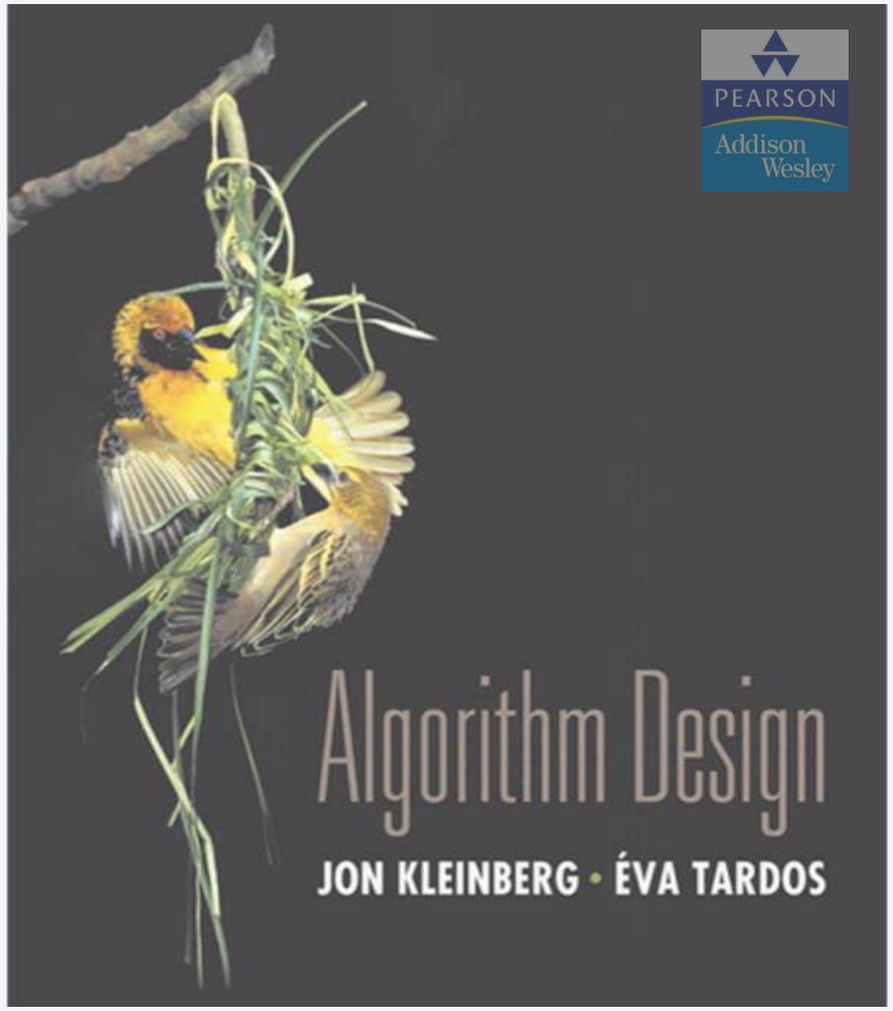
CS101 Algorithms and Data Structures

Reductions, P and NP (Highlighted)

By Zhixuan Zhou



REDUCTIONS, P AND NP

- *poly-time reductions*
- *packing and covering problems*
- *constraint satisfaction problems*
- *graph coloring*
- *P vs. NP*
- *NP-complete*

SECTION 8.1

Algorithm design patterns and antipatterns

Algorithm design patterns.

- Greedy.
- Divide and conquer.
- Dynamic programming.
- Reductions.
- Duality.
- Local search.
- Randomization.

Algorithm design antipatterns.

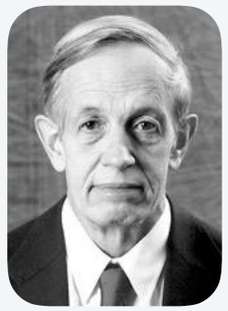
- **NP**-completeness. $O(n^k)$ algorithm unlikely.
- **PSPACE**-completeness. $O(n^k)$ certification algorithm unlikely.
- Undecidability. No algorithm possible.

Classify problems according to computational requirements

Q. Which problems will we be able to solve in practice?

A working definition. Those with poly-time algorithms.

von Neumann (1953)



Nash (1955)

Gödel (1956)

Cobham (1964)

Edmonds (1965)

Rabin (1966)

Theory. Definition is broad and robust.

Turing machine, word RAM, uniform circuits, ...

Practice. Poly-time algorithms scale to huge problems.

constants tend to be small, e.g., $3n^2$

Classify problems according to computational requirements

Q. Which problems will we be able to solve in practice?

A working definition. Those with poly-time algorithms.

yes	probably no
shortest path	longest path
min cut	max cut
2-satisfiability	3-satisfiability
planar 4-colorability	planar 3-colorability
bipartite vertex cover	vertex cover
matching	3d-matching
primality testing	factoring
linear programming	integer linear programming

Classify problems

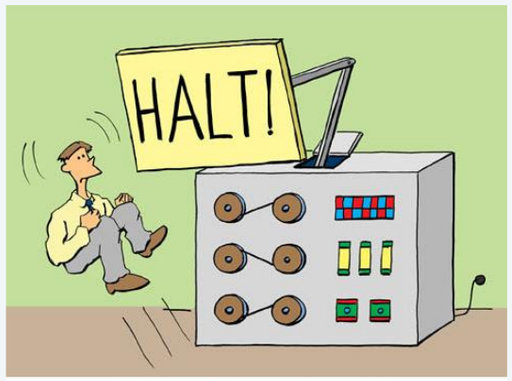
Desiderata. Classify problems according to those that can be solved in polynomial time and those that cannot.

Provably requires exponential time.

input size = $c + \log k$

- Given a constant-size program, does it halt in at most k steps?
- Given a board position in an n-by-n generalization of checkers, can black guarantee a win?

using forced capture rule



Alan designed the perfect computer

Frustrating news. Huge number of fundamental problems have defied classification for decades.

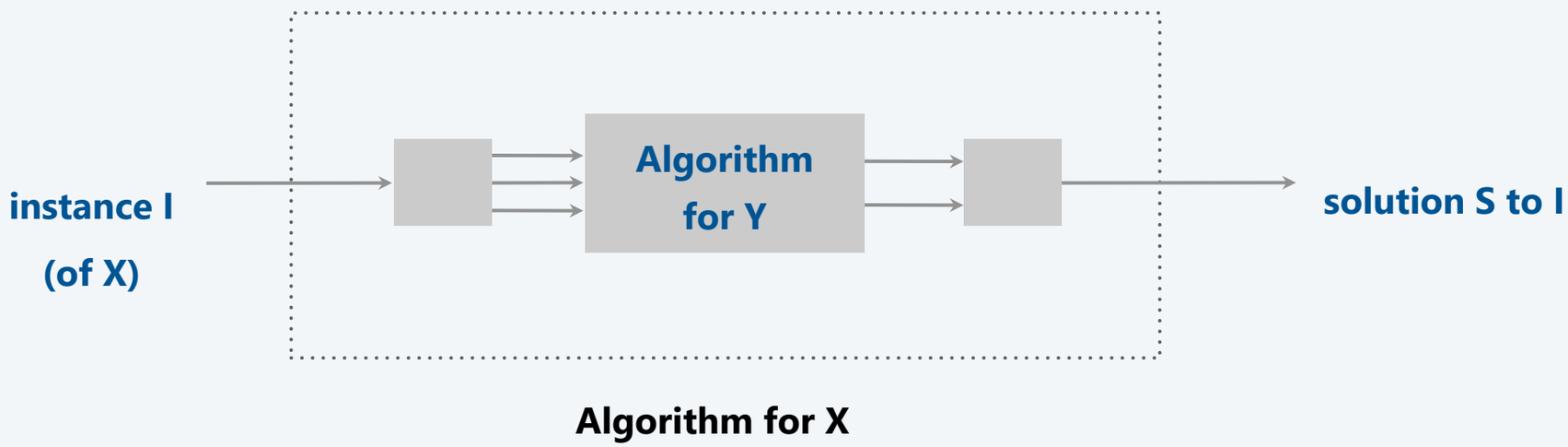
Polynomial-time **Turing** reductions (**Cook** Reductions)

Desiderata'. Suppose we could solve problem Y in polynomial time. What else could we solve in polynomial time?

Reduction. Problem X polynomial-time Turing reduces (Cook) to problem Y if **arbitrary** instances (An instance of a problem consists of an input for the problem) of problem X can be solved using:

- Polynomial number of standard computational steps, plus
- Polynomial number of calls to oracle that solves problem Y.

Abstract machine (black box) that is capable of solving certain problems in a single operation



Polynomial-time **Turing** reductions (**Cook** Reductions)

Desiderata'. Suppose we could solve problem Y in polynomial time. What else could we solve in polynomial time?

Reduction. Problem X polynomial-time Turing reduces (Cook) to problem Y if **arbitrary** instances (An instance of a problem consists of an input for the problem) of problem X can be solved using:

- Polynomial number of standard computational steps, plus
- Polynomial number of calls to oracle that solves problem Y.

Notation. $X \leq_T^P Y$.

Note. We pay for time to write down instances of Y sent to oracle $\Rightarrow$ instances of Y must be of polynomial size.

Decision Problem & Yes-Instance

Def A decision problem is a problem with a yes / no answer.

- Ex Given a graph, is there a path from node s to t?
- Ex Given a map, is there a way to 3-color it?
- Ex Given a number, is it prime?

Def Given a decision problem, the set of yes (resp. no) instances are the instances of the problem for which the answer is yes (resp. no).

- Ex 11 is a yes instance to the prime problem, 10 is a no instance.

Polynomial-time **Many-one** Reductions (**Karp** Reductions)

Def. Problem X polynomial **transforms (Karp)** (polynomial-time **many-one** reduces) to problem Y if we can construct a mapping function f that maps instances of X to instances of Y such that given **any** instance x of X, the instance $y = f(x)$ of Y is a *yes* instance of Y iff x is a *yes* instance of X.

we require $|y|$ to be of size polynomial in $|x|$

Notation. $X \leq_m^P Y$ or $\boldsymbol{X \leq_P Y}$ **for simplicity.**

Note. Polynomial transformation is polynomial reduction with **just one** call to oracle for Y, exactly at the end of the algorithm for X. Almost all reductions in this section (except the one on page 33-35) were of this form.

Novice mistake. Confusing $X \leq_P Y$ with $Y \leq_P X$.

Suppose that $X \leq_{\mathrm{P}} Y$. Which of the following can we infer?

A. If X can be solved in polynomial time, then so can Y.

B. X can be solved in poly time iff Y can be solved in poly time.

C. If X cannot be solved in polynomial time, then neither can Y.

D. If Y cannot be solved in polynomial time, then neither can X.

The answer is C.

$X \leq_{\mathrm{P}} Y$ intuitively means that "X is no harder than Y".

Choice B would be correct if you change the " iff " to " if ".

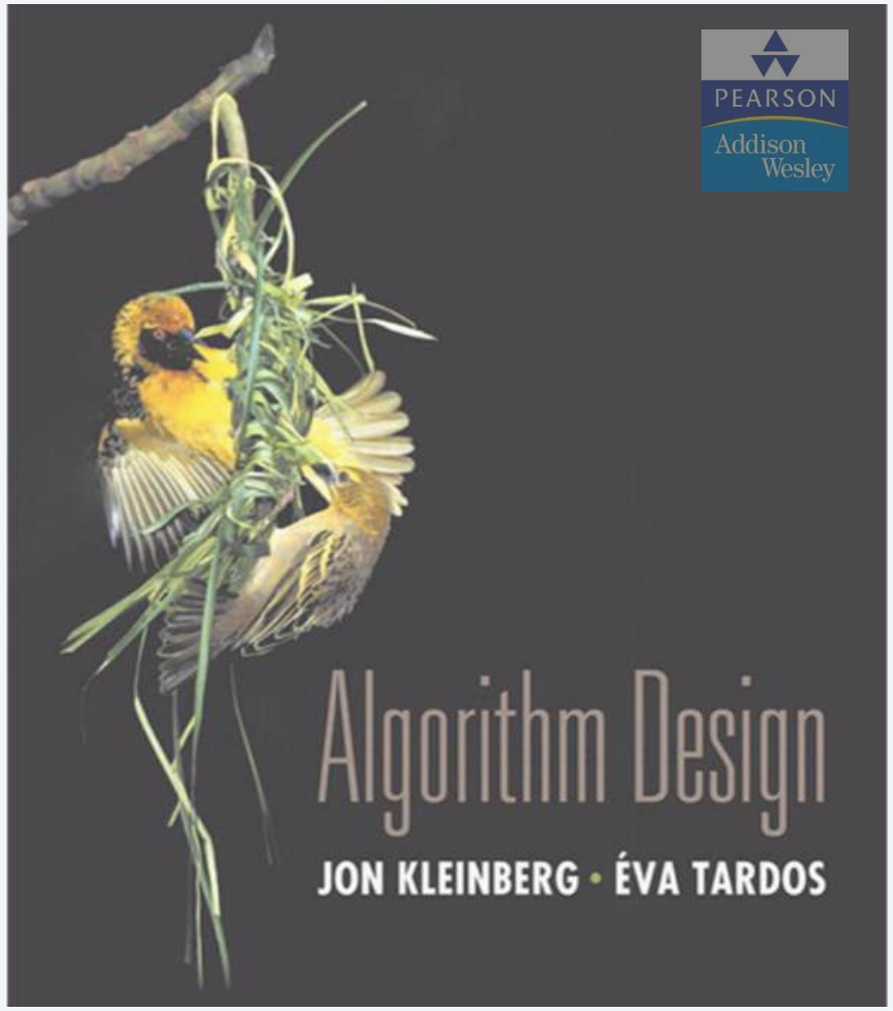
Polynomial-time **Many-one** Reductions (**Karp** Reductions)

Design algorithms. If $X \leq_{\mathrm{P}} Y$ and Y can be solved in polynomial time, then X can be solved in polynomial time.

Establish intractability. If $X \leq_{\mathrm{P}} Y$ and X cannot be solved in polynomial time, then Y cannot be solved in polynomial time.

Establish equivalence. If both $X \leq_{\mathrm{P}} Y$ and $Y \leq_{\mathrm{P}} X$, we use notation $X \equiv_{\mathrm{P}} Y$.
In this case, X can be solved in polynomial time iff Y can be.

Bottom line. Reductions classify problems according to relative difficulty.



