-1\} \; \overline{(u_i \wedge v_{i+1})}$

(4) no two vertices are assigned the same index.

$$\forall u \neq v \quad \forall i \in \{1, \dots, n\} \quad \overline{(u_i \wedge v_i)}$$

The Cook-Levin Theorem

Theorem (Cook-Levin)

SAT *is* ***NP****-complete.*

The Cook-Levin Theorem

Theorem (Cook-Levin)

SAT *is* ***NP****-complete.*

Turns out all the formulas Cook-Levin constructs are all in "Conjunctive Normal Form":

- Formula is the AND of many clauses.
- Each clause is the OR of many variables/negations of variables. literal
- Example: $(a \lor \overline{b} \lor c) \land (b \lor d) \land (\overline{a} \lor b \lor c \lor \overline{d})$

The Cook-Levin Theorem

Theorem (Cook-Levin)

SAT *is* ***NP****-complete.*

Turns out all the formulas Cook-Levin constructs are all in "Conjunctive Normal Form":

- Formula is the AND of many clauses.
- Each clause is the OR of many variables/negations of variables.
- Example: $(a \lor \overline{b} \lor c) \land (b \lor d) \land (\overline{a} \lor b \lor c \lor \overline{d})$

Theorem (Cook-Levin, stronger version)

CNF-SAT $= \{\varphi \mid \varphi$ *is satisfiable and in CNF*$\}$ *is* ***NP****-complete.*

The Cook-Levin Theorem

Theorem (Cook-Levin)

$\mathbf{SAT}$ is $\mathbf{NP}$-complete.

Turns out all the formulas Cook-Levin constructs are all in "Conjunctive Normal Form":

- Formula is the AND of many clauses.
- Each clause is the OR of many variables/negations of variables.
- Example: $(a \vee \overline{b} \vee c) \wedge (b \vee d) \wedge (\overline{a} \vee b \vee c \vee \overline{d})$

Theorem (Cook-Levin, stronger version)

$\mathbf{CNF\text{-}SAT} = \{\varphi \mid \varphi \text{ is satisfiable and in CNF}\}$ is $\mathbf{NP}$-complete.

This means that (assuming $\mathbf{P} \neq \mathbf{NP}$) there is no polynomial time algorithm for $\mathbf{SAT}$ nor $\mathbf{CNF\text{-}SAT}$!

Part III

3SAT

Using Reductions

How do we prove that more problems are ***NP***-complete?

(Without redoing Cook-Levin...)

Using Reductions

How do we prove that more problems are ***NP***-complete?

(Without redoing Cook-Levin...)

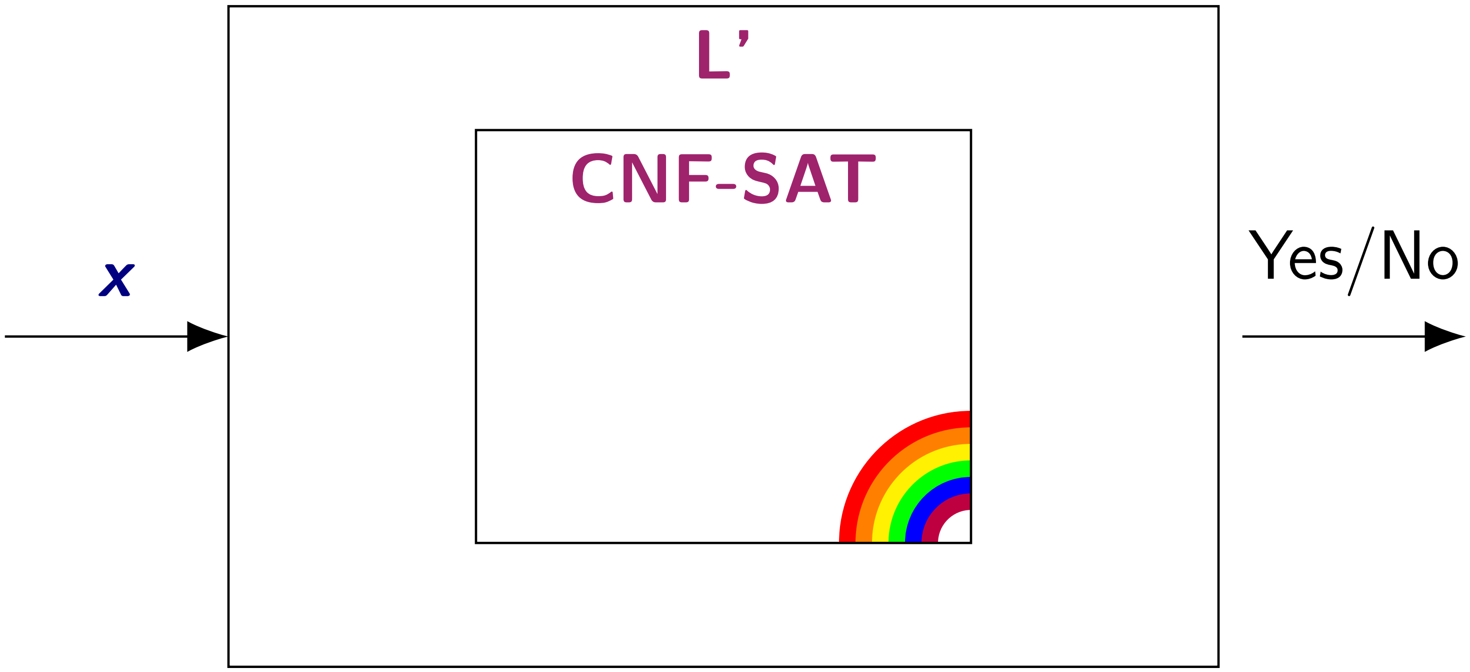
Idea: If ***CNF-SAT*** reduces to $\boldsymbol{L}$ in polynomial time, then $\boldsymbol{L}$ is ***NP***-hard! (So ***NP***-complete as long as $\boldsymbol{L} \in \boldsymbol{NP}$)

