CS311 Computational Structures

NP-completeness

Lecture 18

Andrew P. Black
Andrew Tolmach

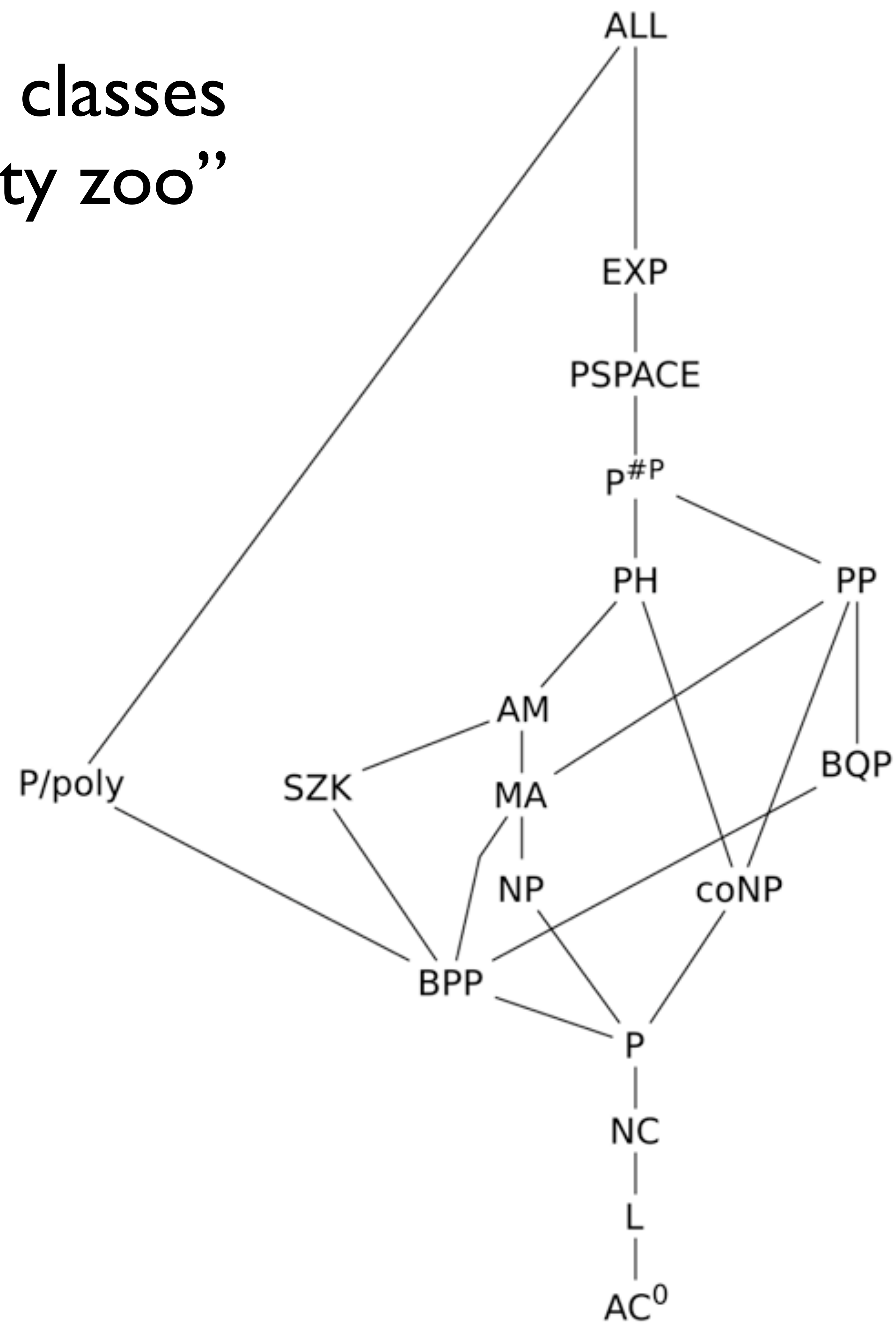
Portland State University

Some complexity classes

- P = Decidable in polynomial time on deterministic TM ("tractable")
- NP = Decidable in polynomial time on non-deterministic TM;

  = *Verifiable* in polynomial time on deterministic TM
- PSPACE = Decidable in polynomial space on a TM
- EXPTIME = Decidable in exponential time on a TM
  - Known to contain "intractable" problems

Some important classes in the "complexity zoo"



Portland State UNIVERSITY

Classifying problems

- Given a problem, we'd like to locate it as far down in the hierarchy as possible.
- This is often quite hard to do.
- Establishing an *upper* bound requires showing an algorithm
  - Someone may find a cleverer algorithm tomorrow!
    - this will give us a better upper bound
- Establishing a *lower* bound requires showing that there **cannot** be an algorithm.

Example: A_{CFG}

Example: A_{CFG}

- $A_{CFG} = \{ \langle G, w \rangle \mid G \text{ is a CFG that generates } w\}$

Example: A_{CFG}

- $A_{CFG} = \{ \langle G, w \rangle \mid G$ is a CFG that generates $w\}$
- A_{CFG} is *decidable*

Example: A_{CFG}

- $A_{CFG} = \{ \langle G, w \rangle \mid G \text{ is a CFG that generates } w\}$
- A_{CFG} is *decidable*
  - *what does this mean?*

Example: A_{CFG}

- $A_{CFG} = \{ \langle G, w \rangle \mid G \text{ is a CFG that generates } w\}$
- A_{CFG} is *decidable*
  - *what does this mean?*
  - *is it decidable "quickly"*

Portland State UNIVERSITY

Example: A_{CFG}

- $A_{CFG} = \{ \langle G, w \rangle \mid G \text{ is a CFG that generates } w\}$
- A_{CFG} is *decidable*
  - *what does this mean?*
  - *is it decidable "quickly"*
- How did we show that A_{CFG} is decidable?

Portland State UNIVERSITY

Attempt 1

Attempt 1

- For any CFG G, there is a PDA that decides whether w is in L(G)

Attempt 1

- For any CFG G, there is a PDA that decides whether w is in L(G)
  - What does this mean?

Portland State UNIVERSITY

Attempt 1

- For any CFG G, there is a PDA that decides whether w is in L(G)
  - What does this mean?
  - Problem:

Attempt 1

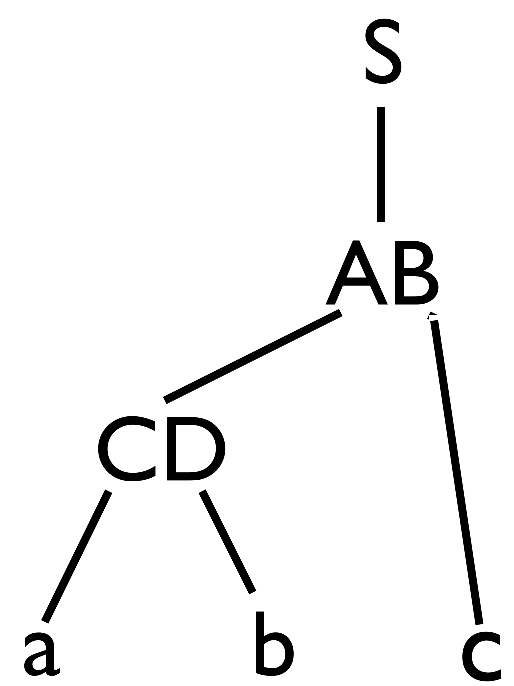
- For any CFG G, there is a PDA that decides whether w is in L(G)
  - What does this mean?
  - Problem:
- But: we can simulate a non-deterministic TM with a deterministic TM!

Attempt 1

- For any CFG G, there is a PDA that decides whether w is in L(G)
  - What does this mean?
  - Problem:
- But: we can simulate a non-deterministic TM with a deterministic TM!
  - Problem:

Attempt 2

- Recall Chomsky Normal Form:
  - Let G be a context-free grammar in Chomsky Normal Form. For any non-empty string $w \in L(G)$, exactly $2|w| - 1$ steps are required in any derivation of w.



