Tractable and Intractable Problems

Lecture 08.01

by Marina Barsky

Design and analysis of algorithms

Main focus: practical algorithms + supporting theory for solving fundamental computational problems:

- **Sorting**
- **Searching**
- **Shortest paths**
- **Sequence alignment**
- **Spanning trees**
- **…**

You might feel that now you can solve any problem efficiently, and always can do better

Design and analysis of algorithms

Main focus: practical algorithms + supporting theory for solving fundamental computational problems:

- Sorting
- Searching
- Shortest paths
- Sequence alignment
- Spanning trees
- …

You might feel that now you can solve any problem efficiently, and always can do better

Bad news: **many important practical problems that you will encounter in your projects do not have known efficient solutions**

New tools

- Classifying problems by hardness
- Identifying intractable problems
- Strategies for dealing with such problems

Complexity class P

We say that the problem is ***tractable*** if there is an algorithm which solves it in time $O(n^k)$ for some constant k, and where n represents the input size [More precisely - the number of bits or keystrokes needed to describe the input]

Tractable:

$O(n)$, $O(n^2)$, $O(n^{1000})$, $O(n^{10,000,000})$

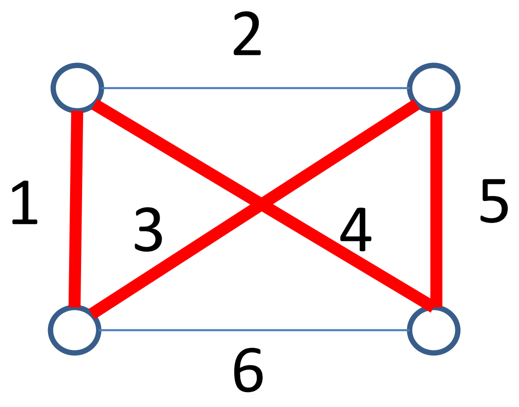
Class P: set of all problems solvable in polynomial time

All the algorithms we developed so far are in class P

Not in P?

Traveling Salesperson Problem (TSP)

Input: complete undirected graph with non-negative edge costs
Output: a min-cost tour - a cycle that visits each vertex exactly once



TSP path: 13

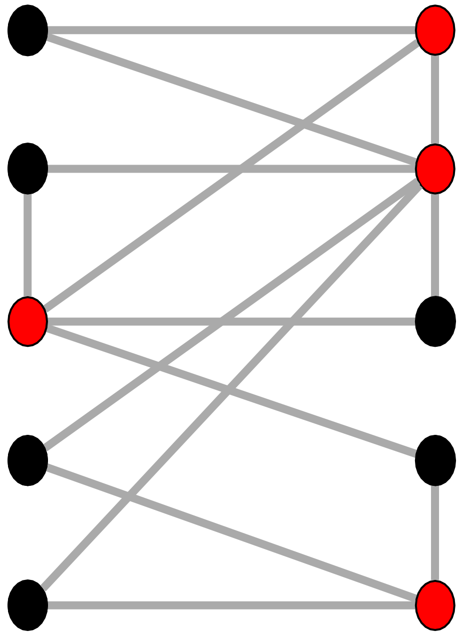
Solution:
Try all permutations of vertices and select the sequence with the cheapest cost

We solved shortest paths, min spanning trees, why not TSP?
60 years of research – and there is no polynomial-time algorithm?

Not in P?

Min Vertex Cover

Input: graph G (V, E)
Output: a minimum-size subset C of vertices such that for each edge (v,w), we have v ∊ C or w ∊ C (C *covers* all the edges)



Vertex Cover of size 4

Solution:
Try every subset of V and for each subset check if it covers all the edges.
Keep subset of a minimum size

Not in P?

Knapsack 01 without repetitions

Input: set of n items with their weights and values and the knapsack capacity W
Output: maximum value of knapsack filled with items that fit into W (each item can be used only once)

Solution:

check 2^n =16 knapsacks

Each verification takes O(n)

Time complexity **$O(n2^n)$ = 4*8 = 32**

[exponential in n]

What about DP solution?

subset	total weight	total value
Ø	0	$0
{1}	7	$42
{2}	3	$12
{3}	4	$40
{4}	5	$25
{1,2}	10	$54
{1,3}	11	-
{1,4}	12	-
{2,3}	7	$52
{2,4}	8	$37
etc. …		

Example:
items = {(7 lbs, $42),
(3 lbs, $12),
(4 lbs, $40),
(5 lbs, $25)}

n=4 (4 items)
W = 10

KnapsackDP(*W*, *n* items)

```
initialize all $maxvalue[0, i] \leftarrow 0$
initialize all $maxvalue[w, 0] \leftarrow 0$
for $i$ from 1 to $n$:
    for $w$ from 1 to $W$:
        $maxvalue[w, i] \leftarrow maxvalue[w, i-1]$
        if $w_i \leq w$:
            $val \leftarrow maxvalue[w - w_i, i - 1] + v_i$
            if $val > maxvalue[w, i]$:
                $maxvalue[w, i] \leftarrow val$
return $maxvalue[W, N]$
```

1,2,3,....W

Exhaustive – running time: $O(n2^n) = 4*8 = 32$

DP – running time $O(nW) = 10*4 = 40$

Running Time of DP Knapsack: *closer look*

- The running time is $O(nW)$
- *W* is not the **size** of the input - after all, the input consists of a single number (total knapsack capacity) - not *W* knapsacks
- We loop over all possible values between 0 and W, and the time is **not proportional** to the size of the input, which is n+1 (n items and 1 number W)
- For example for W=1,125,899,906,842,624 we will perform 1,125,899,906,842,624 loop iterations, while the input still consists of a single number W.

DP Knapsack is not polynomial!

- The running time of an algorithm is defined as a function of the input size

- In normal O(n) complexity we assume that reading each of n input numbers can be done in constant time (each number is using a constant number of bits)

- Say, we use m bits to represent number W. If we were just reading this number once, then the complexity would be proportional to m

- But instead we need to loop from 0 to 2^m

- The complexity is $O(n2^m)$: we need to check 2^m imaginary knapsacks

- The algorithm is exponential in number of bits used to represent the capacity: if we add just one more bit – we double W and double the run time

Pseudo-polynomial running time

- The complexity of an algorithm refers to the **number of input elements**, **not a value of a single element** in the input
- More precisely, the input size n is a number of keystrokes (alternatively number of bits) needed to describe the input
- Thus the complexity of the knapsack remains exponential in input size even with dynamic programming
- The DP knapsack algorithm is ***pseudo-polynomial***
- NP-hard problems with pseudo-polynomial solutions are called *weakly NP-complete*

DP knapsack is exponential:

$$O(n2^m)$$

Input size: number of bits to represent number W

wikipedia link

Polynomial or pseudo-polynomial?

