Algorithm Design and Analysis
NP Completeness (2)

http://ada.miulab.tw

Yun-Nung (Vivian) Chen

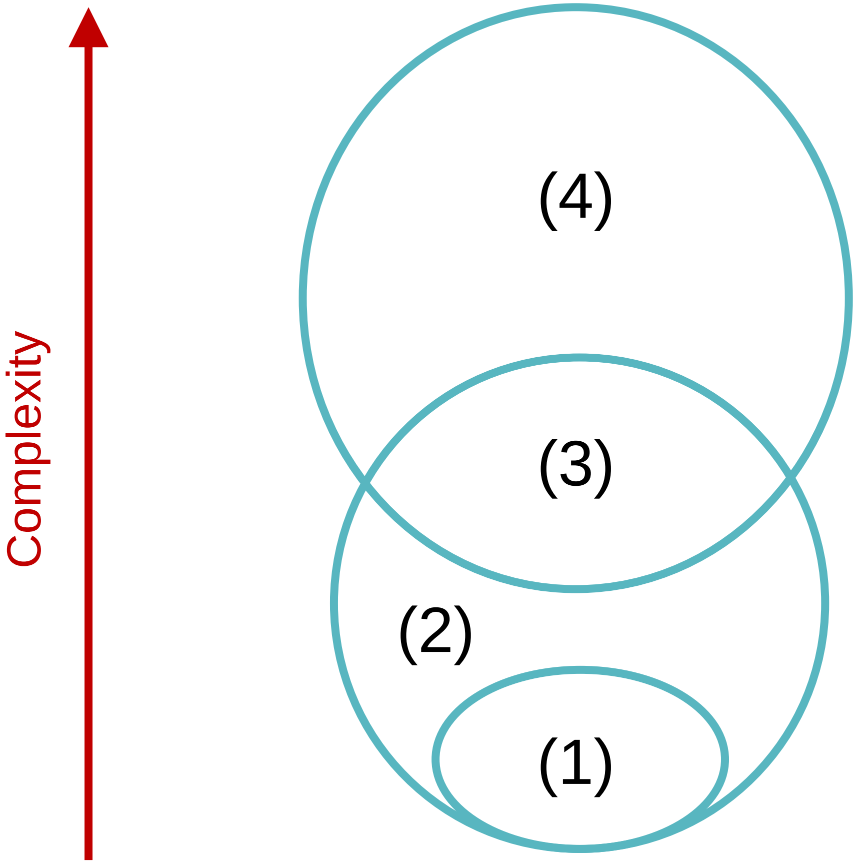
國立臺灣大學 National Taiwan University

Outline

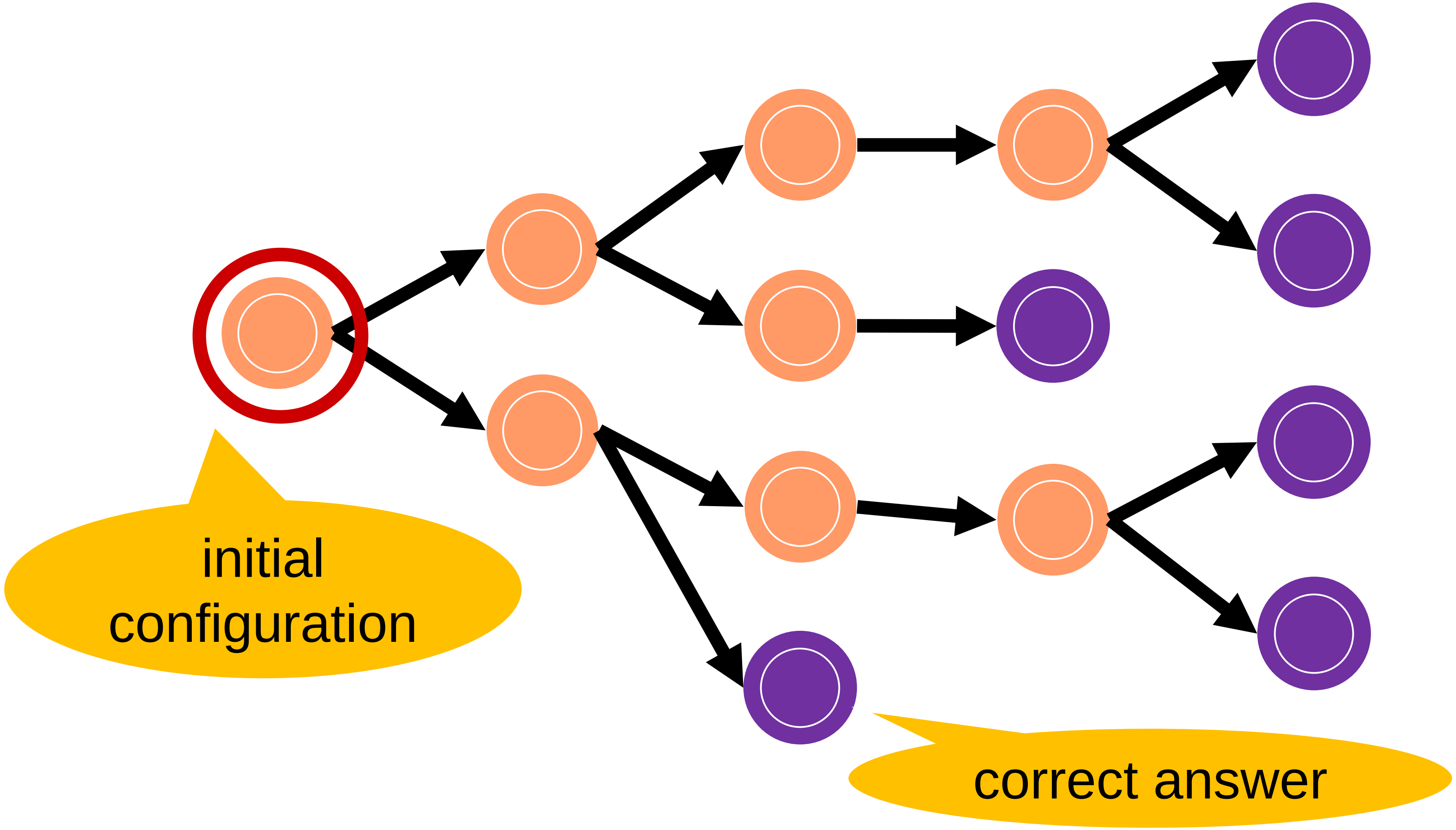
- Polynomial-Time Reduction
- Polynomial-Time Verification
- Proving NP-Completeness
  - 3-CNF-SAT
  - Clique
  - Vertex Cover
  - Independent Set
  - Traveling Salesman Problem

P, NP, NP-Complete, NP-Hard

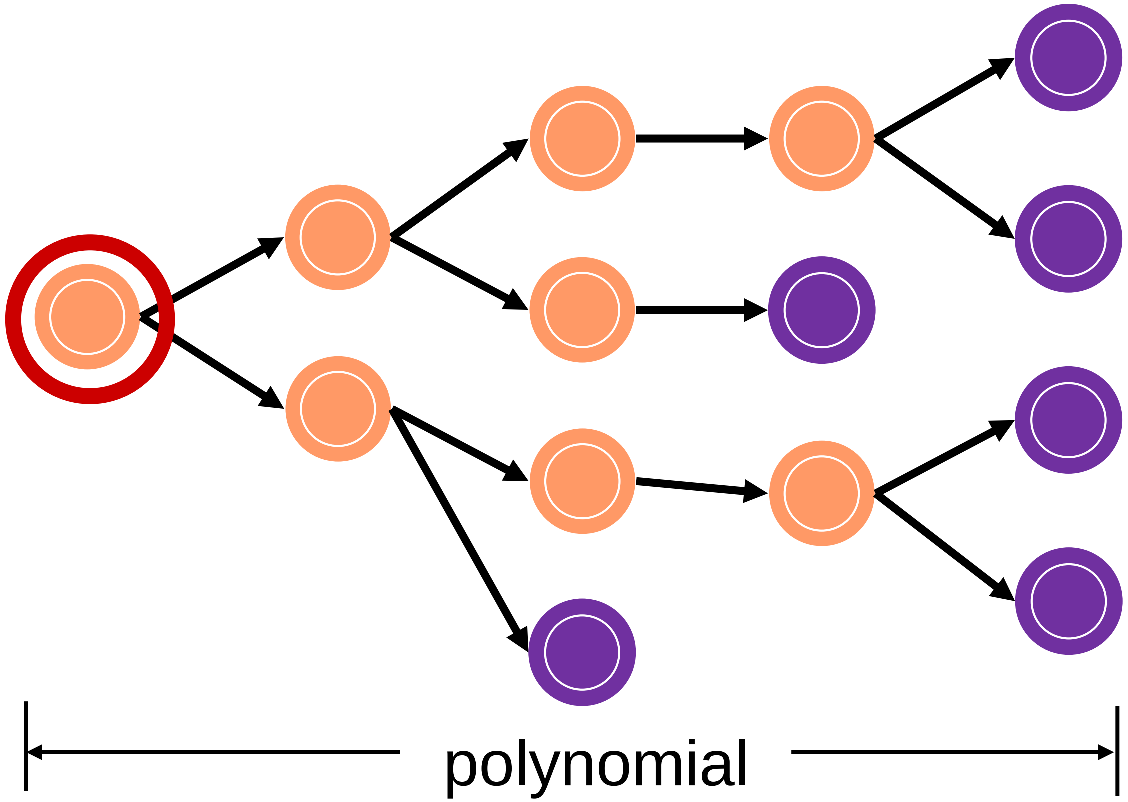
- P ≠ NP



Non-Deterministic Problem Solving



Non-Deterministic Polynomial



"solved" in non-deterministic polynomial time
= "verified" in polynomial time

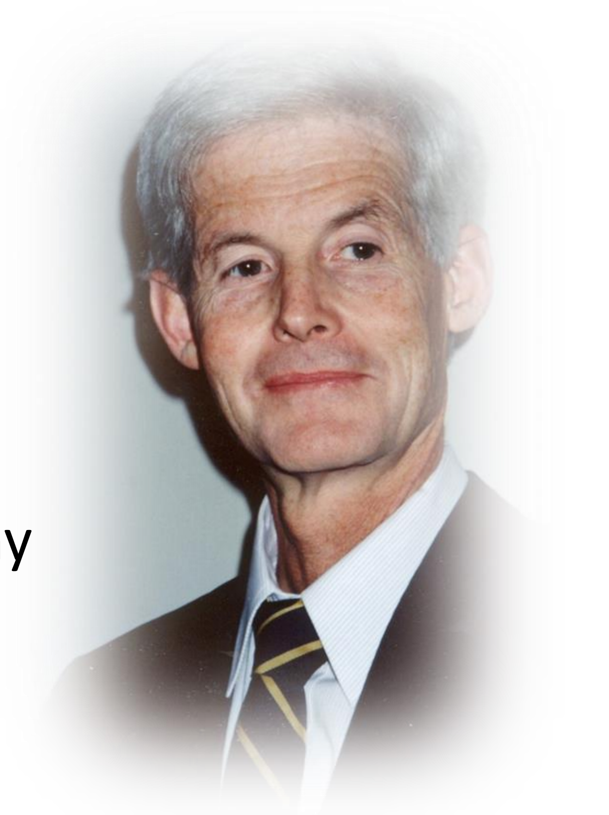
Polynomial-Time Reduction

Textbook Chapter 34.3 – NP-completeness and reducibility

First NP-Complete Problem – SAT (Satisfiability)

- Input: a Boolean formula with variables
- Output: whether there is a truth assignment for the variables that satisfies the input Boolean formula

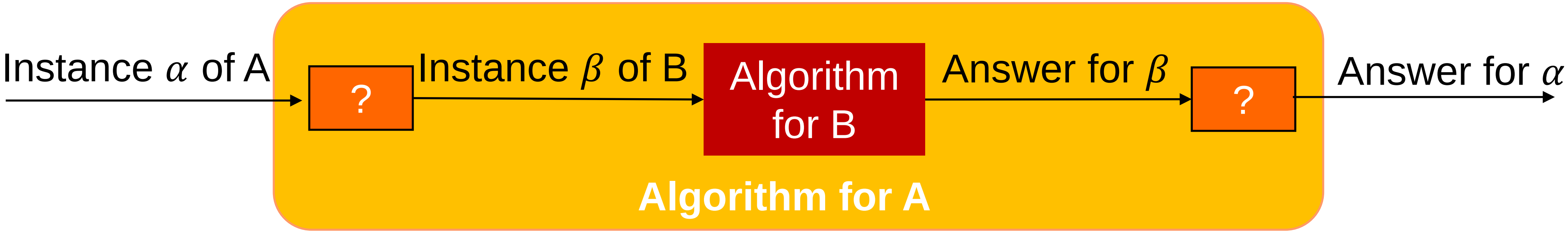
$$(x \vee y \vee z) \wedge (x \vee \bar{y} \vee \bar{z}) \wedge (\bar{x} \vee y)$$



- Stephan A. Cook [FOCS 1971] proved that
  - SAT can be solved in non-deterministic polynomial time → SAT ∈ NP
  - If SAT can be solved in deterministic polynomial time, then so can any NP problems → SAT ∈ NP-hard

Reduction

- Problem A can be reduced (in polynomial time) to Problem B
  = Problem B can be reduced (in polynomial time) from Problem A
  - We can find an algorithm that solves Problem B to help solve Problem A



Instance α of A

?

Instance β of B

Algorithm for B

Answer for β

?

Answer for α

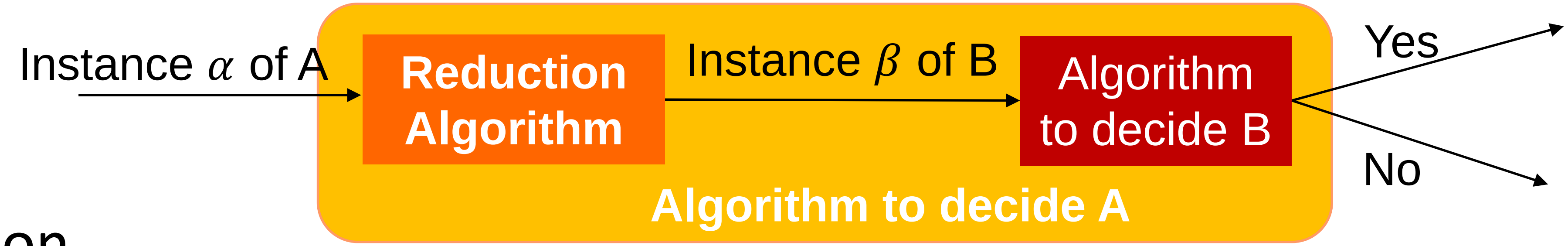
Algorithm for A

What is the complexity of Algorithm for A?

  - If problem B has a polynomial-time algorithm, then so does problem A
- Practice: design a MULTIPLY() function by ADD(), DIVIDE(), and SQUARE()

Reduction

- A reduction is an algorithm for **transforming a problem instance into another**



- Definition
  - Reduction from A to B implies A is not harder than B
  - A $\leq_p$ B if A can be reduced to B in polynomial time
- Applications
  - Designing algorithms: given algorithm for B, we can also solve A
  - Classifying problems: establish relative difficulty between A and B
  - **Proving limits: if A is hard, then so is B**

This is why we need it for proving NP-completeness!

Questions

- If *A* is an NP-hard problem and *B* can be reduced from *A*, then *B* is an NP-hard problem?
- If *A* is an NP-complete problem and *B* can be reduced from *A*, then *B* is an NP-complete problem?
- If *A* is an NP-complete problem and *B* can be reduced from *A*, then *B* is an NP-hard problem?

Problem Difficulty

- Q: Which one is harder?

KNAPSACK: Given a set $\{a_1, \dots, a_n\}$ of non-negative integers, and an integer K, decide if there is a subset $P \subseteq [1, n]$ such that $\sum_{i \in P} a_i = K$.

Polynomial-time reducible?

PARTITION: Given a set of n non-negative integers $\{a_1, \dots, a_n\}$, decide if there is a subset $P \subseteq [1, n]$ such that $\sum_{i \in P} a_i = \sum_{i \notin P} a_i$.

Polynomial-time reducible?

- A: They have equal difficulty.
- Proof:
  - PARTITION $\leq_p$ KNAPSACK
  - KNAPSACK $\leq_p$ PARTITION

Polynomial Time Reduction

Polynomial-time reducible?

KNAPSACK: Given a set $\{a_1, \ldots, a_n\}$ of non-negative integers, and an integer K, decide if there is a subset $P \subseteq [1, n]$ such that $\sum_{i \in P} a_i = K$.

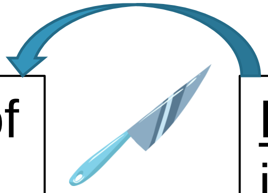
PARTITION: Given a set of n non-negative integers $\{a_1, \ldots, a_n\}$, decide if there is a subset $P \subseteq [1, n]$ such that $\sum_{i \in P} a_i = \sum_{i \notin P} a_i$.

Polynomial-time reducible?

- PARTITION $\leq_p$ KNAPSACK
  - If we can solve KNAPSACK, how can we use that to solve PARTITION?
- KNAPSACK $\leq_p$ PARTITION
  - If we can solve PARTITION, how can we use that to solve KNAPSACK?