$S \to AB$

$A \to CD$

$B \to EF \mid GF \mid c$

$C \to a$

$D \to b \mid x$

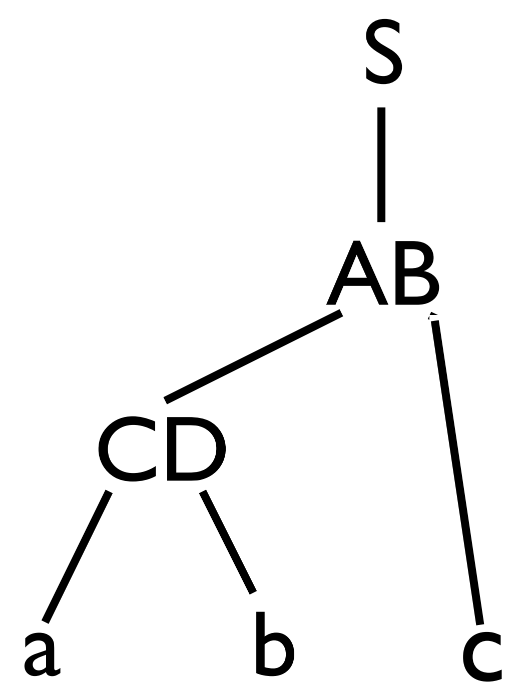
$E \to c \mid f$

$F \to d \mid g$

Attempt 2

- Recall Chomsky Normal Form:
  - Let G be a context-free grammar in Chomsky Normal Form. For any non-empty string $w \in L(G)$, exactly $2|w| - 1$ steps are required in any derivation of w.

- deriving a string of length 3 takes 5 steps

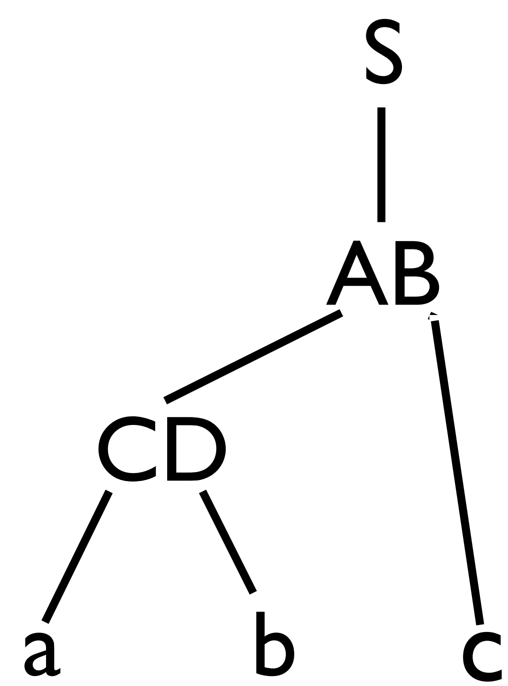


$S \rightarrow AB$
$A \rightarrow CD$
$B \rightarrow EF \mid GF \mid c$
$C \rightarrow a$
$D \rightarrow b \mid x$
$E \rightarrow c \mid f$
$F \rightarrow d \mid g$

Attempt 2

- Recall Chomsky Normal Form:
  - Let G be a context-free grammar in Chomsky Normal Form. For any non-empty string $w \in L(G)$, exactly $2|w| - 1$ steps are required in any derivation of w.

- deriving a string of length 3 takes 5 steps
- deriving a string of length 4 takes 7 steps

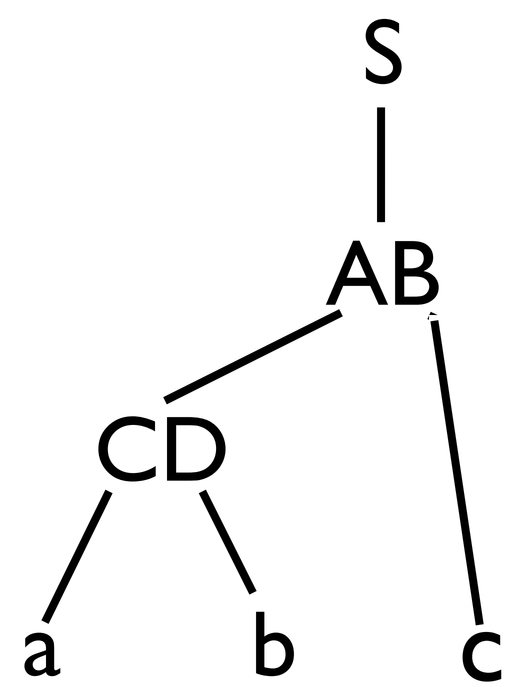


$S \rightarrow AB$
$A \rightarrow CD$
$B \rightarrow EF \mid GF \mid c$
$C \rightarrow a$
$D \rightarrow b \mid x$
$E \rightarrow c \mid f$
$F \rightarrow d \mid g$

Attempt 2

- Recall Chomsky Normal Form:
  - Let G be a context-free grammar in Chomsky Normal Form. For any non-empty string $w \in L(G)$, exactly $2|w| - 1$ steps are required in any derivation of w.

- deriving a string of length 3 takes 5 steps
- deriving a string of length 4 takes 7 steps
- deriving a string of length *n* takes 2n−1 steps

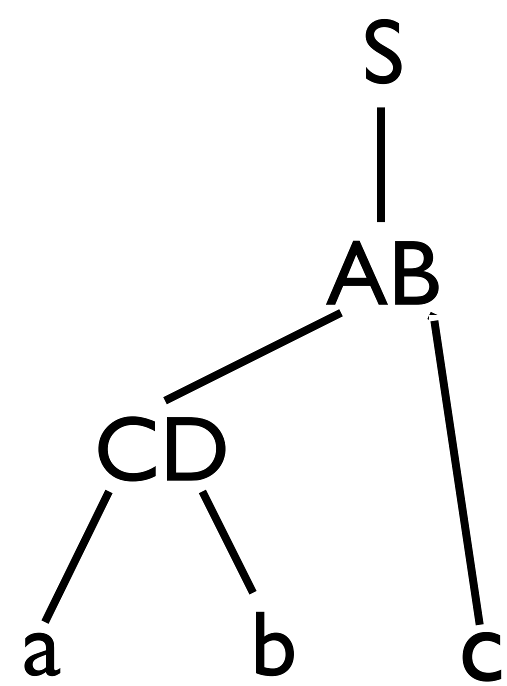


$S \rightarrow AB$
$A \rightarrow CD$
$B \rightarrow EF \mid GF \mid c$
$C \rightarrow a$
$D \rightarrow b \mid x$
$E \rightarrow c \mid f$
$F \rightarrow d \mid g$

Attempt 2

- Recall Chomsky Normal Form:
  - Let G be a context-free grammar in Chomsky Normal Form. For any non-empty string $w \in L(G)$, exactly $2|w| - 1$ steps are required in any derivation of w.

- deriving a string of length 3 takes 5 steps
- deriving a string of length 4 takes 7 steps
- deriving a string of length *n* takes 2n−1 steps



$S \rightarrow AB$

$A \rightarrow CD$

$B \rightarrow EF \mid GF \mid c$

$C \rightarrow a$

$D \rightarrow b \mid x$

$E \rightarrow c \mid f$

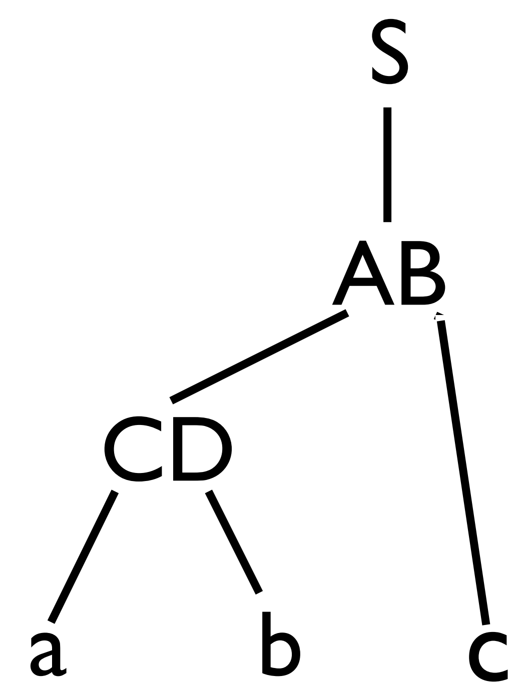
$F \rightarrow d \mid g$

- How many trials must a TM make before it “chooses” the right tree?