- Is prime (num)
- Naïve GCD (a,b)
- Money change (target)
- Subset sum (A of size n, target sum)
- Edit distance (S1 of size m, S2 of size n)
- Bellman-Ford (G(V,E), source s)

Polynomial or pseudo-polynomial?

- Is prime (num) – pseudo-polynomial
- Naïve GCD (a,b) – pseudo-polynomial
- Money change (target) – pseudo-polynomial
- Subset sum (A of size n, target sum) – pseudo-polynomial
- Edit distance (S1 of size m, S2 of size n) -polynomial
- Bellman-Ford (G(V,E), source s) – polynomial

Intractable problems

Not all problems are tractable=can be solved in polynomial time

What is common to all the above problems: they can be solved via exhaustive search

Problem types

Most existing computational problems belong to one of three types:

- **Decision problems: return Boolean answer Yes/No**

- **Optimization problems: return min/max of some function [subject to constraints]**

- **Construction problems: return a structure with desired properties**

Problem types: examples

- **Decision problems: return Boolean answer Yes/No**
  - Is there a subset with sum = k? Yes or no?
  - Is there a cycle in the graph which passes through all vertices and visits every edge exactly once?
- **Optimization problems: return min/max of some function**
  - What is the value of the min-cost path from s to t?
  - What is the max possible value in a knapsack?
- **Construction problems: return a structure with desired properties**
  - Produce a shortest path from s to t
  - Produce a sequence of items in knapsack of maximum value

Complexity theory considers decision problems only

Decision problems: return Boolean answer Yes/No

Problem of any type can be reduced to a decision problem or a sequence of decision problems

Example 1: Optimization

Problem p1: Max value of knapsack

Problem p2: Is there a knapsack of value at least k

Reduction of p1 to p2:

- We ask: is there a knapsack with value V = $(v_i+v_2+v_3+\ldots v_n)$?
  - If the answer is yes, this is the max value – we fit all the available items
  - If the answer is no, next decision problem: is there knapsack with value at least V/2?
- Binary search until we find the max value
- Not always can do a binary search, but nevertheless we can reduce an optimization problem to polynomial number of invocations of some decision problem.

Complexity theory considers decision problems only

Decision problems: return Boolean answer Yes/No

Problem of any type can be reduced to a decision problem or a sequence of decision problems

Example 2: Construction

Problem p1: Sequence of items in max-valued knapsack
Problem p2: Max value of knapsack
Problem p3: Is there a knapsack of value at least k
Reduction of p1 to p2 (which in turn reduces to p3)

- After we found max value V_{max} (see previous slide), we start removing one item i at a time and ask: is there still knapsack with value V_{max}?
- If the answer is no, the solution has to include item i
- We check for all n items in turn

Reductions

- Reduction from one problem to another is a routine approach in algorithm design

- For any new problem we ask: maybe I already know how to solve it? Maybe I can rephrase it as a shortest-path problem? Maybe I can invoke a known algorithm multiple times?

Informally:
Problem p1 reduces to p2 if given a poly-time algorithm for solving p2, we can use this algorithm as a subroutine for solving p1:
$p1 \leq_p p2$

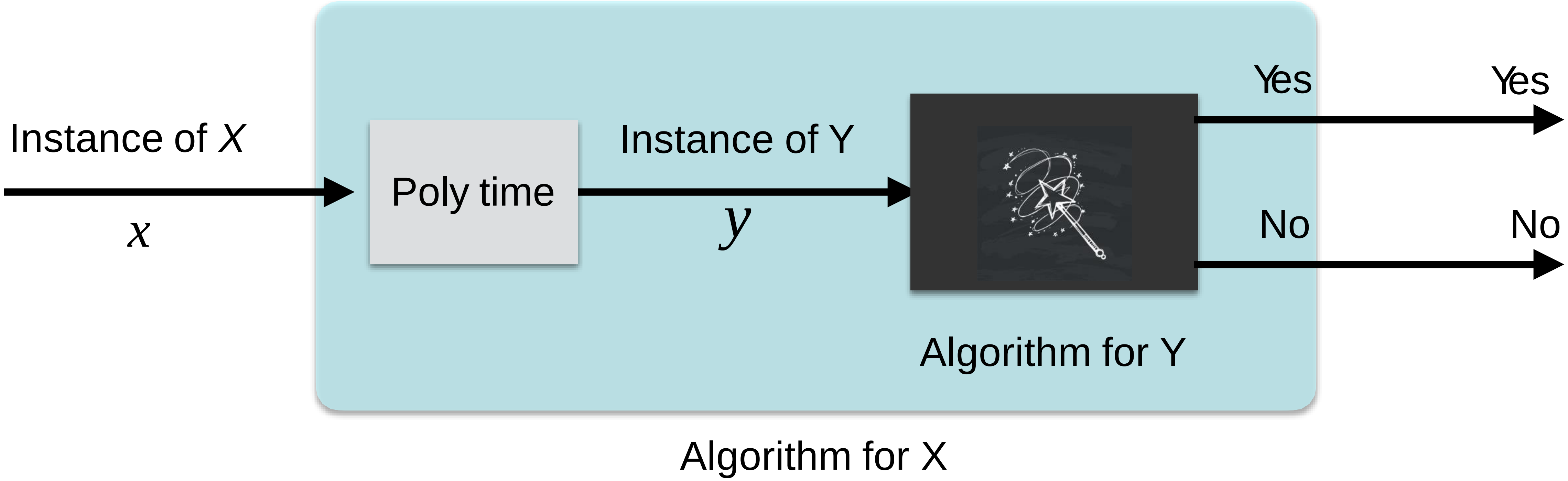
Examples:
- Computing median reduces to sorting
- All pair shortest paths reduces to n invocations of the single-source shortest path

Polynomial-time reductions

Formal definition: Decision problem X is ***(polynomial-time) reducible*** to decision problem Y if we can convert any instance of X into an instance of Y and the following three conditions hold:

- The new converted input for Y has size polynomial in the original input for X
- The conversion and invocation of solution for Y is done in polynomial number of steps
- For any instance of problem X the algorithm for Y returns exactly the same decision

Notation: **$X \leq_p Y$**



Complexity class NP

The complexity class NP is defined to include all the decision problems from class P but allow for the inclusion of problems that may not be in P

