ALGORITHMS AND DATA STRUCTURES · LECTURE 19

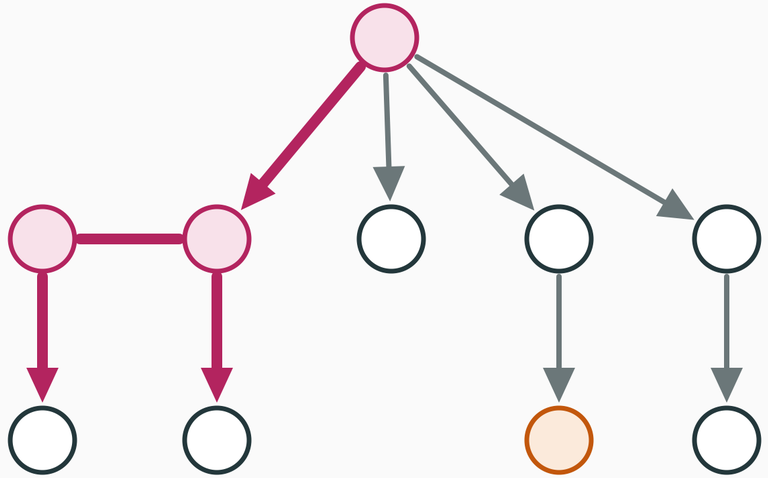
NP-completeness

Non-examinable

Heng Guo

School of Informatics, University of Edinburgh

Monday 23 November 2026 · 16:10–17:00



The problem we could not solve fast

Efficient algorithms: polynomial time

- sorting
- matrix multiplication
- polynomial multiplication
- minimum spanning tree
- maximum flow
- bipartite matching
- linear programming
- longest common subsequence

No efficient algorithm known

- subset sum

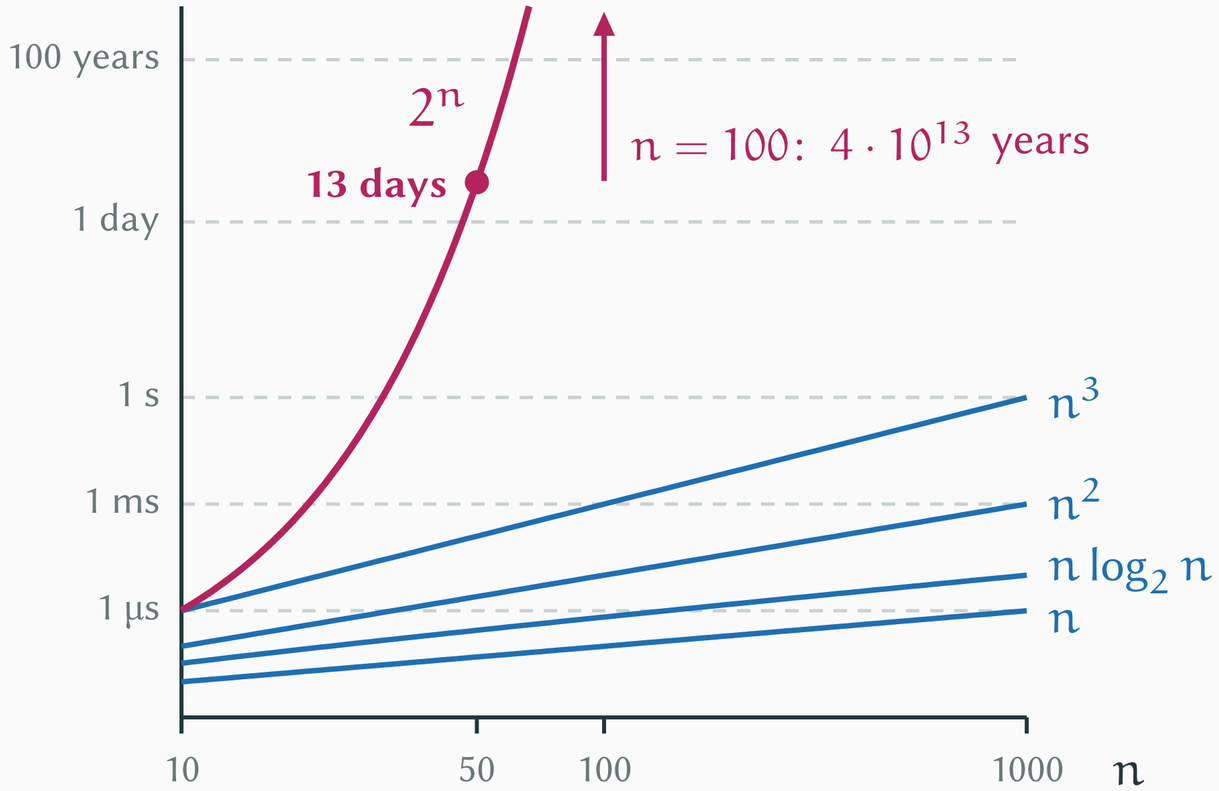
$O(nW)$ is only **pseudo-polynomial**:
polynomial in the **value** of W,
not in its number of bits.