Using Reductions

How do we prove that more problems are ***NP***-complete?

(Without redoing Cook-Levin...)

Idea: If ***CNF-SAT*** reduces to $\boldsymbol{L}$ in polynomial time, then $\boldsymbol{L}$ is ***NP***-hard! (So ***NP***-complete as long as $\boldsymbol{L} \in \boldsymbol{NP}$)

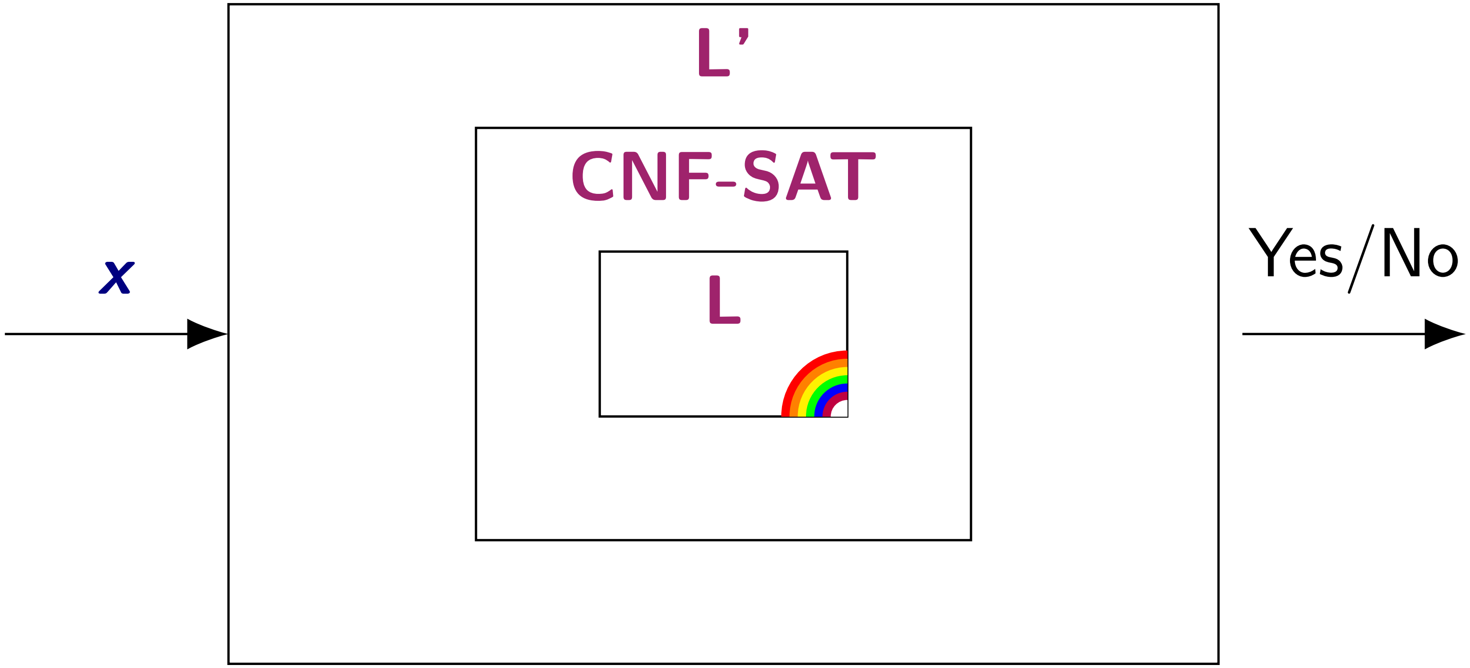


Using Reductions

How do we prove that more problems are ***NP***-complete?

(Without redoing Cook-Levin...)

Idea: If ***CNF-SAT*** reduces to L in polynomial time, then L is ***NP***-hard! (So ***NP***-complete as long as $L \in NP$)



Using Reductions

How do we prove that more problems are $\boldsymbol{NP}$-complete?

(Without redoing Cook-Levin...)

Idea: If $\boldsymbol{CNF\text{-}SAT}$ reduces to $\boldsymbol{L}$ in polynomial time, then $\boldsymbol{L}$ is $\boldsymbol{NP}$-hard! (So $\boldsymbol{NP}$-complete as long as $\boldsymbol{L} \in \boldsymbol{NP}$)

Common form of reduction: give function $\boldsymbol{f}$ (that can be computed in polynomial time) so that $\boldsymbol{x} \in \boldsymbol{CNF\text{-}SAT}$ iff $\boldsymbol{f(x)} \in \boldsymbol{L}$.

CNF-SAT to 3SAT I

A boolean formula is 3CNF if it is CNF and each clause has three literals. Let ***3SAT*** be the language of satisfiable 3CNF formulas.

CNF-SAT to 3SAT I

A boolean formula is 3CNF if it is CNF and each clause has three literals. Let **3SAT** be the language of satisfiable 3CNF formulas.

Claim

3SAT *is* ***NP****-complete.*

CNF-SAT to 3SAT I