Attempt 2

- Recall Chomsky Normal Form:
  - Let G be a context-free grammar in Chomsky Normal Form. For any non-empty string $w \in L(G)$, exactly $2|w| - 1$ steps are required in any derivation of w.

- deriving a string of length 3 takes 5 steps
- deriving a string of length 4 takes 7 steps
- deriving a string of length n takes $2n-1$ steps



$S \to AB$

$A \to CD$

$B \to EF \mid GF \mid c$

$C \to a$

$D \to b \mid x$

$E \to c \mid f$

$F \to d \mid g$

- How many trials must a TM make before it “chooses” the right tree?
  - if there are p productions, we make $\leq p$ choices; in k steps we make p^k choices, so for a string of length n, we make $O(p^n)$ choices.

Attempt 3

- Dynamic Programming—The CYK Algorithm
  - What's Dynamic Programming?
    - accumulate information about small(er) subproblems
    - use this to solve progressively larger subproblems
  - Key: the subproblems overlap
    - We can save work by *memoizing* the answers

Is w in L(G)?

- Look at all the substrings of w
- Build a matrix M where $M[i,j]$ contains the set of variables that can generate $w[i..j]= w_i w_{i+1} w_{i+2} \dots w_j$
- start on the diagonal and work up

	1	2	3	4	…	…	n–1	n
1								
2	–							
3	–	–						
4	–	–	–					
…	–	–	–	–				
…								
n–1	–	–	–	–	–	–		
n	–	–	–	–	–	–	–	

Portland State UNIVERSITY

- suppose w = abcdefgxb
  - productions that yield terminals:

    B→c
    C→a
    D→b | x
    E→c | f
    F→d | g

- Step 1: substrings of w of length 1
  - e.g., $w[1..1]$=a, can be generated from C

	1	2	3	4	…	…	$n-1$	n
1	C							
2	–	D						
3	–	–	B,E					
4	–	–	–	F				
…	–	–	–	–				
…								
$n-1$	–	–	–	–	–	–	D	
n	–	–	–	–	–	–	–	D
	a	b	c	d …		…	x	b

productions: S→AB A→CD B→EF | GF

- Step 2: substrings of w of length 2
  - e.g., $w[1..2]$=ab
    - Split into shorter substrings in all possible ways
    - Use entries already in M to compute M [1..2]
  - $w[2,3]$ can only be derived from DB or DE, and neither is on the rhs of a production
  - $w[3,4]$ can be derived from BF or EF; EF can be derived from B

	1	2	3	4	…	…	$n-1$	n
1	C	A						
2	–	D	∅					
3	–	–	B,E	B				
4	–	–	–	F				
…	–	–	–	–				
…								
$n-1$	–	–	–	–	–	–	D	
n	–	–	–	–	–	–	–	D
	a	b	c	d	…	…	x	b

Portland State UNIVERSITY

productions: S→AB A→CD B→EF | GF

- Step k: substrings of w of length k
  - e.g., $w[1..k]$
    - Split into 2 shorter substrings in all k–1 possible ways
    - Use entries already in M to compute $M[1..k]$
- Step n: $M[1, n]$ can be broken into 2 shorter substrings in n–1 ways
- S $\in M[1, n] \equiv w \in$ L(G)

	1	2	3	4	…	…	n–1	n
1	C	A						
2	–	D	∅					
3	–	–	B,E	B				
4	–	–	–	F				
…	–	–	–	–				
…								
n–1	–	–	–	–	–	–	D	
n	–	–	–	–	–	–	–	D
	a	b	c	d	…	…	x	b

Portland State UNIVERSITY

- CYK algorithm is $O(n^3)$, where n is the length of the input.
- So every context-free language is a member of P

Thanks to Cocke, Younger and Kasami

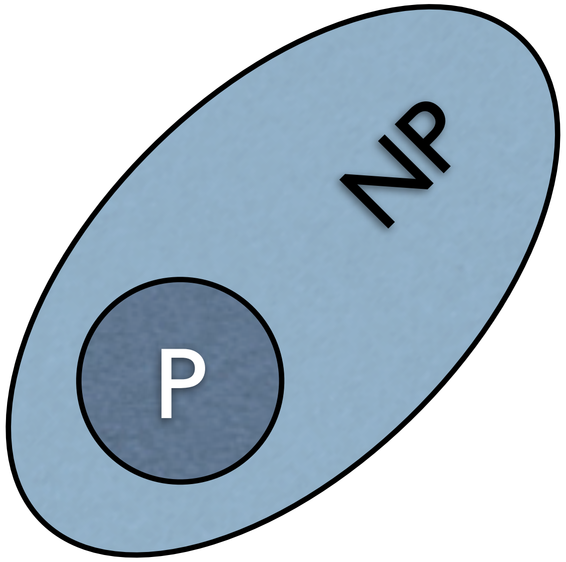
Hopcroft pp 304–307

P vs. NP

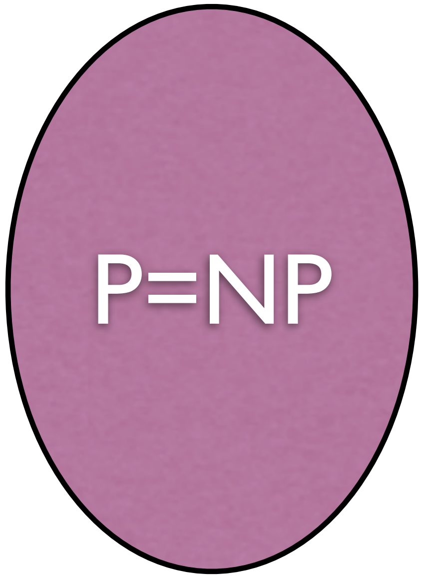
P = the class of languages for which membership can be *decided* quickly

NP = the class of languages for which membership can be *verified* quickly

"quickly" means "in polynomial time"



or



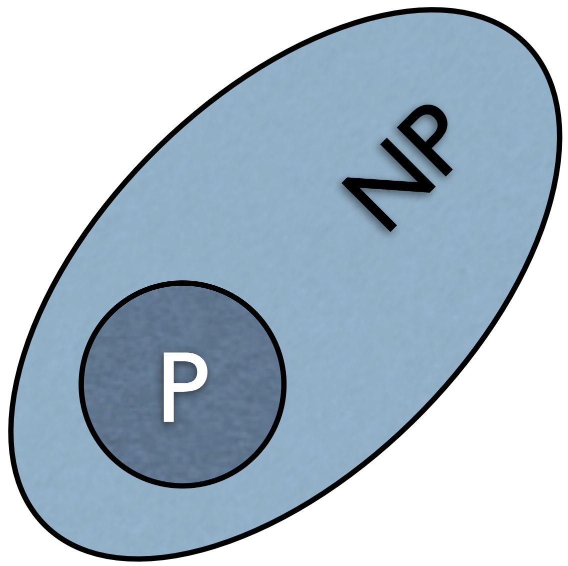
We don't know for sure which of these diagrams is correct

P vs. NP

P = the class of languages for which membership can be *decided* quickly

NP = the class of languages for which membership can be *verified* quickly

“quickly” means “in polynomial time”



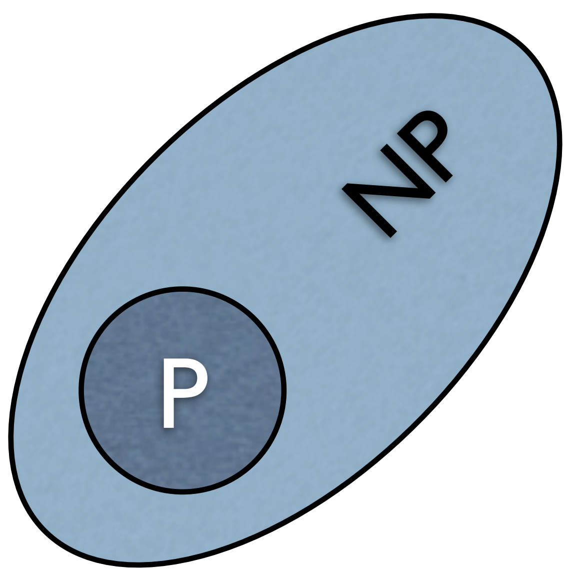
We don’t know for sure which of these diagrams is correct

P vs. NP

P = the class of languages for which membership can be *decided* quickly

NP = the class of languages for which membership can be *verified* quickly

"quickly" means "in polynomial time"