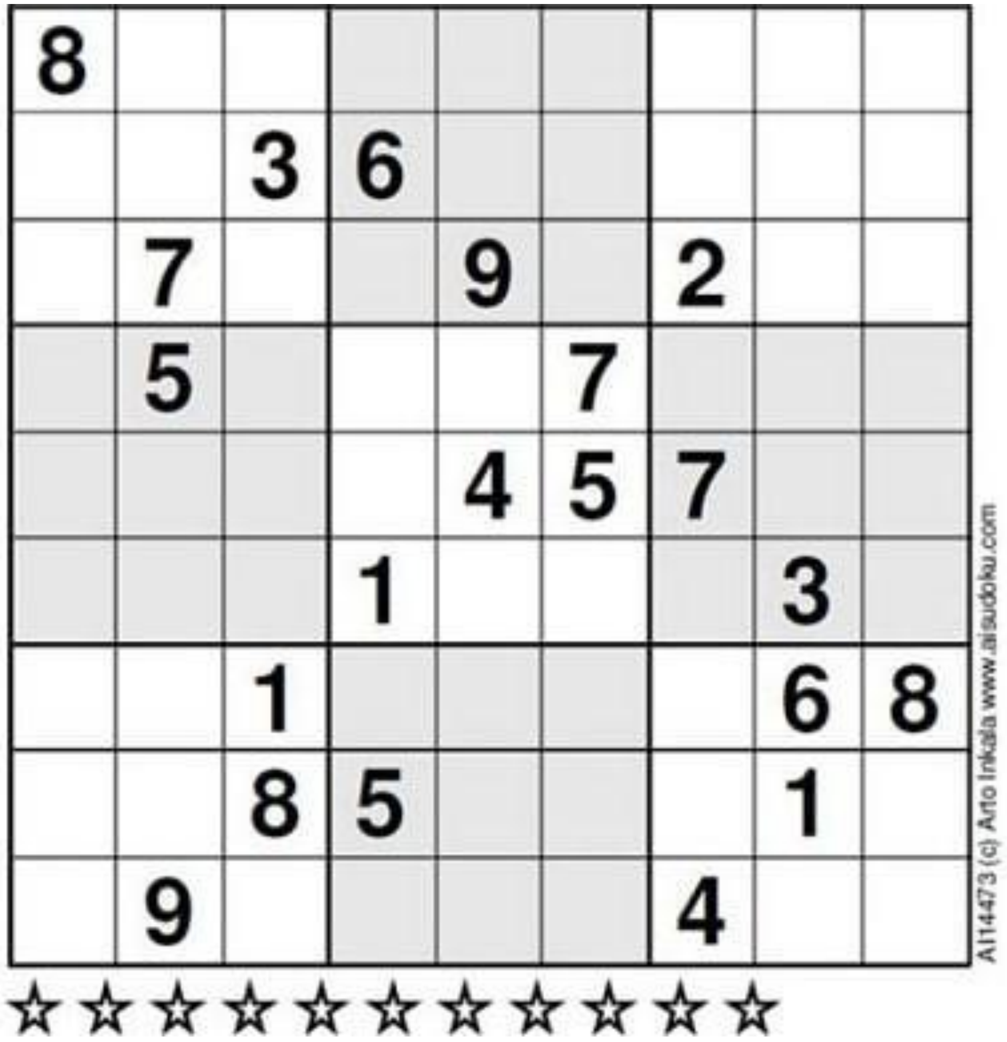
Every problem in NP can be solved in exponential time via Exhaustive Search

The solution must be efficiently verifiable:

- Solutions (certificates) always have length polynomial in input size
- Proposed solution can be verified in polynomial time

Checking a given solution is polynomial,
number of candidates can be exponential

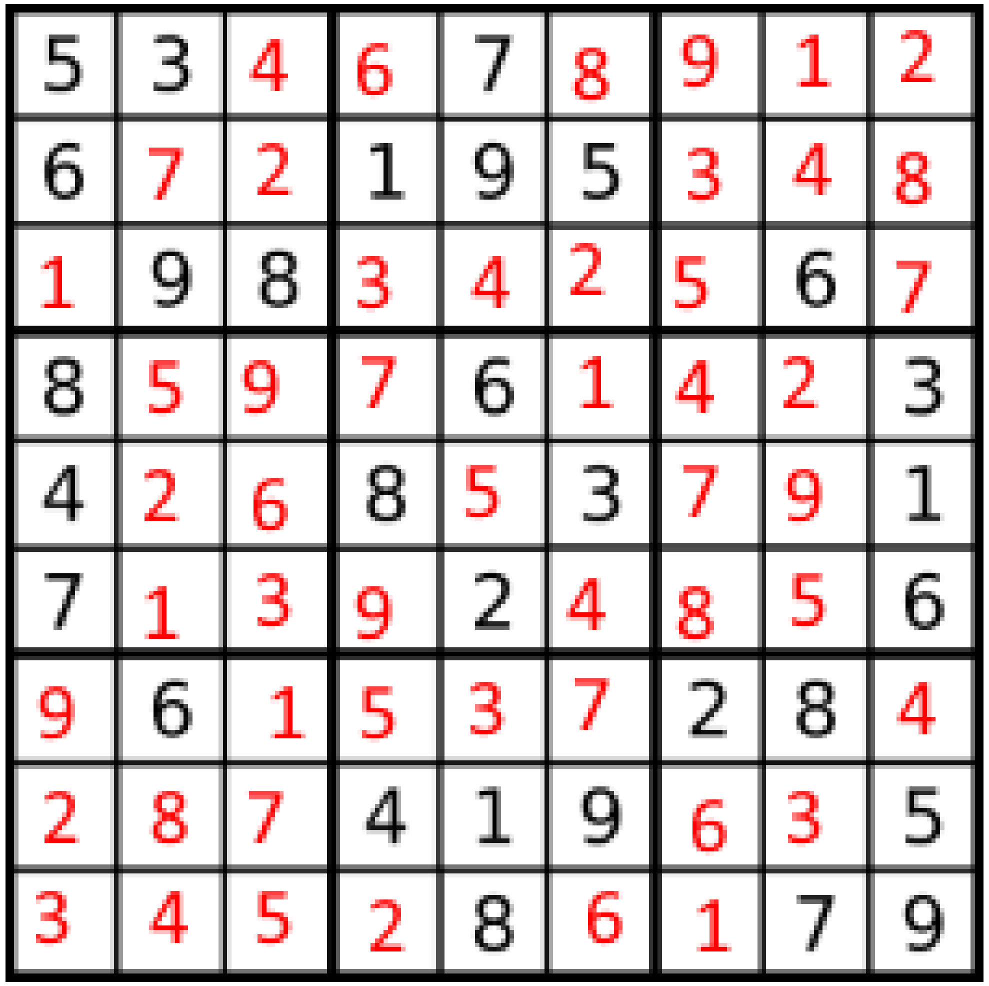
Class NP—example



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

AI14473 (c) Arto Inkala www.aisudoku.com

☆☆☆☆☆☆☆☆☆☆☆



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

Checking solution to Sudoku can be done in polynomial time. So sudoku is in **NP**