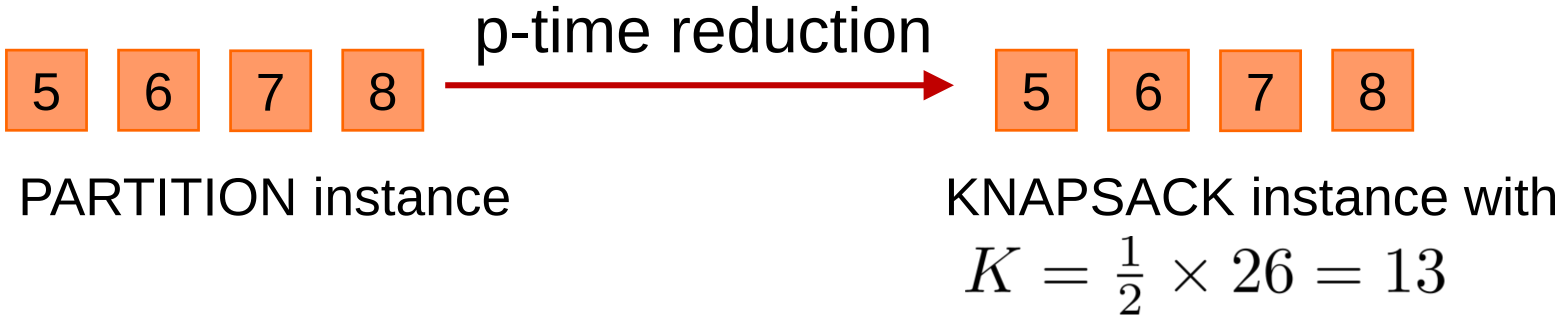
PARTITION $\leq_p$ KNAPSACK

KNAPSACK: Given a set $\{a_1, \ldots, a_n\}$ of non-negative integers, and an integer K, decide if there is a subset $P \subseteq [1, n]$ such that $\sum_{i \in P} a_i = K$.



PARTITION: Given a set of n non-negative integers $\{a_1, \ldots, a_n\}$, decide if there is a subset $P \subseteq [1, n]$ such that $\sum_{i \in P} a_i = \sum_{i \notin P} a_i$.

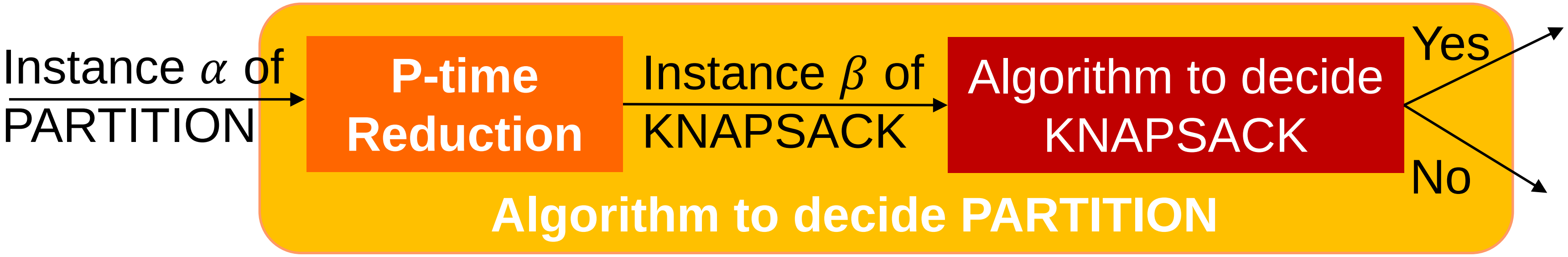
- If we can solve KNAPSACK, how can we use that to solve PARTITION?
- Polynomial-time reduction
  - Set $K = \frac{1}{2} \sum_{i=1}^{n} a_i$



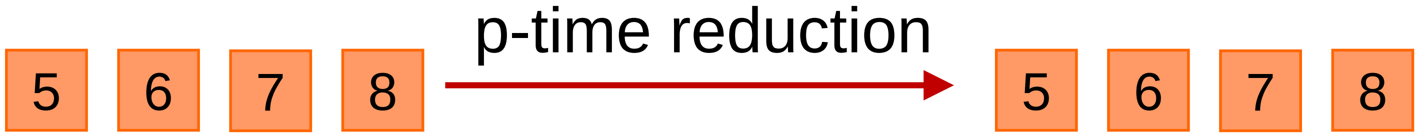
PARTITION instance

KNAPSACK instance with $K = \frac{1}{2} \times 26 = 13$

PARTITION $\leq_p$ KNAPSACK


Instance α of PARTITION
P-time Reduction
Instance β of KNAPSACK
Algorithm to decide KNAPSACK
Yes
No
Algorithm to decide PARTITION

- If we can solve KNAPSACK, how can we use that to solve PARTITION?
- Polynomial-time reduction
  - Set $K = \frac{1}{2}\sum_{i=1}^{n} a_i$



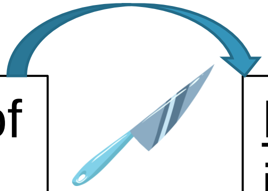
PARTITION instance

KNAPSACK instance with $K = \frac{1}{2} \times 26 = 13$

- Correctness proof: KNAPSACK returns yes if and only if an equal-size partition exists

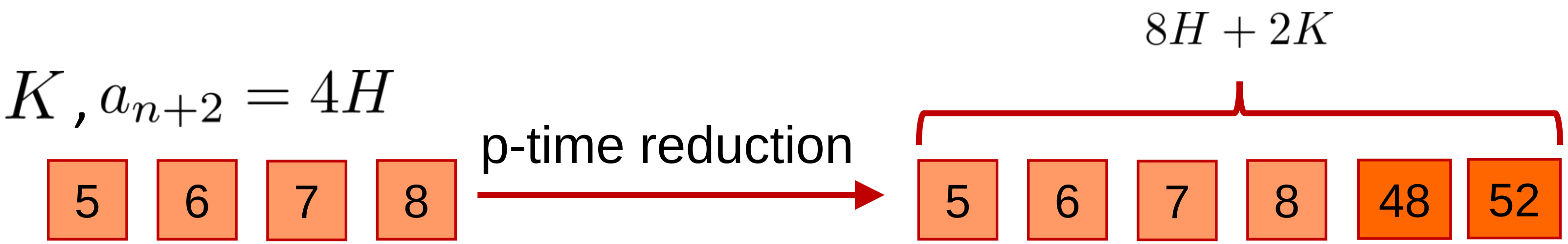
KNAPSACK $\leq_p$ PARTITION

KNAPSACK: Given a set $\{a_1, \ldots, a_n\}$ of non-negative integers, and an integer K, decide if there is a subset $P \subseteq [1, n]$ such that $\sum_{i \in P} a_i = K$.



PARTITION: Given a set of n non-negative integers $\{a_1, \ldots, a_n\}$, decide if there is a subset $P \subseteq [1, n]$ such that $\sum_{i \in P} a_i = \sum_{i \notin P} a_i$.

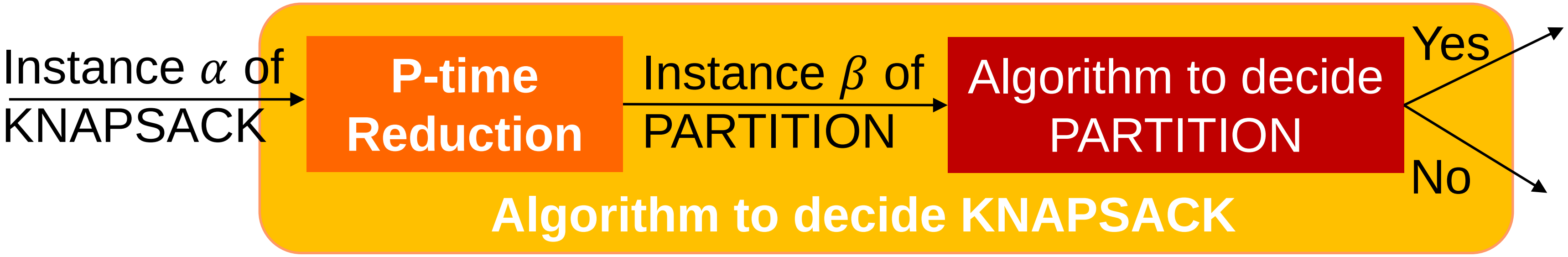
- If we can solve PARTITION, how can we use that to solve KNAPSACK?
- Polynomial-time reduction
  - Set $H = \frac{1}{2} \sum_{i=1}^{n} a_i$
  - Add $a_{n+1} = 2H + 2K$, $a_{n+2} = 4H$



KNAPSACK instance with $K = 11$

PARTITION instance

KNAPSACK ≤$_p$ PARTITION



Instance α of KNAPSACK

P-time Reduction

Instance β of PARTITION

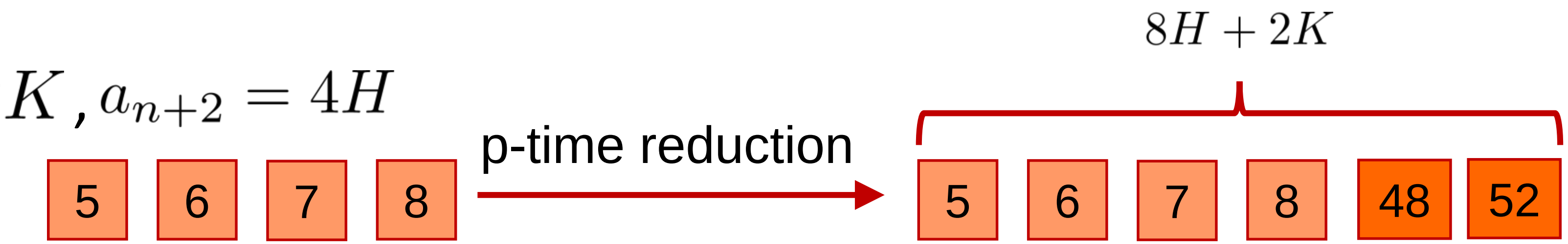
Algorithm to decide PARTITION

Yes

No

Algorithm to decide KNAPSACK

- If we can solve PARTITION, how can we use that to solve KNAPSACK?
- Polynomial-time reduction
  - Set $H = \frac{1}{2}\sum_{i=1}^{n} a_i$
  - Add $a_{n+1} = 2H + 2K$, $a_{n+2} = 4H$



$8H + 2K$

5 6 7 8 $\xrightarrow{\text{p-time reduction}}$ $\overbrace{5\ 6\ 7\ 8\ 48\ 52}^{8H+2K}$

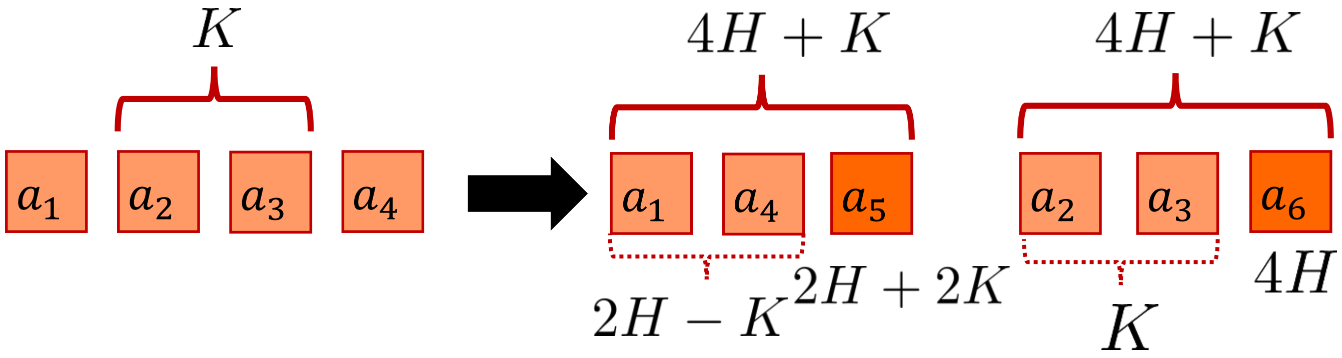
KNAPSACK instance with $K = 11$

PARTITION instance

- Correctness proof: PARTITION returns yes if and only if there is a subset $P \subseteq [1, n]$ such that $\sum_{i \in P} a_i = K$

KNAPSACK $\leq_p$ PARTITION

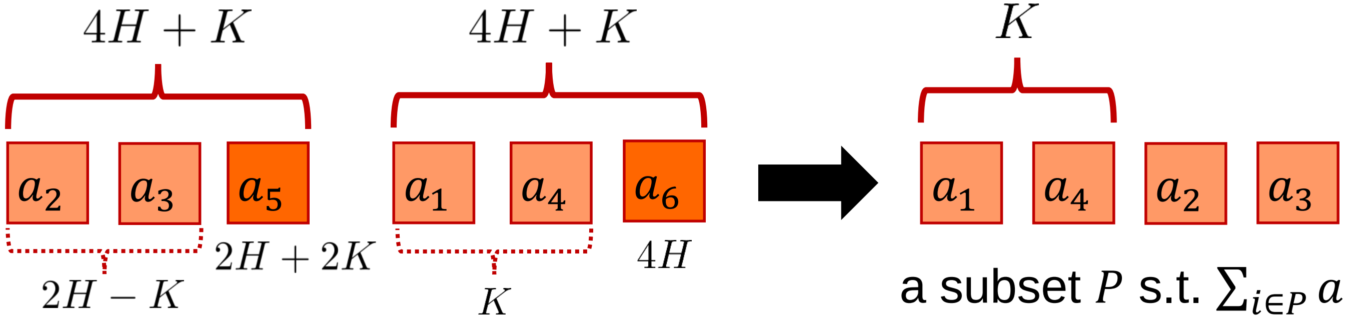
- Polynomial-time reduction
  - Set $H = \frac{1}{2}\sum_{i=1}^{n} a_i$
  - Add $a_{n+1} = 2H + 2K$, $a_{n+2} = 4H$
- Correctness proof: PARTITION returns yes if and only if there is a subset $P \subseteq [1, n]$ such that $\sum_{i \in P} a_i = K$
  - “if” direction



PARTITION returns yes!

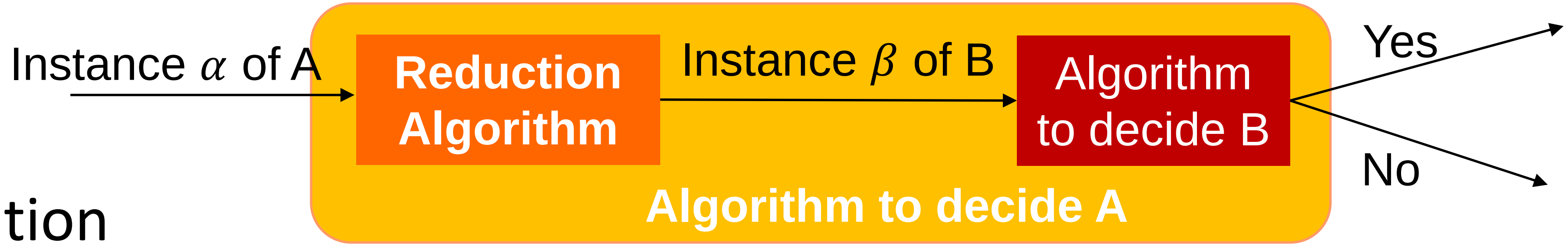
KNAPSACK $\leq_p$ PARTITION

- Polynomial-time reduction
  - Set $H = \frac{1}{2}\sum_{i=1}^{n} a_i$
  - Add $a_{n+1} = 2H + 2K$, $a_{n+2} = 4H$
- Correctness proof: PARTITION returns yes if and only if there is a subset $P \subseteq [1, n]$ such that $\sum_{i \in P} a_i = K$
  - “only if” direction
    - Because $\sum_{i=1}^{n+2} a_i = 8H + 2K$, if PARTITION returns yes, each set has $4H + K$
    - $\{a_1, \dots, a_n\}$ must be divided into $2H - K$ and K



a subset P s.t. $\sum_{i \in P} a_i = K$

Reduction for Proving Limits



- Definition
  - Reduction from A to B implies A is not harder than B
  - $A \le_p B$ if A can be reduced to B in polynomial time
- NP-completeness proofs
  - Goal: prove that B is NP-hard
  - Known: A is NP-complete/NP-hard
  - Approach: construct a polynomial-time reduction algorithm to convert α to β
  - Correctness: if we can solve B, then A can be solved → $A \le_p B$
  - B is no easier than A → A is NP-hard, so B is NP-hard

If the reduction is not p-time, does this argument hold?

Proving NP-Completeness

Formal Language Framework

- Focus on decision problems
- A language L over ∑ is any set of strings made up of symbols from ∑
- Every language L over ∑ is a subset of $\sum^*$

$$\sum^* = \{\epsilon; 0; 1; 10; 11; 101; 111; \cdots\}$$

The formal-language framework allows us to express concisely the relation between decision problems and algorithms that solve them.