? *Is subset sum really harder,*
or have we just not found the algorithm?

Efficiency

What does it mean for an algorithm to be fast?

Polynomial time: efficient (?)



Time at 10^9 steps per second. Both axes logarithmic.

Efficient = polynomial time:
$O(n^c)$ for a constant c,
where n is the **size of the input**,
the number of bits that write it down.

All problems solvable in polynomial time:
the class **P**.

A good theoretical model.
In practice, constants matter.

Subset sum: $O(nW)$ is not of this form,
as W has only $\log_2 W$ bits.

Intractable problems

Intractable:
no efficient algorithm is **known**.

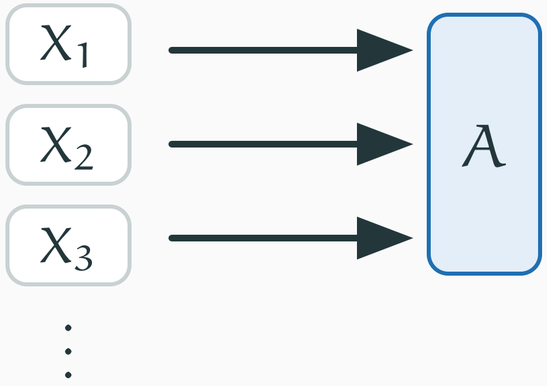
Ideal definition: no efficient algorithm **exists**.

Very few tools available.

A clever algorithm may be hiding.

? *Why do we believe a problem is hard?*

1. We tried for a long time, and failed.
2. Many clever people tried and failed.
3. An efficient algorithm for it would solve many other problems that nobody can solve efficiently.

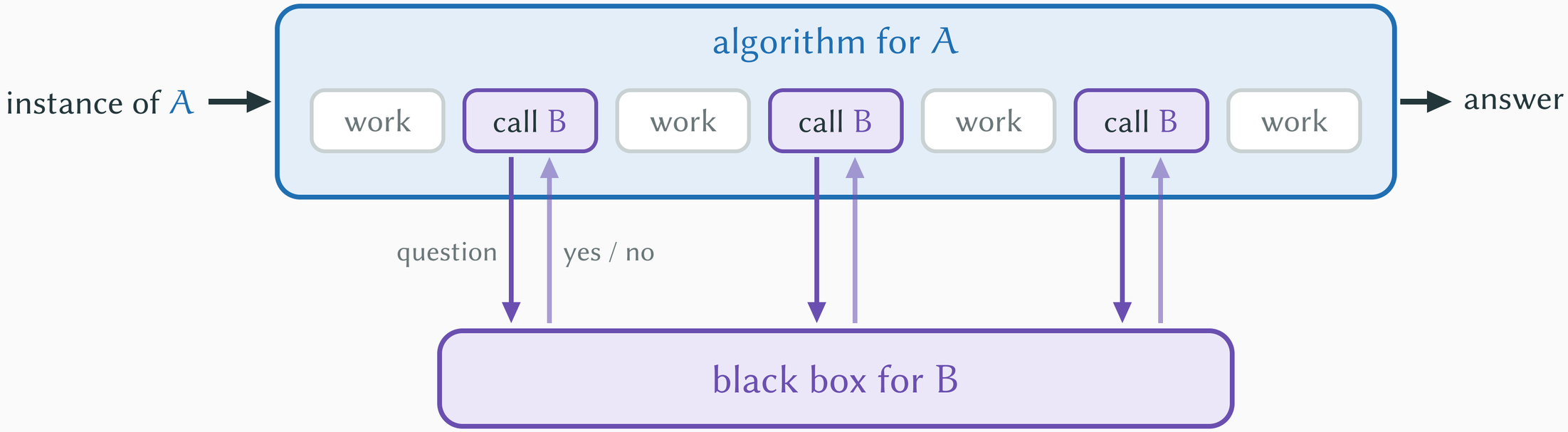


? *How can an algorithm for A solve other problems?*

Reductions

How do we use one problem to solve another?