REDUCTIONS, P AND NP

- *poly-time reductions*
- *packing and covering problems*
- *constraint satisfaction problems*
- *graph coloring*
- *P vs. NP*
- *NP-complete*

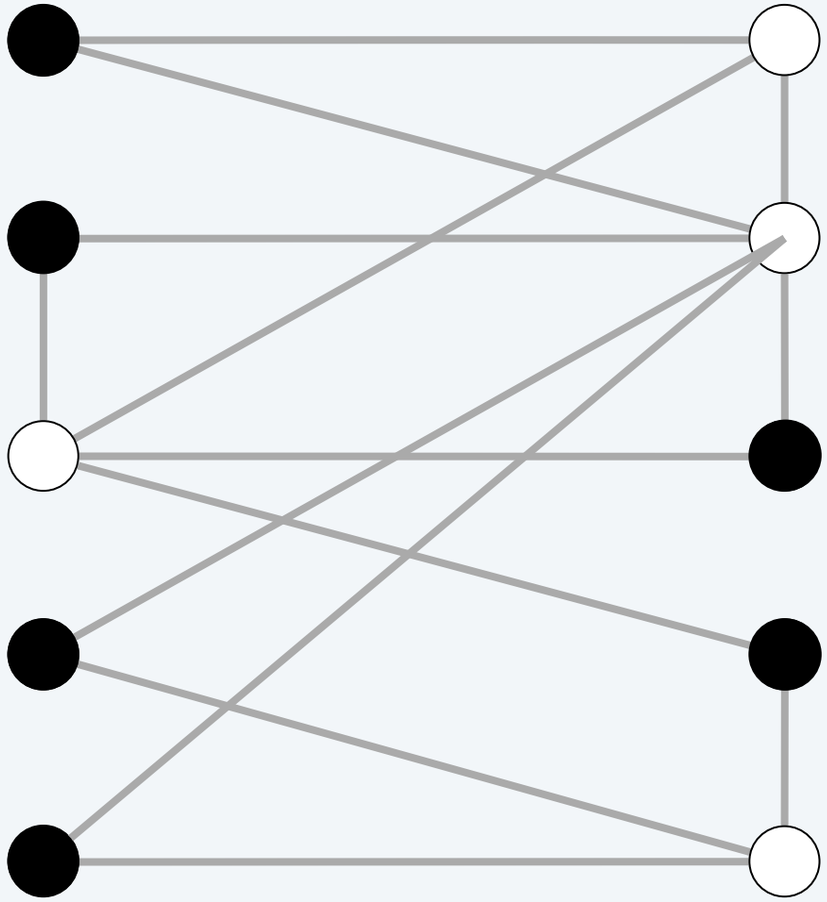
SECTION 8.1

Independent set

INDEPENDENT-SET. Given a graph $G = (V, E)$ and an integer k, is there a subset of k (or more) vertices such that no two are adjacent?

Ex. Is there an independent set of size ≥ 6?

Ex. Is there an independent set of size ≥ 7?



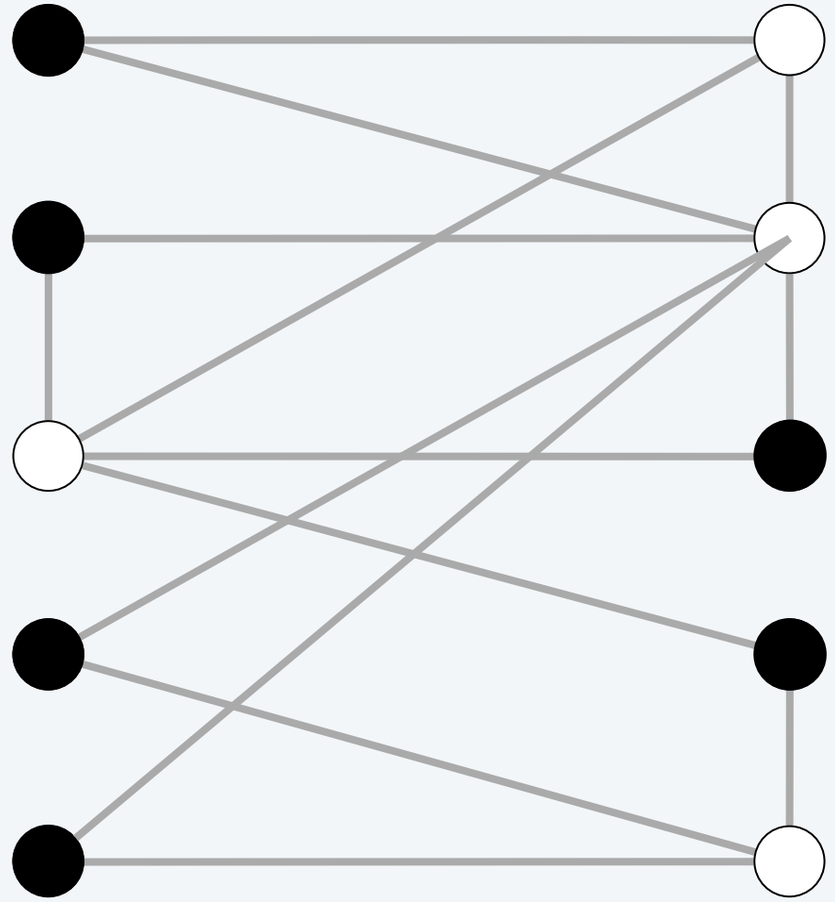
independent set of size 6

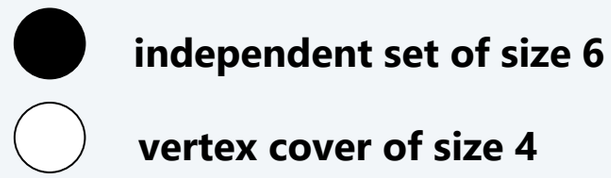
Vertex cover

VERTEX-COVER. Given a graph $G = (V, E)$ and an integer k, is there a subset of k (or fewer) vertices such that each edge is incident to at least one vertex in the subset?

Ex. Is there a vertex cover of size ≤ 4 ?

Ex. Is there a vertex cover of size ≤ 3 ?





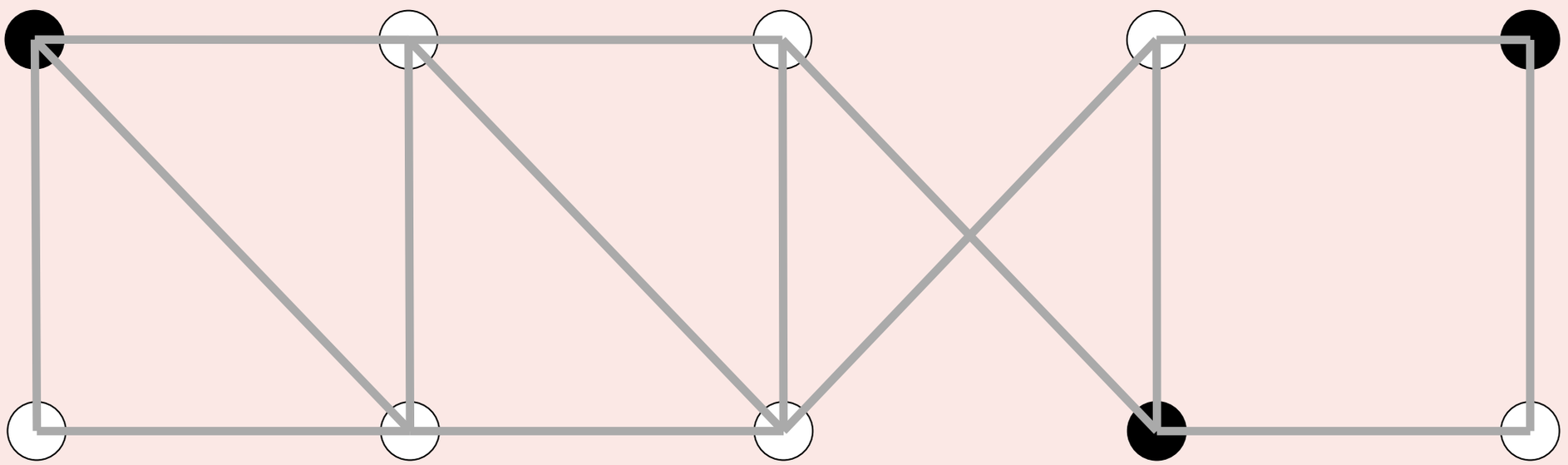
Consider the following graph G. Which are true?

A. The white vertices are a vertex cover of size 7.

B. The black vertices are an independent set of size 3.

C. Both A and B.

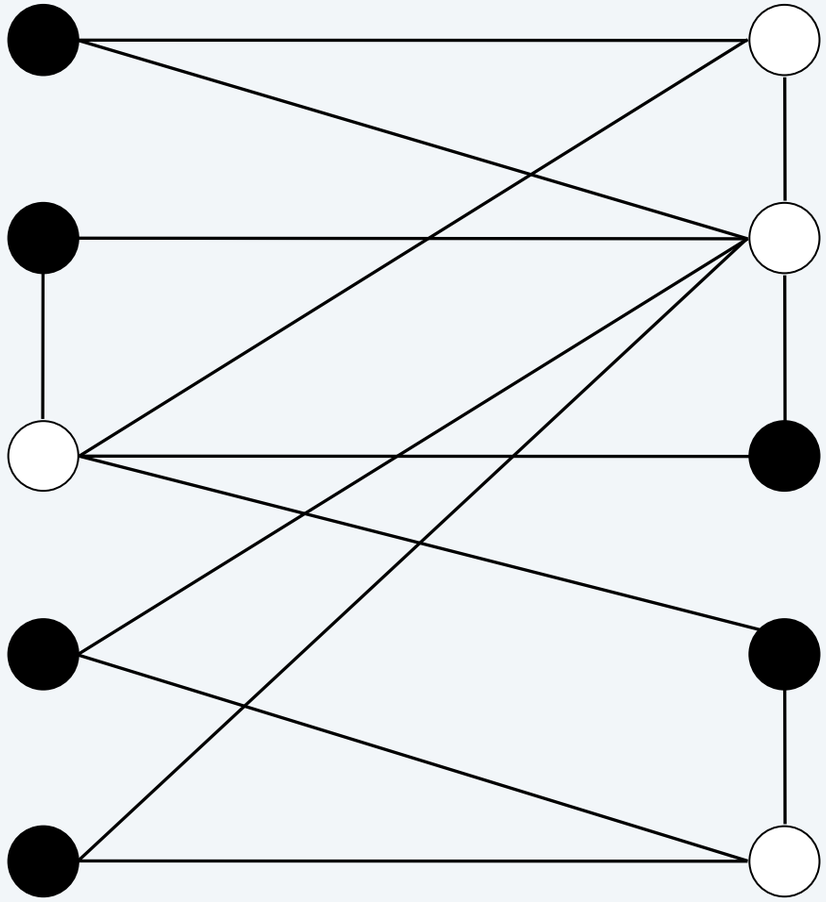
D. Neither A nor B.

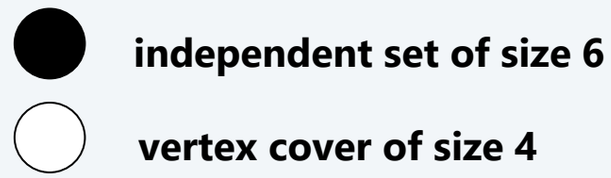


Vertex cover and independent set reduce to one another

Theorem. INDEPENDENT-SET $\equiv_P$ VERTEX-COVER.

Pf. We show S is an independent set of size k **iff** $V - S$ is a vertex cover of size $n - k$.





Vertex cover and independent set reduce to one another

Theorem. INDEPENDENT-SET $\equiv_P$ VERTEX-COVER.

Pf. We show S is an independent set of size k **iff** $V - S$ is a vertex cover of size $n - k$.

$\Rightarrow$

- Let S be any independent set of size k.
- $V - S$ is of size $n - k$.
- Consider an arbitrary edge $(u, v) \in E$.
- S independent $\Rightarrow$ either $u \notin S$, or $v \notin S$, or both.
  $\Rightarrow$ either $u \in V - S$, or $v \in V - S$, or both.
- Thus, $V - S$ covers (u, v). ▪

Vertex cover and independent set reduce to one another

Theorem. INDEPENDENT-SET $\equiv_P$ VERTEX-COVER.

Pf. We show S is an independent set of size k **iff** $V - S$ is a vertex cover of size $n - k$.

$\Leftarrow$

- Let $V - S$ be any vertex cover of size $n - k$.
- S is of size k.
- Consider an arbitrary edge $(u, v) \in E$.
- $V - S$ is a vertex cover $\Rightarrow$ either $u \in V - S$, or $v \in V - S$, or both.
  $\Rightarrow$ either $u \notin S$, or $v \notin S$, or both.
- Thus, S is an independent set. ▪

Set cover

SET-COVER. Given a set U of elements, a collection S of subsets of U, and an integer k, are there $\leq k$ of these subsets whose union is equal to U?

Sample application.

- m available pieces of software.
- Set U of n capabilities that we would like our system to have.
- The i^{th} piece of software provides the set $S_i \subseteq U$ of capabilities.
- Goal: achieve all n capabilities using fewest pieces of software.

$U = \{ 1, 2, 3, 4, 5, 6, 7 \}$

$S_a = \{ 3, 7 \}$ $S_b = \{ 2, 4 \}$

$S_c = \{ 3, 4, 5, 6 \}$ $S_d = \{ 5 \}$

$S_e = \{ 1 \}$ $S_f = \{ 1, 2, 6, 7 \}$

$k = 2$

a set cover instance

Given the universe U = { 1, 2, 3, 4, 5, 6, 7 } and the following sets, which is the minimum size of a set cover?

A. 1

B. 2

C. 3

D. None of the above.

$U = \{ 1, 2, 3, 4, 5, 6, 7 \}$

$S_a = \{ 1, 4, 6 \}$ $S_b = \{ 1, 6, 7 \}$

$S_c = \{ 1, 2, 3, 6 \}$ $S_d = \{ 1, 3, 5, 7 \}$

$S_e = \{ 2, 6, 7 \}$ $S_f = \{ 3, 4, 5 \}$

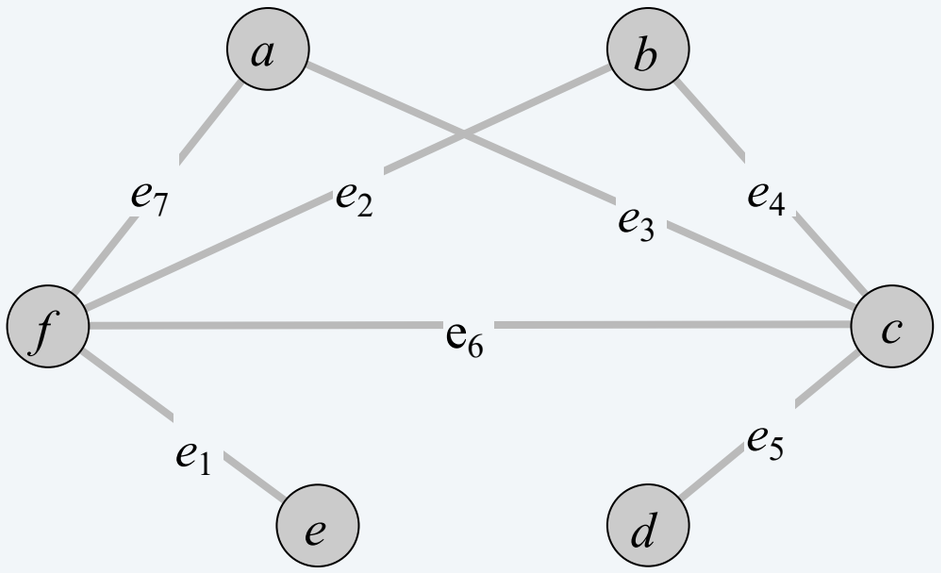
Vertex cover reduces to set cover

Theorem. VERTEX-COVER $\leq_P$ SET-COVER.