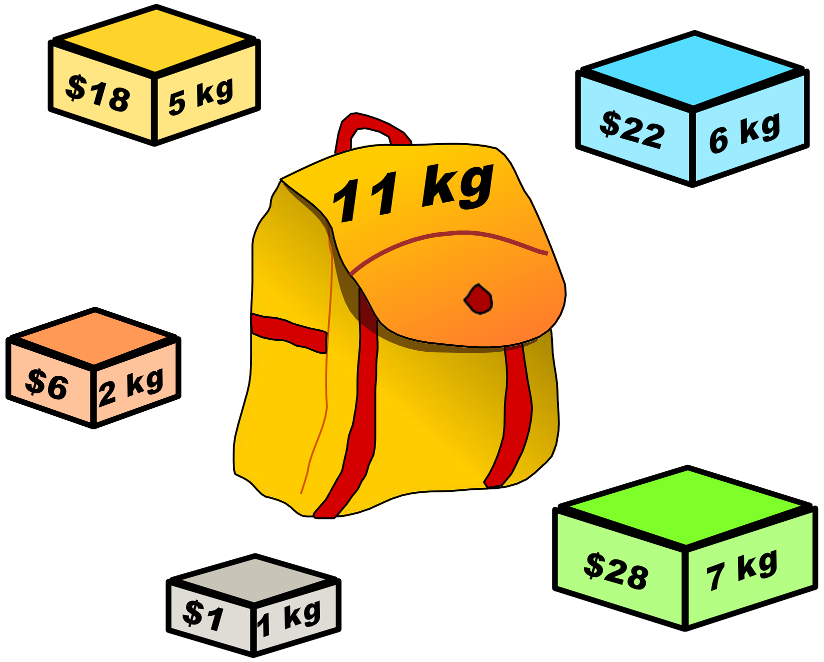
Class NP—example

Problem: is there a knapsack with value $40?

- {3, 4} has value $40 (and weight 11)

i	v_i	w_i
1	$1	1 kg
2	$6	2 kg
3	$18	5 kg
4	$22	6 kg
5	$28	7 kg

knapsack instance
(weight limit W = 11)



Checking the total value of a proposed knapsack can be done in polynomial time. So knapsack is in NP

Classic problem in NP: Satisfiability

Given a logical expression, can we assign “True” and “False” to the variables to *satisfy* the equation (make the expression True)?

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

3-SAT. A SAT formula where each clause contains exactly 3 literals (corresponding to different variables)

$$\phi = (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_4)$$

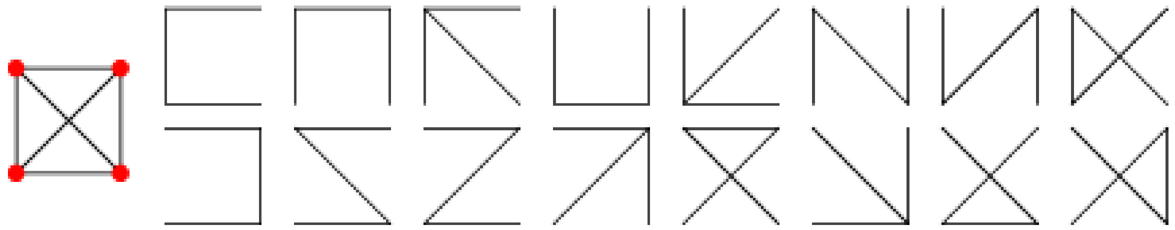
Satisfying instance: $x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 0$

SAT, 3-SAT $\in$ NP

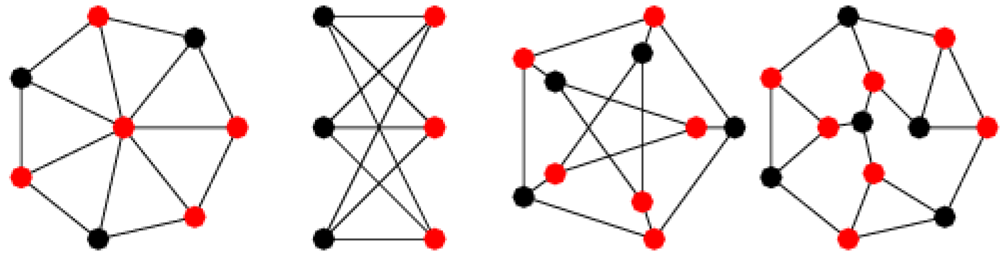
- Certificate: truth assignment to variables (poly-size)
- Poly-time verifier: check if assignment makes ϕ true

Which problems in NP are tractable?