Polynomial-time reductions

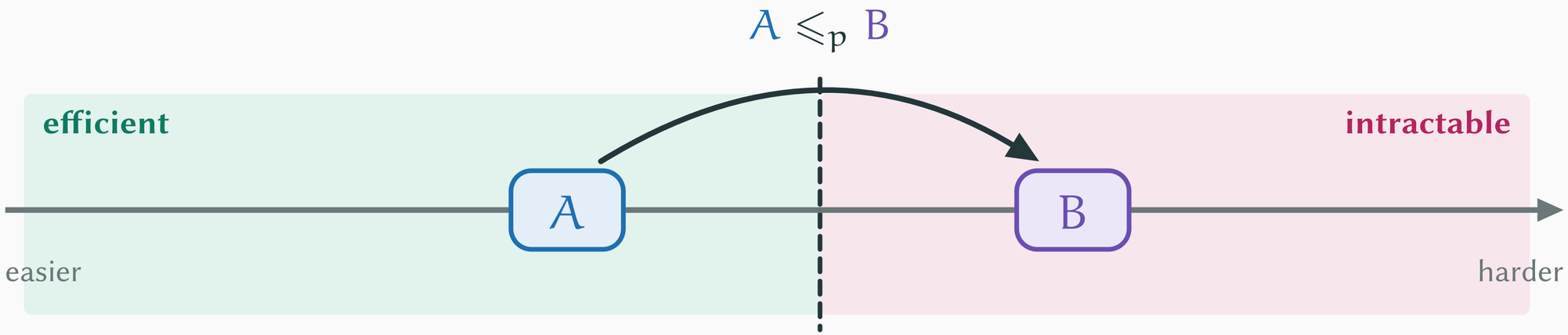


Reduction. A **reduces** to B, denoted $A \leqslant_p B$, if A can be solved with **polynomial work** and **polynomially many calls** to a black box for B.

A is no harder than B.

Lecture 12: matching $\leqslant_p$ maximum flow, with one call.

Two ways to use a reduction



If B is easy, then A is easy.

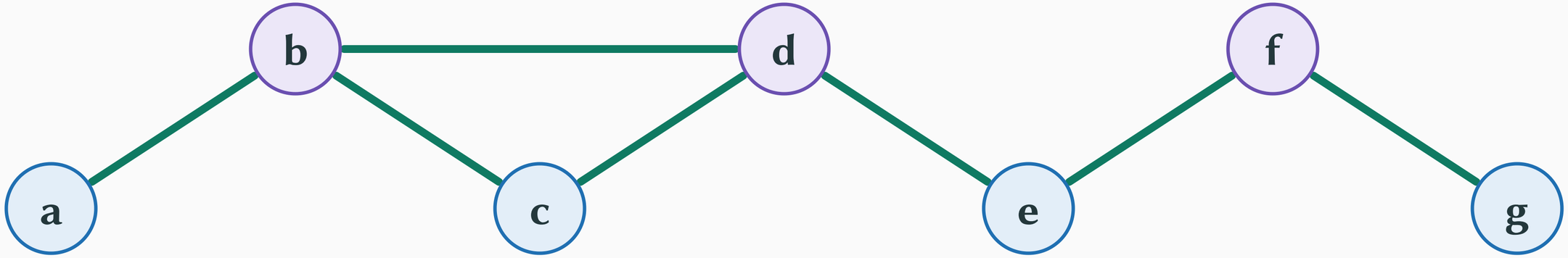
To solve a new problem A, reduce it to a B that we can solve.

If A is hard, then B is hard.

To show that B is hard, reduce a known hard A to B.

Direction matters. Reduce *from* the hard problem *to* the new one.
A easy and B hard is possible, so a hard B tells us nothing about A.

A first reduction: independent set and vertex cover



Independent set S: no edge has both ends in S.

Vertex cover C: every edge has an end in C.