← Widely suspected

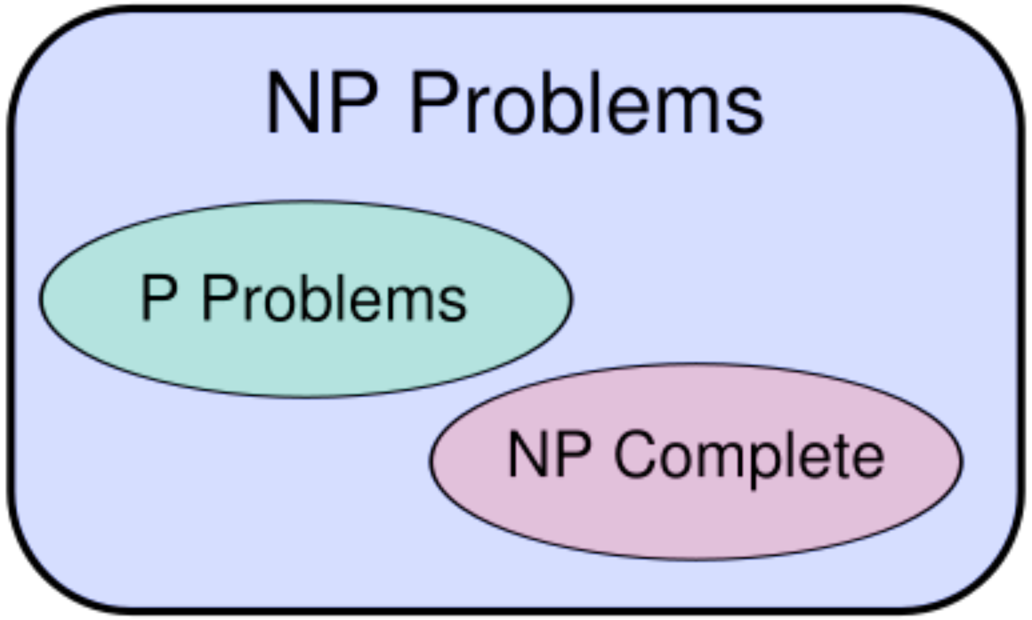
We don't know for sure which of these diagrams is correct

P vs. NP

- There are many interesting problems that we can show to be in NP, but that no one has shown to be in P.
- So is P ≠ NP ? Nobody knows!
- To investigate this question, it makes sense to look at the **hardest** problems in NP
  - these are least likely to be in P
  - if we can show that one is in P — they all will be

"Completeness"

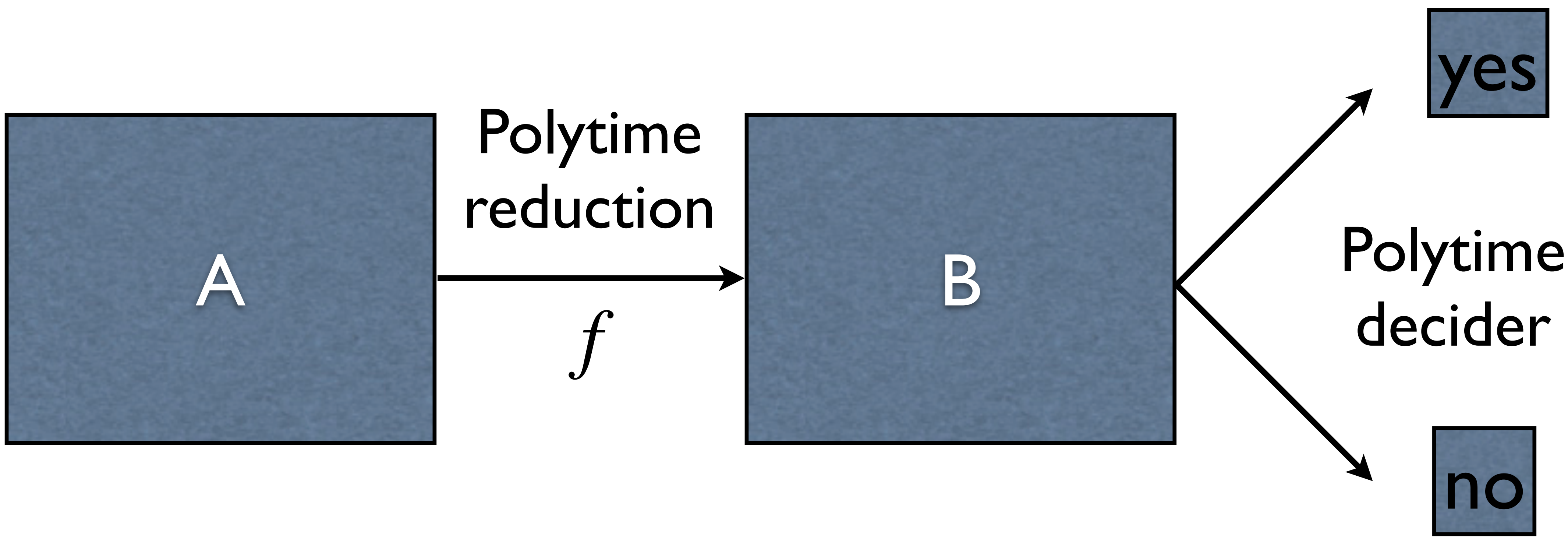
- A "complete" problem is one that is "as hard as possible" in its category.
- How can we formalize the idea of one problem being as hard as any other?
- We use the idea of *Reducibility*