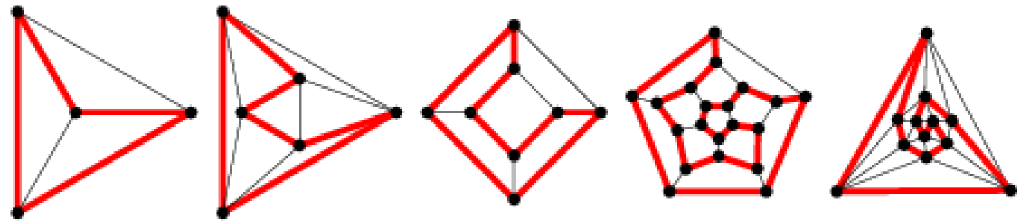
Spanning tree ✓



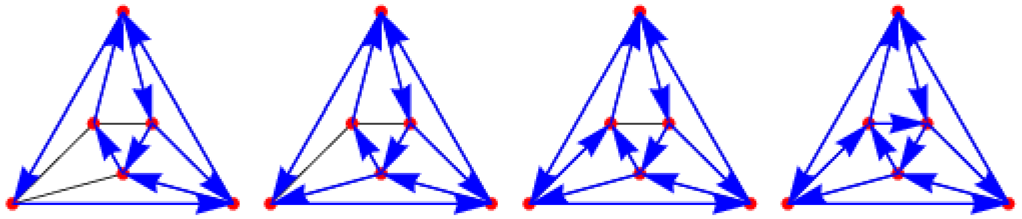
Min vertex cover



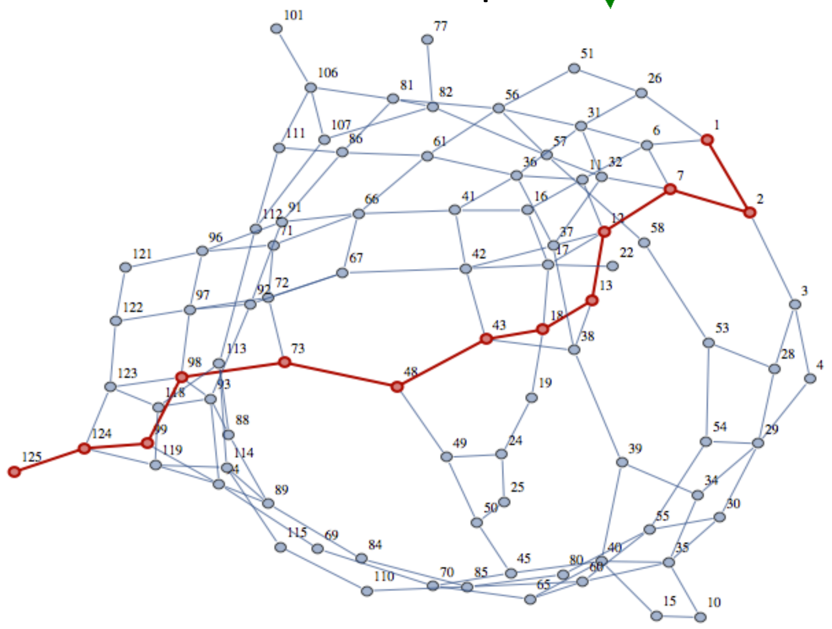
Hamiltonian Cycle



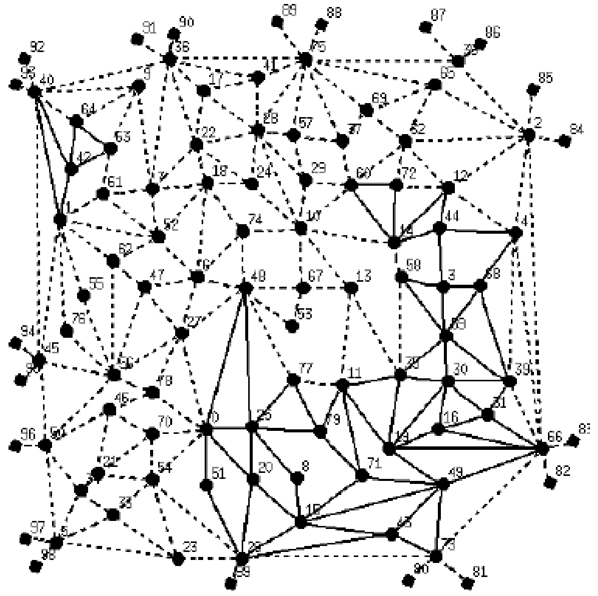
Eulerian cycle ✓



Shortest path ✓



Longest path



The definitions seem very similar…

Defining Intractability

- **Complexity class NP contains different problems: some of them we call *tractable* and others *intractable***

- **For the latter we do not know if a polynomial solution exists**

- **Problems in NP are still decidable (solvable): if only we could magically guess the right solution, we could then quickly test it**

- **How do we formally define intractability?**

- **Evidence of intractability: relative difficulty**
  TSP is “at least as hard as” the list of really hard problems

Completeness, or relative hardness

- **Suppose p1 reduces to p2: p1 $\leq_p$ p2**

- **If we know that p1 cannot be solved efficiently in poly-time, then p2 cannot be solved in poly-time either**

- **Contrapositive use of reductions:**
  If p1 is not in P then neither is p2

- **p2 is at least as hard as p1**

- **To use this, we need to have at least one problem that is not in P: this is the hardest problem in NP, and we call it NP-complete**

NP-completeness

- **By definition: solving 1 NP-complete problem in poly-time will provide a solution to all NP problems [P=NP]**

- **Interpretation: an NP-complete problem encodes simultaneously all problems for which the solution can be efficiently recognized (a “universal problem”)**

- **Can such problems really exist?**

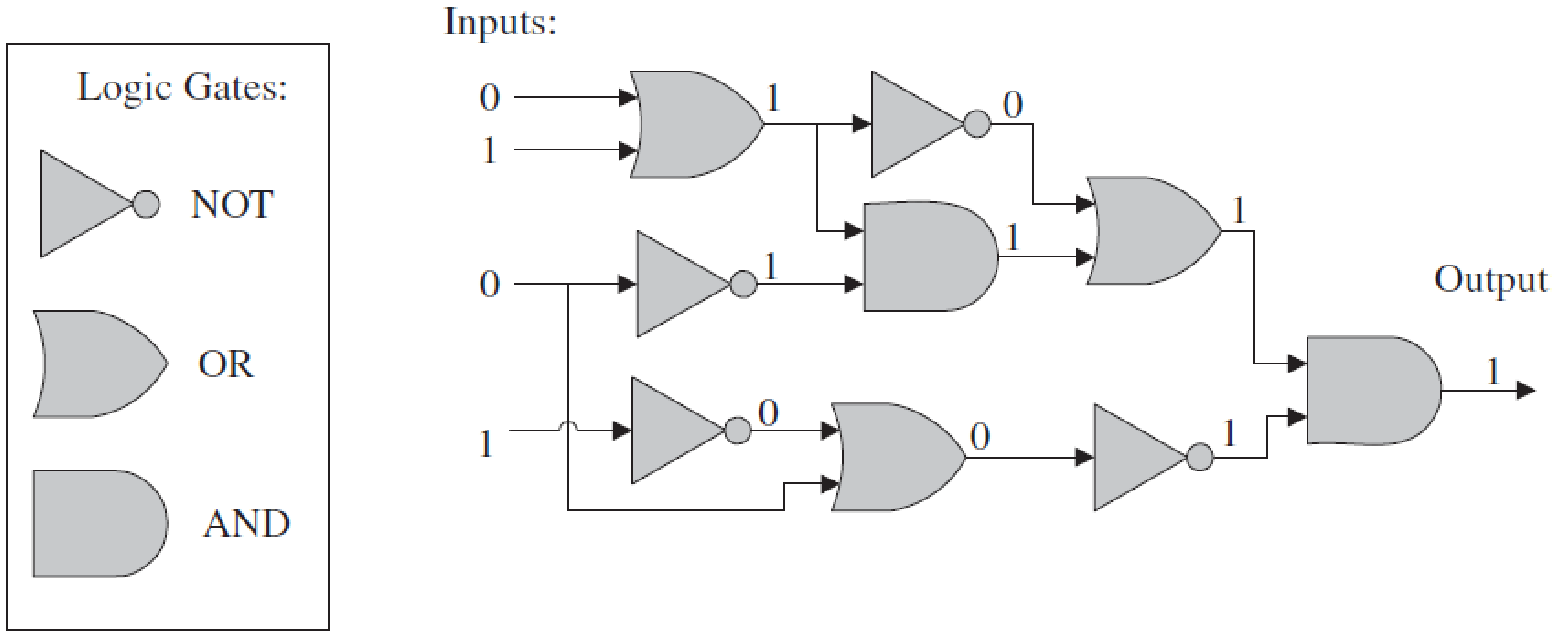
Cook-Levin theorem

(Cook 71, Levin 73)

NP-complete problems exist

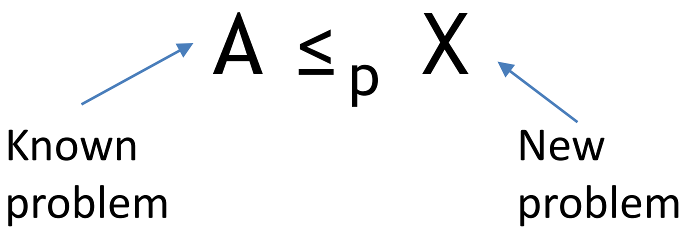
Any computer program can be represented by a circuit-SAT.