Pf. Given a VERTEX-COVER instance $G = (V, E)$ and k, we construct a SET-COVER instance (U, S, k) that has a set cover of size k **iff** G has a vertex cover of size k.

Construction.

- Universe $U = E$.
- Include one subset for each node $v \in V$: $S_v = \{e \in E : e \text{ incident to } v\}$.



vertex cover instance (k = 2)

$U = \{ 1, 2, 3, 4, 5, 6, 7 \}$

$S_a = \{ 3, 7 \}$ $S_b = \{ 2, 4 \}$

$S_c = \{ 3, 4, 5, 6 \}$ $S_d = \{ 5 \}$

$S_e = \{ 1 \}$ $S_f = \{ 1, 2, 6, 7 \}$

set cover instance (k = 2)

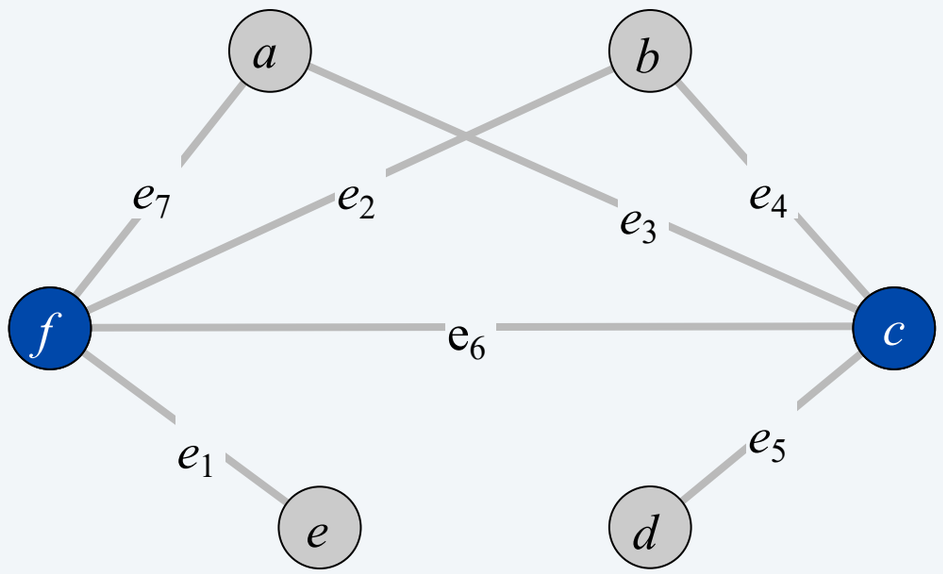
Vertex cover reduces to set cover

Lemma. $G = (V, E)$ contains a vertex cover of size k iff (U, S, k) contains a set cover of size k.

Pf. $\Rightarrow$ Let $X \subseteq V$ be a vertex cover of size k in G.

- Then $Y = \{ S_v : v \in X \}$ is a set cover of size k. ▪

"yes" instances of VERTEX-COVER are solved correctly



vertex cover instance (k = 2)

$U = \{ 1, 2, 3, 4, 5, 6, 7 \}$

$S_a = \{ 3, 7 \}$ $S_b = \{ 2, 4 \}$

$S_c = \{ 3, 4, 5, 6 \}$ $S_d = \{ 5 \}$

$S_e = \{ 1 \}$ $S_f = \{ 1, 2, 6, 7 \}$

set cover instance (k = 2)

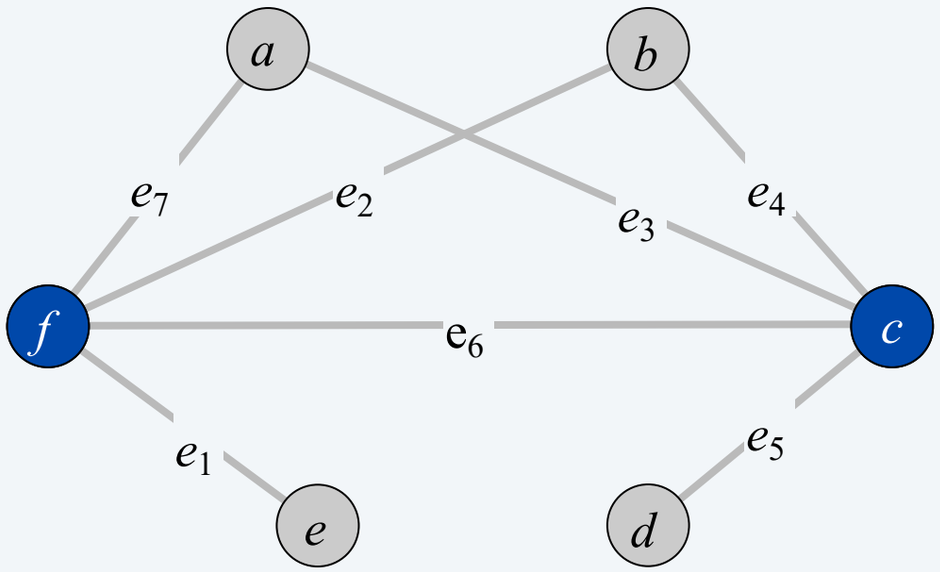
Vertex cover reduces to set cover

Lemma. $G = (V, E)$ contains a vertex cover of size k iff (U, S, k) contains a set cover of size k.

Pf. $\Leftarrow$ Let $Y \subseteq S$ be a set cover of size k in (U, S, k).

- Then $X = \{ v : S_v \in Y \}$ is a vertex cover of size k in G. ▪

"no" instances of VERTEX-COVER are solved correctly



vertex cover instance (k = 2)

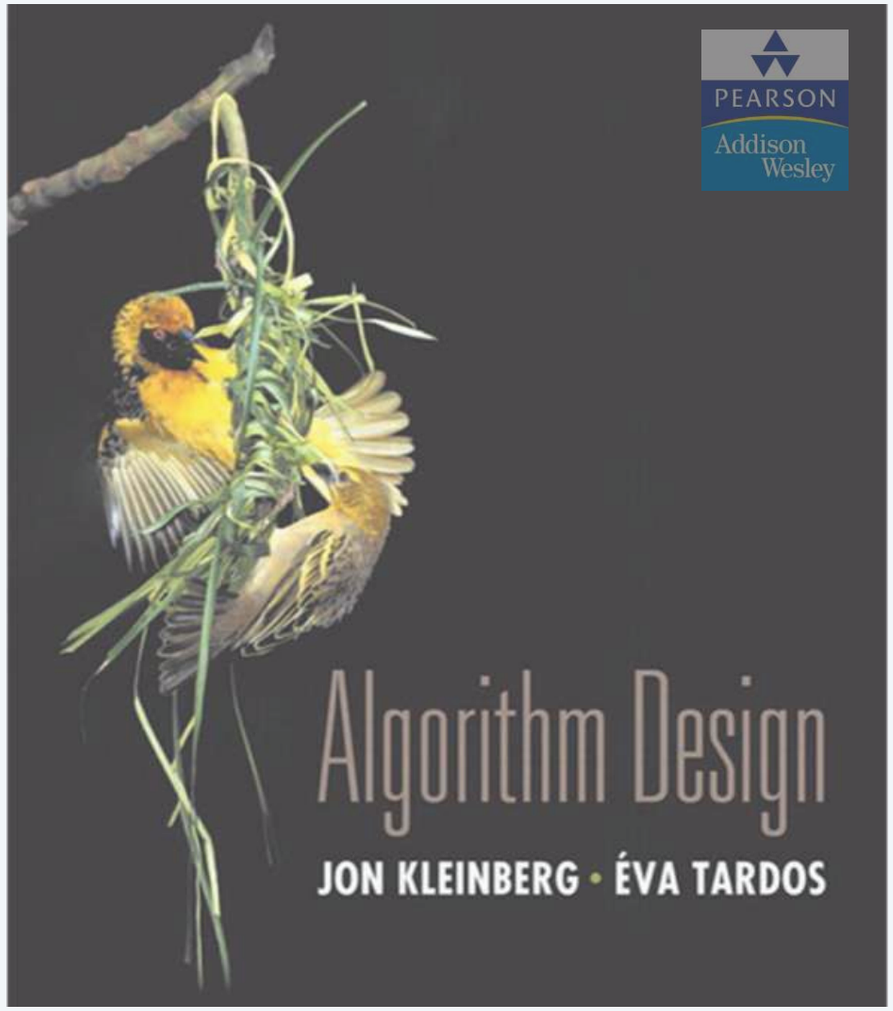
$U = \{ 1, 2, 3, 4, 5, 6, 7 \}$

$S_a = \{ 3, 7 \}$ $S_b = \{ 2, 4 \}$

$S_c = \{ 3, 4, 5, 6 \}$ $S_d = \{ 5 \}$

$S_e = \{ 1 \}$ $S_f = \{ 1, 2, 6, 7 \}$

set cover instance (k = 2)



SECTION 8.2

REDUCTIONS, P AND NP

- *poly-time reductions*
- *packing and covering problems*
- *constraint satisfaction problems*
- *graph coloring*
- *P vs. NP*
- *NP-complete*

Satisfiability

Literal. A Boolean variable or its negation. x_i or $\overline{x_i}$

Clause. A disjunction of literals. $C_j = x_1 \vee \overline{x_2} \vee x_3$

Conjunctive normal form (CNF). A propositional formula Φ that is a conjunction of clauses. $\Phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4$

SAT. Given a CNF formula Φ, does it have a satisfying truth assignment?

3-SAT. SAT where each clause contains exactly 3 literals (and each literal corresponds to a different variable).

$$\Phi = \left(\overline{x_1} \vee x_2 \vee x_3\right) \wedge \left(x_1 \vee \overline{x_2} \vee x_3\right) \wedge \left(\overline{x_1} \vee x_2 \vee x_4\right)$$

yes instance: x_1 = true, x_2 = true, x_3 = false, x_4 = false

Key application. Electronic design automation (EDA).

3-satisfiability reduces to independent set

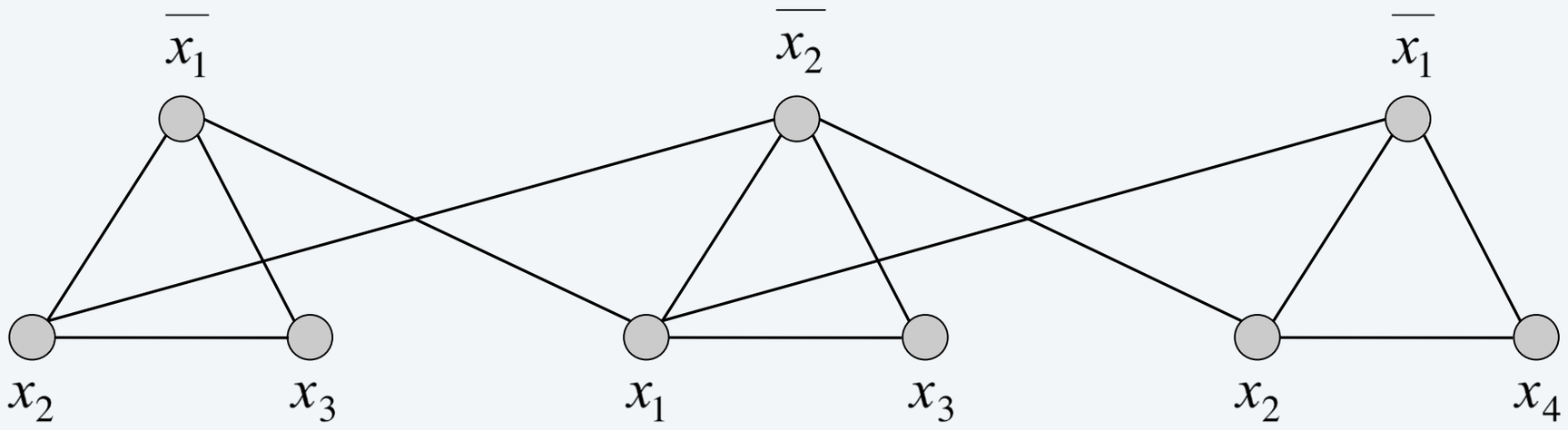
Theorem. 3-SAT $\leq_P$ INDEPENDENT-SET. (Please just **keep this conclusion in mind** and skip the **proof** below if you are doing the first-round self study. Also, we won't have corresponding HW or exam questions based on the **proof** below.)

Pf. Given an instance Φ of 3-SAT, we construct an instance (G, k) of INDEPENDENT-SET that has an independent set of size $k = |\Phi|$ iff Φ is satisfiable.

Construction.

- G contains 3 nodes for each clause, one for each literal.
- Connect 3 literals in a clause in a triangle.
- Connect literal to each of its negations.

G



k = 3

$$\Phi = \left(\overline{x_1} \vee x_2 \vee x_3\right) \wedge \left(x_1 \vee \overline{x_2} \vee x_3\right) \wedge \left(\overline{x_1} \vee x_2 \vee x_4\right)$$

3-satisfiability reduces to independent set

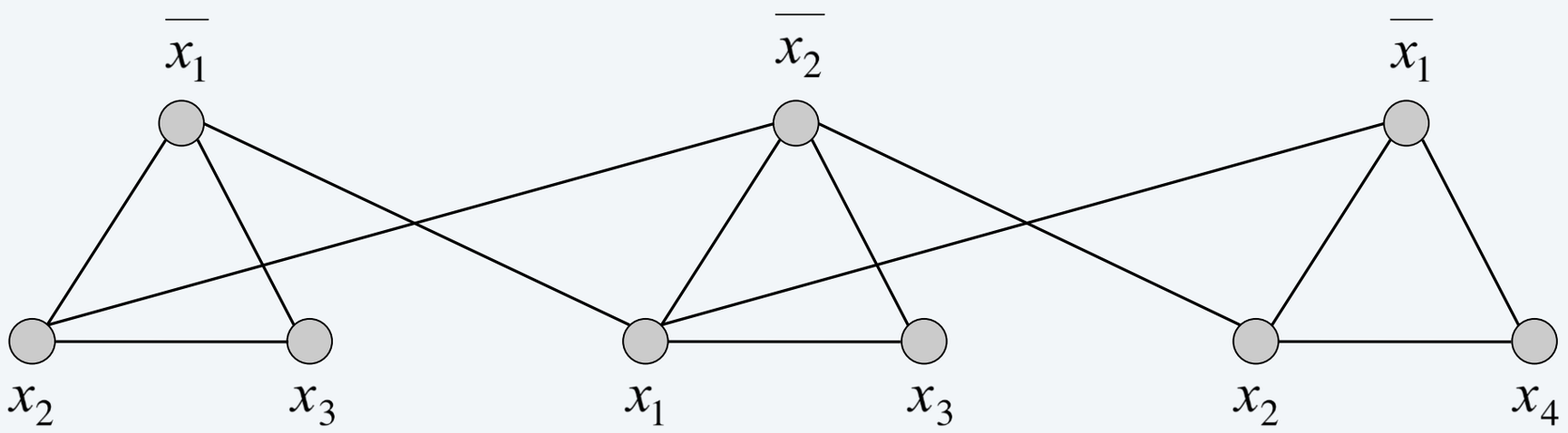
Lemma. Φ is satisfiable iff G contains an independent set of size $k = |\Phi|$.