- An algorithm A accepts a string $x \in \{0,1\}^*$ if $A(x) = 1$
- The language accepted by an algorithm A is the set of strings

$$L = \{x \in \{0,1\}^* : A(x) = 1\}$$

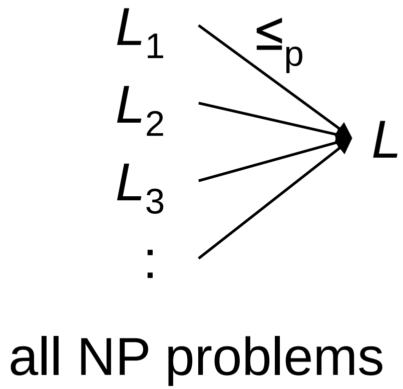
- An algorithm A rejects a string x if $A(x) = 0$

Proving NP-Completeness

- NP-Complete (NPC): class of decision problems in both NP and NP-hard
- In other words, a decision problem L is NP-complete if

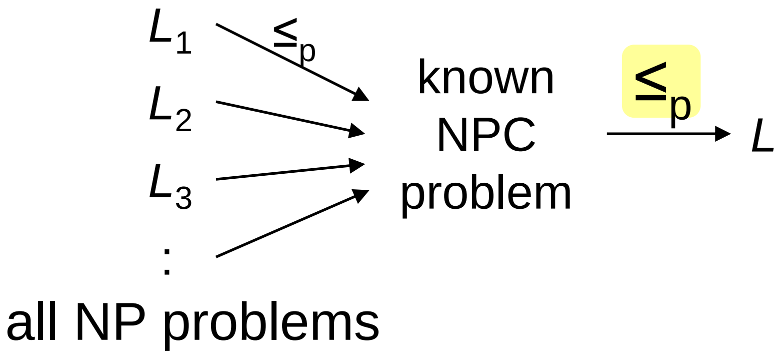
  1. $L \in$ NP
  2. $L \in$ NP-hard (that is, $L' \leq_p L$ for every $L' \in$ NP)

How to prove *L* is NP-hard ?



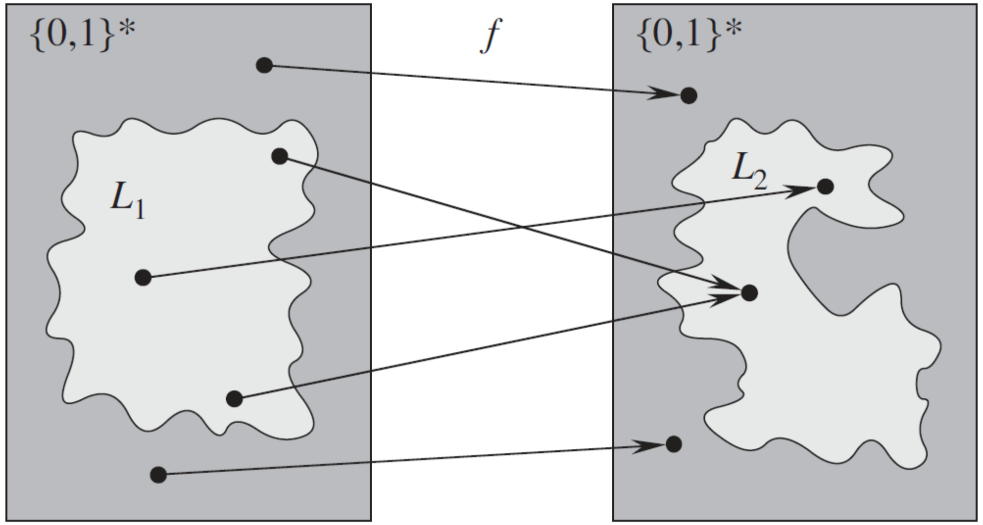
held by definition

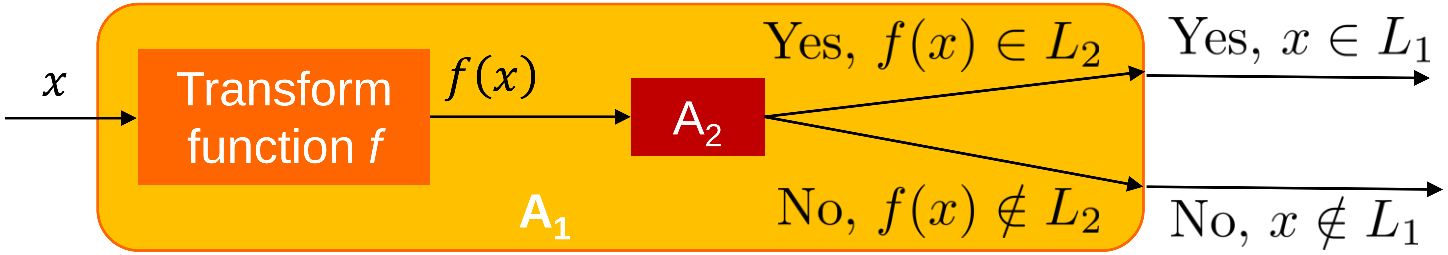
Goal: prove polynomial-time reduction



Polynomial-Time Reducible

- If $L_1, L_2 \subset \{0,1\}^*$ are languages s.t. $L_1 \le_p L_2$, then L2 ∈ P implies L1 ∈ P.



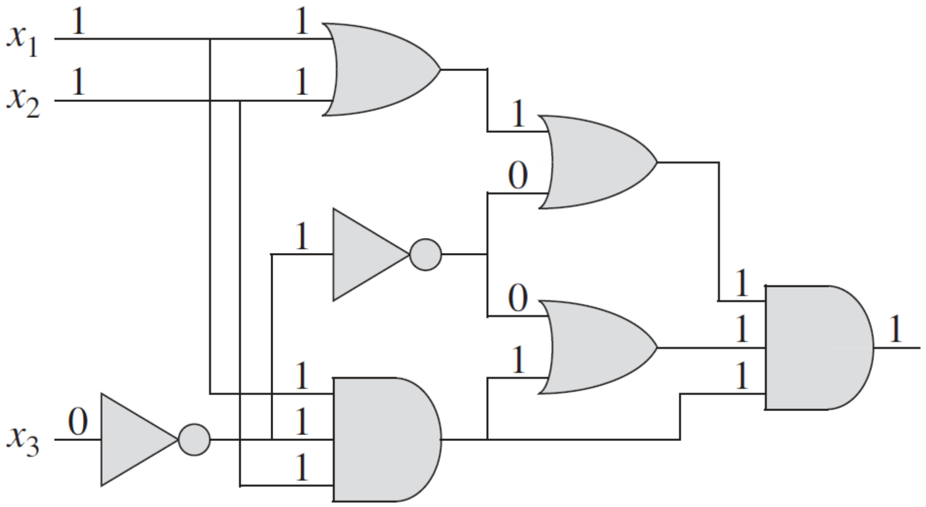


P v.s. NP

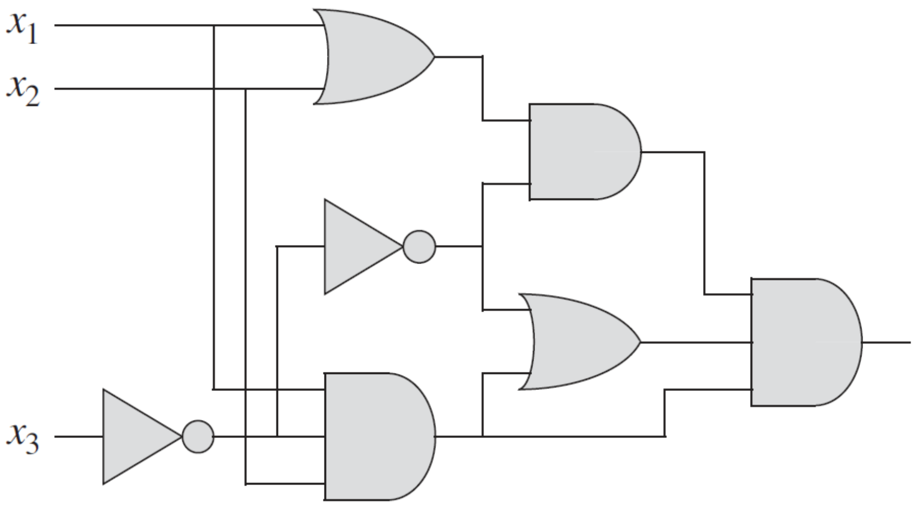
- If one proves that SAT can be solved by a polynomial-time algorithm, then NP = P.
- If somebody proves that SAT cannot be solved by any polynomial-time algorithm, then NP ≠ P.

Circuit Satisfiability Problem

- Given a Boolean combinational circuit composed of AND, OR, and NOT gates, is it satisfiable?
  - Satisfiable: there exists an assignment s.t. outputs = 1



Satisfiable



Unsatisfiable

CIRCUIT-SAT

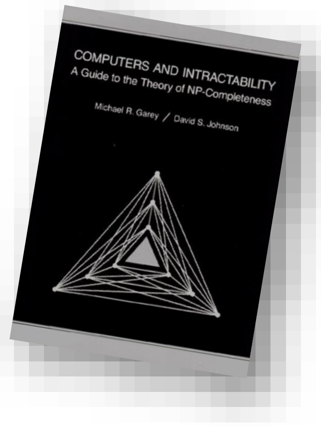
CIRCUIT-SAT = {<C>: C is a satisfiable Boolean combinational circuit}

- CIRCUIT-SAT can be solved in non-deterministic polynomial time

  → ∈ NP
- If CIRCUIT-SAT can be solved in deterministic polynomial time, then so can any NP problems

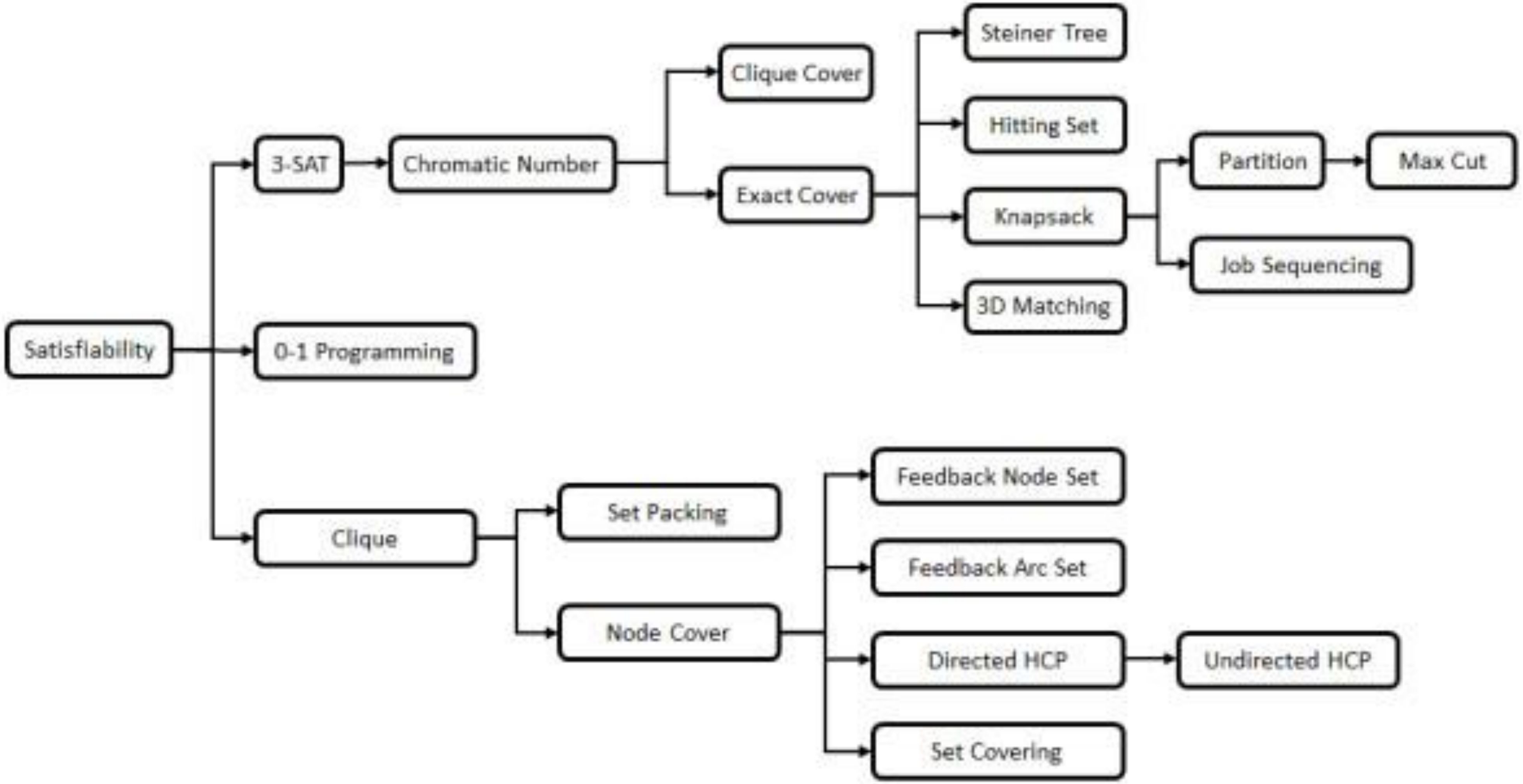
  → ∈ NP-hard
- (proof in textbook 34.3)
- CIRCUIT-SAT is NP-complete

Karp’s NP-Complete Problems


COMPUTERS AND INTRACTABILITY
A Guide to the Theory of NP-Completeness
Michael R. Garey / David S. Johnson

1. CNF-SAT
2. 0-1 INTEGER PROGRAMMING
3. CLIQUE
4. SET PACKING
5. VERTEX COVER
6. SET COVERING
7. FEEDBACK ARC SET
8. FEEDBACK NODE SET
9. DIRECTED HAMILTONIAN CIRCUIT
10. UNDIRECTED HAMILTONIAN CIRCUIT
11. 3-SAT
12. CHROMATIC NUMBER
13. CLIQUE COVER
14. EXACT COVER
15. 3-dimensional MATCHING
16. STEINER TREE
17. HITTING SET
18. KNAPSACK
19. JOB SEQUENCING
20. PARTITION
21. MAX-CUT

Karp’s NP-Complete Problems



Formula Satisfiability Problem (SAT)

- Given a Boolean formula Φ with variables, is there a variable assignment satisfying Φ

$$\phi = ((x_1 \to x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$

  - ∧ (AND), ∨ (OR), ¬ (NOT), → (implication), ↔ (if and only if)
  - Satisfiable: Φ is evaluated to 1

$$x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1$$

$$\begin{aligned}\phi &= ((0 \to 0) \vee \neg((\neg 0 \leftrightarrow 1) \vee 1)) \wedge \neg 0 \\ &= (1 \vee \neg((1 \leftrightarrow 1) \vee 1)) \wedge 1 \\ &= (1 \vee \neg(1 \vee 1)) \wedge 1 \\ &= (1 \vee 0) \wedge 1 \\ &= 1 \wedge 1 \\ &= 1\end{aligned}$$

SAT

SAT = {Φ | Φ is a Boolean formula with a satisfying assignment }

- Is SAT ∈ NP-Complete?
- To prove that SAT is NP-Complete, we show that
  - SAT ∈ NP
  - SAT ∈ NP-hard (CIRCUIT-SAT $\leq_p$ SAT)

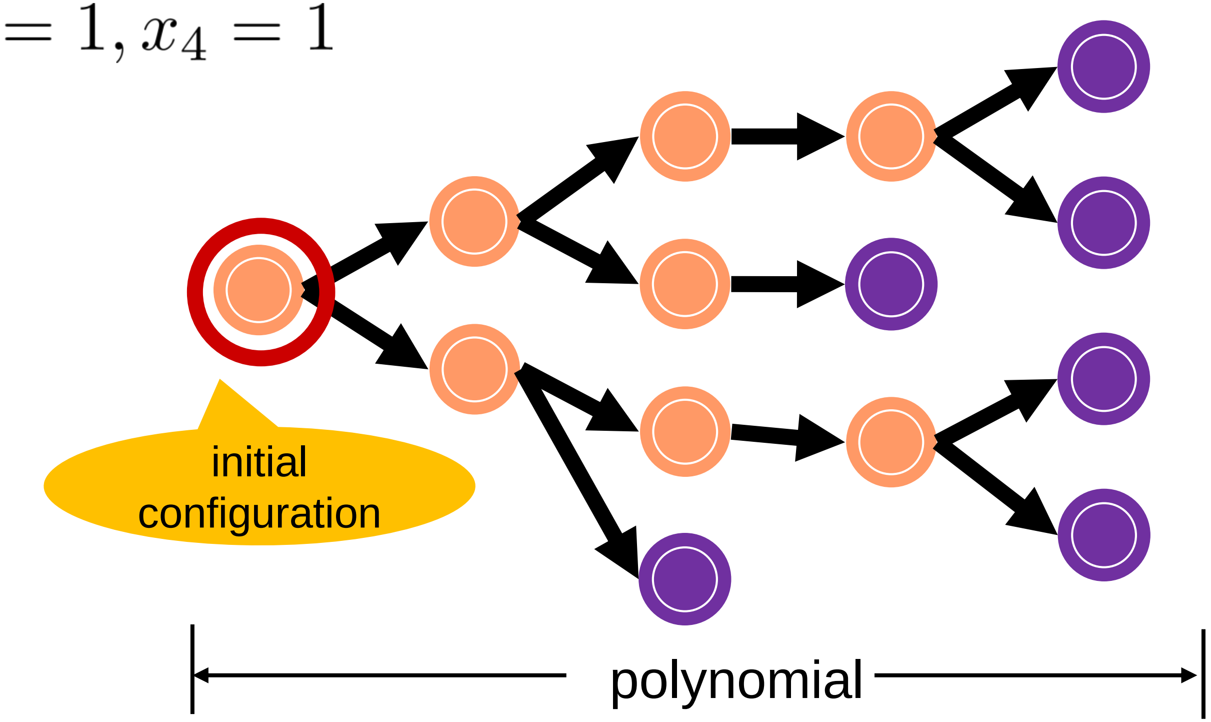
1) CIRCUIT-SAT is a known NPC problem
2) Construct a reduction *f* transforming every CIRCUIT-SAT instance to an SAT instance
3) Prove that x ∈ CIRCUIT-SAT iff *f*(x) ∈ SAT
4) Prove that *f* is a polynomial time transformation