Circuit-SAT is NP-complete



You’ll see the proof in CSCI 361

The logic of reductions



$A \leq_p X$

Known problem

New problem

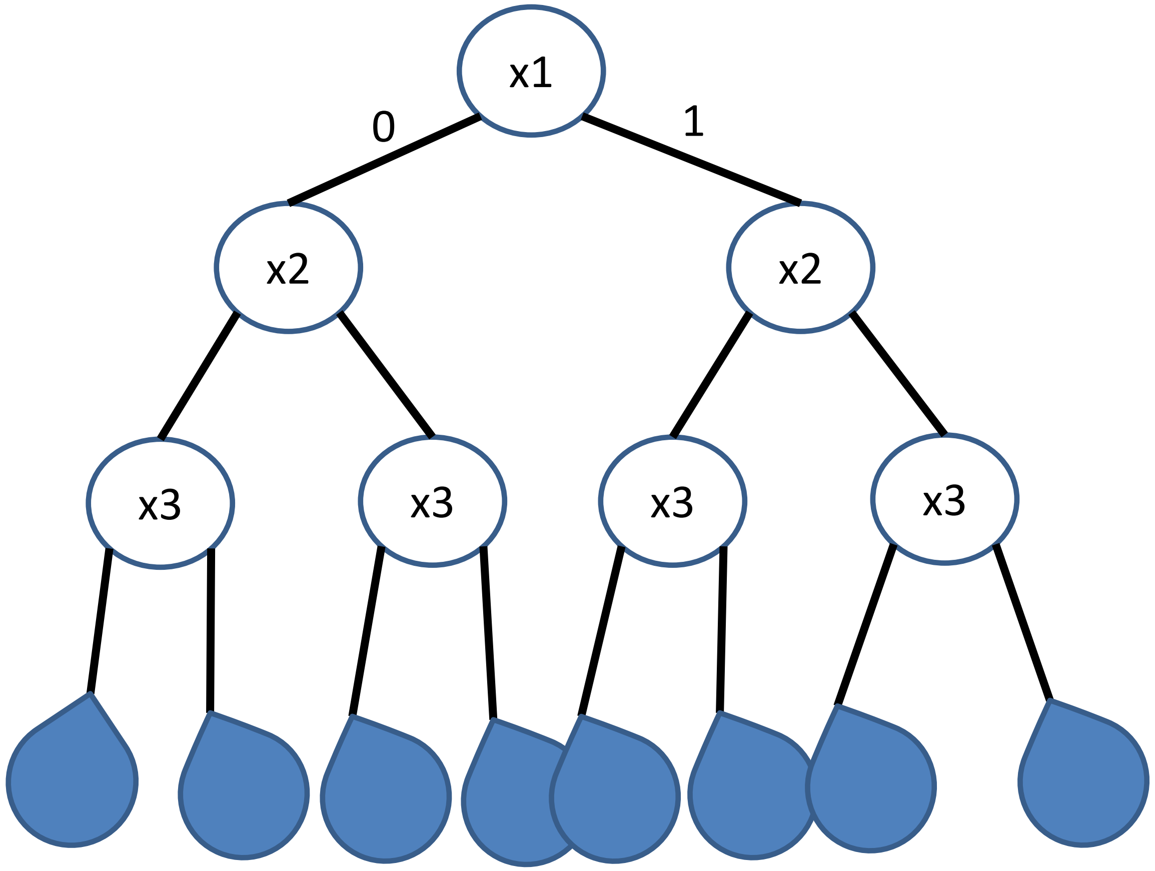
Suppose we can reduce problem A to problem X in polynomial time

- If A is in P then nothing is known about X - we can always encode the easy instance into a hard one
- If X is in P then A is also in P: solve X in poly time + poly time of reduction
- If A is not in P then X is not in P (contrapositive): suppose X is in P then A should also be in P - but it is not
- If X is not in P then nothing known about A - A might have a poly solution - we just encoded an easy problem into a hard problem

To prove that a new problem X is NP-complete, reduce a known NP complete problem A to X

Decision tree

Decision problems can be expressed as a tree of decisions or choices



$$\phi = (\overline{x_1} \lor x_2 \lor x_3) \land (x_1 \lor \overline{x_2} \lor x_3) \land (\overline{x_1} \lor x_2 \lor x_3)$$

Alternative definition of class NP:

The decision tree may have exponential number of leaves

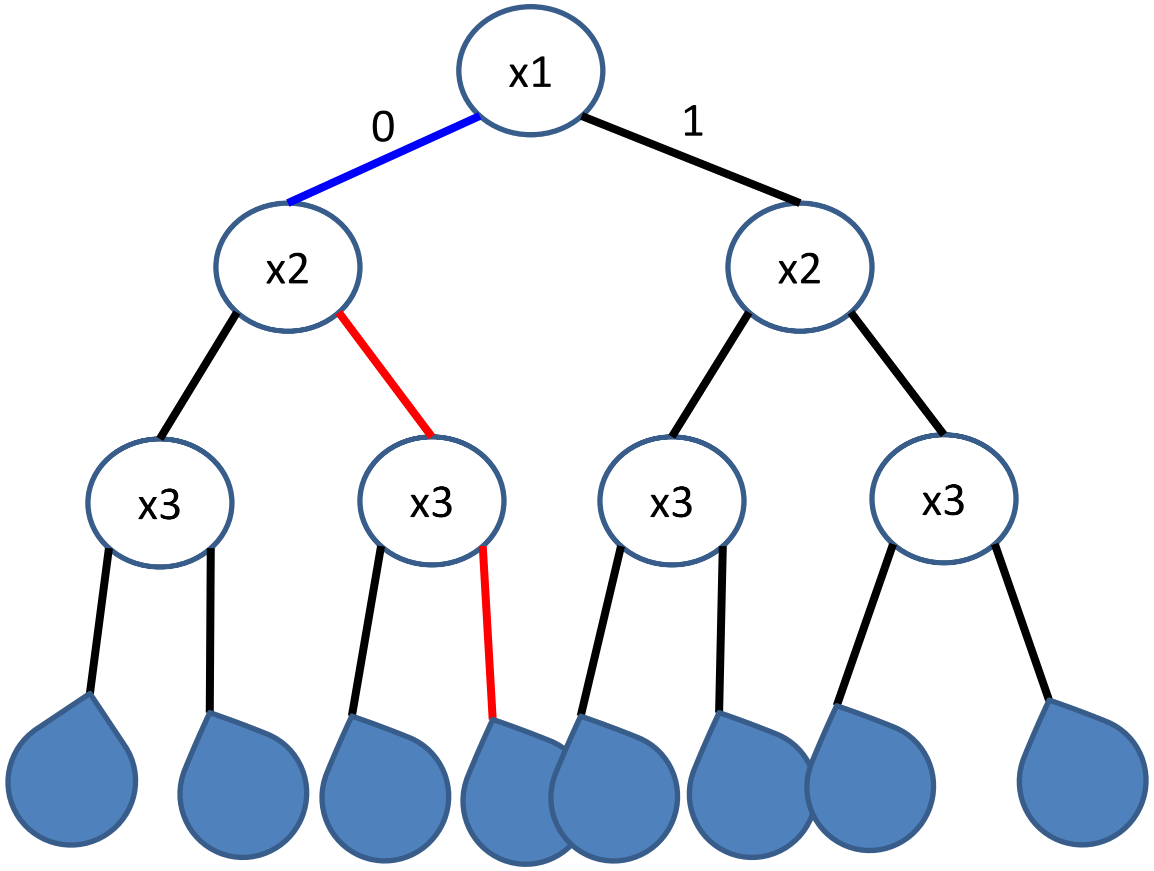
The height of the decision tree must be polynomial in input size