Pf. $\Rightarrow$ Consider any satisfying assignment for Φ.

- Select one true literal from each clause/triangle.
- This is an independent set of size $k = |\Phi|$. ▪

"yes" instances of 3-SAT are solved correctly

G



k = 3

$$\Phi = \left(\overline{x_1} \vee x_2 \vee x_3 \right) \wedge \left(x_1 \vee \overline{x_2} \vee x_3 \right) \wedge \left(\overline{x_1} \vee x_2 \vee x_4 \right)$$

3-satisfiability reduces to independent set

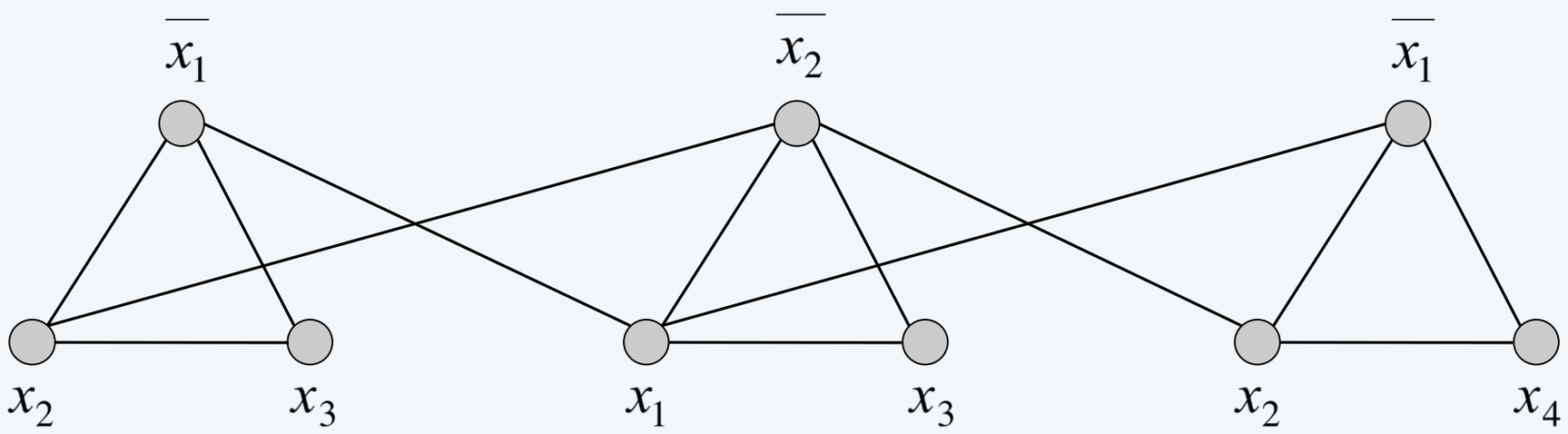
Lemma. Φ is satisfiable iff G contains an independent set of size $k = |\Phi|$.

Pf. $\Leftarrow$ Let S be independent set of size k.

- S must contain exactly one node in each triangle.
- Set these literals to *true* (and remaining literals consistently).
- All clauses in Φ are satisfied. ▪

"no" instances of 3-SAT are solved correctly

G



k = 3

$$\Phi = \left(\overline{x_1} \vee x_2 \vee x_3 \right) \wedge \left(x_1 \vee \overline{x_2} \vee x_3 \right) \wedge \left(\overline{x_1} \vee x_2 \vee x_4 \right)$$

Review

Basic reduction strategies.

- Simple equivalence: INDEPENDENT-SET $\equiv_P$ VERTEX-COVER.
- Special case to general case: VERTEX-COVER $\leq_P$ SET-COVER.
- Encoding with gadgets: 3-SAT $\leq_P$ INDEPENDENT-SET.

Transitivity. If $X \leq_P Y$ and $Y \leq_P Z$, then $X \leq_P Z$.

Pf idea. Compose the two mapping functions.

Ex. 3-SAT $\leq_P$ INDEPENDENT-SET $\leq_P$ VERTEX-COVER $\leq_P$ SET-COVER.

Transitivity of Reductions

Transitivity. If $X \leq_P Y$ and $Y \leq_P Z$, then $X \leq_P Z$.

Proof idea. Compose the two mapping functions.

Proof:

- Since $X \leq_P Y$, there's a polytime mapping f from instances of X to instances of Y.
- Since $Y \leq_P Z$, there's a polytime mapping g from instances of Y to instances of Z.
- Given an instance x of X, let $y = f(x)$, and $z = g(y) = g(f(x))$.
- Then x is a yes instance of $X \Leftrightarrow y$ is a yes instance of $Y \Leftrightarrow z$ is a yes instance of Z.
- So $g \circ f$ is a valid mapping of X to Z.
- Since f and g are both polytime, $g \circ f$ is also polytime.

Decision, search, and optimization problems

Decision problem. Does there exist a vertex cover of size $\leq k$?

Search problem. Find a vertex cover of size $\leq k$.

Optimization problem. Find a vertex cover of minimum size.

Goal. Show that all three problems poly-time Turing reduce (Cook reduce) to one another.

(Please just **keep this conclusion in mind** and skip the following **proof** in the next 2 pages if you are doing the first-round self study. Also, we won't have corresponding HW or exam questions based on the **proof** below.)

Search problems vs. decision problems

Vertex-Cover. Does there exist a vertex cover of size $\leq k$?

Find-Vertex-Cover. Find a vertex cover of size $\leq k$.

Theorem. Vertex-Cover $\equiv_T^P$ Find-Vertex-Cover.

Pf. $\leq_T^P$ Decision problem is a special case of search problem. ▪

Pf. $\geq_T^P$

To find a vertex cover of size $\leq k$:

- Determine if there exists a vertex cover of size $\leq k$.
- Find a vertex v such that $G - \{ v \}$ has a vertex cover of size $\leq k - 1$.
  (any vertex in any vertex cover of size $\leq k$ will have this property)
- Include v in the vertex cover.
- Recursively find a vertex cover of size $\leq k - 1$ in $G - \{ v \}$. ▪

delete v and all incident edges

Optimization problems vs. search problems

Find-Vertex-Cover. Find a vertex cover of size $\leq k$.

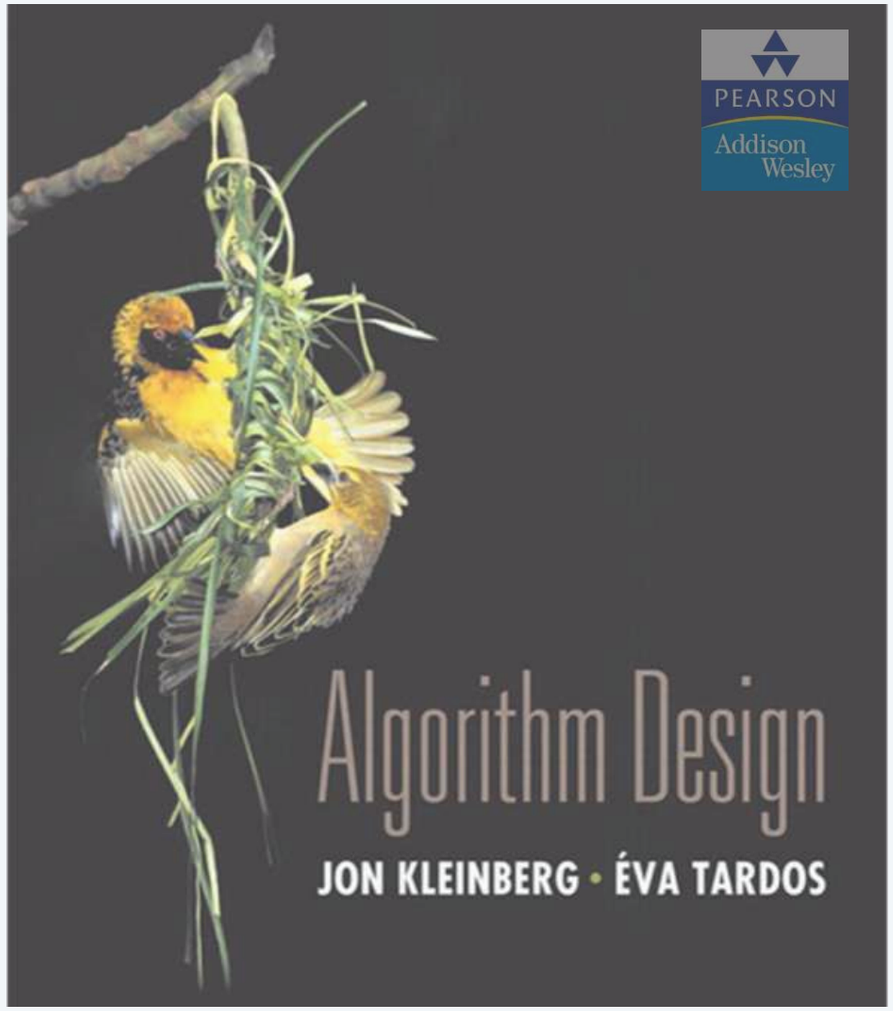
Find-Min-Vertex-Cover. Find a vertex cover of minimum size.

Theorem. Find-Vertex-Cover $\equiv_T^P$ Find-Min-Vertex-Cover.

Pf. $\leq_T^P$ Search problem is a special case of optimization problem. ▪

Pf. $\geq_T^P$ To find vertex cover of minimum size:

- Binary search (or linear search) for size k^* of min vertex cover.
- Solve search problem for given k^*. ▪



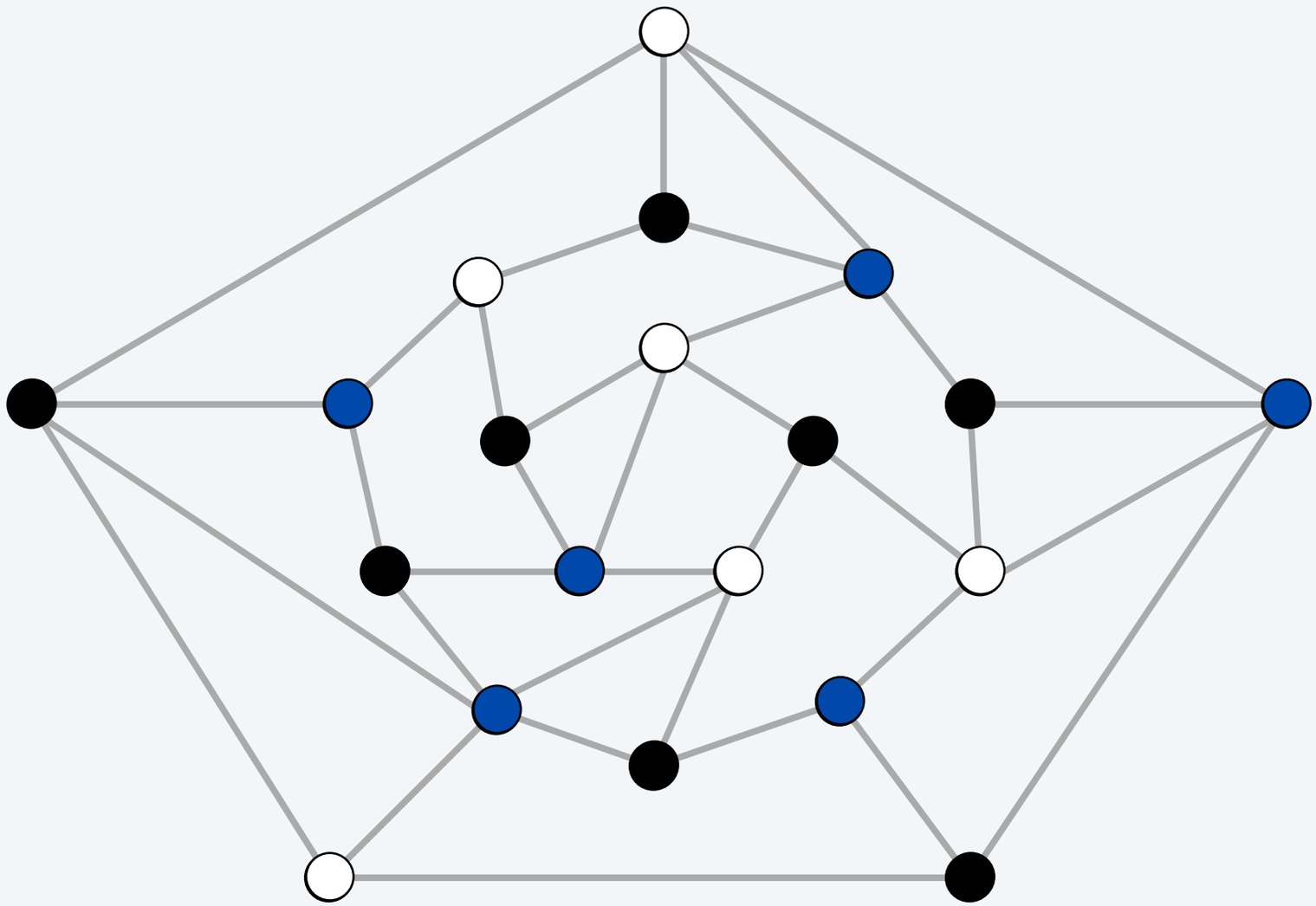
SECTION 8.7

REDUCTIONS, P AND NP

- *poly-time reductions*
- *packing and covering problems*
- *constraint satisfaction problems*
- *graph coloring*
- *P vs. NP*
- *NP-complete*

3-colorability

3-COLOR. Given an undirected graph G, can the nodes be colored black, white, and blue so that no adjacent nodes have the same color?



yes instance

How difficult to solve 2-COLOR?