- $S = \{a, c, e, g\}$ independent,
- $V \setminus S = \{b, d, f\}$ covers every edge.

Claim. S is independent $\iff$ $V \setminus S$ is a vertex cover.

No edge inside S $\iff$ every edge has an end outside S.

Reduction. $(G, k) \mapsto (G, n - k)$ turns IS into VC, and VC into IS.

NP-completeness

3-SAT: checking an assignment

3-SAT. Given $\varphi = C_1 \wedge \cdots \wedge C_m$, is there an assignment to $x_1, \ldots, x_n$ that makes every clause true?

Each clause is an OR of 3 **literals** (x_i or $\neg x_i$).

Checking one assignment takes $O(m)$ time.

Finding one: 2^n candidates, about 10^{30} for $n = 100$.

$x_1 = 1 \qquad x_2 = 0 \qquad x_3 = 0 \qquad x_4 = 1$

$$(\ x_1 \vee \neg x_2 \vee x_3\ ) \checkmark$$
$$\wedge$$
$$(\ \neg x_1 \vee x_2 \vee x_4\ ) \checkmark$$
$$\wedge$$
$$(\ \neg x_2 \vee \neg x_3 \vee \neg x_4\ ) \checkmark$$

All 3 clauses are true: a satisfying assignment.

A satisfying assignment is a **short certificate**.

The class NP: easy to check

NP. A decision problem is in **NP** if every yes-instance s has a **certificate** t of polynomial length that a polynomial-time **verifier** accepts.

NP = nondeterministic polynomial time (not "non-polynomial").

problem	certificate t	the check $V(s, t)$
3-SAT	an assignment	evaluate every clause
Subset sum	a subset of the numbers	add up, compare with W
Independent set	k vertices	no edge between two of them
Hamiltonian cycle	the vertices in cyclic order	consecutive ones are adjacent

P $\subseteq$ **NP**: the verifier ignores t and solves s itself.

? *Is* **P** $=$ **NP***? Open: one of the seven Millennium Prize Problems.*

NP-hard and NP-complete

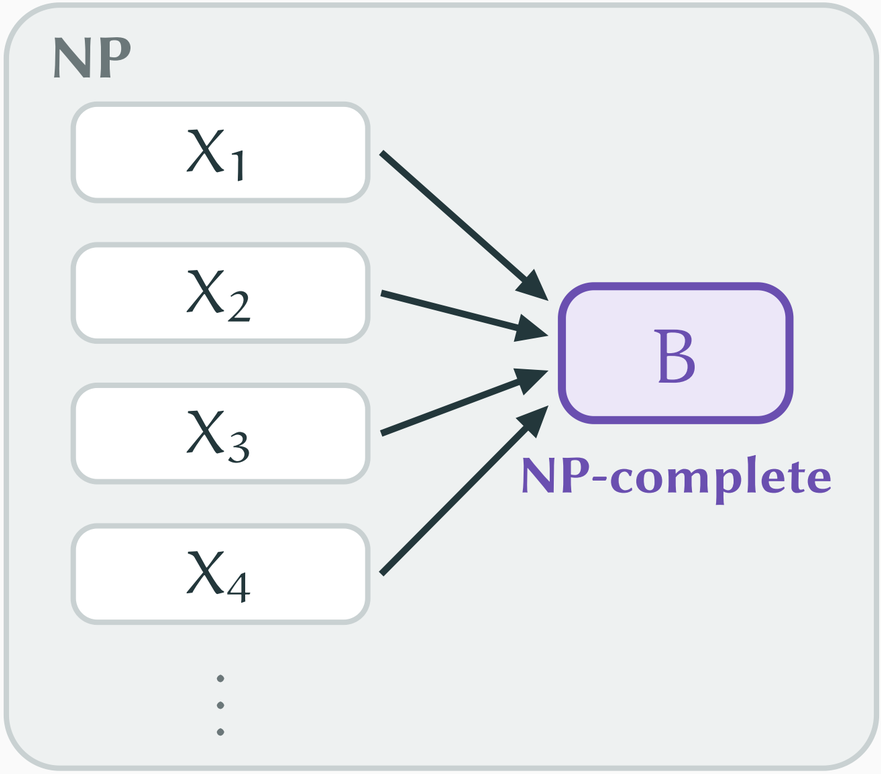
NP-hard. $X \leqslant_p B$ for **every** $X \in \mathbf{NP}$.