Each node of the tree must be encoded in polynomial number of bits

Each leaf represents a solution (certificate) and can be reached in polynomial number of steps

Class NP: Non-deterministic Polynomial

- **When searching for a satisfying assignment, we allow to use function *choose(b)* which randomly (non-deterministically) chooses the next decision**
- **Once all the selections have been made - the solution can be easily verified**
- **We allow a random “guess”. If we are lucky - we found the satisfying assignment**



$$\phi = (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_3)$$

Vast majority of natural computational problems are in NP

P vs NP

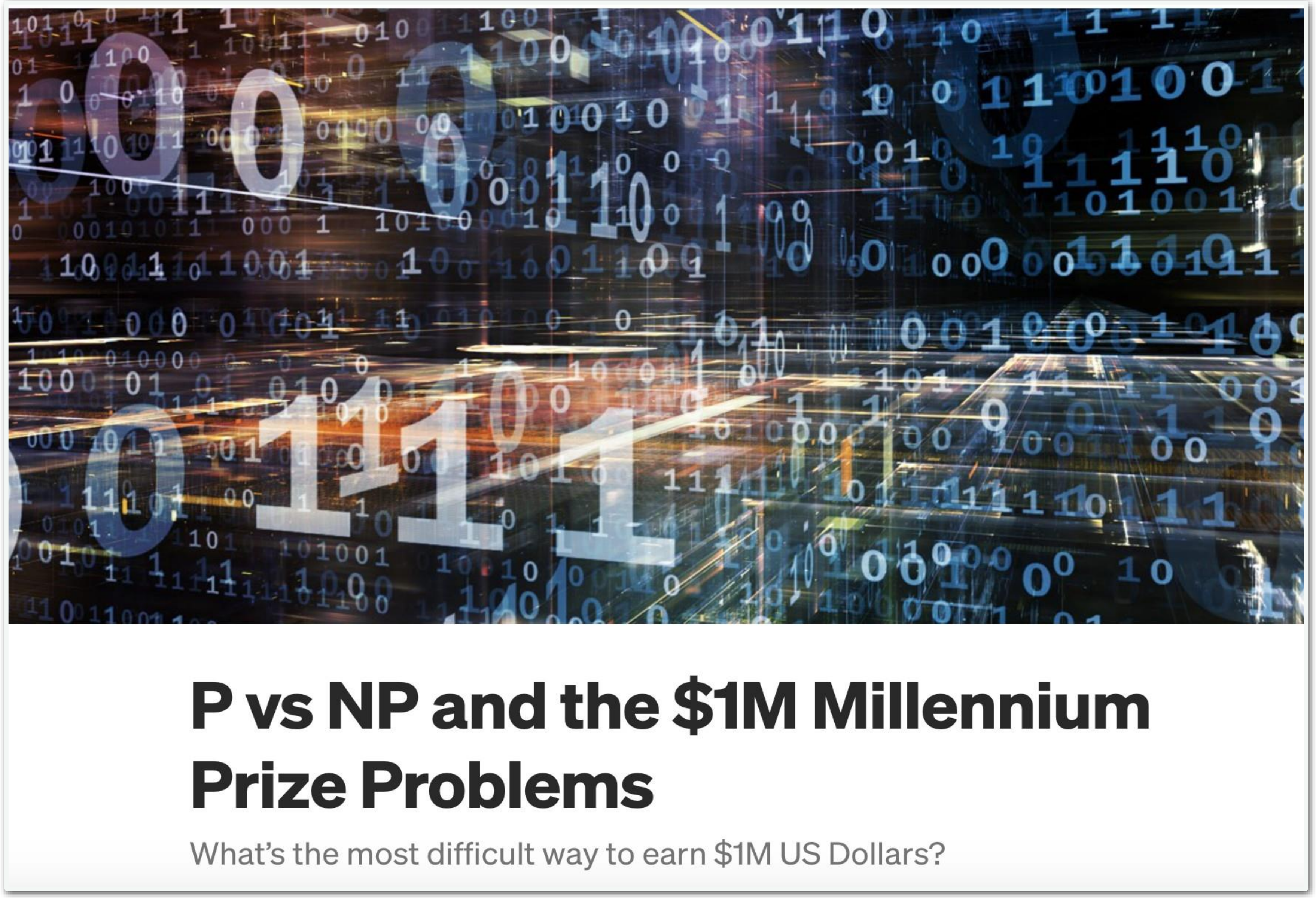
We know that every problem in P is also in NP

What about the reverse?

- If a problem can be efficiently *verified*, does that mean it can be efficiently solved in the first place?
- Or, do there exist problems that can be verified quickly that are provably *impossible* to solve quickly?

The answer: we do not know

Million Dollar Question: P vs NP



P vs NP and the $1M Millennium Prize Problems

What's the most difficult way to earn $1M US Dollars?

https://medium.com/@mpreziuso/

Is P=NP?

Widely believed: P≠ NP
But this has not been proved!

Arguments:

1. **P≠ NP (psychological). Many smart people tried to solve at least one NP-complete problem and never succeeded**
2. **P≠ NP (philosophical). To prove something is much more difficult than to verify somebody else's proof. Verifying in poly-time does not imply that we can solve in poly-time. Can mathematical creativity be automated?**
3. **P=NP (mathematical). There are surprisingly efficient polynomial-time algorithms (i.e. Number of inversions, Matrix multiplication) which seem count-intuitive and difficult to discover. So maybe we just need to try harder?**

NP: name

Non-deterministic Polynomial (Knuth, Terminological Proposal, 1974)