SAT ∈ NP

- **Polynomial-time verification**: replaces each variable in the formula with the corresponding value in the certificate and then evaluates the expression

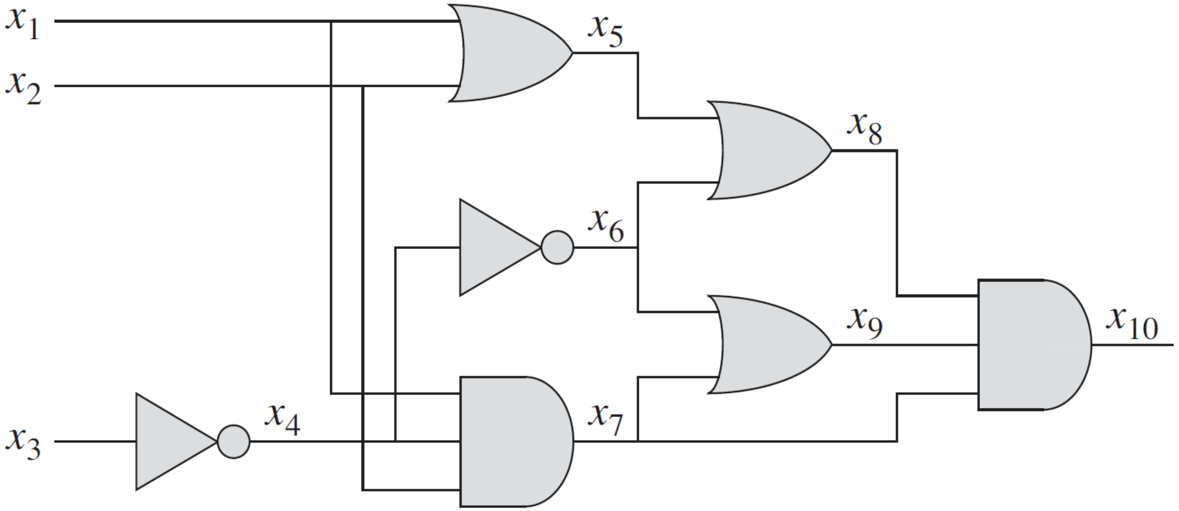
$$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$

$$x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1$$

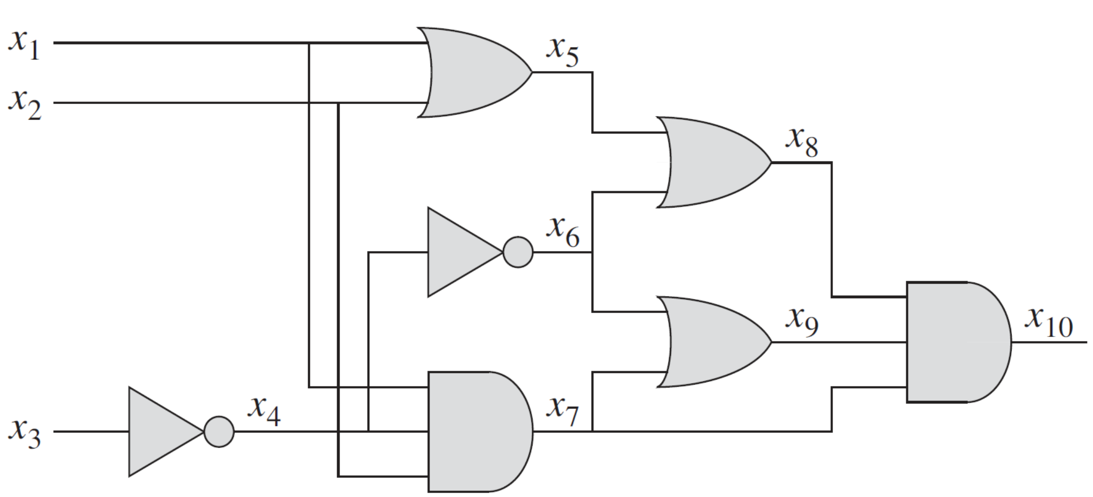


SAT ∈ NP-Hard

1) CIRCUIT-SAT is a known NPC problem
2) Construct a reduction *f* transforming every CIRCUIT-SAT instance to an SAT instance
   - Assign a variable to each **wire** in circuit C
   - Represent the operation of each gate using a formula, e.g.
   - Φ = AND the **output variable** and the **operations of all gates** $x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9)$



SAT ∈ NP-Hard



$$\begin{aligned}\phi = x_{10} &\wedge (x_4 \leftrightarrow \neg x_3)\\ &\wedge (x_5 \leftrightarrow (x_1 \vee x_2))\\ &\wedge (x_6 \leftrightarrow \neg x_4)\\ &\wedge (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4))\\ &\wedge (x_8 \leftrightarrow (x_5 \vee x_6))\\ &\wedge (x_9 \leftrightarrow (x_6 \vee x_7))\\ &\wedge (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9))\end{aligned}$$

- Prove that x ∈ CIRCUIT-SAT ↔ f(x) ∈ SAT
  - x ∈ CIRCUIT-SAT → f(x) ∈ SAT
  - f(x) ∈ SAT → x ∈ CIRCUIT-SAT
- f is a polynomial time transformation

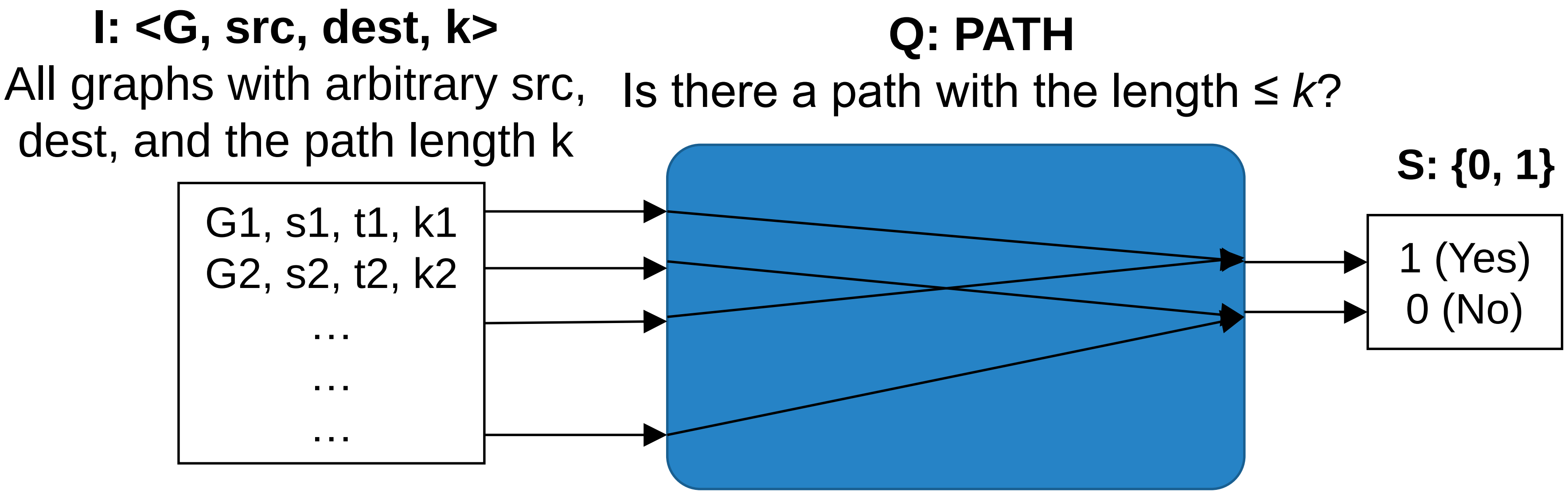
CIRCUIT-SAT $\leq_p$ SAT → SAT ∈ NP-hard

Polynomial-Time Verification

Chapter 34.1 – Polynomial-time
Chapter 34.2 – Polynomial-time verification

Abstract Problems

- Example of a decision problem, PATH
- **I**: a set of problem instances
- **S**: a set of problem solutions
- **Q**: abstract problem, defined as a *binary relation* on I and S



I: <G, src, dest, k>

All graphs with arbitrary src, dest, and the path length k

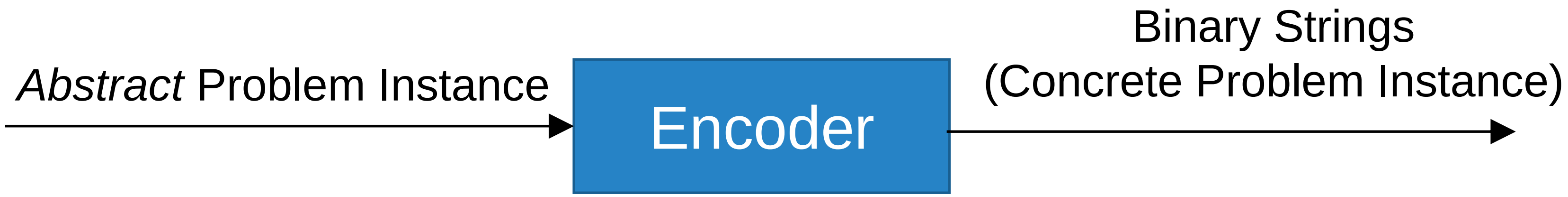
Q: PATH

Is there a path with the length ≤ *k*?

S: {0, 1}

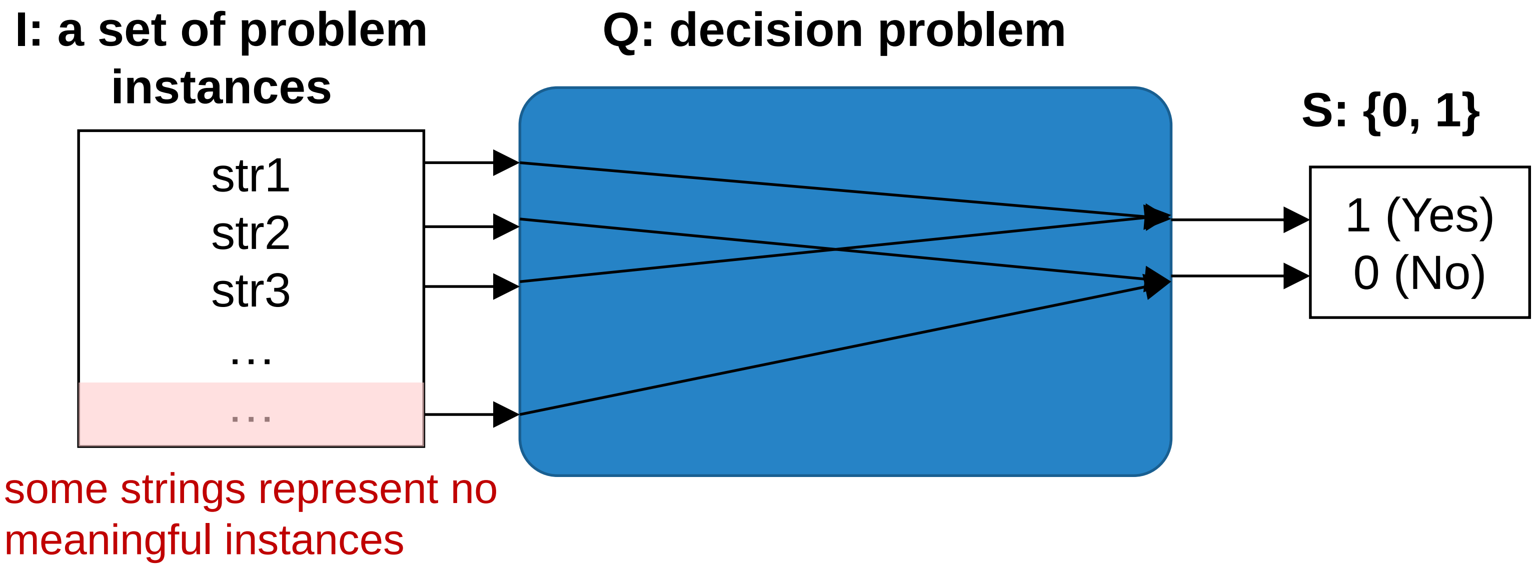
Problem Instance Encoding

- Convert an abstract problem instance into a binary string fed to a computer program



- A concrete problem is **polynomial-time solvable** if there exists an algorithm that solves any concrete instance of length *n* in time $O(n^k)$ for some constant *k*
  - Solvable = can produce a solution

Decision Problem Representation


I: a set of problem instances
Q: decision problem
S: {0, 1}
str1
str2
str3
…
…
1 (Yes)
0 (No)
some strings represent no meaningful instances

- I: a set of problem instances $\sum^* = \{\epsilon; 0; 1; 10; 11; 101; 111; \cdots\}$
- Q: a decision problem

  = a language *L* over $\sum = \{0, 1\}$ s.t. $L = \{x \in \{0,1\}^* : Q(x) = 1\}$

以答案為1的instances定義decision problem Q (L = {str1, str3} in this example)

P in Formal Language Framework

A **decision problem** Q can be defined as a **language** *L* over $\sum = \{0,1\}$ s.t. $L = \{x \in \{0,1\}^* : Q(x) = 1\}$

- An algorithm A ***accepts*** a string $x \in \{0,1\}^*$ if $A(x) = 1$
- An algorithm A ***rejects*** a string $x \in \{0,1\}^*$ if $A(x) = 0$
- An algorithm A **accepts** a language *L* if A accepts every string $x \in L$
  - If the string is in *L*, A outputs yes.
  - If the string is not in *L*, A may output no or loop forever.
- An algorithm A **decides** a language *L* if A accepts *L* and A rejects every string $x \notin L$
  - For every string, A can output the correct answer.

P in Formal Language Framework

- **Class P:** a class of decision problems *solvable* in polynomial time
- Given an instance *x* of a decision problem Q, its solution Q(*x*) (i.e., YES or NO) can be found in polynomial time
- An alternative definition of P:

P = {*L* ⊆ {0,1}* | there exists an algorithm that **decides *L* in polynomial time**}

- P is the class of language that can be accepted in polynomial time

P = {*L* | *L* is accepted by a polynomial algorithm}

Hamiltonian-Cycle Problem

- Problem: find a cycle that visits each vertex exactly once
- Formal language:

HAM-CYCLE = {<G> | G has a Hamiltonian cycle}

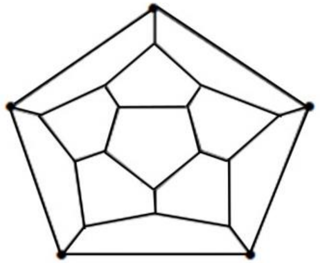
  - Is this language decidable? Yes
  - Is this language decidable in polynomial time? Probably not
- Given a **certificate** – the vertices in order that form a Hamiltonian cycle in G, how much time does it take to **verify** that G indeed contains a Hamiltonian cycle?

Verification Algorithm

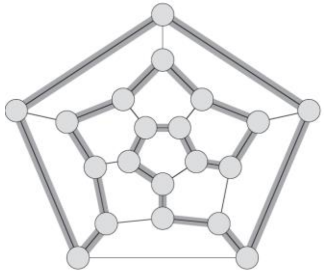
- **Verification algorithms** verify memberships in language

HAM-CYCLE = {<G> | G has a Hamiltonian cycle}

Input *x*



Certificate *y*



Verification Algorithm

Is *y* a Hamiltonian cycle in the graph (encoded in *x*)?

YES

x is in HAM-CYCLE

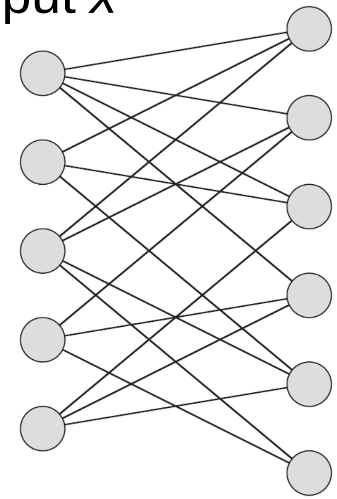
There exists a certificate for each YES instance

Verification Algorithm

- **Verification algorithms** verify memberships in language

HAM-CYCLE = {<G> | G has a Hamiltonian cycle}

Input *x*



Certificate *y*

??