NP-complete. **NP**-hard, and $B \in \mathbf{NP}$.

Consequence. If one **NP**-complete problem is in **P**, then $\mathbf{P} = \mathbf{NP}$.

Every $X \in \mathbf{NP}$: $X \leqslant_p B$ and $B \in \mathbf{P}$, so $X \in \mathbf{P}$.

? *Does an NP-complete problem exist?*

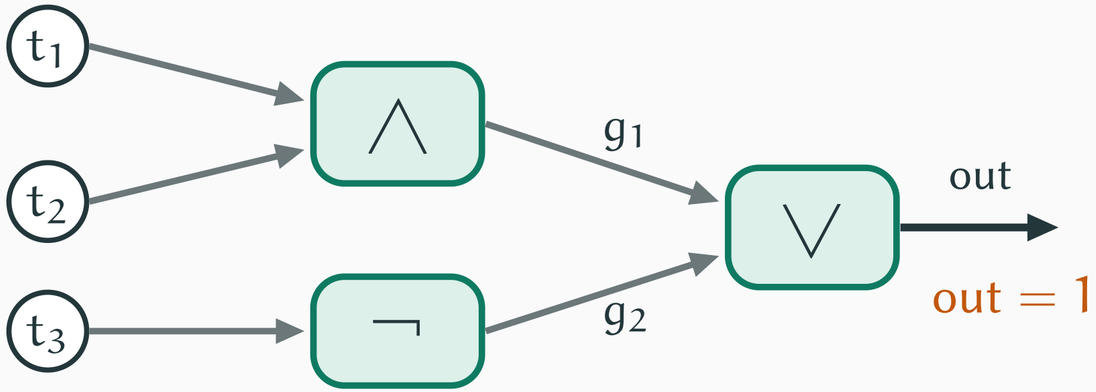


Cook–Levin: a first NP-complete problem

Theorem (Cook, 1971; Levin, 1973). SAT is **NP**-complete. As SAT $\leqslant_p$ 3-SAT, so is **3-SAT**.

Why? Let $X \in \mathbf{NP}$, with verifier V:
$s \in X \iff \exists t : V(s,t)$ accepts.

1. $V(s,\cdot)$ runs in polynomial time:
   it is a **circuit** C_s of polynomial size.
2. A variable per wire, a few clauses per gate:
   wire = gate of its inputs.
3. Add the clause (out). Then
   $s \in X \iff$ the formula is satisfiable.