A boolean formula is 3CNF if it is CNF and each clause has three literals. Let $\mathbf{3SAT}$ be the language of satisfiable 3CNF formulas.

Claim

$\mathbf{3SAT}$ is $\mathbf{NP}$-complete.

$\mathbf{3SAT}$ is in $\mathbf{NP}$: $\boldsymbol{w}$ is a satisfying assignment to the variables.

CNF-SAT to 3SAT I

A boolean formula is 3CNF if it is CNF and each clause has three literals. Let ***3SAT*** be the language of satisfiable 3CNF formulas.

Claim

3SAT *is* ***NP****-complete.*

3SAT is in ***NP***: ***w*** is a satisfying assignment to the variables.

Given a CNF formula, want to make each clause have 3 literals.

- How do we fix clauses with too few? (eg $(a \vee \overline{b})$)

create a new variable c $(a \vee \bar{b} \vee c) \wedge (a \vee \bar{b} \vee \bar{c})$

- How do we fix clauses with too many? (eg $(a \vee \overline{b} \vee c \vee \overline{d})$)

create a new variable e $(a \vee \bar{b} \vee e) \wedge (\bar{e} \vee c \vee \bar{d})$

CNF-SAT to 3SAT II

Claim

3SAT *is* ***NP****-complete.*

Given a CNF formula φ, create $\boldsymbol{f(\varphi)}$ by for every $\boldsymbol{C} \in \varphi$:

CNF-SAT to 3SAT II

Claim

$\mathbf{3SAT}$ is $\mathbf{NP}$-complete.

Given a CNF formula φ, create $f(\varphi)$ by for every $C \in \varphi$:

- If $C = (\ell_1)$ has one literal, include clauses $(\ell_1 \vee x_C \vee y_C)$, $(\ell_1 \vee \overline{x_C} \vee y_C)$, $(\ell_1 \vee x_C \vee \overline{y_C})$, and $(\ell_1 \vee \overline{x_C} \vee \overline{y_C})$ in $f(\varphi)$, where x_C and y_C are new variables.

CNF-SAT to 3SAT II

Claim

$\mathbf{3SAT}$ is $\mathbf{NP}$-complete.