Verification Algorithm

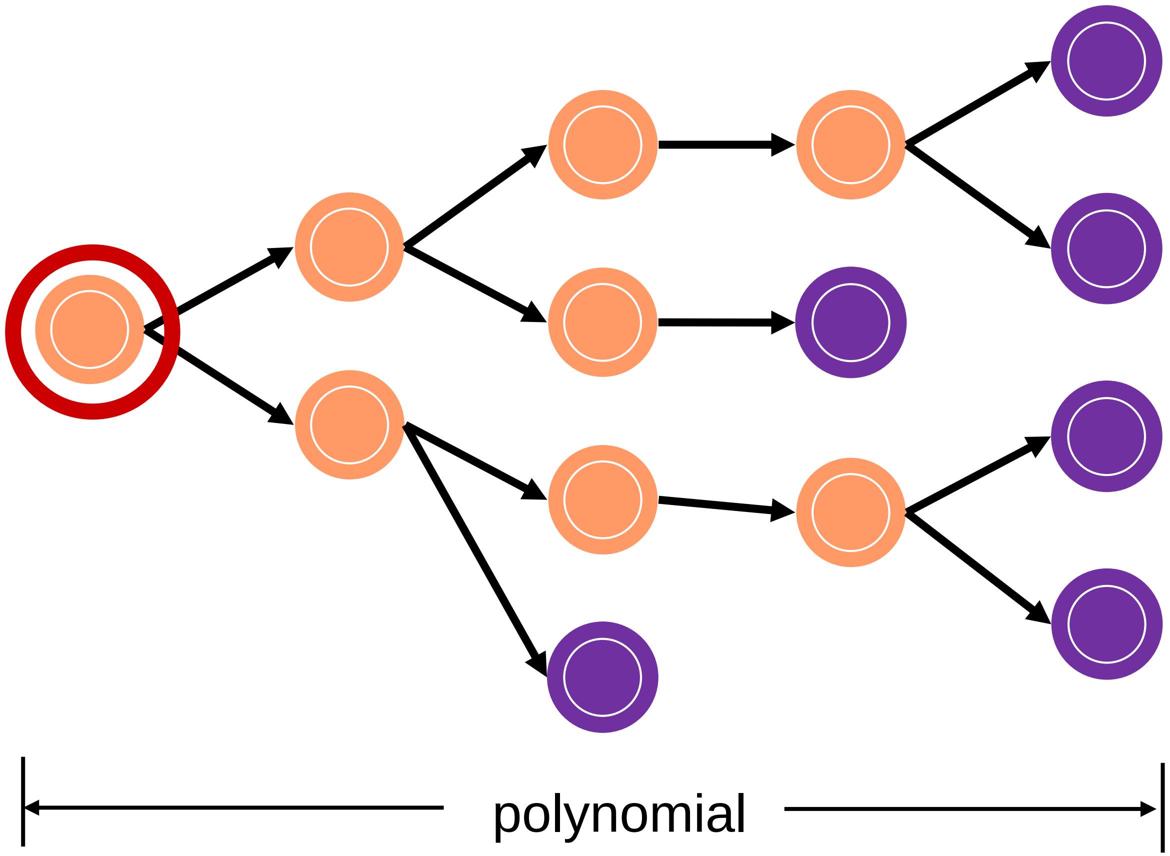
Is *y* a Hamiltonian cycle in the graph (encoded in *x*)?

→ NO

No conclusion

There exists no certificate for NO instance

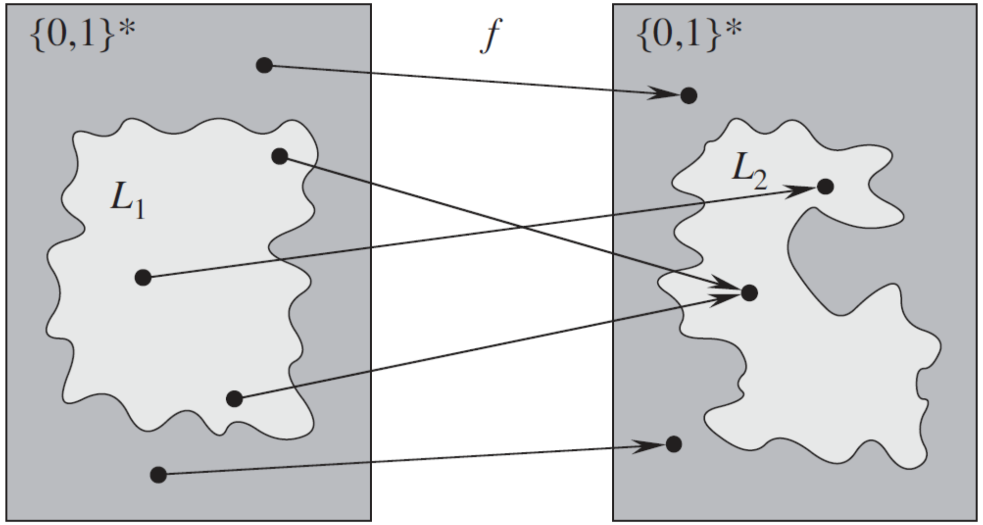
Non-Deterministic Polynomial

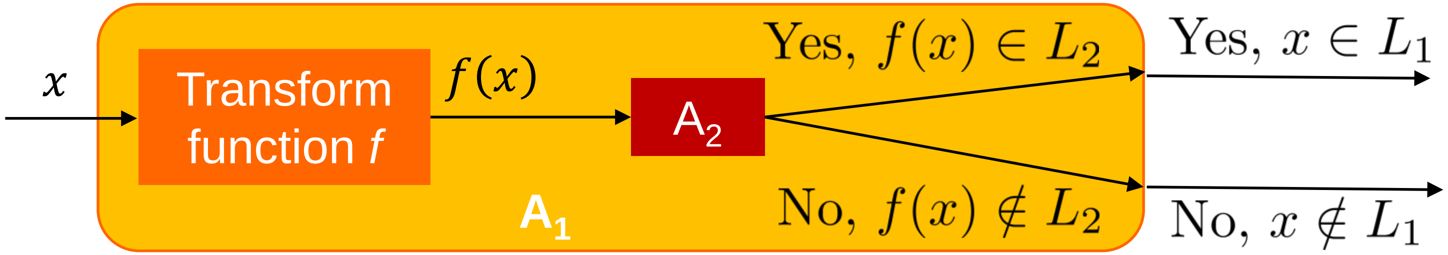


“solved” in non-deterministic polynomial time
= “verified” in polynomial time

Polynomial-Time Reducible

- If $L_1, L_2 \subset \{0,1\}^*$ are languages s.t. $L_1 \leq_p L_2$, then L2 ∈ P implies L1 ∈ P.



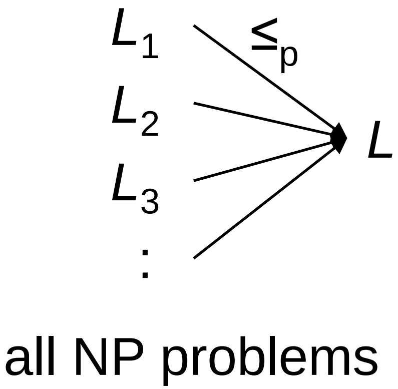


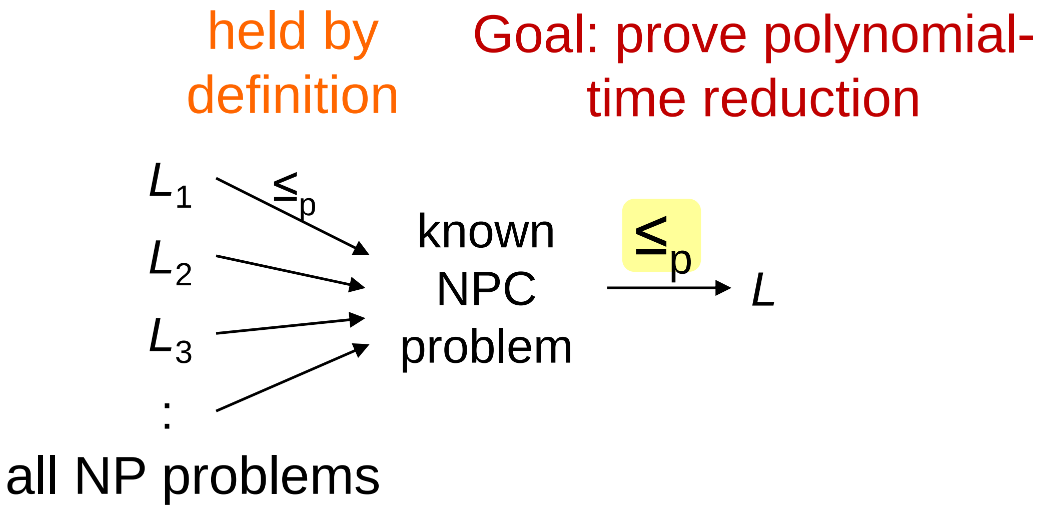
Proving NP-Completeness

- NP-Complete (NPC): class of decision problems in both NP and NP-hard
- In other words, a decision problem L is NP-complete if

1. $L \in$ NP
2. $L \in$ NP-hard (that is, $L' \leq_p L$ for every $L' \in$ NP)

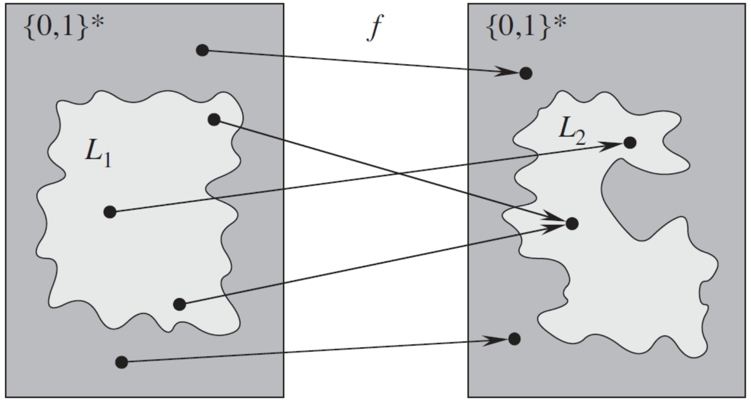
How to prove L is NP-hard ?





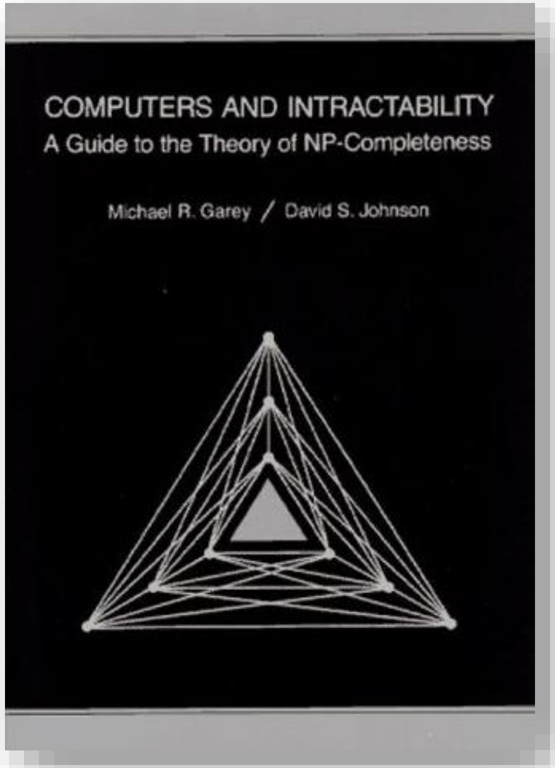
Proving NP-Completeness

- *L* ∈ NPC iff *L* ∈ NP and *L* ∈ NP-hard
- Proof of *L* in NPC:
  - Prove L ∈ NP
  - Prove L ∈ NP-hard
    1) Select a known NPC problem C
    2) Construct a reduction f transforming every instance of C to an instance of *L*
    3) Prove that $x \in C \iff f(x) \in L, \forall x \in \{0,1\}^*$
    4) Prove that *f* is a polynomial time transformation



More NP-Complete Problems

- “Computers and Intractability” by Garey and Johnson includes more than 300 NP-complete problems
  - All except SAT are proved by Karp’s polynomial-time reduction

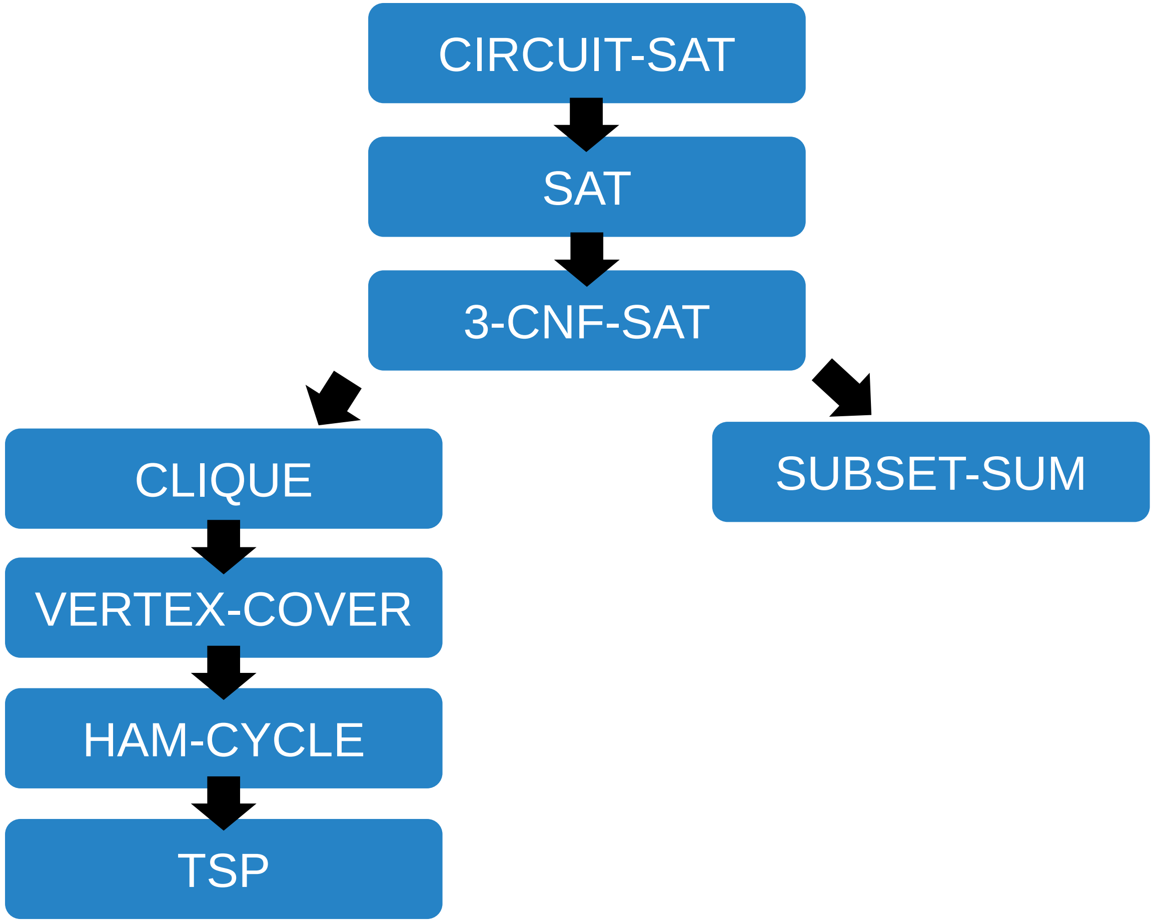

COMPUTERS AND INTRACTABILITY
A Guide to the Theory of NP-Completeness
Michael R. Garey / David S. Johnson

Proving NP-Completeness

Chapter 34.5 – NP-complete problems

Roadmap for NP-Completeness

- A → B: $A \le_p B$



3-CNF-SAT Problem

- 3-CNF-SAT: Satisfiability of Boolean formulas in 3-*conjunctive normal form* (3-CNF)

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

  - 3-CNF = AND of clauses, each of which is the OR of exactly 3 distinct literals
  - A literal is an occurrence of a variable or its negation, e.g., x_1 or ¬x_1

$x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1$ → satisfiable

3-CNF-SAT

3-CNF-SAT = {Φ | Φ is a Boolean formula in 3-conjunctive normal form (3-CNF) with a satisfying assignment }

- Is 3-CNF-SAT ∈ NP-Complete?
- To prove that 3-CNF-SAT is NP-Complete, we show that
  - 3-CNF-SAT ∈ NP
  - 3-CNF-SAT ∈ NP-hard (SAT $\leq_p$ 3-CNF-SAT)

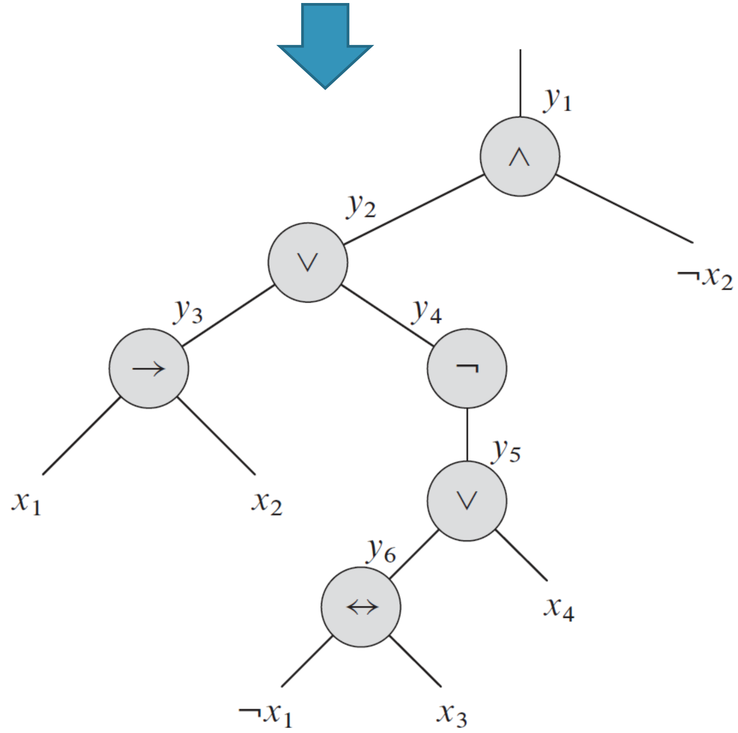
1) SAT is a known NPC problem
2) Construct a reduction *f* transforming every SAT instance to an 3-CNF-SAT instance
3) Prove that x ∈ SAT iff *f*(x) ∈ 3-CNF-SAT
4) Prove that *f* is a polynomial time transformation

We focus on the reduction construction from now on, but remember that a full proof requires showing that all other conditions are true as well