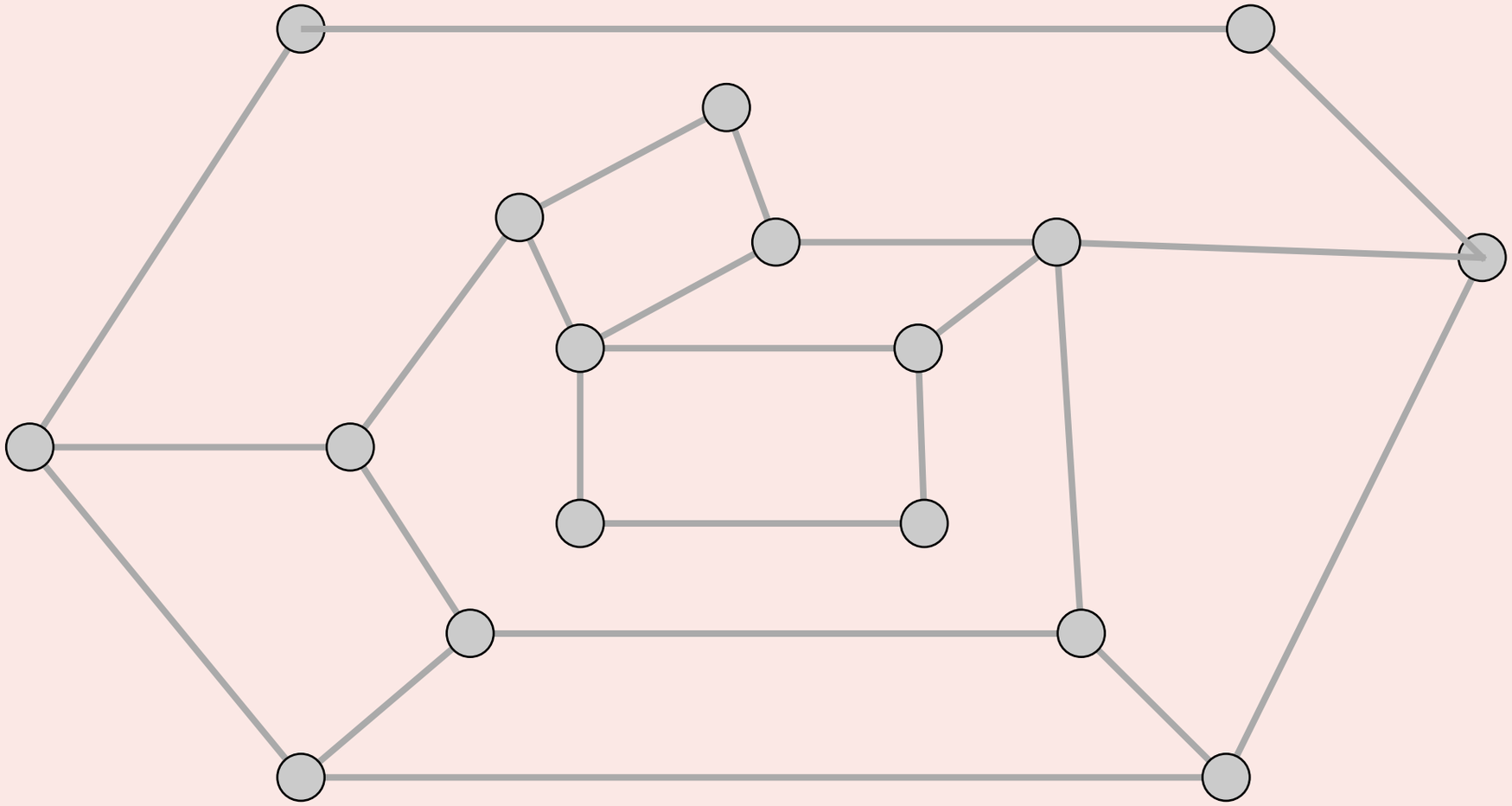
A. $O(m + n)$ using BFS or DFS.

B. $O(mn)$ using maximum flow.

C. $\Omega(2^n)$ using brute force.

D. Not even Tarjan knows.

The answer is A.



3-satisfiability reduces to 3-colorability

Theorem. 3-SAT $\leq_P$ 3-COLOR.

(Please just **keep this conclusion in mind** and skip the following **proof** in the next 5 pages if you are doing the first-round self study. The **proof** below can be really complicated for beginners. Also, we won't have corresponding HW or exam questions based on the **proof** below.)

Pf. Given 3-SAT instance Φ, we construct an instance of 3-COLOR that is 3-colorable iff Φ is satisfiable.

3-satisfiability reduces to 3-colorability

Construction.

(i) Create a graph G with a node for each literal.

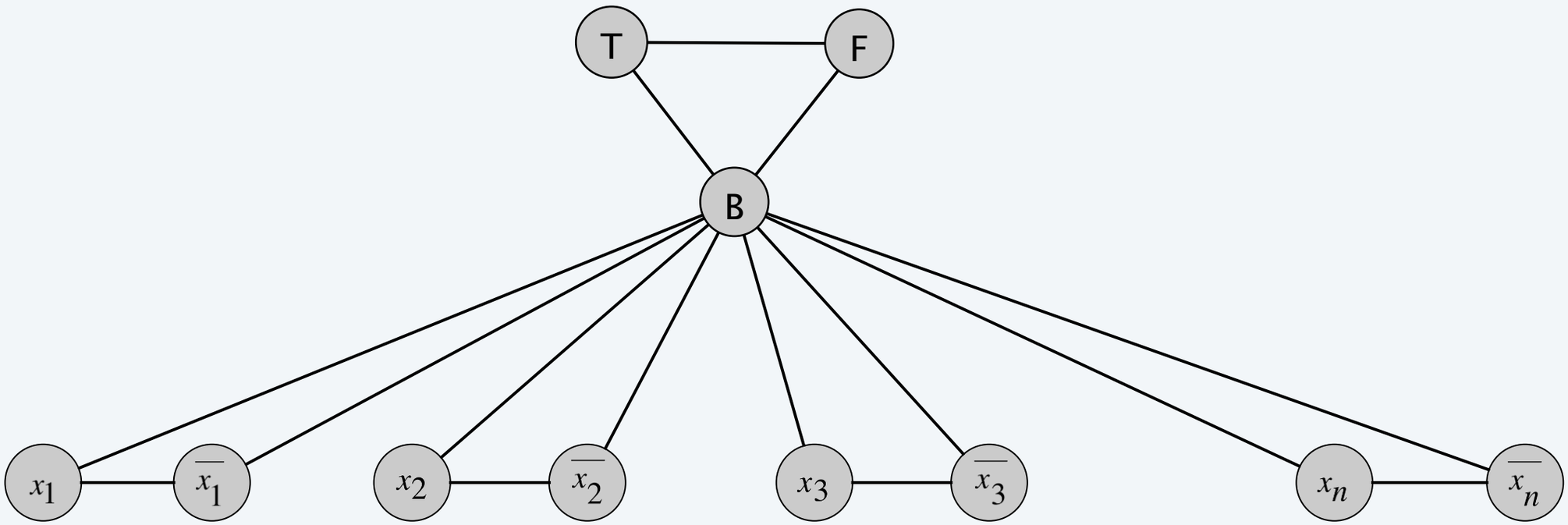
(ii) Connect each literal to its negation.

(iii) Create 3 new nodes T, F, and B; connect them in a triangle.

(iv) Connect each literal to B.

(v) For each clause C_j, add a gadget of 6 nodes and 13 edges.

to be described later

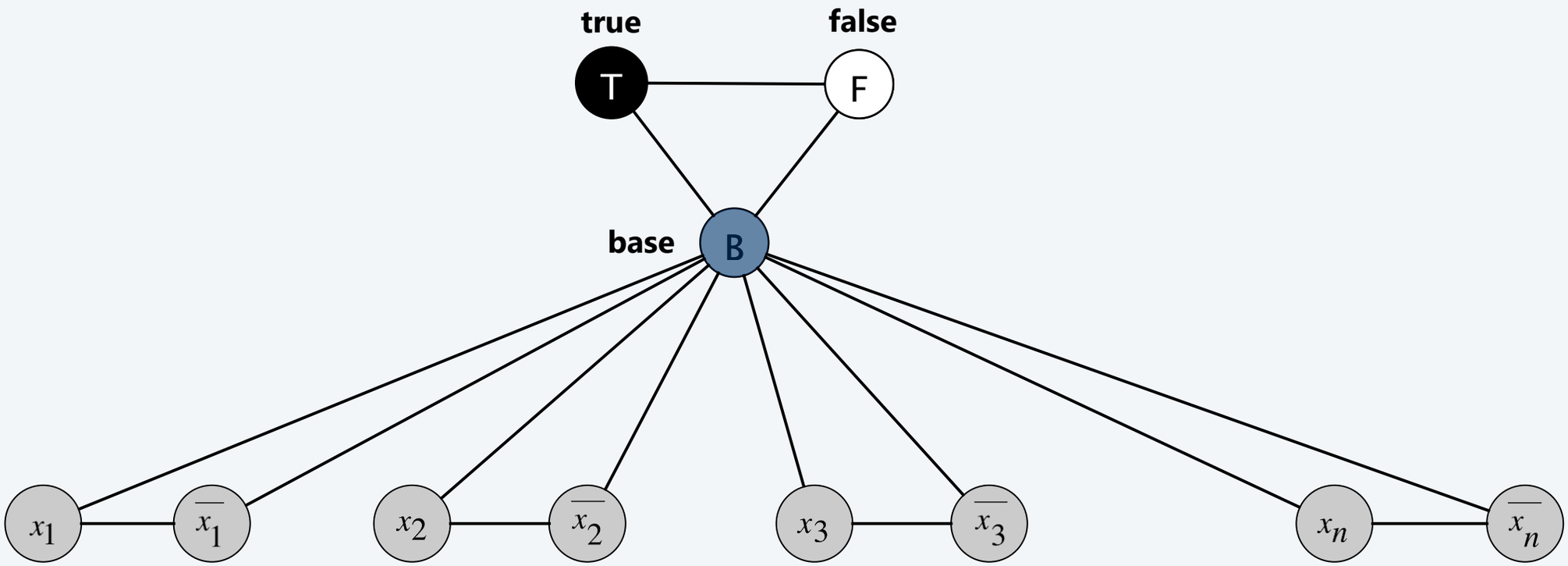


3-satisfiability reduces to 3-colorability

Lemma. Graph G is 3-colorable iff Φ is satisfiable.

Pf. $\Rightarrow$ Suppose graph G is 3-colorable.

- WLOG, assume that node T is colored *black*, F is *white*, and B is *blue*.
- Consider assignment that sets all *black* literals to *true* (and *white* to *false*).
- (iv) ensures each literal is colored either *black* or *white*.
- (ii) ensures that each literal is *white* if its negation is *black* (and vice versa).

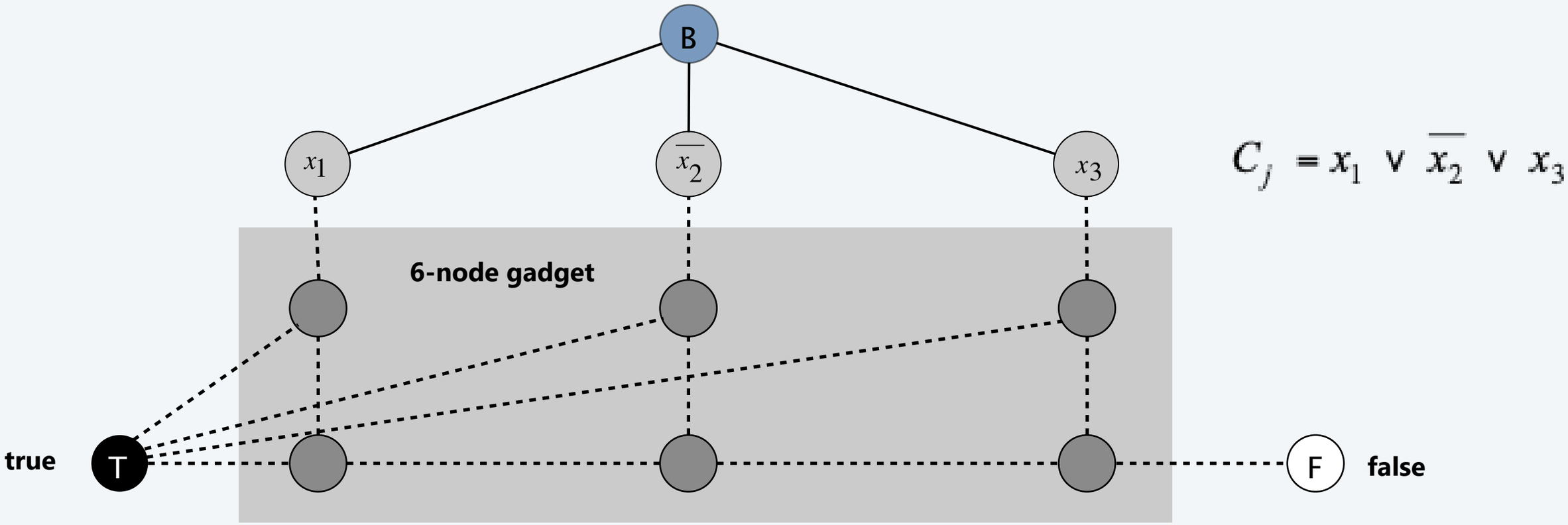


3-satisfiability reduces to 3-colorability

Lemma. Graph G is 3-colorable iff Φ is satisfiable.

Pf. $\Rightarrow$ Suppose graph G is 3-colorable.

- WLOG, assume that node T is colored *black*, F is *white*, and B is *blue*.
- Consider assignment that sets all *black* literals to *true* (and *white* to *false*).
- (iv) ensures each literal is colored either *black* or *white*.
- (ii) ensures that each literal is *white* if its negation is *black* (and vice versa).
- (v) ensures at least one literal in each clause is *black*.