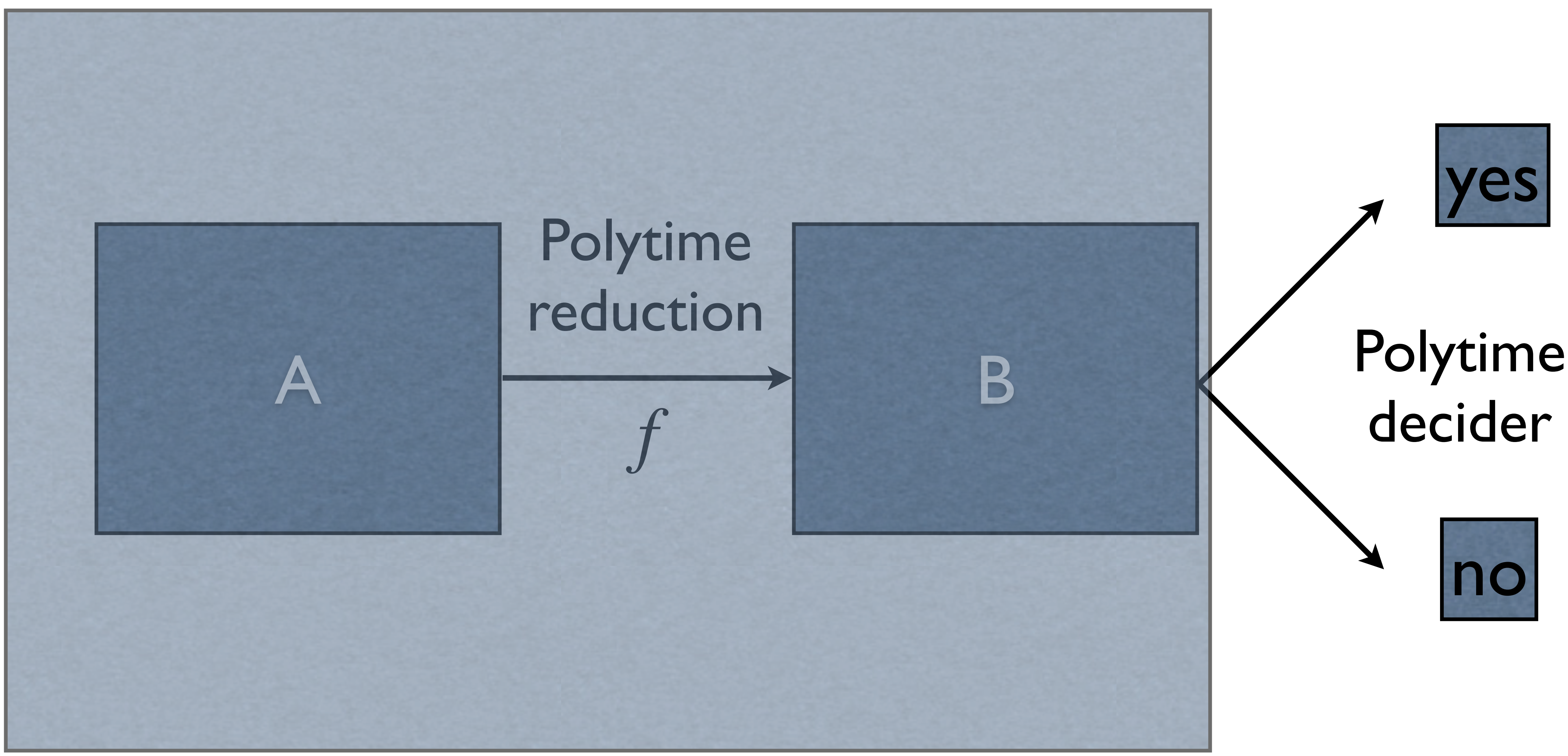


Portland State University

Reducibility, again

- A problem A is **polynomial-time reducible** to a problem B if there is a polynomial-time function f that maps instances of A to instances of B s.t.
  - I is a yes-instance of A $\equiv f(\mathrm{I})$ is a yes-instance of B
- Suppose we have an algorithm for B
  - We can solve an instance of A by first using f to transform it to an instance of B, and then solving the B-instance.
  - If our B-algorithm is polynomial, then so is this one for A.


A
Polytime
reduction
f
B
Polytime
decider
yes
no



Portland State University

NP-complete problems

- A language L is NP-complete if
  - L is in NP
  - Every language in NP is polynomial-time reducible to L.
- If L is NP-complete and $L \in P$, then $P = NP$
  - This is unlikely, so proving a problem is NP-complete strongly suggest that it is intractable
- If A is NP-complete and A is polynomial-time reducible to B, then B is also NP-complete

Some NP-complete problems

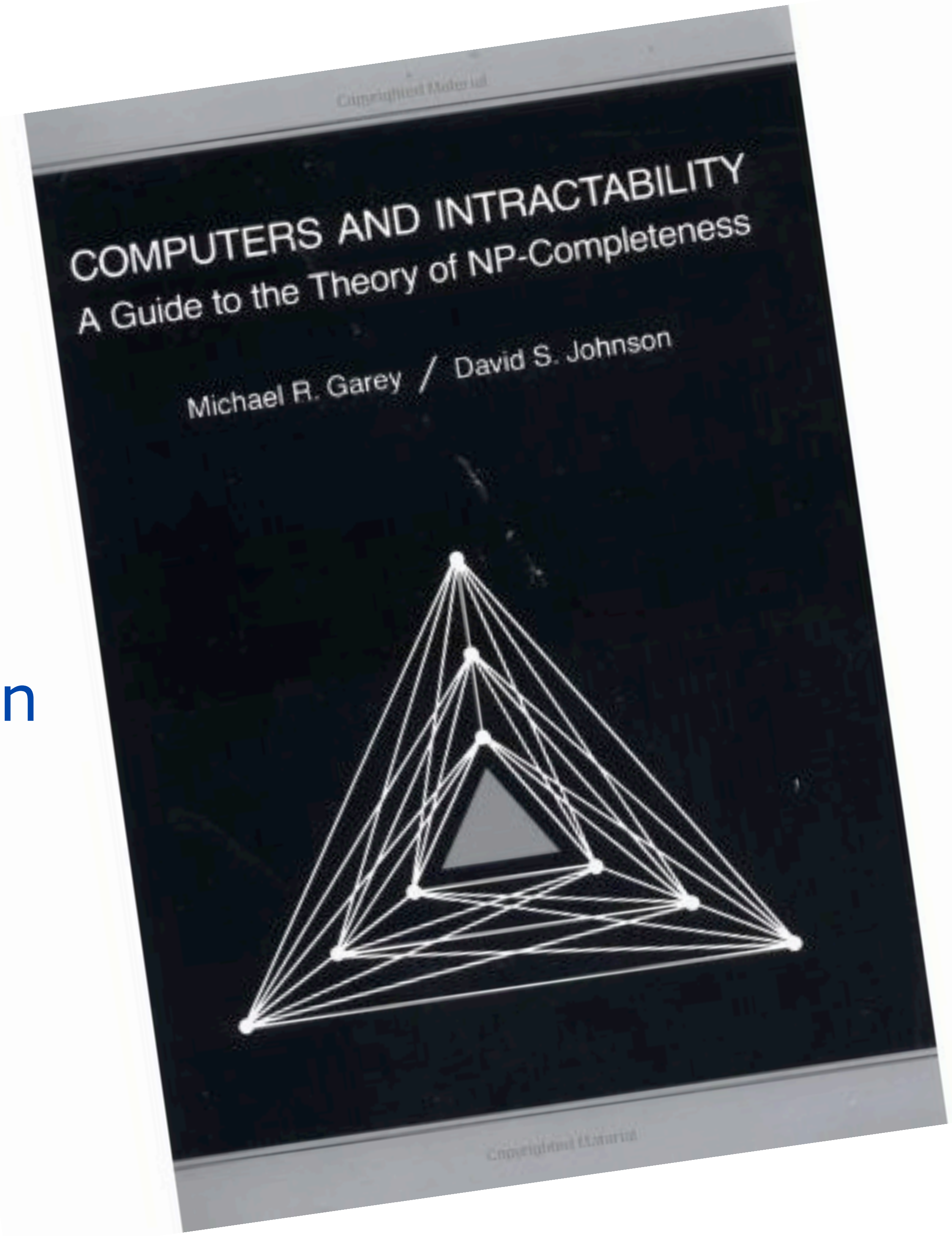
— see Garey and Johnson

- Hamiltonian circuit: Given a directed graph, is there a path that visits each vertex once?
- Traveling salesman: Given a set of cities, can they be toured traveling no more than a specified maximum distance?
- Partition: Given a finite set of positive integers, can they be partitioned into two subsets that sum to the same value?
- Graph isomorphism: Given two graphs, are they isomorphic?
- Quadratic diophantine equations: Given positive integers a,b,c, are there positive integers x and y such that $ax^2 + by = c$?
- Multiprocessor scheduling: Given a set of tasks with specified lengths, a number of processors, and a deadline, can the tasks be scheduled to complete by the deadline?

Computers and Intractability: A Guide to the Theory of NP-Completeness

M. R. Garey & D. S. Johnson

W. H. Freeman



SAT: Boolean Satisfiability

- Consider formulas over boolean variables and the operations AND ($\wedge$), OR ($\vee$), and NOT ($\neg$).
  - Ex. $\phi_1 = (x \wedge y) \vee (\neg y \wedge z)$ $\phi_2 = (x \vee y) \wedge \neg y \wedge \neg x$
- A formula is *satisfiable* if we can assign a value True ($\mathfrak{tt}$) or False ($\mathfrak{ff}$) to each variable such that the formula is True
  - Ex. $x=\mathfrak{tt}$, $y=\mathfrak{ff}$, $z=\mathfrak{tt}$ satisfies ϕ_1, but ϕ_2 is unsatisfiable

Portland State UNIVERSITY

SAT is NP-complete

- SAT = $\{\langle \phi \rangle \mid \phi$ is a satisfiable Boolean formula $\}$ is the paradigmatic example of an NP-complete language
  - Cook-Levin Theorem.
- A closely-related NP-complete language is 3SAT: the satisfiable 3CNF-formulae
  - CFN = conjunctive normal form: an AND of ORs of literals (variables or their negations)

Cook-Levin Theorem [1971]

Cook-Levin Theorem [1971]

- SAT is NP-complete

Cook-Levin Theorem [1971]

- SAT is NP-complete

Cook-Levin Theorem [1971]

- SAT is NP-complete
- What does this mean?