SAT $\leq_p$ 3-CNF-SAT

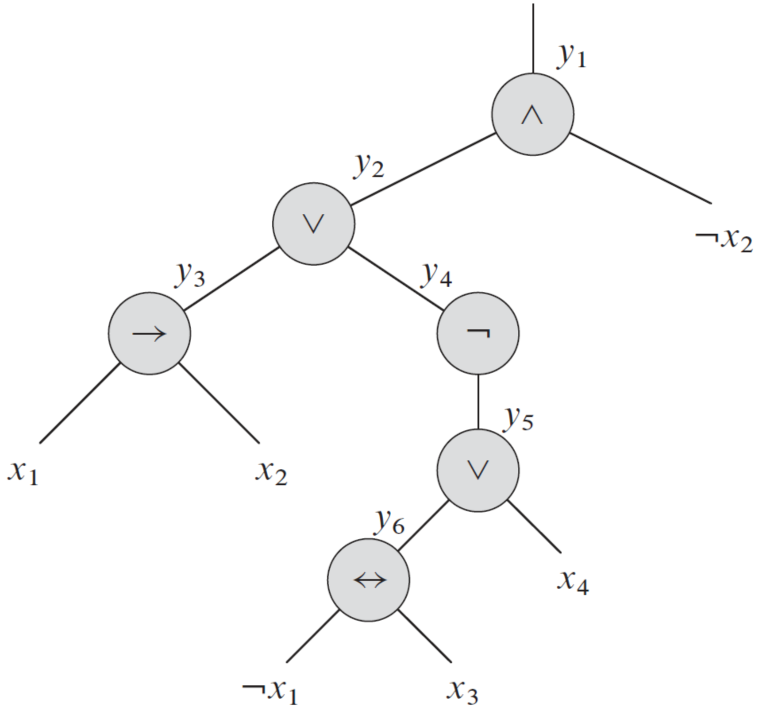
a) Construct a binary parser tree for an input formula Φ and introduce a variable y_i for the output of each internal node

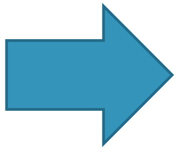
$$\phi = ((x_1 \to x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$



SAT $\leq_p$ 3-CNF-SAT

b) Rewrite Φ as the AND of the root variable and clauses describing the operation of each node





$$\begin{aligned}\phi' = y_1 &\wedge (y_1 \leftrightarrow (y_2 \wedge \neg x_2))\\ &\wedge (y_2 \leftrightarrow (y_3 \vee y_4))\\ &\wedge (y_3 \leftrightarrow (x_1 \wedge x_2))\\ &\wedge (y_4 \leftrightarrow \neg y_5)\\ &\wedge (y_5 \leftrightarrow (y_6 \vee x_4))\\ &\wedge (y_6 \leftrightarrow (\neg x_1 \wedge x_3))\end{aligned}$$

SAT $\leq_p$ 3-CNF-SAT

$$\begin{aligned}\phi' = y_1 &\wedge (y_1 \leftrightarrow (y_2 \wedge \neg x_2))\\ &\wedge (y_2 \leftrightarrow (y_3 \vee y_4))\\ &\wedge (y_3 \leftrightarrow (x_1 \wedge x_2))\\ &\wedge (y_4 \leftrightarrow \neg y_5)\\ &\wedge (y_5 \leftrightarrow (y_6 \vee x_4))\\ &\wedge (y_6 \leftrightarrow (\neg x_1 \wedge x_3))\end{aligned}$$

c) Convert each clause Φ_i' to CNF

- Construct a truth table for each clause Φ_i'
- Construct the disjunctive normal form for $\neg\Phi_i'$
- Apply DeMorgan's Law to get the CNF formula Φ_i''

y_1	y_2	y_2	Φ_1'	$\neg\Phi_1'$
1	1	1	0	1
1	1	0	1	0
1	0	1	0	1
1	0	0	0	1
0	1	1	1	0
0	1	0	0	1
0	0	1	1	0
0	0	0	1	0

$$\begin{aligned}\neg\phi_1' = &(y_1 \wedge y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge x_2)\\ &\vee (y_1 \wedge \neg y_2 \wedge \neg x_2) \vee (\neg y_1 \wedge y_2 \wedge \neg x_2)\end{aligned}$$

$$\Downarrow$$

$$\begin{aligned}\phi_1' = &(\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2)\\ &\wedge (\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee x_2)\end{aligned}$$

$$\boxed{\begin{aligned}\neg(a \wedge b) = \neg a \vee \neg b\\ \neg(a \vee b) = \neg a \wedge \neg b\end{aligned}}$$

SAT $\leq_p$ 3-CNF-SAT

d) Construct Φ''' in which each clause C_i exactly 3 distinct literals

- 3 distinct literals: $C_i = l_1 \vee l_2 \vee l_3$
- 2 distinct literals: $C_i = l_1 \vee l_2$

$$C_i = l_1 \vee l_2 = (l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$$

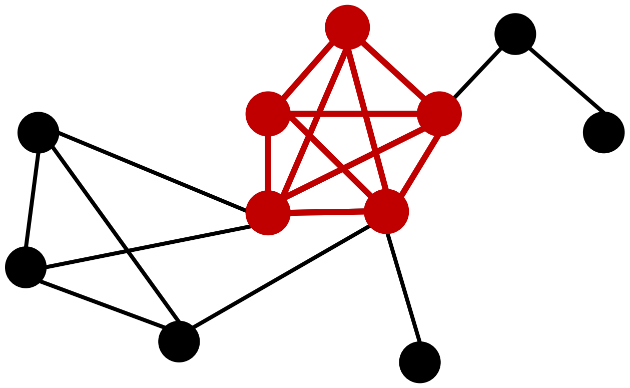
- 1 literal only: $C_i = l$

$$C_i = l = (l \vee p \vee q) \wedge (l \vee \neg p \vee q) \wedge (l \vee p \vee \neg q) \wedge (l \vee \neg p \vee \neg q)$$

- Φ''' is satisfiable iff Φ is satisfiable
- All transformation can be done in polynomial time
- → 3-CNF-SAT is NP-Complete

Clique Problem

- A clique in G = (V, E) is a *complete* subgraph of G
  - Each pair of vertices in a clique is connected by an edge in *E*
  - Size of a clique = # of vertices it contains
- Optimization problem: find a max clique in G
- Decision problem: is there a clique with size larger than *k*



Does G contain a clique of size 4? Yes

Does G contain a clique of size 5? Yes

Does G contain a clique of size 6? No

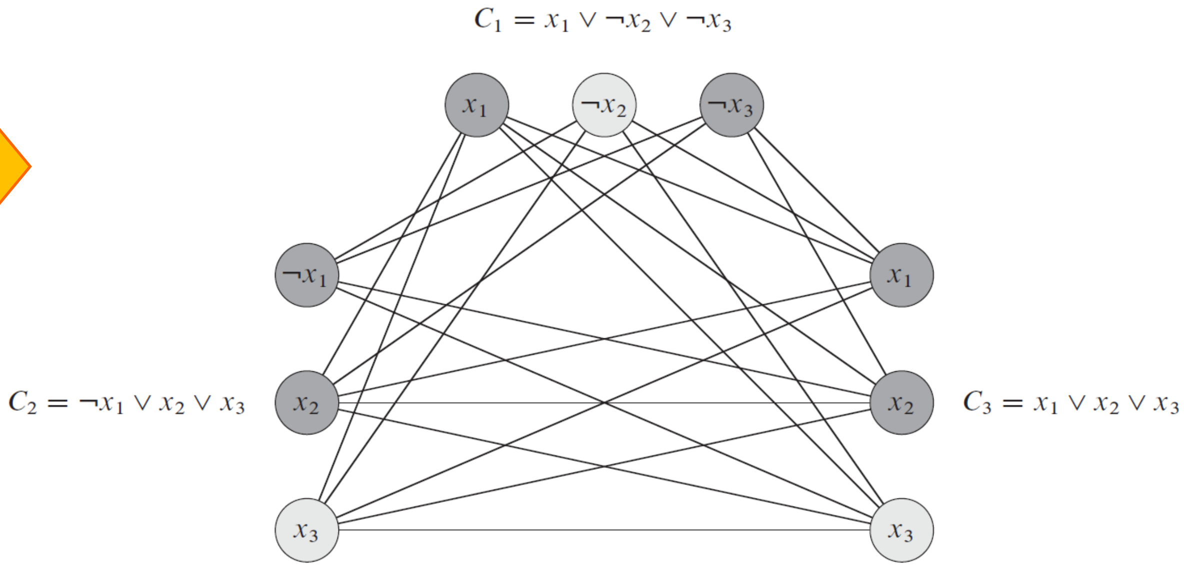
CLIQUE ∈ NP-Complete

CLIQUE = {<G, *k*>: G is a graph containing a clique of size *k*}

- Is CLIQUE ∈ NP-Complete? 3-CNF-SAT $\le_p$ CLIQUE
- Construct a reduction *f* transforming every 3-CNF-SAT instance to a CLIQUE instance
- **a graph G s.t. Φ with *k* clauses is satisfiable ⇔ G has a clique of size k**

$$\phi = (x_1 \lor \neg x_2 \lor \neg x_3) \land (\neg x_1 \lor x_2 \lor x_3) \land (x_1 \lor x_2 \lor x_3)$$

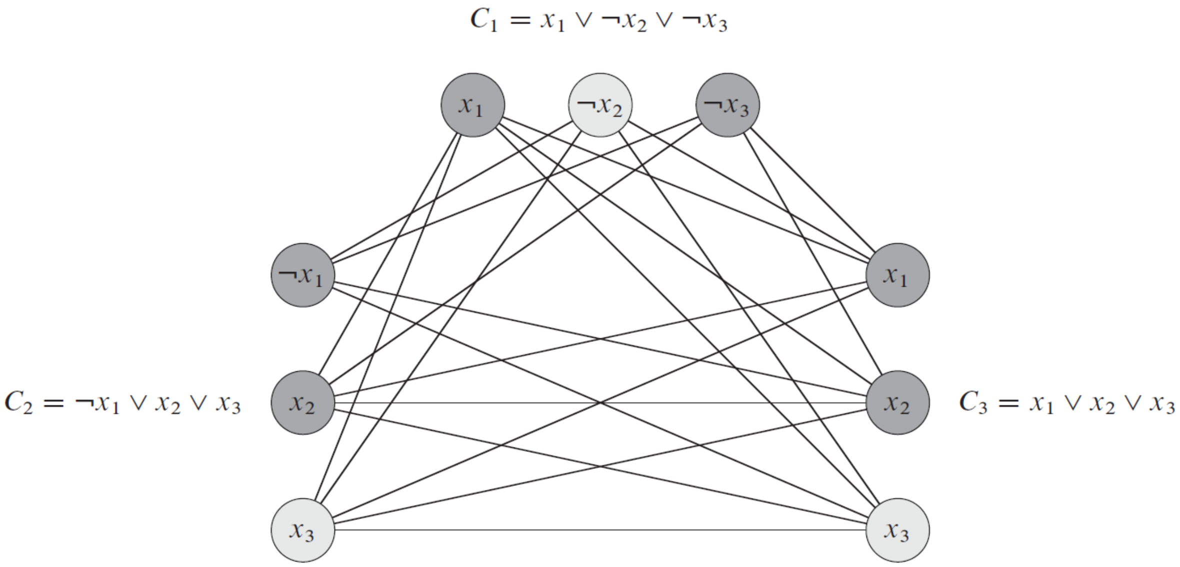
$\le_p$



$C_1 = x_1 \lor \neg x_2 \lor \neg x_3$
x_1
$\neg x_2$
$\neg x_3$
$\neg x_1$
x_1
$C_2 = \neg x_1 \lor x_2 \lor x_3$
x_2
x_2
$C_3 = x_1 \lor x_2 \lor x_3$
x_3
x_3

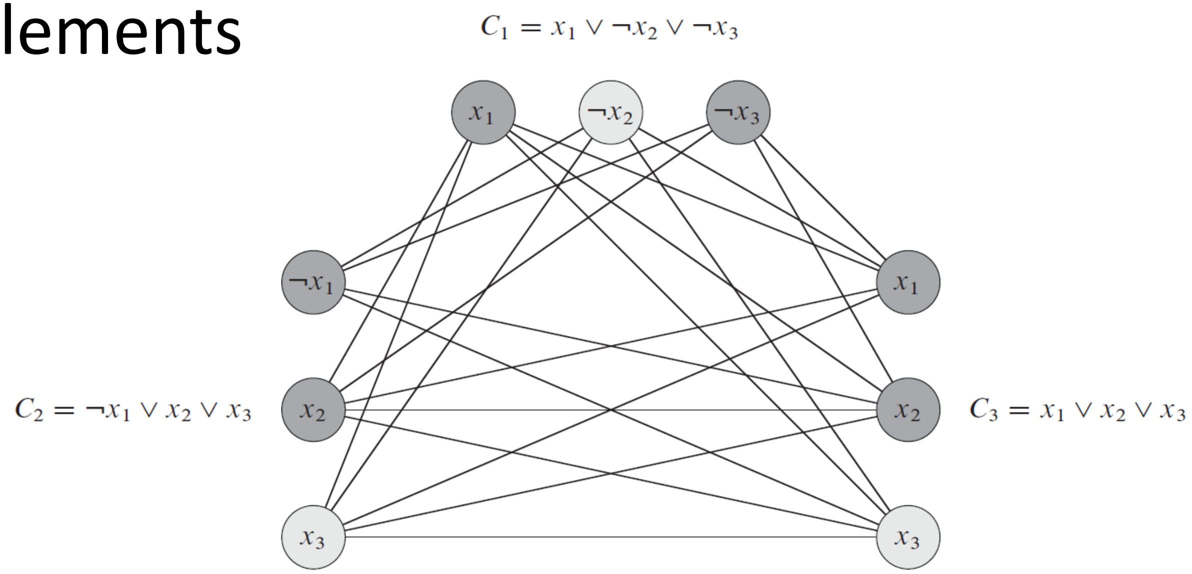
CLIQUE ∈ NP-Complete

- Polynomial-time reduction:
- Let $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_k$ be a Boolean formula in 3-CNF with *k* clauses, and each C_r has exactly 3 distinct literals l_1^r, l_2^r, l_3^r
- For each $C_r = (l_1^r \vee l_3^r \vee l_3^r)$, introduce a triple of vertices v_1^r, v_2^r, v_3^r in V
- Build an edge between v_i^r, v_j^s if both of the following hold:
  - v_i^r and v_j^s are in different triples
  - l_i^r is not the negation of l_j^s


$C_1 = x_1 \vee \neg x_2 \vee \neg x_3$
x_1
$\neg x_2$
$\neg x_3$
$\neg x_1$
x_2
x_3
$C_2 = \neg x_1 \vee x_2 \vee x_3$
x_1
x_2
x_3
$C_3 = x_1 \vee x_2 \vee x_3$

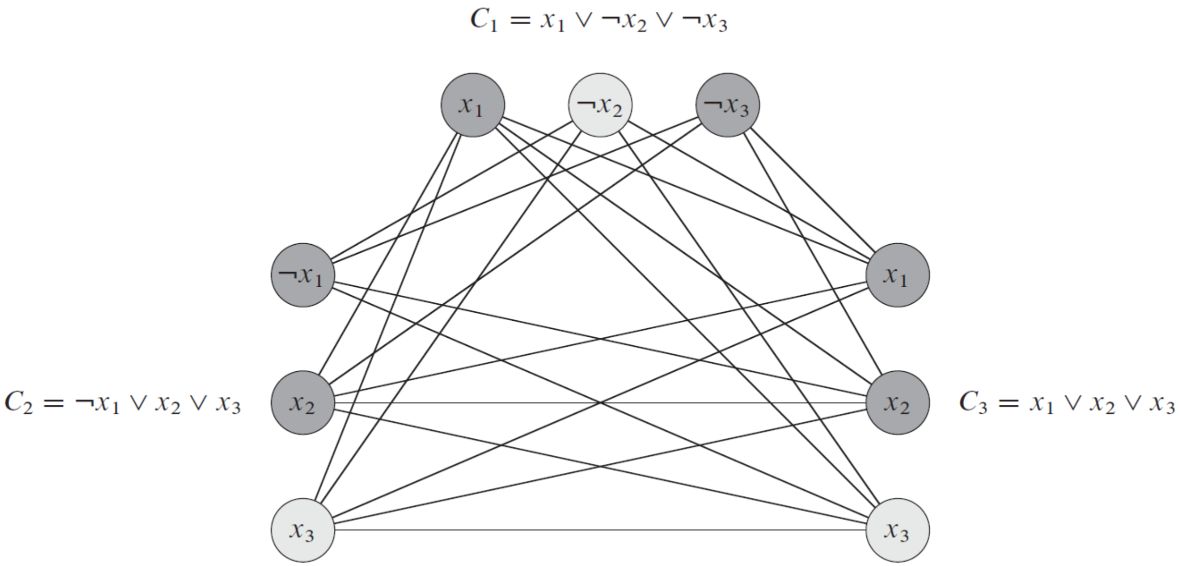
3-CNF-SAT $\leq_p$ CLIQUE

- Correctness proof: **Φ is satisfiable → G has a clique of size *k***
- If Φ is satisfiable
- → Each C_r contains at least one $l_i^r = 1$ and such literal corresponds to v_i^r
- → Pick a TRUE literal from each C_r forms a set of V’ of *k* vertices
- → For any two vertices $v_i^r, v_j^s \in V'(r \neq s)$, edge $(v_i^r, v_j^s) \in E$, because $l_i^r = l_j^s = 1$ and they cannot be complements


$C_1 = x_1 \vee \neg x_2 \vee \neg x_3$
x_1
$\neg x_2$
$\neg x_3$
$C_2 = \neg x_1 \vee x_2 \vee x_3$
$\neg x_1$
x_2
x_3
$C_3 = x_1 \vee x_2 \vee x_3$
x_1
x_2
x_3