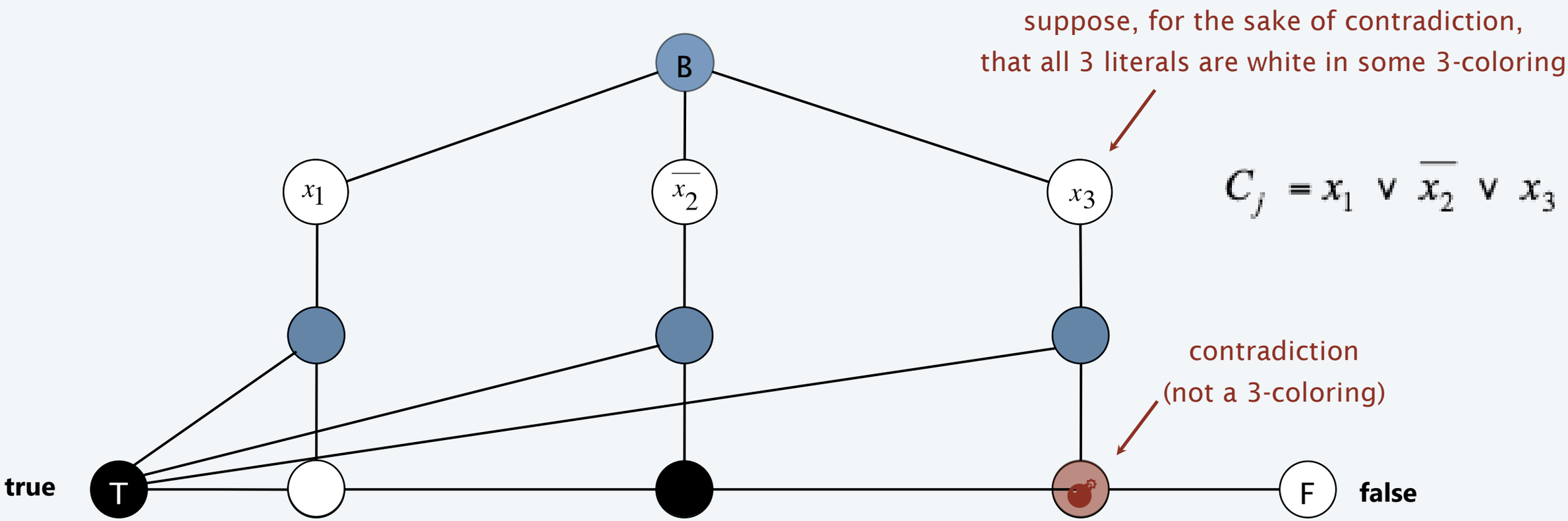


3-satisfiability reduces to 3-colorability

Lemma. Graph G is 3-colorable iff Φ is satisfiable.

Pf. $\Rightarrow$ Suppose graph G is 3-colorable.

- WLOG, assume that node T is colored *black*, F is *white*, and B is *blue*.
- Consider assignment that sets all *black* literals to *true* (and *white* to *false*).
- (iv) ensures each literal is colored either *black* or *white*.
- (ii) ensures that each literal is *white* if its negation is *black* (and vice versa).
- (v) ensures at least one literal in each clause is *black*. ▪

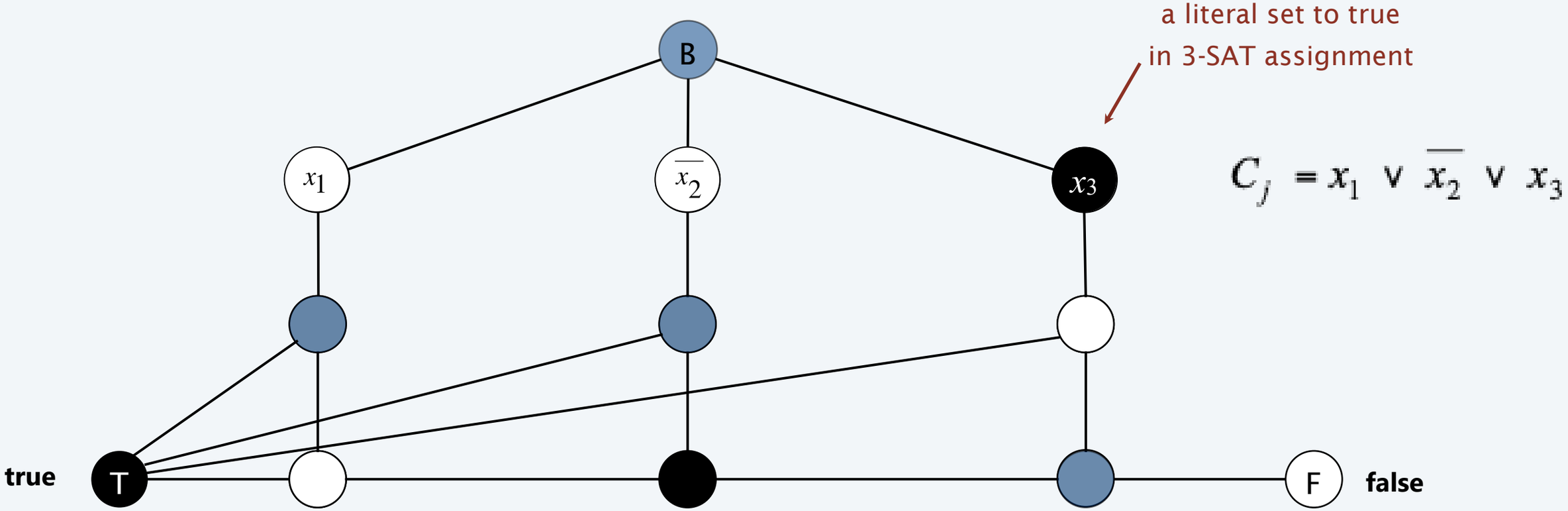


3-satisfiability reduces to 3-colorability

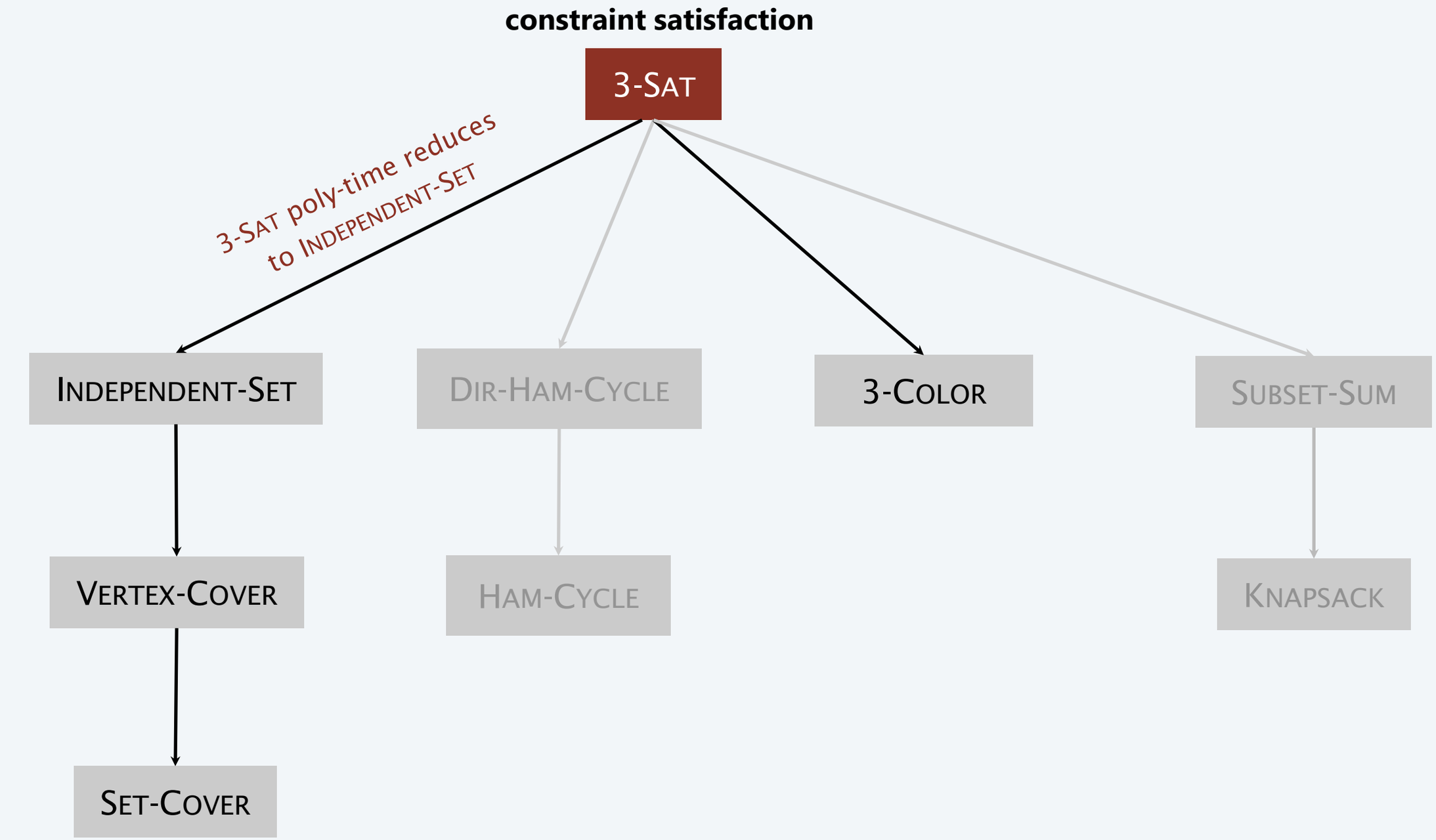
Lemma. Graph G is 3-colorable iff Φ is satisfiable.

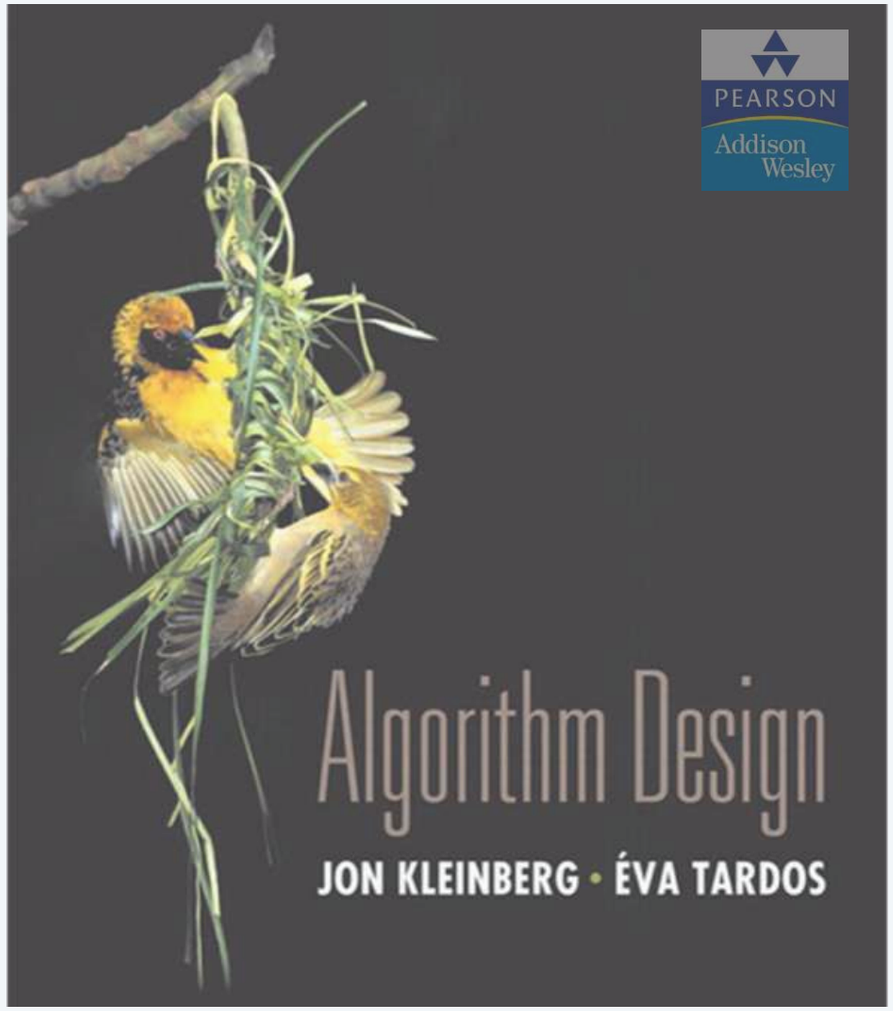
Pf. $\Leftarrow$ Suppose 3-SAT instance Φ is satisfiable.

- Color all *true* literals *black* and all *false* literals *white*.
- Pick one *true* literal; color node below that node *white*, and node below that *blue*.
- Color remaining middle row nodes *blue*.
- Color remaining bottom nodes *black* or *white*, as forced. ▪



Poly-time reductions





SECTION 8.3

REDUCTIONS, P AND NP

- *poly-time reductions*
- *packing and covering problems*
- *constraint satisfaction problems*
- *graph coloring*
- *P vs. NP*
- *NP-complete*

P & NP

Def. A polynomial time (polytime) algorithm is one that runs in $O(n^k)$ time, for some constant k, when input has size n.
Def. P is the set of all problems that can be solved by a polytime algorithm.

The following is an intuitive understanding of the concept of NP and how NP compares to P:

P is the class of problems for which all instances can be solved in polynomial time by some algorithm.
NP is the class of problems for which the solvability of an instance can be verified in polynomial time.

- The verification is done by a "verifier (certifier)" algorithm.
- The verifier(certifier) needs an additional "hint" to work correctly.
  - The hint is also called a "witness" or "certificate".
- The verifier(certifier) doesn't find a solution to a problem instance, but only checks that the instance has been solved.

NP Continue (Certifier)

The verifier(certifier) has the following properties.

- The verifier(certifier)'s input is a problem instance x, and a certificate y.
- The verifier(certifier)'s output is either "accept(yes)" or "reject(no)".
- If x has a solution, then if y is a "good" certificate, the verifier(certifier) will output accept.
  - If y is not a "good" certificate, the verifier(certifier) can either accept or reject.
- If x has no solution, the verifier(certifier) rejects no matter what y is.
- Intuitively, the certificate y indicates x is solvable.
  - For example, y can be a solution to x.
  - But y can also be an indirect representation of a solution.
- The verifier(certifier) is efficient, i.e. runs in polynomial time.

The class NP, formally

Def Given a decision problem A, a polynomial time verifier(certifier) V for A is an algorithm that does the following

- V's input is an instance x of A, and a certificate string y.
- $V(x, y) \in \{0,1\}$, representing "reject(no)" and "accept(yes)", respectively.
- If x is a yes instance, there exists a y for which V outputs 1, i.e. $\exists y: V(x, y) = 1$.
- If x is a no instance, every y makes V output 0, i.e. $\forall y: V(x, y) = 0$.
- V runs in polynomial time.

NP is the set of all decision problems with polytime verifiers(certifiers).

To show a decision problem is in NP, give a polynomial time verifier(certifier) for the problem satisfying the properties on the previous slide.

- This requires specifying what the certificates are, and how the verifier(certifier) operates, given an instance of the problem and a certificate.

P (Again)

Decision problem.