Cook-Levin Theorem [1971]

- SAT is NP-complete

- What does this mean?
  - SAT $\in$ NP

Portland State UNIVERSITY

Cook-Levin Theorem [1971]

- SAT is NP-complete
- What does this mean?
  - SAT $\in$ NP
  - every A $\in$ NP is poly-time reducible to SAT

Cook-Levin Theorem [1971]

- SAT is NP-complete
- What does this mean?
  - SAT $\in$ NP
  - every A $\in$ NP is poly-time reducible to SAT

Portland State UNIVERSITY

Cook-Levin Theorem [1971]

- SAT is NP-complete

- What does this mean?
  - SAT $\in$ NP
  - every A $\in$ NP is poly-time reducible to SAT

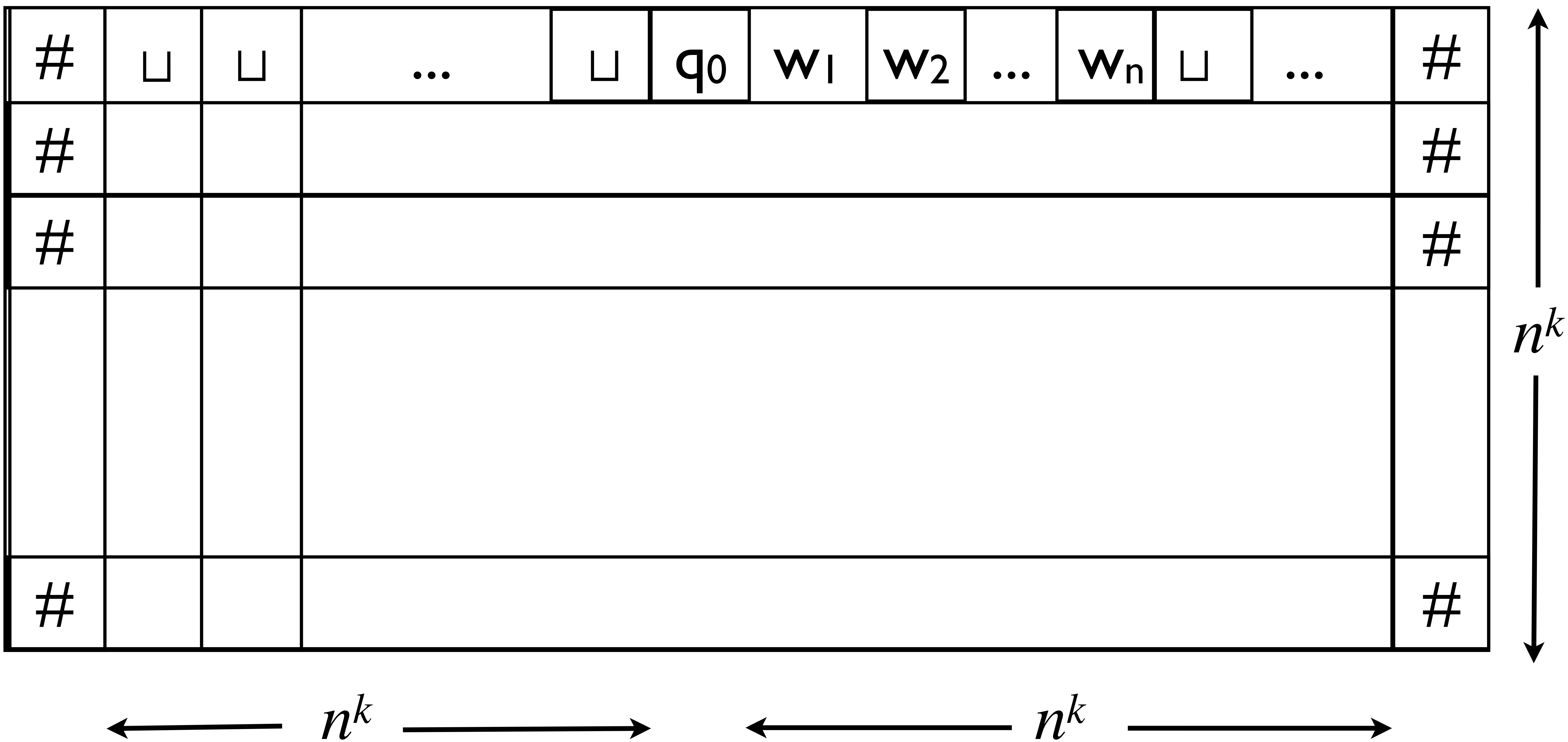
- How on earth can we prove such a thing?

Cook-Levin Theorem

- Basic Idea:
  - Any problem in NP has a NDTM that solves it in poly-time, say in time n^k
  - Let's look at the n^k steps that the NDTM must take in solving it
  - Represent each of those steps as a boolean formula

- An accepting NDTM computation on an input w is described by a finite **tableau** where
  - each row is a machine configuration
  - the first row is the start configuration with w
  - each row leads to the next by a legal transition
  - some row describes an accepting configuration

Format of tableau



Encoding the tableau

- N accepts w iff there exists an accepting tableau for N on w.
- We define a boolean formula ϕ that is satisfiable iff such a tableau exists
  - Each tableau cell[i,j] contains a symbol in $C = Q \cup \Gamma \cup \{\#\}$
  - Represent cell contents using boolean variables $x_{i,j,s}$ where $x_{i,j,s} = 1$ iff cell[i,j] = s
  - Define $\phi = \phi_{cell} \wedge \phi_{start} \wedge \phi_{move} \wedge \phi_{accept}$

Portland State UNIVERSITY

Details

- ϕ_{cell} ensures that exactly one symbol appears in each cell

$$\phi_{cell} = \bigwedge_{\substack{1 \le i \le n^k \\ 1 \le j \le 2n^k+3}} \left[\left(\bigvee_{s \in C} x_{i,j,s} \right) \wedge \left(\bigwedge_{\substack{s,t \in C \\ s \neq t}} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}}) \right) \right]$$

- ϕ_{start} ensures that the first row of the tableau is the starting configuration

$$\begin{aligned} \phi_{start} = \quad & x_{1,1,\#} \wedge x_{1,2,\sqcup} \wedge \ldots \wedge x_{1,n^k+1,\sqcup} \wedge \\ & x_{1,n^k+2,q_0} \wedge x_{1,n^k+3,w_1} \wedge x_{1,n^k+4,w_2} \wedge \ldots \wedge x_{1,n^k+n+2,w_n} \wedge \\ & x_{1,n^k+n+3,\sqcup} \wedge \ldots \wedge x_{1,2n^k+2,\sqcup} \wedge x_{1,2n^k+3,\#} \end{aligned}$$

- ϕ_{accept} ensures that an accepting configuration occurs somewhere in the tableau

$$\phi_{accept} = \bigvee_{\substack{1 \leq i \leq n^k \\ 1 \leq j \leq 2n^k+3 \\ q_a \in F}} x_{i,j,q_a}$$

- ϕ_{move} ensures that rows of tableau represent legal transitions of the machine

$$\phi_{move} = \bigwedge_{\substack{1 \leq i < n^k \\ 1 \leq j < 2n^k+3}} (\text{legal\_window\_at}(i,j))$$

$$\text{legal\_window\_at}(i,j) = \bigvee_{\text{legal\_window}(a_1,\ldots,a_6)} \left(\begin{array}{l} x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge \\ x_{i+1,j-1,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6} \end{array} \right)$$

Legal windows

- A 2×3 window is **legal** if it might appear when one configuration correctly follows another in the tableau
- Set of legal windows for machine is defined by alphabets, states and transition function
- Straightforward but tedious to define all legal windows

Example window constructions

- Suppose we have $\delta(q_1,a) = \{(q_1,b,R)\}$ and $\delta(q_1,b) = \{(q_2,c,L),(q_2,a,R)\}$
- Here are some legal windows:

a	q_1	b
q_2	a	c

a	q_1	b
a	a	q_2

a	a	q_1
a	a	b

#	b	a
#	b	a

a	b	a
a	b	q_2

b	b	b
c	b	b

- Here are some illegal windows:

a	b	a
a	a	a

a	q_1	b
q_1	a	a

b	q_1	b
q_2	b	q_2

Checking polynomial time

- It is crucial that ϕ can be constructed in polynomial time
- This follows from
  - size of tableau: ~$2n^{2k}$ cells
  - finite number of symbols: $|Q| + |\Gamma| + 1$
  - hence $O(n^{2k})$ variables
  - ϕ contains fixed-size fragment per cell

Proving a problem is NP-complete by reduction

3CNF

- a *literal* is a variable, or a negation of a variable, e.g.
  - $x_1, \neg x_2, x_3, \neg x_2$
- a 3CNF formula is a boolean formula in conjunctive normal form in which each conjunction has exactly 3 literals, e.g.
  - $(x_1 \vee \neg x_2 \vee x_3) \wedge (x_3 \vee \neg x_5 \vee x_6) \wedge (x_3 \vee x_6 \vee \neg x_4)$