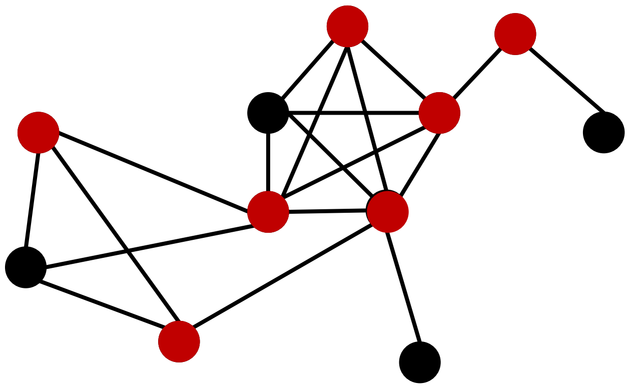
3-CNF-SAT $\leq_p$ CLIQUE

- Correctness proof: **G has a clique of size *k* → Φ is satisfiable**
- G has a clique V' of size k
- → V' contains exactly one vertex per triple since no edges connect vertices in the same triple
- → Assign 1 to each l_i^r where $v_i^r \in V'$ s.t. each C_r is satisfiable, and so is Φ


$C_1 = x_1 \vee \neg x_2 \vee \neg x_3$
x_1
$\neg x_2$
$\neg x_3$
$\neg x_1$
x_2
x_3
$C_2 = \neg x_1 \vee x_2 \vee x_3$
x_1
x_2
x_3
$C_3 = x_1 \vee x_2 \vee x_3$

Vertex Cover Problem

- A vertex cover of G = (V, E) is a subset V’ ⊆ V s.t. if (w, v) ∈ E, then w ∈ V’ or v ∈ V’
  - A vertex cover “covers” every edge in G
- Optimization problem: find a minimum size vertex cover in G
- Decision problem: is there a vertex cover with size smaller than *k*



Does G have a vertex cover of size 11? Yes

Does G have a vertex cover of size 7? Yes

Does G have a vertex cover of size 6? No

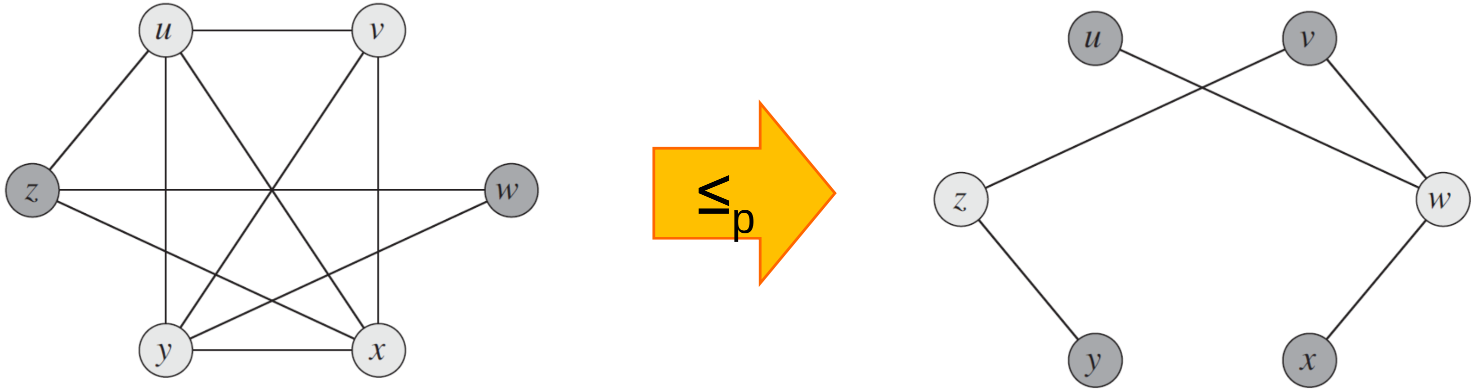
VERTEX-COVER ∈ NP-Complete

VERTEX-COVER = {<G, *k*>: G is a graph containing a vertex cover of size *k*}

- Is VERTEX-COVER ∈ NP-Complete?

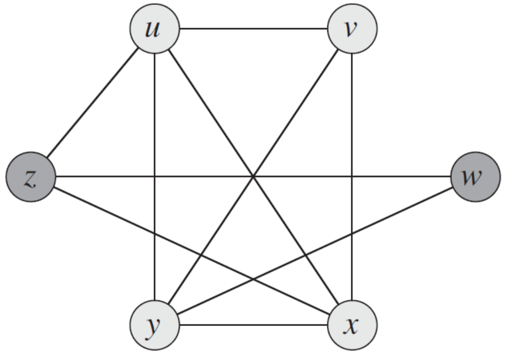
CLIQUE $\leq_p$ VERTEX-COVER

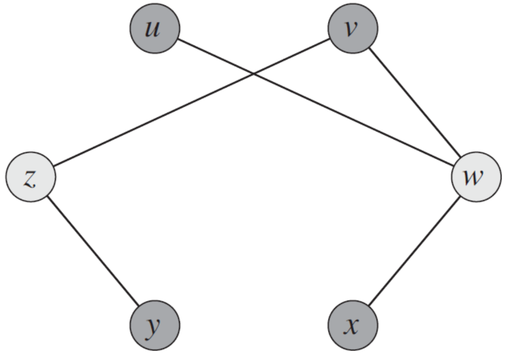
- Construct a reduction *f* transforming every CLIQUE instance to a VERTEX-COVER instance (polynomial-time reduction)
  - Compute the *complement* of G
  - Given G = <V, E>, G_c is defined as <V, E_c> s.t. E_c = {(u,v) | (u,v) ∉ E}
- **a graph G has a clique of size *k* ⇔ G_c has a vertex cover of size |V| - *k***



CLIQUE $\leq_p$ VERTEX-COVER

- Correctness proof:
  a graph G has a clique of size *k* → G_c has a vertex cover of size |V| - *k*
- If G has a clique V’ ⊆ V with |V’| = *k*
- → for all $(w, v) \in E_c$, at least one of w or $v \notin V'$
- → $w \in V - V'$ or $v \in V - V'$ (or both)
- → edge (w, v) is covered by $V - V'$
- → $V - V'$ forms a vertex cover of G_c, and |V - V'| = |V| - *k*

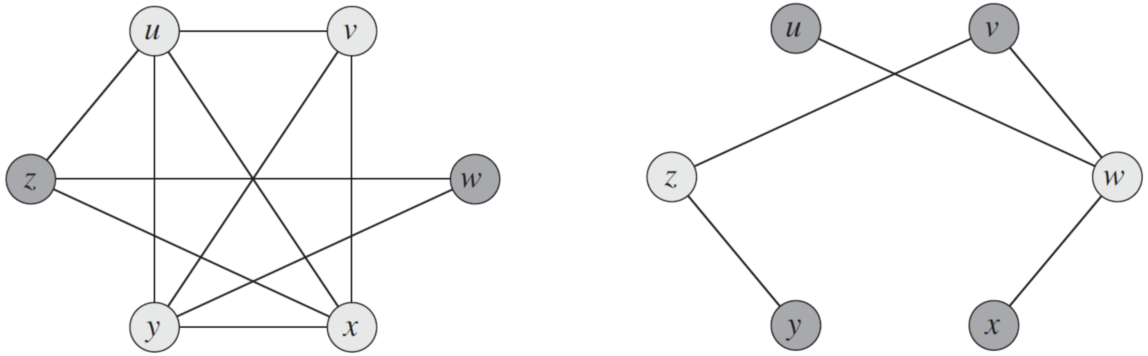




CLIQUE $\leq_p$ VERTEX-COVER

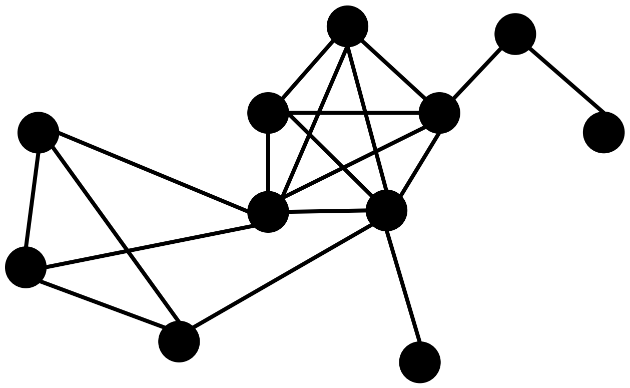
- Correctness proof:

  G_c has a vertex cover of size |V| - *k* → a graph G has a clique of size *k*
- If G_c has a vertex cover V’ ⊆ V with |V’| = |V| - *k*
- → for all $w, v \in V$, if $(w, v) \in E_c$, then $w \in V'$ or $v \in V'$ or both
- → for all $w, v \in V$, if $w \notin V'$ and $v \notin V'$, $(w, v) \in E$
- → $V - V'$ is a clique where |V - V'| = *k*



Independent-Set Problem

- An independent set of G = (V, E) is a subset V’ ⊆ V such that G has no edge between any pair of vertices in V’
  - A vertex cover “covers” every edge in G
- Optimization problem: find a maximum size independent set
- Decision problem: is there an independent set with size larger than *k*



Does G have an independent set of size 1?

Does G have an independent set of size 4?

Does G have an independent set of size 5?

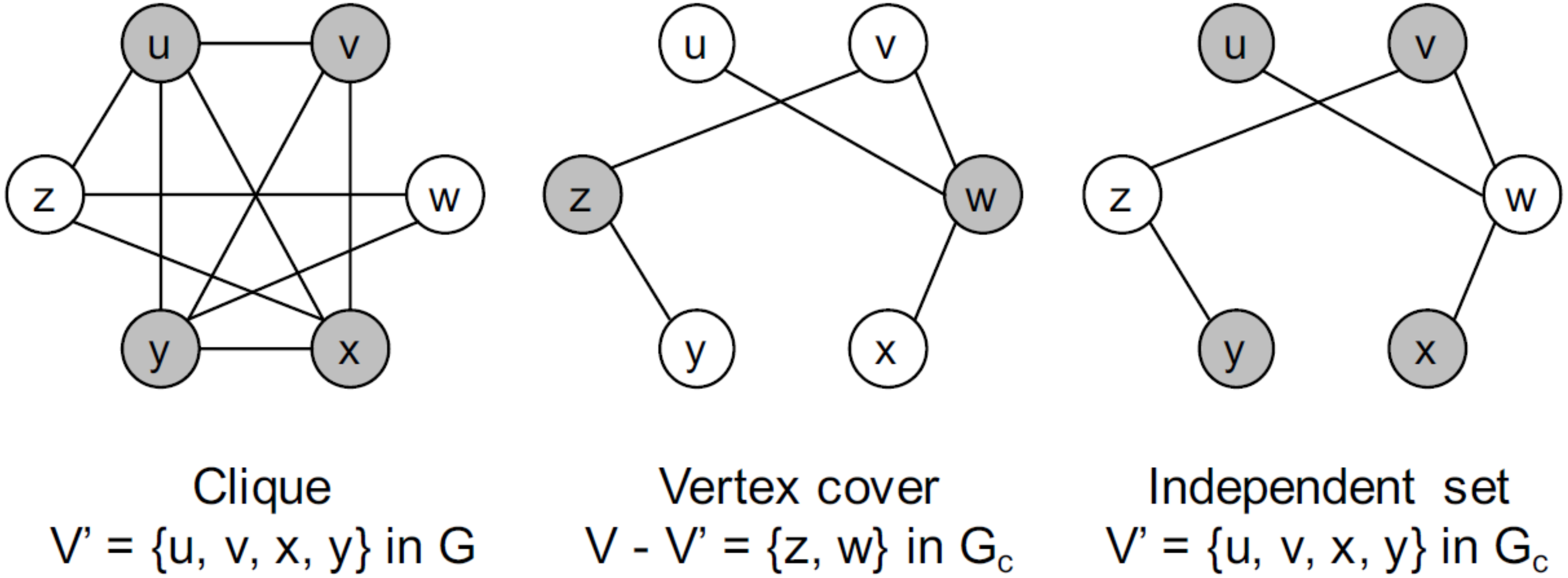
IND-SET ∈ NP-Complete

IND-SET = {<G, *k*>: G is a graph containing an independent set of size *k*}

- Is IND-SET ∈ NP-Complete?
- Practice by yourself (textbook problem 34-1)

CLIQUE, VERTEX-COVER, IND-SET

- The following are equivalent for G = (V, E) and a subset V’ of V:
  1) V' is a clique of G
  2) V-V' is a vertex cover of G_c
  3) V' is an independent set of G_c



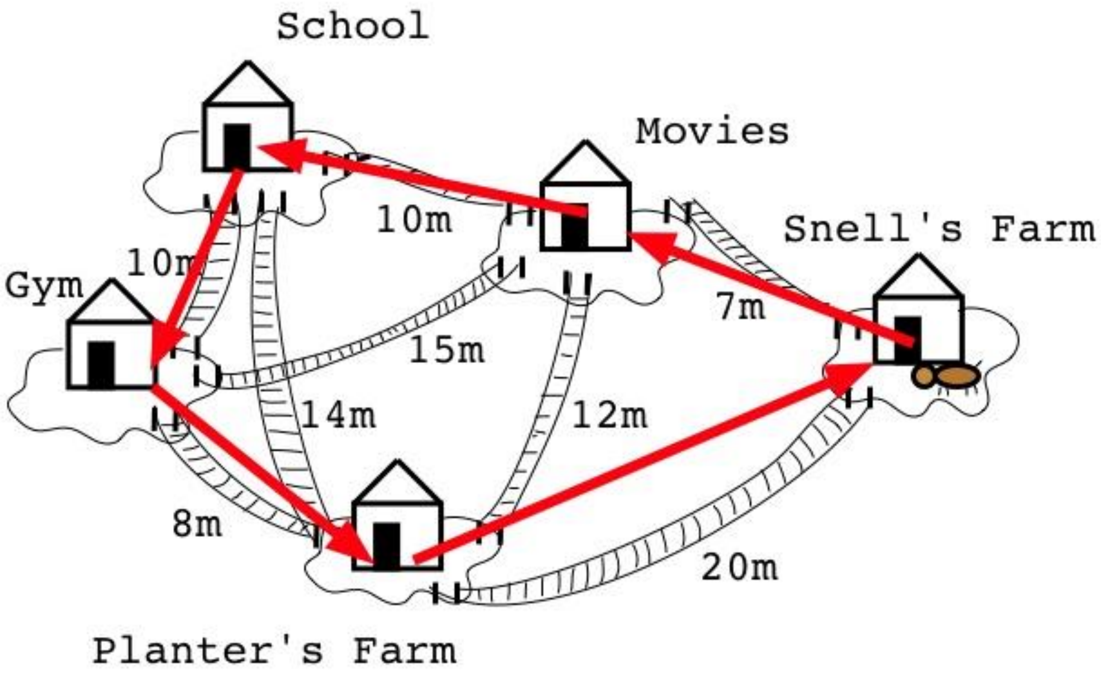
Clique
V’ = {u, v, x, y} in G

Vertex cover
V - V’ = {z, w} in G_c

Independent set
V’ = {u, v, x, y} in G_c

Traveling Salesman Problem (TSP)

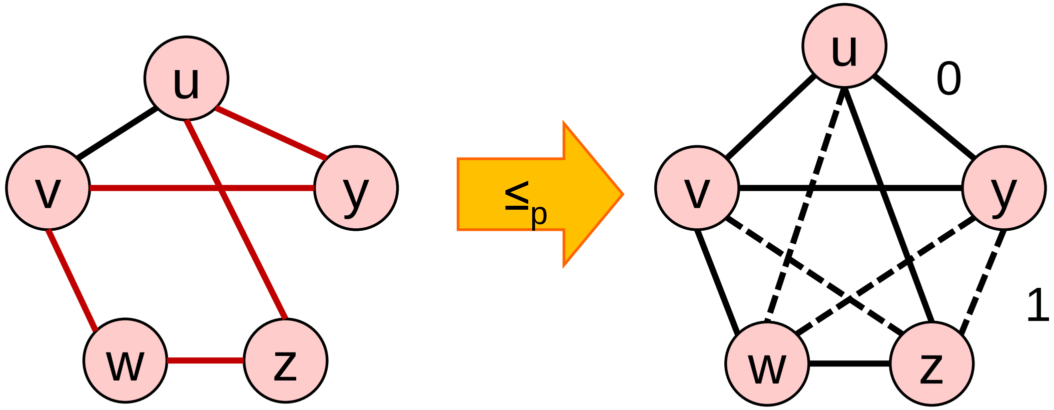
- Optimization problem: Given a set of cities and their pairwise distances, find a tour of lowest cost that visits each city exactly once.
- Decision problem: is there a traveling salesman tour with cost at most *k*



TSP ∈ NP-Complete

TSP = {<G, c, k>: G = (V,E) is a complete graph, c is a cost function for edges, G has a traveling-salesman tour with cost at most k}