- Problem X is a set of strings.
- Instance s is one string.
- Algorithm A solves problem X: $A(s) = \begin{cases} yes & \text{if } s \in X \\ no & \text{if } s \notin X \end{cases}$

Def. Algorithm A runs in polynomial time if for **every** string s, $A(s)$ terminates in $\leq poly(|s|)$ "steps," where $poly(\cdot)$ is some polynomial function.

length of s

Def. **P** = set of problems for which there exists a poly-time algorithm.

problem PRIMES: { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, … }

instance s: 592335744548702854681

algorithm: Agrawal–Kayal–Saxena (2002)

NP (Again)

Def. Algorithm $C(s, t)$ is a certifier for problem X if for every string s :

$s \in X$ **iff <u>there exists</u>** a string t such that $C(s, t) = yes$.

Def. **NP** = set of decision problems for which there exists a poly-time certifier.

- $C(s, t)$ is a poly-time algorithm.
- Certificate t is of polynomial size: $|t| \leq poly(|s|)$ for some polynomial $poly(\cdot)$.

"certificate" or "witness"

problem COMPOSITES:	{ 4, 6, 8, 9, 10, 12, 14, 15, 16, 18, 20, …. }
instance s:	437669
certificate t:	541 ← 437,669 = 541 × 809
certifier C(s, t):	grade-school division

Certifiers and certificates: satisfiability

SAT. Given a CNF formula Φ, does it have a satisfying truth assignment?

3-SAT. SAT where each clause contains exactly 3 literals.

Certificate. An assignment of truth values to the Boolean variables.

Certificate. Check that each clause in Φ has at least one true literal.

instance s $\Phi = (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_4)$

certificate t $x_1 = true,\ x_2 = true,\ x_3 = false,\ x_4 = false$

Conclusions. SAT $\in$ **NP**, 3-SAT $\in$ **NP**.

Which of the following graph problems are known to be in NP?

A. Is the length of the longest simple path $\leq k$?

B. Is the length of the longest simple path $\geq k$?

C. Is the length of the longest simple path $= k$?

D. Find the length of the longest simple path.

E. All of the above.

The answer is B.

Here the certificate should be a simple path. If the certificate path has length $\geq$ k, then we know that there exists a path with length $\geq$ k in the graph. In this way, we may certify that the longest simple path must have length at least k.

Intractability: quiz 6

In complexity theory, the abbreviation NP stands for...

A. Nope.

B. No problem.

C. Not polynomial time.

D. Not polynomial space.

E. Nondeterministic polynomial time.

The answer is E.

Significance of NP

NP. Decision problems for which there exists a poly-time certifier.

> *" NP captures vast domains of computational, scientific, and mathematical endeavors, and seems to roughly delimit what mathematicians and scientists have been aspiring to compute feasibly. "* — *Christos Papadimitriou*

> *" In an ideal world it would be renamed P vs VP. "* — *Clyde Kruskal*

P, NP, and EXP

P. Decision problems for which there exists a poly-time algorithm.

NP. Decision problems for which there exists a poly-time certifier.

EXP. Decision problems for which there exists an exponential-time algorithm.

Proposition. **P** $\subseteq$ **NP**.

Pf. Consider any problem $X \in$ **P**.

- By definition, there exists a poly-time algorithm $A(s)$ that solves X.
- Certificate $t = \varepsilon$ (empty string), certifier $C(s, t) = A(s)$. ▪

Proposition. **NP** $\subseteq$ **EXP**. (Please just **keep this conclusion in mind** and skip the **proof** below if you are doing the first-round self study. Also, we won't have corresponding HW or exam questions based on the **proof** below.)

Pf. Consider any problem $X \in$ **NP**.

- By definition, there exists a poly-time certifier $C(s, t)$ for X, where certificate t satisfies $|t| \leq p(|s|)$ for some polynomial $p(\cdot)$.
- To solve instance s, run $C(s, t)$ on all strings t with $|t| \leq p(|s|)$.
- Return *yes* iff $C(s, t)$ returns *yes* for any of these potential certificates. ▪

All problems in P are in NP

Let A be a problem in P. I.e. there's a polytime algorithm S s.t. on every instance x of A

- If x has a solution, S returns a solution.
- If x has no solution, S returns fail.

Verifier (Certifier)

- V runs S. If S finds a solution, V outputs 1. Otherwise V outputs 0.

If x has a solution.

- S finds a solution, so V outputs 1.

If x has no solution.

- S returns fail, so V outputs 0.

V runs in polytime.

- Because V just runs S, which runs in polytime.

Notice that for problems in P, V doesn't need a certificate y.

- For problems in P, it's easy to determine if they're solvable or not.

But for hard problems (not in P), V isn't powerful enough to determine solvability by itself.

- So it needs a hint / witness / certificate.

The main question: P vs. NP

Q. How to solve an instance of 3-SAT with n variables?

A. Exhaustive search: try all 2^n truth assignments.

Q. Can we do anything substantially more clever?

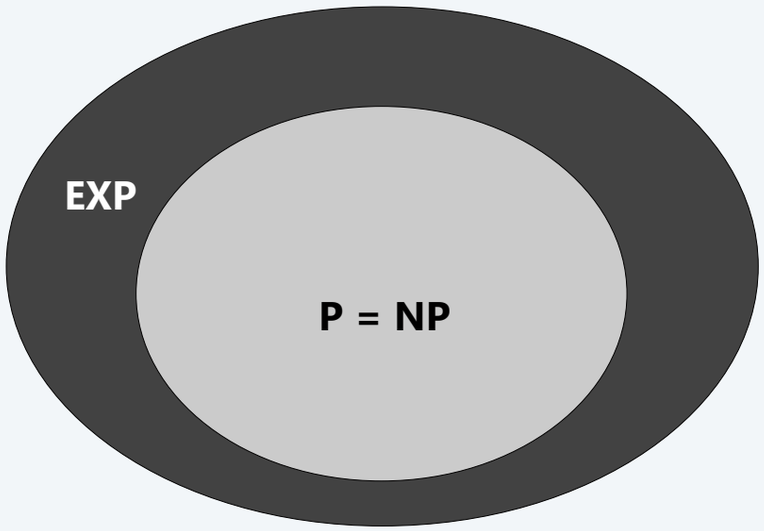
Conjecture. No poly-time algorithm for 3-SAT.

"intractable"

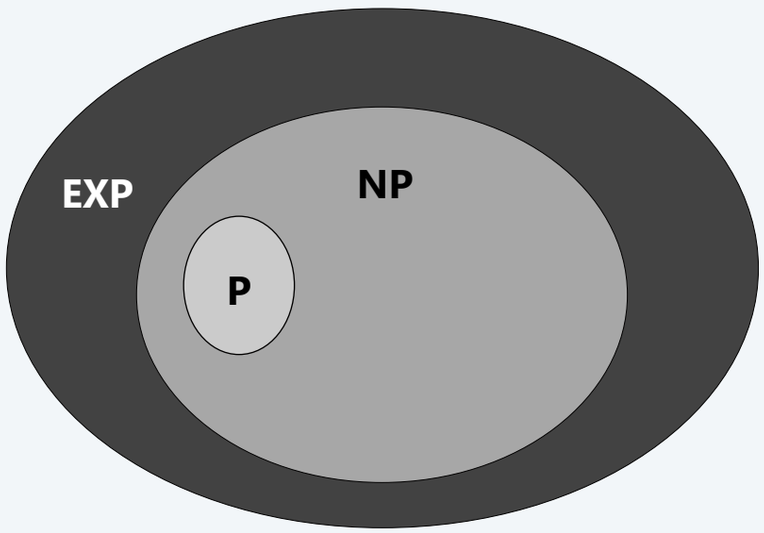
The main question: P vs. NP

Does P = NP? [Cook 1971, Edmonds, Levin, Yablonski, Gödel]

Is the decision problem as easy as the certification problem?



If P = NP

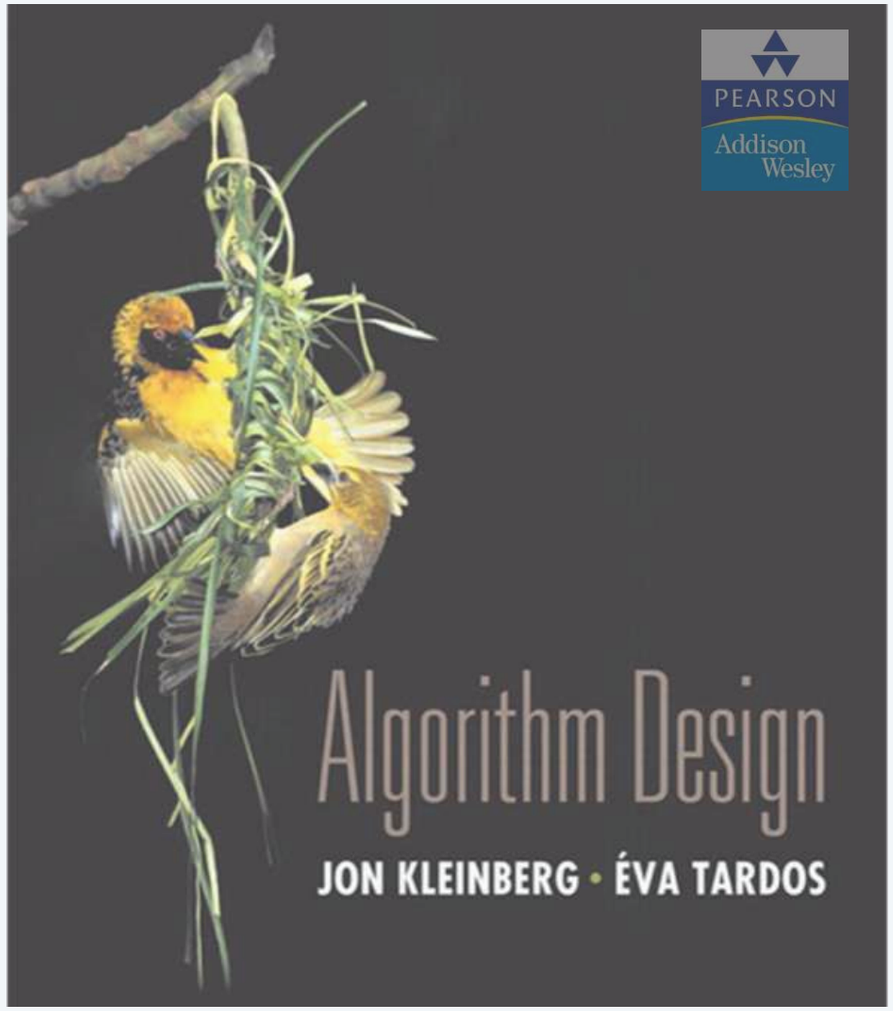


If P ≠ NP

If yes… Efficient algorithms for 3-SAT, TSP, VERTEX-COVER, FACTOR, …

If no… No efficient algorithms possible for 3-SAT, TSP, VERTEX-COVER, …

Consensus opinion. Probably no.



SECTION 8.4

REDUCTIONS, P AND NP

- *poly-time reductions*
- *packing and covering problems*
- *constraint satisfaction problems*
- *graph coloring*
- *P vs. NP*
- *NP-complete*

Polynomial transformations

Def. Problem X polynomial (Cook) reduces to problem Y if arbitrary instances of problem X can be solved using:

- Polynomial number of standard computational steps, plus
- Polynomial number of calls to oracle that solves problem Y.

Def. Problem X polynomial (Karp) transforms to problem Y if given any instance x of X, we can construct an instance y of Y such that x is a *yes* instance of X **iff** y is a *yes* instance of Y.

we require $|y|$ to be of size polynomial in $|x|$

Note. Polynomial transformation is polynomial reduction with just one call to oracle for Y, exactly at the end of the algorithm for X. Almost all previous reductions were of this form.

Open question. Are these two concepts the same with respect to **NP**?

we abuse notation $\leq_P$ and blur distinction

NP-complete

NP-complete. A problem $Y \in$ **NP** with the property that for **every** problem $X \in$ **NP**, $X \leq_P Y$.

Proposition. Suppose $Y \in$ **NP**-complete. Then, $Y \in$ **P** iff **P = NP**.

Pf. $\Leftarrow$ If **P = NP**, then $Y \in$ **P** because $Y \in$ **NP**.

Pf. $\Rightarrow$ Suppose $Y \in$ **P**.

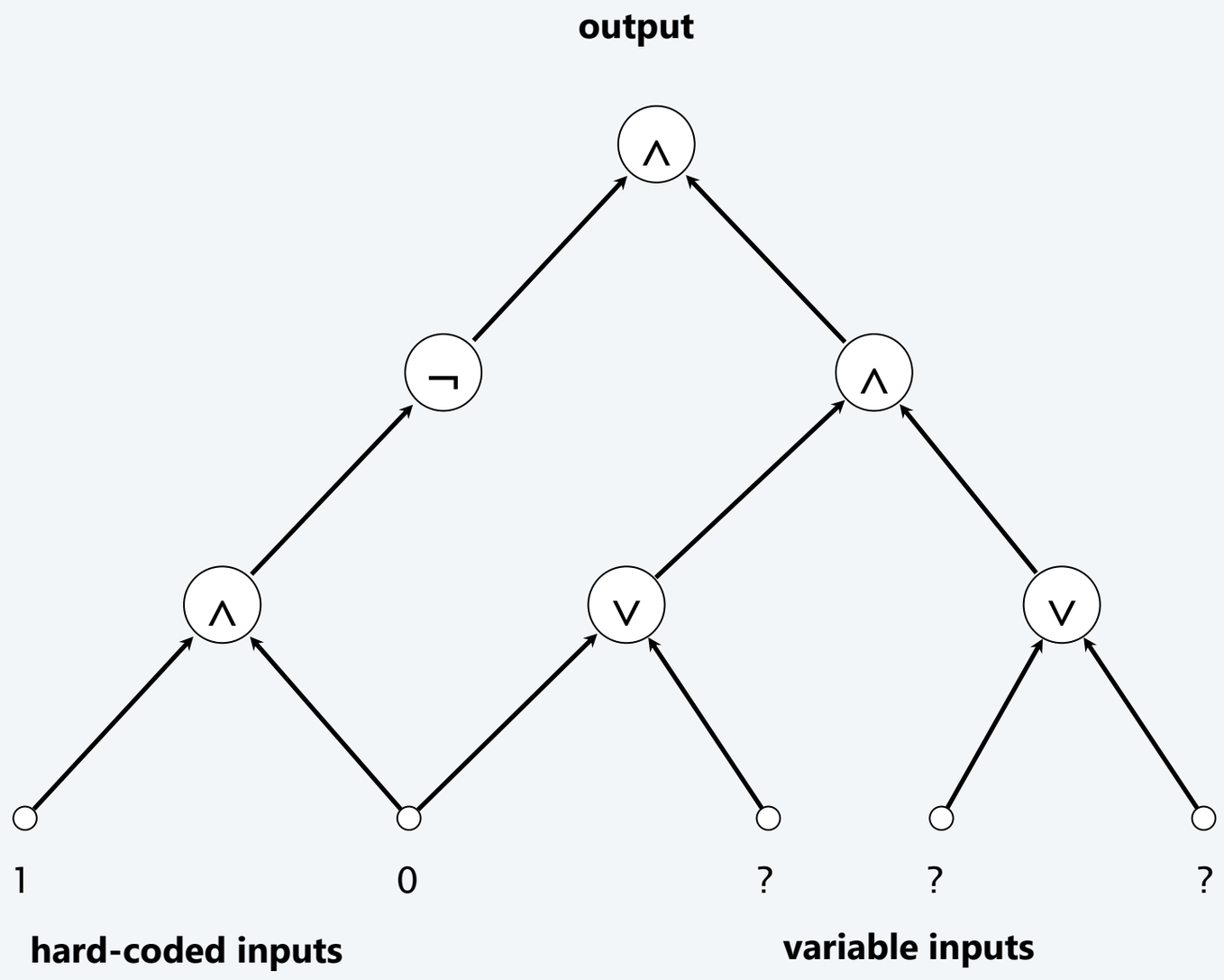
- Consider any problem $X \in$ **NP**. Since $X \leq_P Y$, we have $X \in$ **P**.
- This implies **NP** $\subseteq$ **P**.
- We already know **P** $\subseteq$ **NP**. Thus **P = NP**. ▪

Fundamental question. Are there any "natural" **NP**-complete problems?