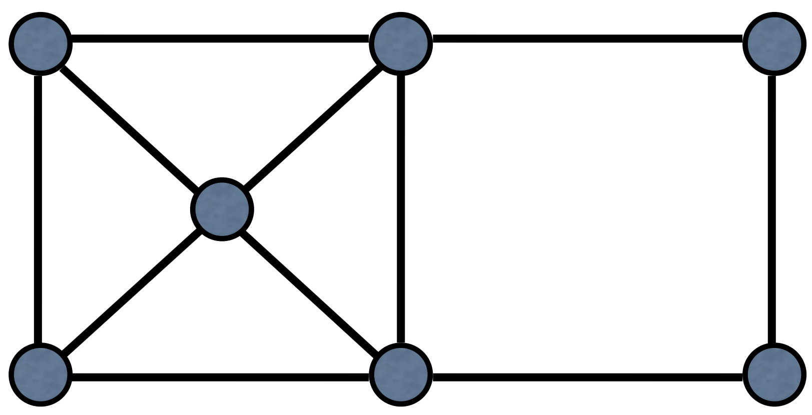
SAT can be poly-time reduced to 3SAT

- Details are in Hopcroft §10.3.2
- Key idea:

$$(a \vee b \vee c \vee d) \equiv (a \vee b \vee x) \wedge (c \vee d \vee \neg x)$$

The Clique Problem

- The Clique problem is to decide if a graph contains a clique of a certain size
  - a clique is a subgraph in which every pair of nodes is connected by an edge
- CLIQUE = $\{\langle G, k\rangle \mid G$ is an undirected graph with a k-clique$\}$

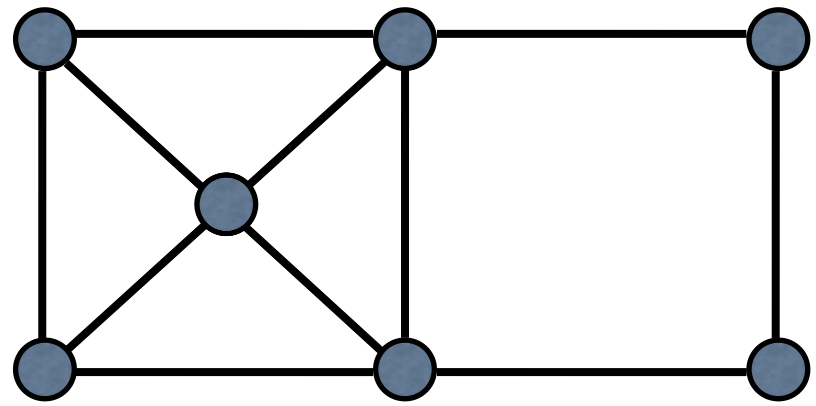


Portland State University

CLIQUE is in NP

- Here is a verifier for CLIQUE:
  - Input is $\langle G, k \rangle$, c
    1. Test whether c is a set of k nodes in G: — $O(|c|)$ time
    2. Test whether G contains all edges connecting nodes in c: — $O(|c|^2)$ time
    3. If both tests pass, *accept,* otherwise, *reject.*
- It runs in time polynomial in the length of the input

Is CLIQUE in P?



- What's the time complexity of a search for *k*-cliques in a graph with *n* nodes?
- No polynomial time algorithm is known

3SAT is polynomial-time reducible to CLIQUE

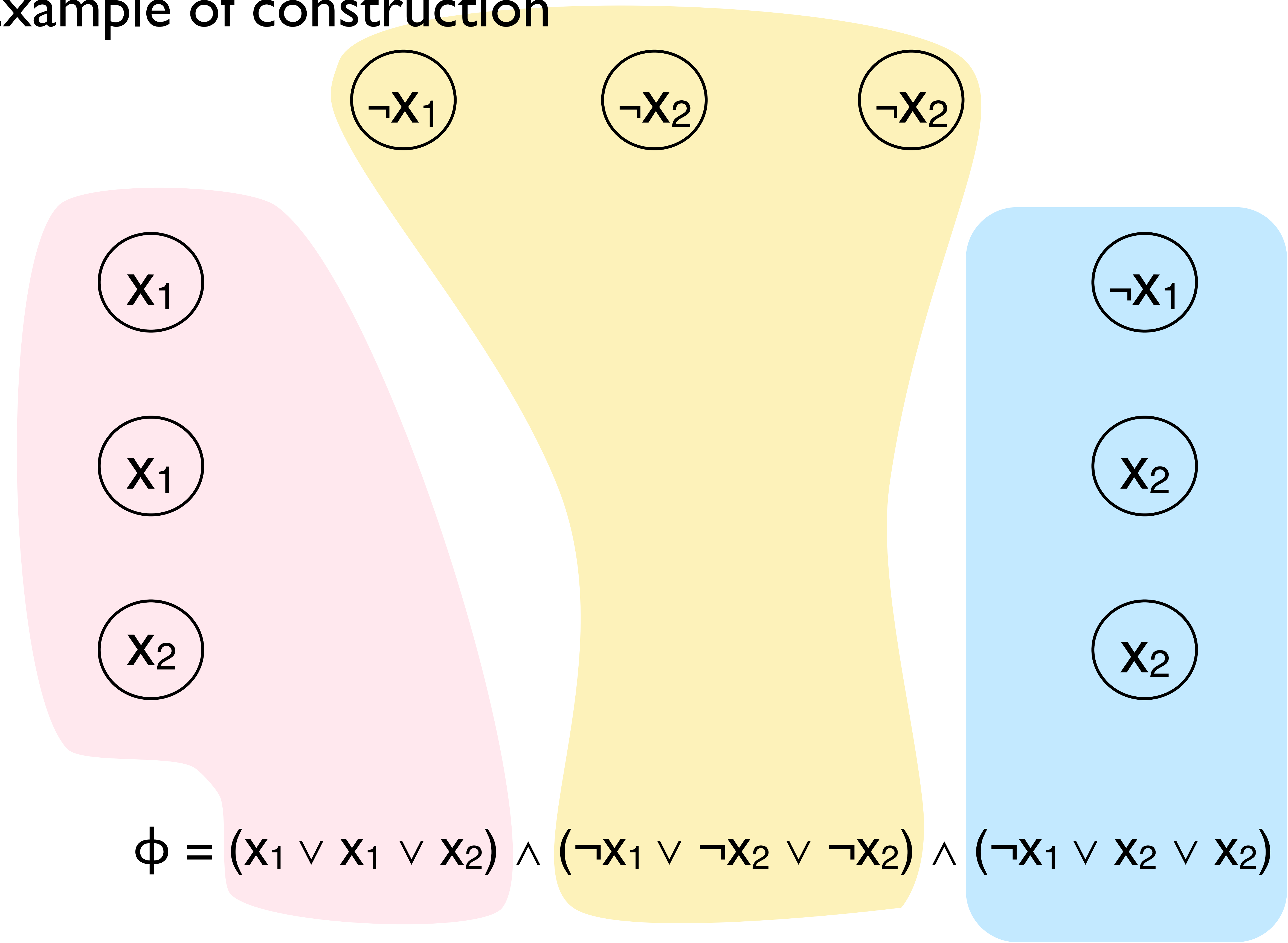
Idea:

- Convert formulae to graphs in a certain form.
- Find cliques in the graph
  - each clique corresponds to a satisfying assignment in the formula.

Construction

- Given a formula ϕ with k conjuncts we build a graph G and look for k-cliques.
  - One node in G for each *occurrence* of a literal in ϕ. Each node is labeled by that literal.
  - Organize the nodes into groups of 3, called triples. Each triple corresponds to a conjunct in ϕ.
  - Each node in the triple corresponds to a literal in the clause.