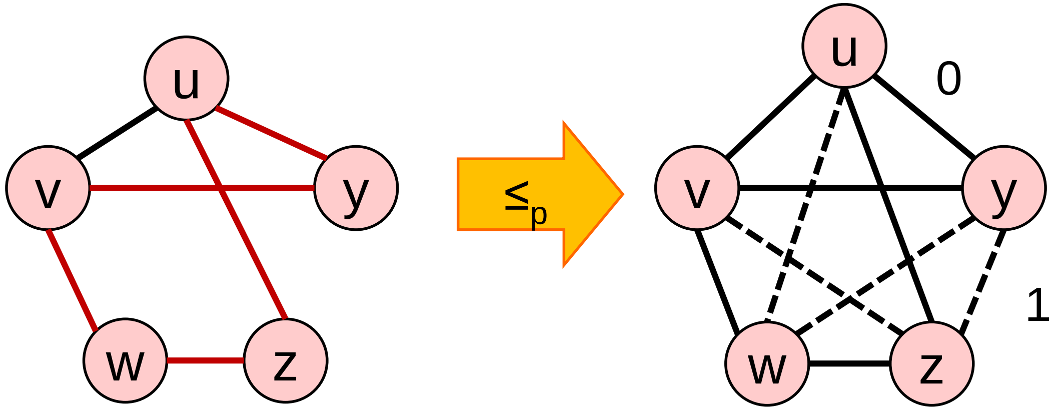
- Is TSP ∈ NP-Complete? HAM-CYCLE $\leq_p$ TSP
- Construct a reduction *f* transforming every HAM-CYCLE instance to a TSP instance (polynomial-time reduction)
- **G contains a Hamiltonian cycle *h* = <v_1, v_2, …, v_n, v_1> ⇔ <v_1, v_2, …, v_n, v_1> is a traveling-salesman tour with cost 0**



$$c(i,j) = \begin{cases} 0 & \text{if } (i,j) \in E \\ 1 & \text{if } (i,j) \notin E \end{cases}$$

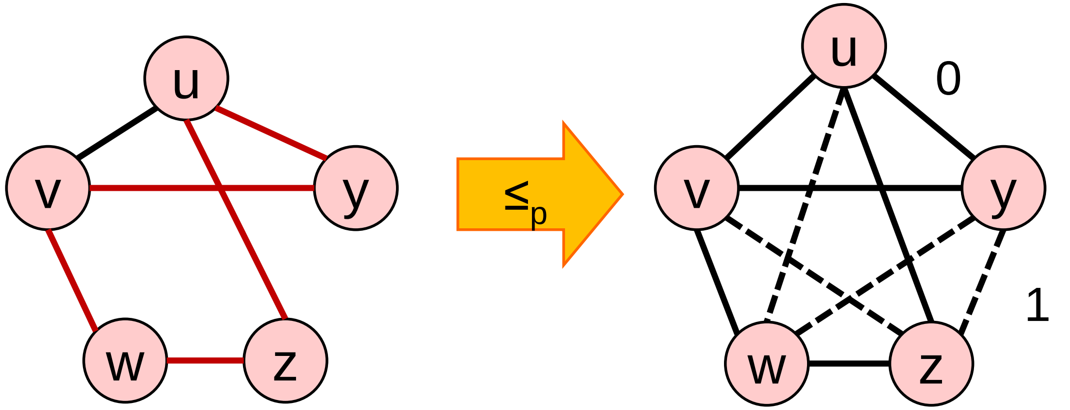
HAM-CYCLE $\leq_p$ TSP

- Correctness proof: ***x*** **∈ HAM-CYCLE →** ***f*(*x*) ∈ TSP**
- If Hamiltonian cycle is $h = \langle v_1, v_2, \ldots, v_n, v_1 \rangle$
- → *h* is also a tour in the transformed TSP instance
- → The distance of the tour *h* is *0* since there are *n* consecutive edges in *E*, and so has distance 0 in *f*(*x*)
- → *f*(*x*) ∈ TSP (*f*(*x*) has a TSP tour with cost ≤ 0)



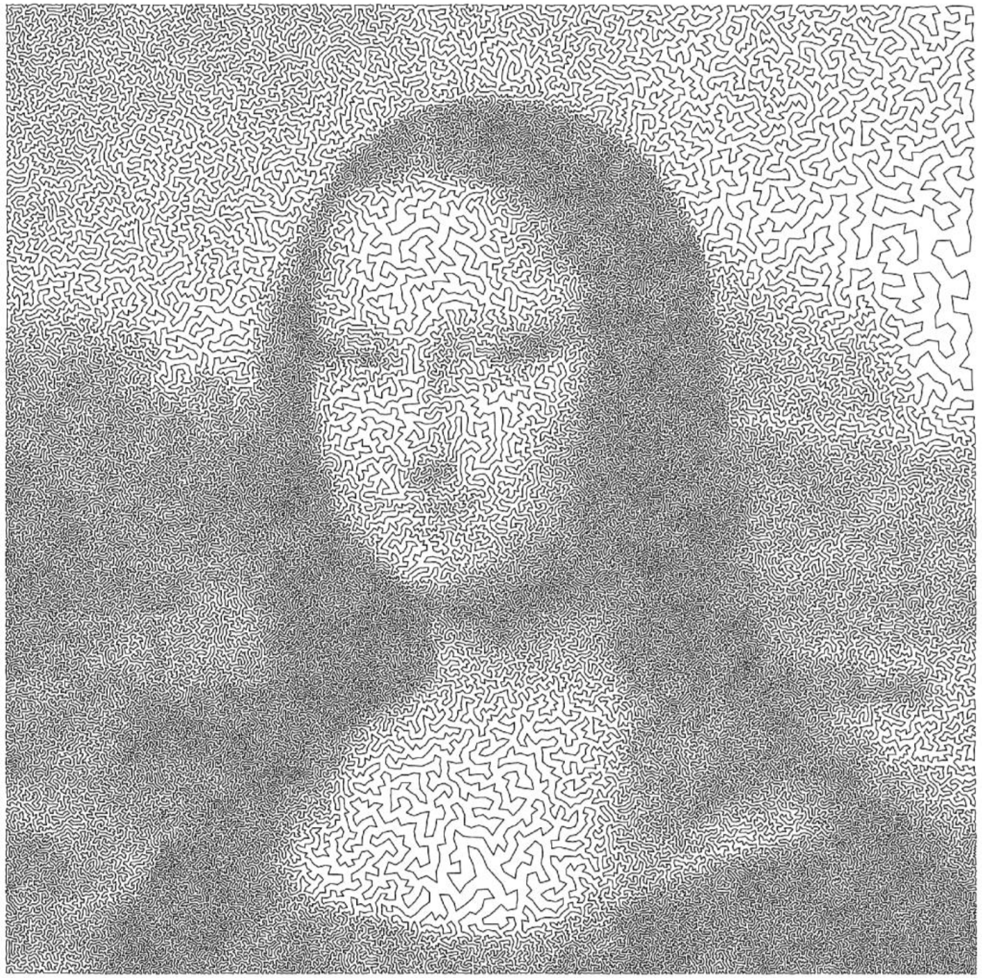
HAM-CYCLE $\leq_p$ TSP

- Correctness proof: ***f*(*x*) ∈ TSP → *x* ∈ HAM-CYCLE**
- **After reduction**, if a TSP tour with cost ≤ 0 as $<v_1, v_2, \ldots, v_n, v_1>$
- → The tour contains only edges in *E*
- → Thus, $<v_1, v_2, \ldots, v_n, v_1>$ is a Hamiltonian cycle



TSP Challenges

- Mona Lisa TSP: $1,000 Prize for 100,000-city



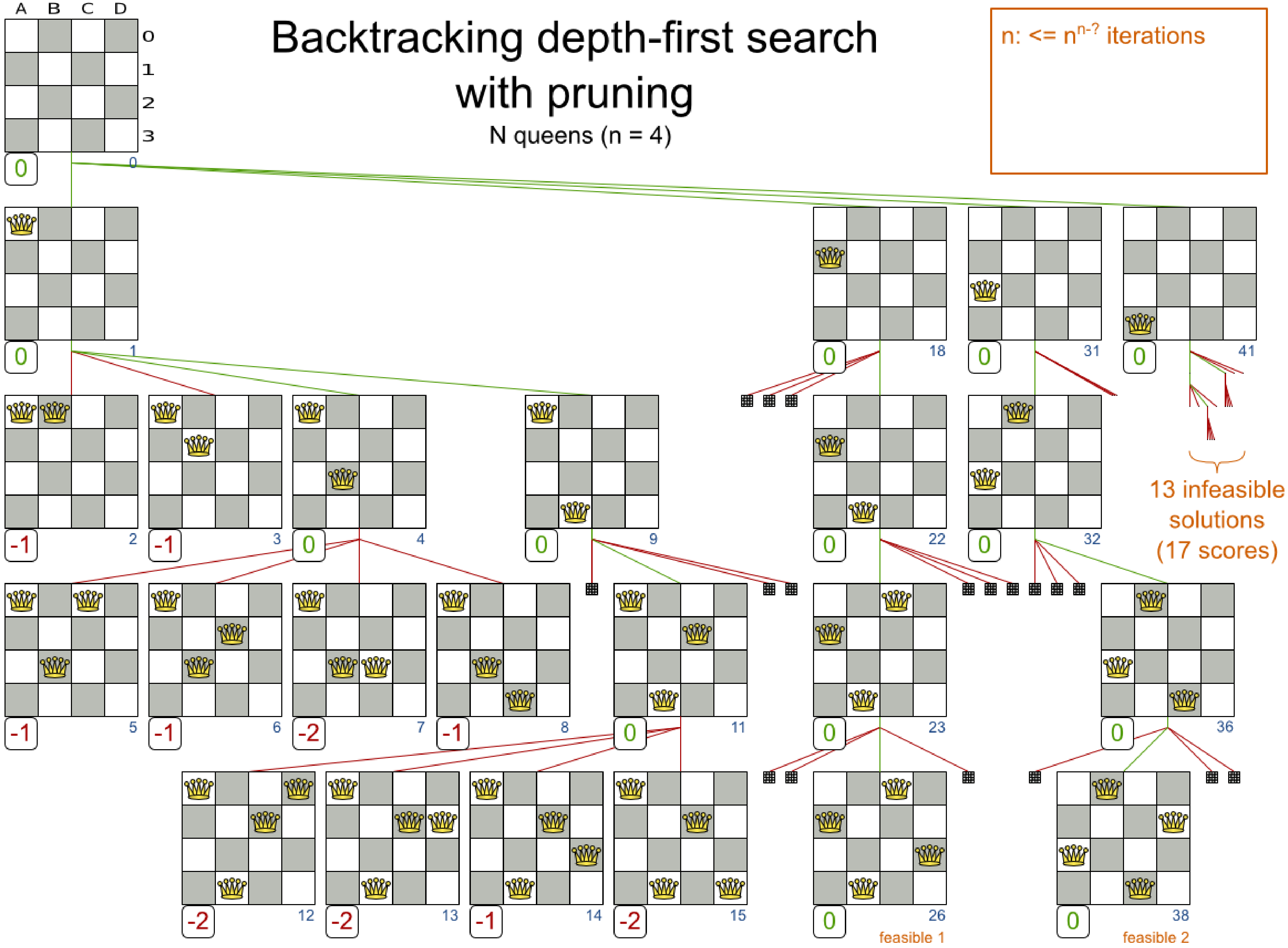
Strategies for NP-Complete/NP-Hard Problems

- NP-complete/NP-hard problems are unlikely to have polynomial-time solutions (unless P = NP), we must sacrifice either **optimality**, **efficiency**, or **generality**
  - **Approximation algorithms:** guarantee to be a fixed percentage away from the optimum
  - **Local search:** simulated annealing (hill climbing), genetic algorithms, etc
  - **Heuristics:** no formal guarantee of performance
  - **Randomized algorithms:** use a randomizer (random number generator) for operation
  - **Pseudo-polynomial time algorithms:** e.g., DP for 0-1 knapsack
  - **Exponential algorithms/Branch and Bound/Exhaustive search:** feasible only when the problem size is small
  - **Restriction:** work on some special cases of the original problem. e.g., the maximum independent set problem in circle graphs

Backtracking depth-first search with pruning

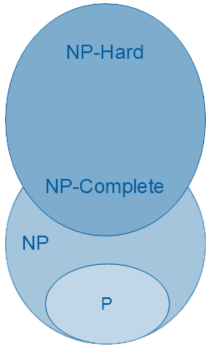
N queens (n = 4)

n: <= $n^{n-?}$ iterations

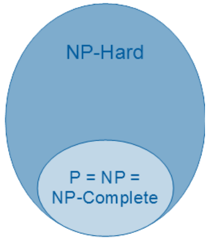


13 infeasible solutions (17 scores)

Concluding Remarks

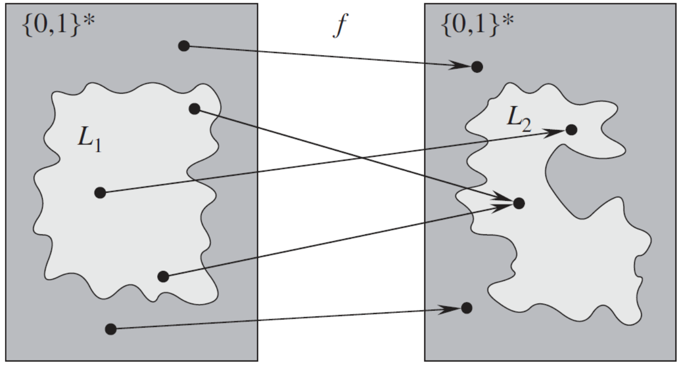


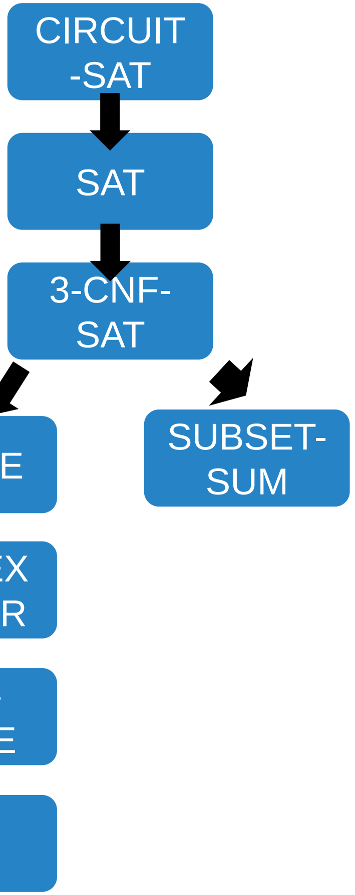
P ≠ NP



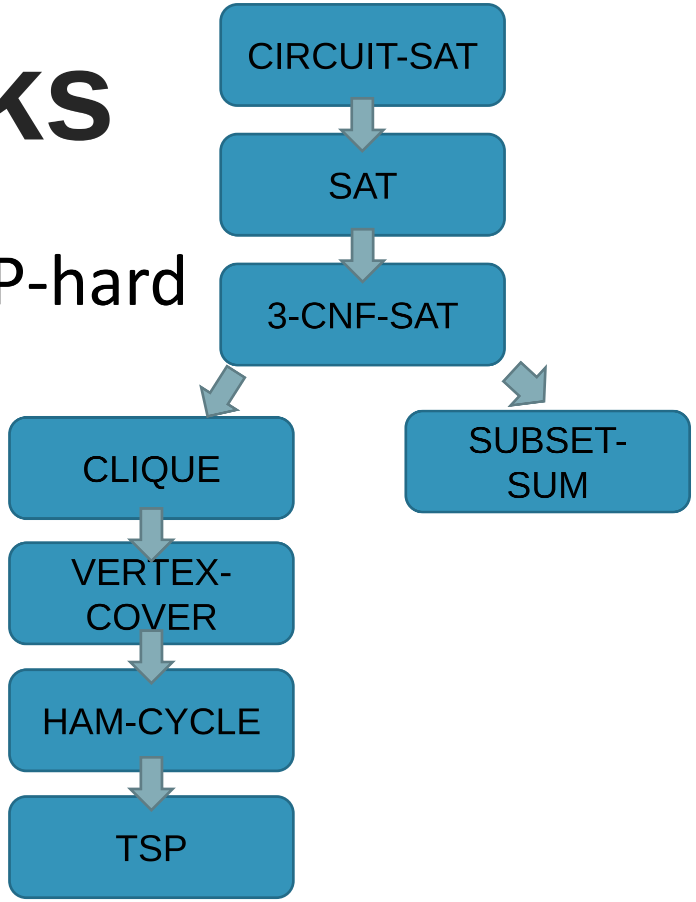
P = NP

- Proving NP-Completeness: *L* ∈ NPC iff *L* ∈ NP and *L* ∈ NP-hard
- Polynomial-time verification
- Step-by-step approach for proving L in NPC:
  - Prove L ∈ NP
  - Prove L ∈ NP-hard
    - Select a known NPC problem C
    - Construct a reduction f transforming every instance of C to an instance of *L*
    - Prove that $x \in C \iff f(x) \in C, \forall x \in \{0,1\}^*$
    - Prove that f is a polynomial time transformation L ∈ NP
- Strategies for NP-complete/NP-hard problems





Concluding Remarks


CIRCUIT-SAT

SAT

3-CNF-SAT

CLIQUE

SUBSET-SUM

VERTEX-COVER

HAM-CYCLE

TSP

- Proving NP-Completeness: *L* ∈ NPC iff *L* ∈ NP and *L* ∈ NP-hard
- Polynomial-time verification
- Step-by-step approach for proving *L* in NPC:
  - Prove *L* ∈ NP
  - Prove *L* ∈ NP-hard
    1) Select a known NPC problem C
    2) Construct a reduction f transforming every instance of C to an instance of L
    3) Prove that
    4) Prove that *f* is a polynomial time transformation

$$x \in C \iff f(x) \in L, \forall x \in \{0,1\}^*$$

- Strategies for NP-complete/NP-hard problems

Question?

Important announcement will be sent to
@ntu.edu.tw mailbox & post to the course website

Course Website: http://ada.miulab.tw
Email: ada-ta@csie.ntu.edu.tw