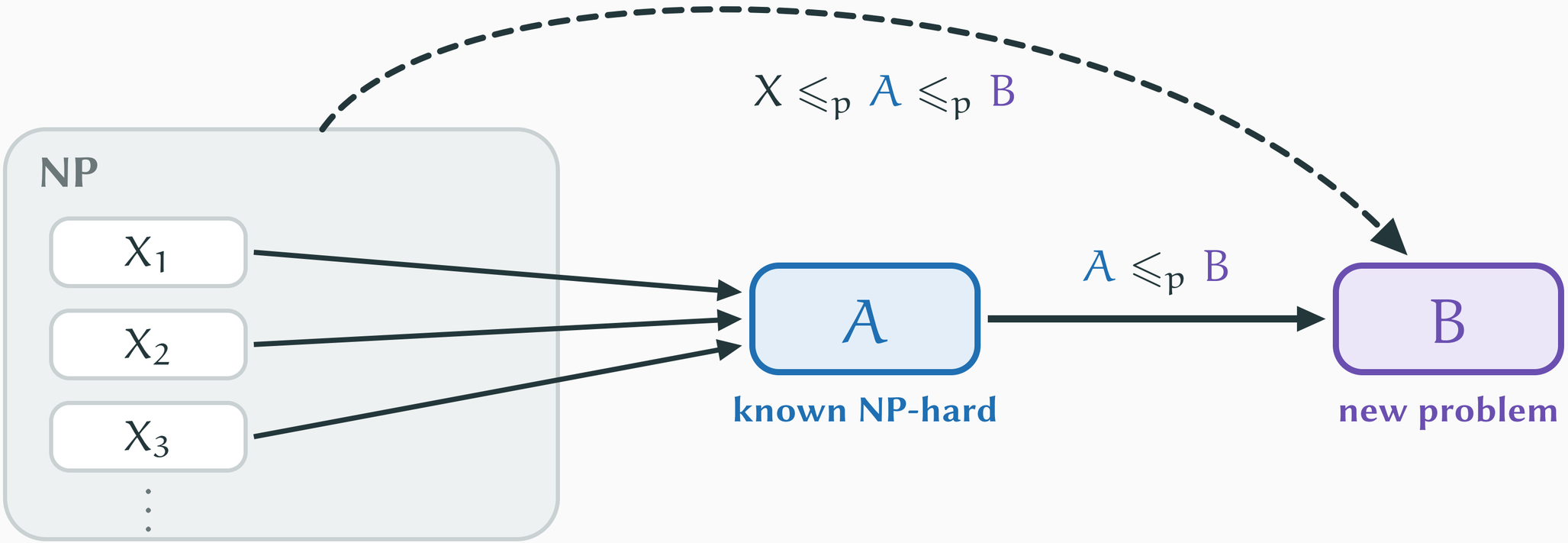


$z = x \wedge y$: $(\neg z \vee x) \wedge (\neg z \vee y) \wedge (z \vee \neg x \vee \neg y)$

$z = \neg x$: $(z \vee x) \wedge (\neg z \vee \neg x)$

$z = x \vee y$: $(z \vee \neg x) \wedge (z \vee \neg y) \wedge (\neg z \vee x \vee y)$

Hardness propagates



Fact. If A is **NP**-hard and $A \leqslant_p B$, then B is **NP**-hard.

$X \leqslant_p A$ and $A \leqslant_p B$ give $X \leqslant_p B$: replace every call to A by the reduction to B.
Polynomially many calls, polynomial work: still polynomial.

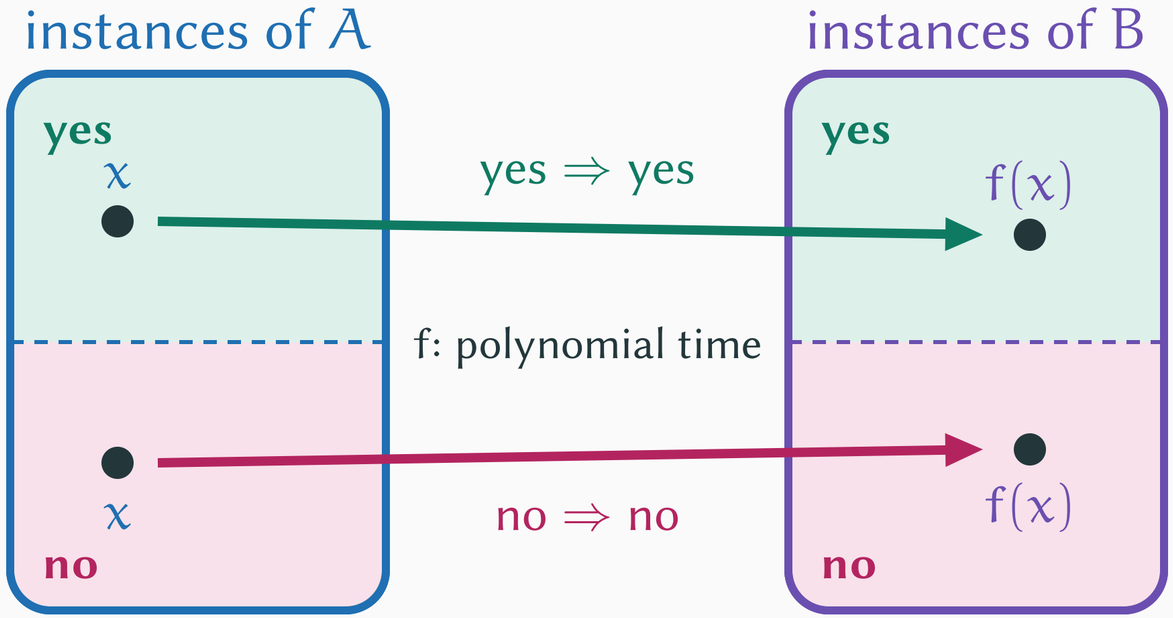
Proving that B is NP-complete

1. $B \in$ **NP:** a certificate, checkable in polynomial time.
2. B **is NP-hard:** pick a *known* **NP**-hard A and show $A \leqslant_p B$.