Answer: Yes! CIRCUIT-SAT $\in$ **NP**-complete by Cook-Levin Theorem.

Circuit satisfiability

CIRCUIT-SAT. Given a combinational circuit built from AND, OR, and NOT gates, is there a way to set the circuit inputs so that the output is 1?



yes: 1 0 1

Cook-Levin Theorem

Cook-Levin theorem says 2 things.

- CIRCUIT-SAT ∈ **NP**.
  - Prove this yourself.
- Every **NP** problem reduces to CIRCUIT-SAT. I.e. every problem A in NP can be mapped to an combinational circuit K in polytime, such that
  - If A is true, then K is satisfiable.
  - If A is false, then K is not satisfiable.

(Please just **keep this conclusion in mind** and skip the following **proof** in the next 4 pages if you are doing the first-round self study. The **proof** below can be really complicated for beginners. Also, we won't have corresponding HW or exam questions based on the **proof** below.)

The "first" NP-complete problem

Theorem. CIRCUIT-SAT ∈ **NP**-complete.

Pf sketch.

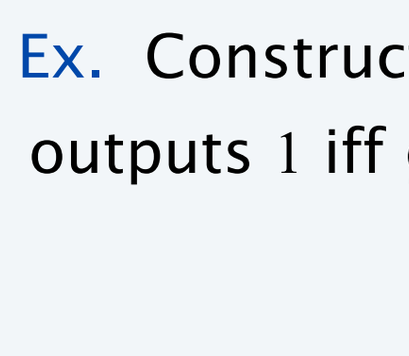
- Clearly, CIRCUIT-SAT ∈ **NP**.
- Any algorithm that takes a fixed number of bits n as input and produces a *yes* or *no* answer can be represented by such a circuit.
- Moreover, if algorithm takes poly-time, then circuit is of poly-size.

sketchy part of proof; fixing the number of bits is important, and reflects basic distinction between algorithms and circuits

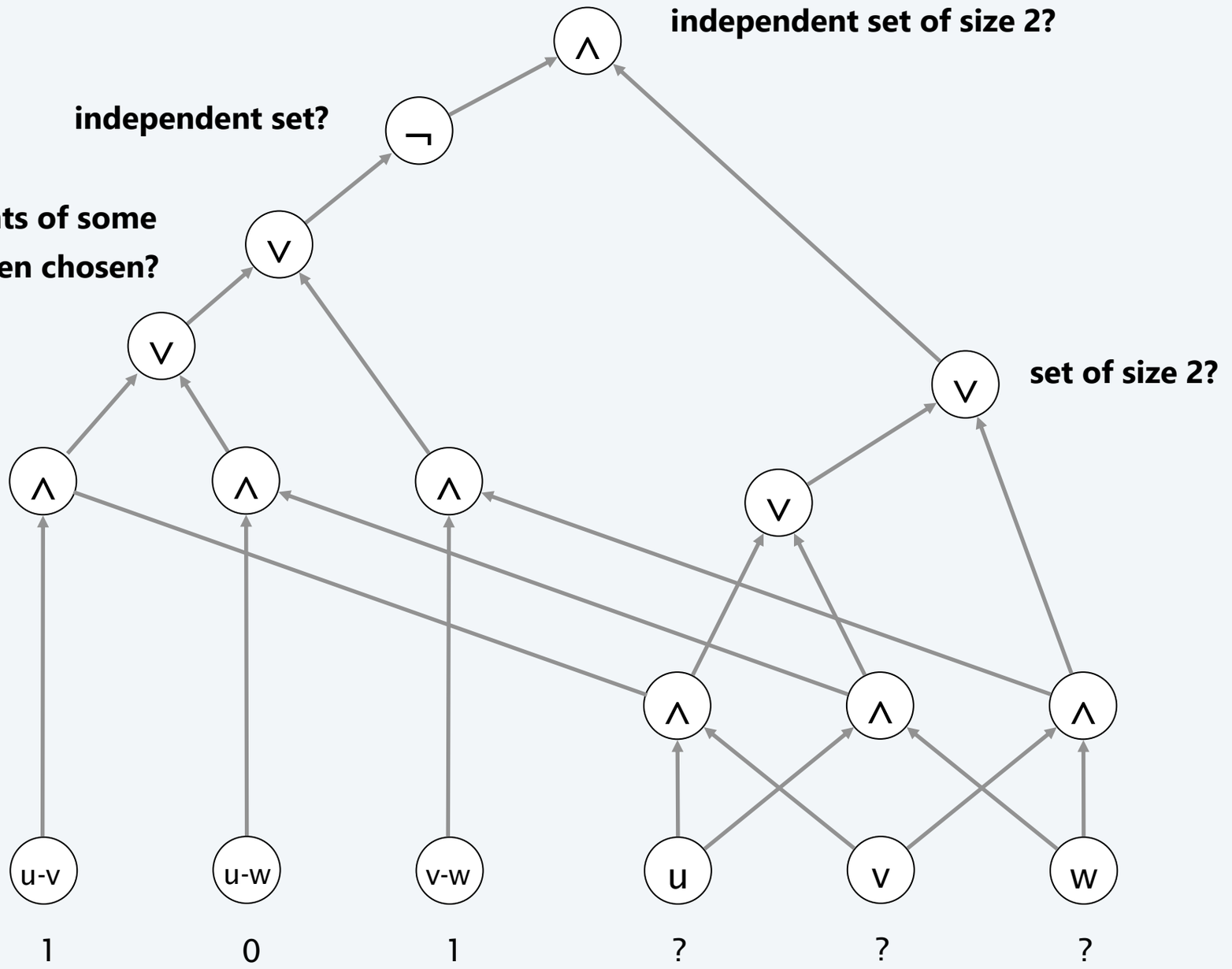
- Consider any problem $X \in$ **NP**. It has a poly-time certifier $C(s, t)$, where certificate t satisfies $|t| \leq p(|s|)$ for some polynomial $p(\cdot)$.
- View $C(s, t)$ as an algorithm with $|s| + p(|s|)$ input bits and convert it into a poly-size circuit K.
  - first $|s|$ bits are hard-coded with s
  - remaining $p(|s|)$ bits represent (unknown) bits of t
- Circuit K is satisfiable iff $C(s, t) = yes$.

Example

Ex. Construction below creates a circuit K whose inputs can be set so that it outputs 1 iff graph G has an independent set of size 2.



G = (V, E), n = 3



$\binom{n}{2}$ **hard-coded inputs (graph description)**

n inputs (nodes in independent set)

Establishing NP-completeness

Remark. Once we establish first "natural" **NP**-complete problem, others fall like dominoes.

Recipe. To prove that $Y \in$ **NP**-complete:

- Step 1. Show that $Y \in$ **NP.** (This is usually not hard.).
- Step 2. Choose an **NP**-complete problem X.
- Step 3. Prove that $X \leq_P Y$. (This can sometimes be quite challenging.)

Proposition. If $X \in$ **NP**-complete, $Y \in$ **NP**, and $X \leq_P Y$, then $Y \in$ **NP**-complete.

Pf. Consider any problem $W \in$ **NP**. Then, both $W \leq_P X$ and $X \leq_P Y$.

- By transitivity, $W \leq_P Y$.
- Hence $Y \in$ **NP**-complete. ▪

by definition of **NP**-complete

by assumption

Reductions: quiz 7

Suppose that $X \in$ NP-COMPLETE, $Y \in$ NP, and $X \leq_{\mathrm{P}} Y$. Which can you infer?

A. Y is **NP**-complete.

B. If $Y \notin$ **P**, then **P ≠ NP**.

C. If **P ≠ NP**, then neither X nor Y is in **P**.

D. All of the above.

The answer is D.

3-satisfiability is NP-complete

Theorem. 3-SAT $\in$ **NP**-complete.

Pf.

- Suffices to show that CIRCUIT-SAT $\leq_P$ 3-SAT since 3-SAT $\in$ **NP**.

(Please just **keep this conclusion in mind** and skip the following **proof** in the next 3 pages if you are doing the first-round self study. The **proof** below can be really complicated for beginners. Also, we won't have corresponding HW or exam questions based on the **proof** below.)

- Given a combinational circuit K, we construct an instance Φ of 3-SAT that is satisfiable iff the inputs of K can be set so that it outputs 1.

3-satisfiability is NP-complete

Construction. Let K be any circuit.

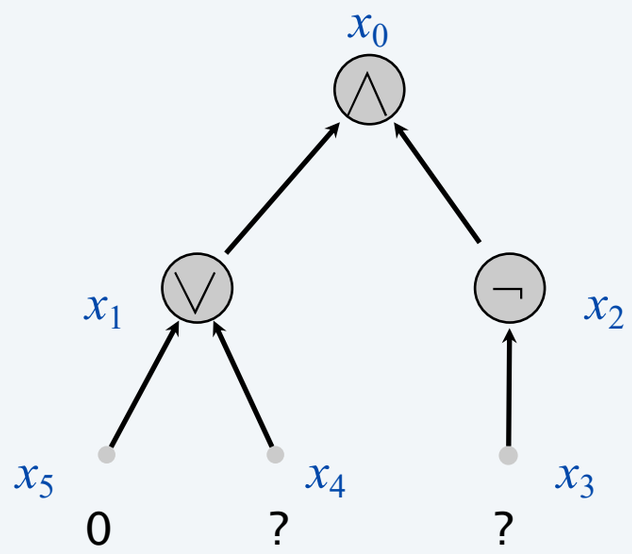
Step 1. Create a 3-SAT variable x_i for each circuit element i.

Step 2. Make circuit compute correct values at each node:

- $x_2 = \neg x_3$ ⇒ add 2 clauses: $x_2 \vee x_3$, $\overline{x_2} \vee \overline{x_3}$
- $x_1 = x_4 \vee x_5$ ⇒ add 3 clauses: $x_1 \vee \overline{x_4}$, $x_1 \vee \overline{x_5}$, $\overline{x_1} \vee x_4 \vee x_5$
- $x_0 = x_1 \wedge x_2$ ⇒ add 3 clauses: $\overline{x_0} \vee x_1$, $\overline{x_0} \vee x_2$, $x_0 \vee \overline{x_1} \vee \overline{x_2}$

Step 3. Hard-coded input values and output value.

- $x_5 = 0$ ⇒ add 1 clause: $\overline{x_5}$
- $x_0 = 1$ ⇒ add 1 clause: x_0



Construction. [continued]

Step 4. Turn clauses of length 1 or 2 into clauses of length 3.

- Create four new variables z_1, z_2, z_3, and z_4.
- Add 8 clauses to force $z_1 = z_2 = 0$:

$$(\overline{z_1} \vee z_3 \vee z_4),\ (\overline{z_1} \vee z_3 \vee \overline{z_4}),\ (\overline{z_1} \vee \overline{z_3} \vee z_4),\ (\overline{z_1} \vee \overline{z_3} \vee \overline{z_4})$$
$$(\overline{z_2} \vee z_3 \vee z_4),\ (\overline{z_2} \vee z_3 \vee \overline{z_4}),\ (\overline{z_2} \vee \overline{z_3} \vee z_4),\ (\overline{z_2} \vee \overline{z_3} \vee \overline{z_4})$$

- Replace any clause with a single term (t_i) with $(t_i \vee z_1 \vee z_2)$.
- Replace any clause with two terms $(t_i \vee t_j)$ with $(t_i \vee t_j \vee z_1)$.

3-satisfiability is NP-complete

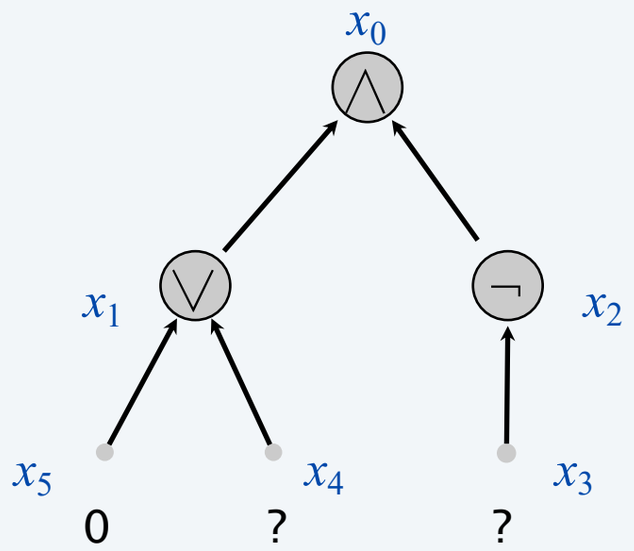
Lemma. Φ is satisfiable iff the inputs of K can be set so that it outputs 1.

Pf. $\Leftarrow$ Suppose there are inputs of K that make it output 1.

- Can propagate input values to create values at all nodes of K.
- This set of values satisfies Φ.

Pf. $\Rightarrow$ Suppose Φ is satisfiable.

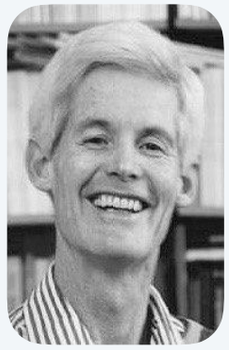
- We claim that the set of values corresponding to the circuit inputs constitutes a way to make circuit K output 1.
- The 3-SAT clauses were designed to ensure that the values assigned to all node in K exactly match what the circuit would compute for these nodes. ▪



Implications of Karp's Work

SAT poly-time reduces to all of these problems (and many, many more)

Implications of Cook–Levin Theorem



All of these problems (and many, many more) poly-time reduce to SAT.

Implications of Karp + Cook–Levin

All of these problems are **NP**-complete; they are manifestations of the same really hard problem.