Example of construction



$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$$

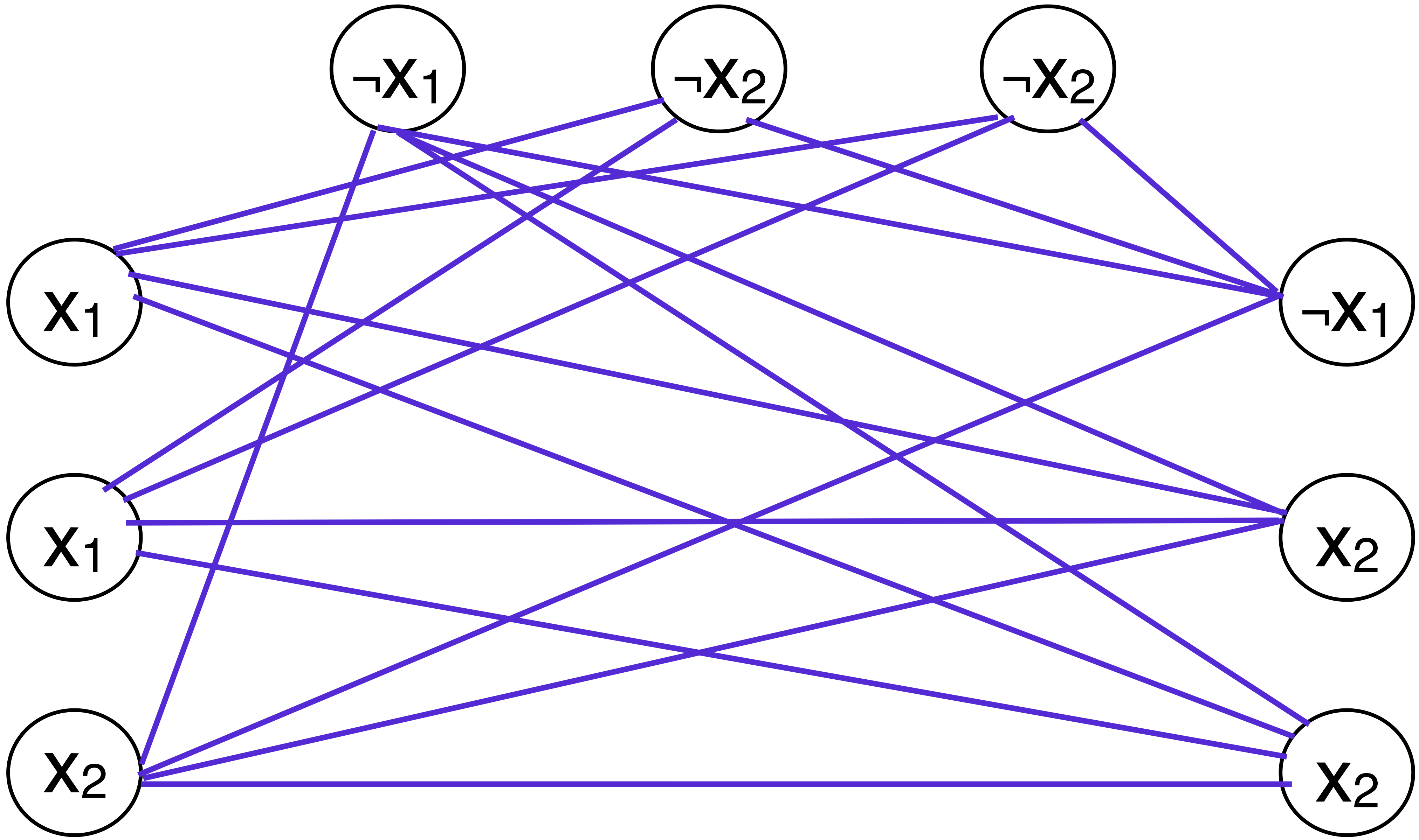
Portland State University

Construction (continued):

- Connect all the nodes in *G except:*

  1. Don't connect two nodes if they are in the same triple
  2. Don't connect two nodes if one is labeled x and the other is labeled $\neg x$

Example of construction

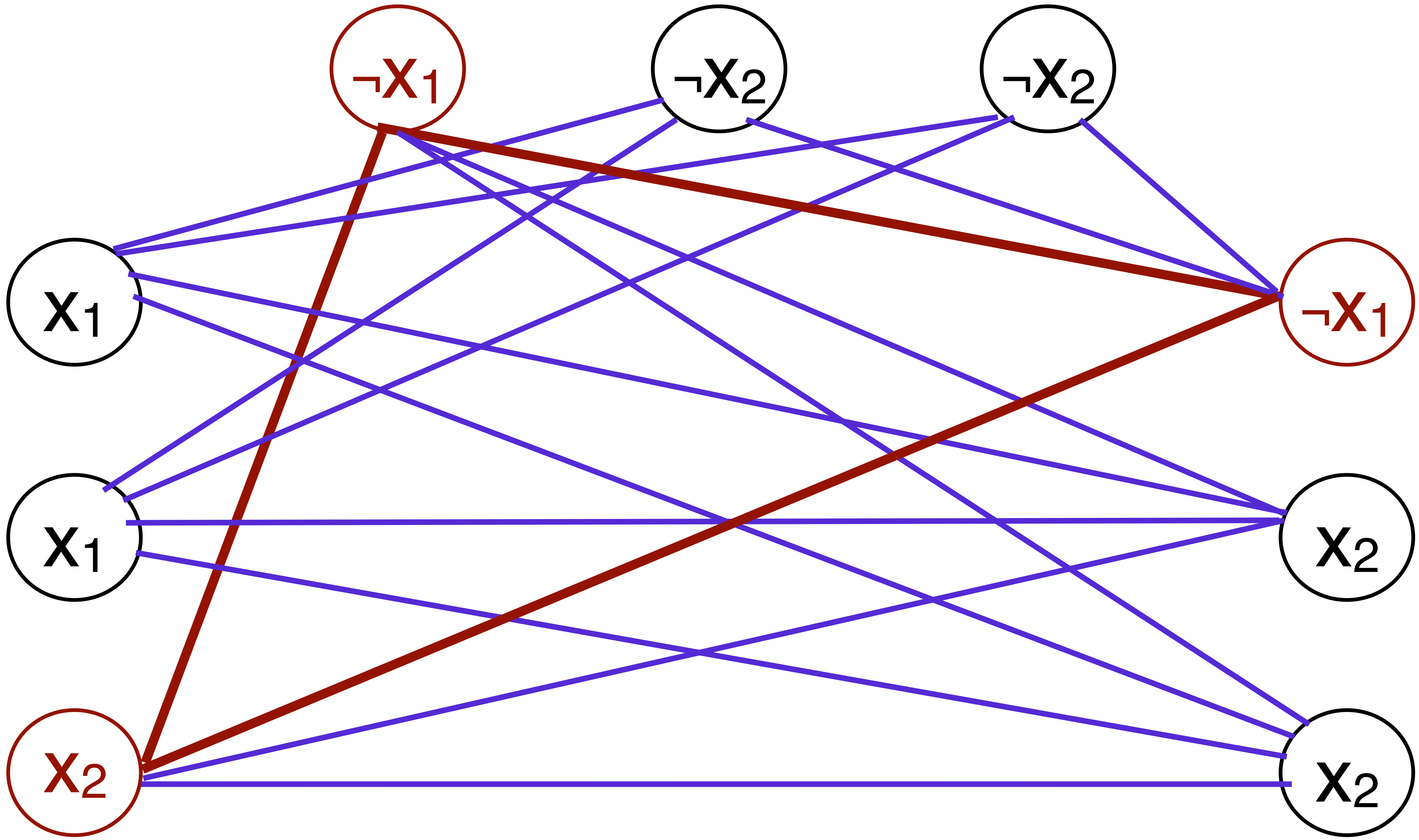


$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$$

Proof

- Suppose that ϕ has a satisfying assignment
  - Then at least one literal is true in every clause
  - In G, select one node in each triple whose label is true: those nodes form a k-clique

Example of construction



$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$$

Proof

- Suppose that ϕ has a satisfying assignment
  - Then at least one literal is true in every clause
  - In G, select one node in each triple whose label is true: those nodes form a k-clique
    - There are k of them, because we chose one per clause
    - Each pair is joined by an edge, because no two are in the same triple, and no two are labeled with contradictory literals

- Suppose that G has a k-clique c
  - No two nodes are in the same triple
    - because there are no edges joining such nodes
  - Each triple contains exactly one node of c
    - because there are k triples
  - Assign truth values to the literals so that each node in c is TRUE
    - This is always possible, because no edge joins x and $\neg x$
  - This assignment satisfies ϕ, because one literal in each of the k-clauses of ϕ is TRUE
- So CLIQUE is NP-complete