Build $f(x)$ in polynomial time from every instance x of A.

Prove: x is a yes-instance $\iff$ $f(x)$ is.

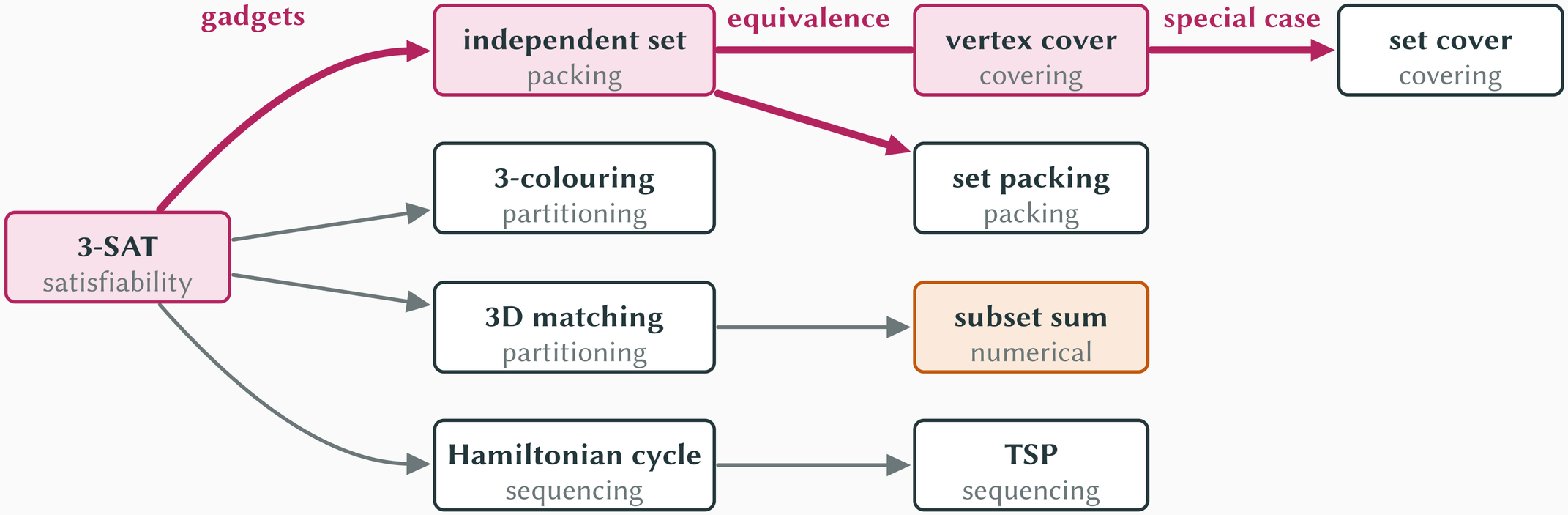
Both directions.



Reduce **from** the known hard problem **to** the new one, B.

Complexity landscape

A web of NP-complete problems



Gadgets: a clause is a triangle.