Given a CNF formula φ, create $f(\varphi)$ by for every $C \in \varphi$:

- If $C = (\ell_1)$ has one literal, include clauses $(\ell_1 \vee x_C \vee y_C)$, $(\ell_1 \vee \overline{x_C} \vee y_C)$, $(\ell_1 \vee x_C \vee \overline{y_C})$, and $(\ell_1 \vee \overline{x_C} \vee \overline{y_C})$ in $f(\varphi)$, where x_C and y_C are new variables.
- If $C = (\ell_1 \vee \ell_2)$ has two literals, include clauses $(\ell_1 \vee \ell_2 \vee x_C)$ and $(\ell_1 \vee \ell_2 \vee \overline{x_C})$ in $f(\varphi)$, where x_C is a new variable.

CNF-SAT to 3SAT II

Claim

$3SAT$ is NP-complete.

Given a CNF formula φ, create $f(\varphi)$ by for every $C \in \varphi$:

- If $C = (\ell_1)$ has one literal, include clauses $(\ell_1 \vee x_C \vee y_C)$, $(\ell_1 \vee \overline{x_C} \vee y_C)$, $(\ell_1 \vee x_C \vee \overline{y_C})$, and $(\ell_1 \vee \overline{x_C} \vee \overline{y_C})$ in $f(\varphi)$, where x_C and y_C are new variables.
- If $C = (\ell_1 \vee \ell_2)$ has two literals, include clauses $(\ell_1 \vee \ell_2 \vee x_C)$ and $(\ell_1 \vee \ell_2 \vee \overline{x_C})$ in $f(\varphi)$, where x_C is a new variable.
- If C has three literals, include C in $f(\varphi)$

CNF-SAT to 3SAT II

Claim

$\mathbf{3SAT}$ is $\mathbf{NP}$-complete.

Given a CNF formula φ, create $f(\varphi)$ by for every $C \in \varphi$:

- If $C = (\ell_1)$ has one literal, include clauses $(\ell_1 \vee x_C \vee y_C)$, $(\ell_1 \vee \overline{x_C} \vee y_C)$, $(\ell_1 \vee x_C \vee \overline{y_C})$, and $(\ell_1 \vee \overline{x_C} \vee \overline{y_C})$ in $f(\varphi)$, where x_C and y_C are new variables.
- If $C = (\ell_1 \vee \ell_2)$ has two literals, include clauses $(\ell_1 \vee \ell_2 \vee x_C)$ and $(\ell_1 \vee \ell_2 \vee \overline{x_C})$ in $f(\varphi)$, where x_C is a new variable.
- If C has three literals, include C in $f(\varphi)$
- If $C = (\ell_1 \vee \ldots \vee \ell_k)$ has $k \geq 4$ literals, include clauses $(\ell_1 \vee \ell_2 \vee x_{C1})$, $(\overline{x_{C1}} \vee \ell_3 \vee x_{C2})$, ..., $(\overline{x_{C(k-3)}} \vee \ell_{k-1} \vee \ell_k)$ in $f(\varphi)$, where $(x_{C1}, \ldots, x_{C(k-3)})$ are new variables.

CNF-SAT to 3SAT III

Claim

3SAT *is* ***NP****-complete.*

Our construction of f clearly runs in polynomial time. (In fact, quadratic.)

CNF-SAT to 3SAT III

Claim

$\mathbf{3SAT}$ is $\mathbf{NP}$-complete.

Our construction of f clearly runs in polynomial time. (In fact, quadratic.)

Exercise: formally prove that φ is satisfiable if and only if $f(\varphi)$ is.

- If direction: given a satisfiable assignment for $f(\varphi)$, find a satisfiable assignment for φ
- Only if direction: given a satisfiable assignment for φ, find a satisfiable assignment for $f(\varphi)$

Takeaway Points

Definitions of $\boldsymbol{P}$ and $\boldsymbol{NP}$.

- If $L \in P$, we can efficiently decide if $x \in L$.
- If $L \in NP$, we can efficiently verify proofs that $x \in L$.
- We will assume that $P \neq NP$.

$\boldsymbol{NP}$-hardness and $\boldsymbol{NP}$-completeness

- A problem is $\boldsymbol{NP}$-hard if *every* problem in $\boldsymbol{NP}$ reduces to it. If it is also in $\boldsymbol{NP}$ itself, we call it $\boldsymbol{NP}$-complete.
- If you can reduce an $\boldsymbol{NP}$-hard problem to L in polynomial time, L is also $\boldsymbol{NP}$-hard.

Known $\boldsymbol{NP}$-complete languages

- ***SAT***
- ***CNF-SAT***
- ***3SAT***