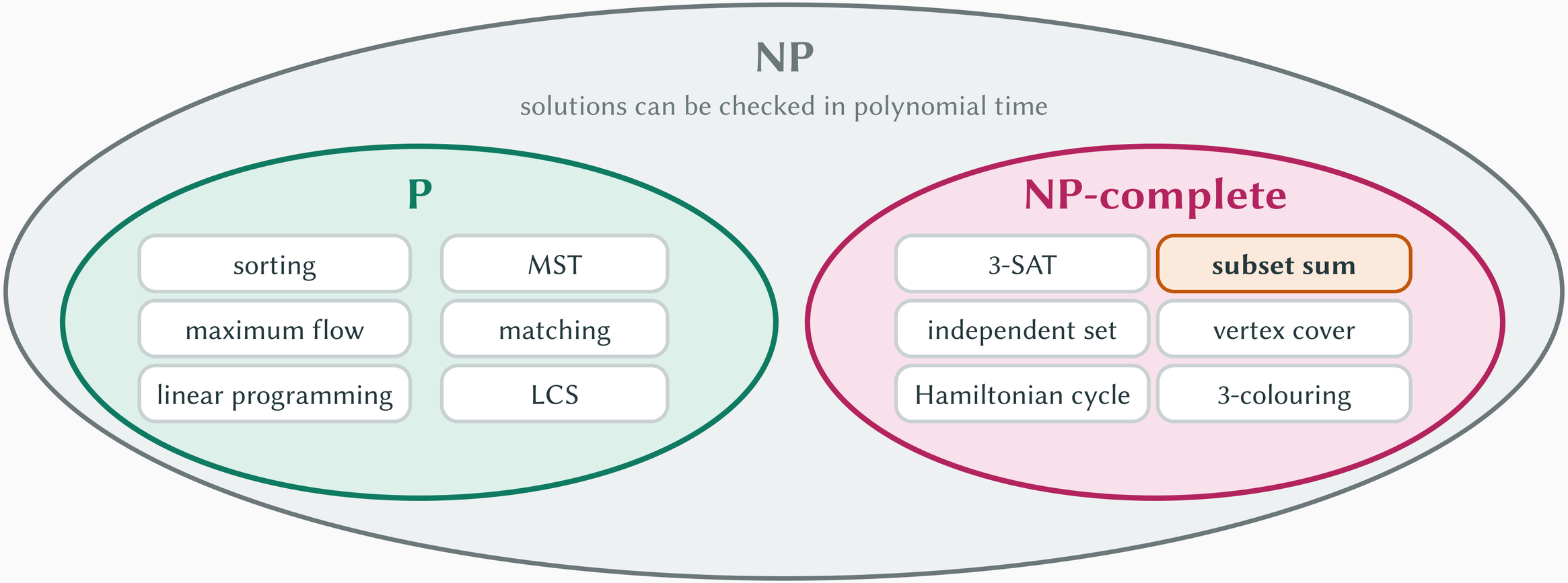
Equivalence: S and $V \setminus S$.

Special case: edges are the elements, a vertex is the set of its edges.

- **Subset sum**, the problem of Lecture 18, is **NP**-complete too.

Karp, *Reducibility among combinatorial problems* (1972).

The landscape of complexity



If one NP-complete problem is in P, everything in NP is.

Most experts believe **P** $\neq$ **NP**.

How hard is 3-SAT?

Brute force: 2^n assignments (slide 11).

Best known: about 1.31^n, randomised. Better, but still exponential.

(Paturi, Pudlák, Saks and Zane, 2005; Hertli, 2011)

Exponential Time Hypothesis (Impagliazzo and Paturi, 2001).
3-SAT on n variables needs time $2^{\Omega(n)}$. Stronger than **P** $\neq$ **NP**.

26 October 2026. OpenAI and Anthropic announced $2^{o(n)}$-time algorithms for 3-SAT, checked in Lean. Hence **ETH is false**.

Their time: $2^{O(n \log\log n/\sqrt{\log n})}$. The gain is tiny:

with $\log = \log_2$, the factor $\log\log n/\sqrt{\log n}$ is 1 at $n = 2^{16}$, and still $1/2$ at $n = 2^{256}$.

These algorithms are still not polynomial: **P** versus **NP** remains open.

A fine line between easy and hard

Polynomial time		NP-complete
2-SAT	vs	3-SAT
bipartite matching	vs	3-dimensional matching
minimum cut	vs	maximum cut
2-colouring	vs	3-colouring
linear programming	vs	integer linear programming

My problem is NP-hard: what now?

? *NP-hardness is a **worst-case** statement. Many workarounds.*

Special cases

Your instances may be easy:
2-SAT, trees, small tree-width.

Approximation

Provably close to optimal:
vertex cover within a factor 2.

Heuristics and solvers

Local search.
SAT and integer-programming solvers can handle large instances.

Summary

1. **Efficient** (theoretically) means polynomial time in the input size:
   $O(nW)$ for subset sum is only pseudo-polynomial.
2. $A \leqslant_p B$: A is no harder than B.
   Algorithms flow back to A, hardness forward to B.
3. **NP:** yes-answers have short certificates.
   NP-complete: the hardest in **NP**.
   Cook–Levin: 3-SAT is **NP**-complete.
4. **To prove** B **NP-complete:** $B \in$ **NP**, and a known **NP**-complete problem reduces to B:
   3-SAT $\leqslant_p$ independent set $\equiv$ vertex cover.

> A polynomial-time algorithm for one **NP**-complete problem would give **P** = **NP**.

Reference: KT, Chapter 8