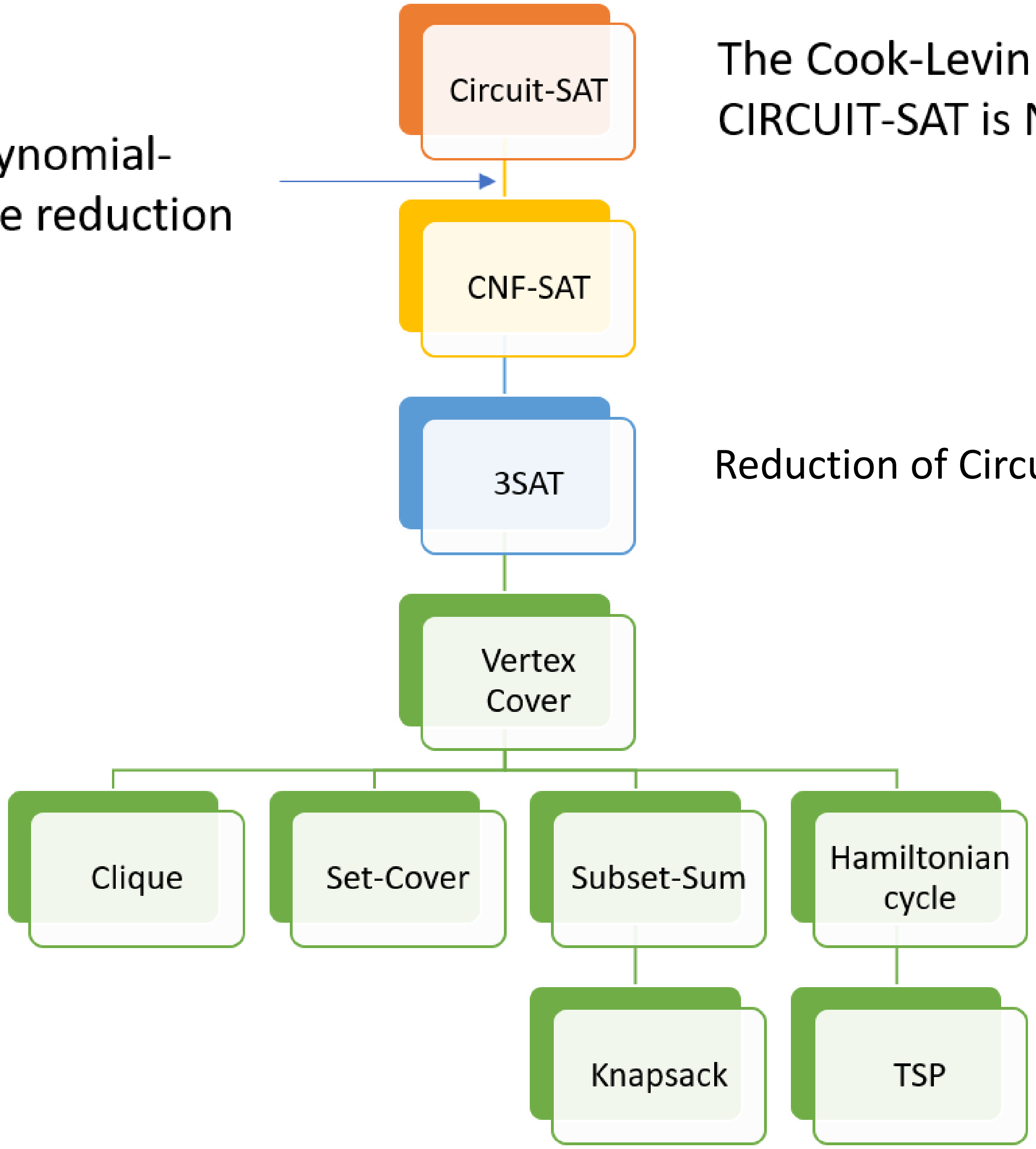
Alternative name:

PET

Possibly Exponential Time (currently)
Provably Exponential Time (if proven that $P \neq NP$)
Previously Exponential Time (if proven that $P = NP$)

23 NP-complete problems

(Karp, 72)



Polynomial-time reduction

The Cook-Levin Theorem:
CIRCUIT-SAT is NP-complete

Reduction of Circuit-SAT to 3-SAT

Some of NP-complete problems discovered by using reductions from Circuit-SAT

Proving NP-completeness

The new problem X is NP-complete if:

1. X is in NP (solution is verifiable in polynomial time)

2. A known NP-complete problem is polynomial-time reducible to X

Example 1. Vertex cover

VERTEX-COVER as a decision problem

Input: undirected graph G(V,E) and an integer k
Output: Yes, if there is a subset C of k vertices such that, for every edge (v,w) of G, $v \in C$ or $w \in C$ (possibly both). No, otherwise.

So if we can show that $A \leq_p X$ AND we know that A is hard (NP-complete), then X must be NP-complete.

As an example, we will prove that Vertex-Cover is NP-complete

1. **Vertex-Cover is in NP**
2. **3-SAT $\leq_p$Vertex-Cover**

1. Vertex-Cover is in NP

Let’s number vertices of G from 1 to N.
If somebody hands us a collection C of k numbers each in interval from 1 to N, we can verify if this is a vertex cover in polynomial time.

For this, we insert all the numbers of C into a dictionary, and then we examine each of the edges in G to make sure that, for each edge (v,w) in G, v is in C or w is in C.

- **If we ever find an edge with neither of its end-vertices in G, then we output “no.”**
- **If we run through all the edges of G so that each has an end-vertex in C, then we output “yes.”**

Such a verification runs in polynomial time O(m) = O(n^2).

Thus, VERTEX-COVER is in NP.

2. Reduction of 3-SAT to Vertex-Cover

We take a general instance of 3-SAT problem
Each 3-SAT instance contains n literals x$_1$, x$_2$, … x$_n$ and m clauses

3 clauses

$$\phi = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_2} \vee \overline{x_3} \vee \overline{x_4})$$

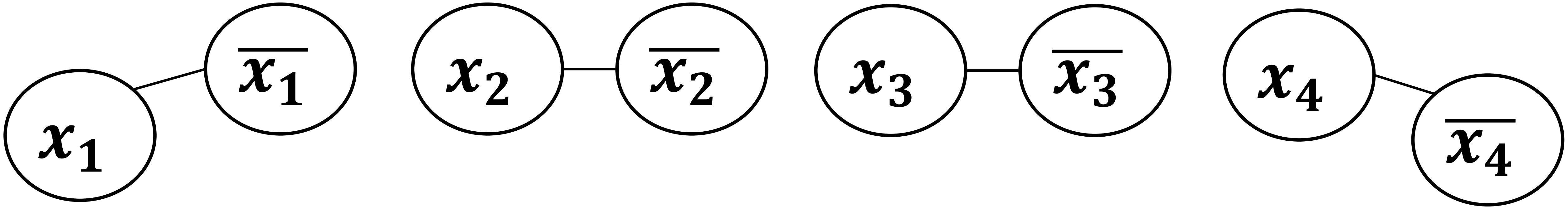
4 literals

We convert the instance into a graph as following:
For each literal i, we create 2 nodes x_i and $\overline{x_i}$ with an edge between them: truth-setting component

For each clause we create 3 nodes connected into a triangle. Each node has an additional edge to the corresponding literal: clause-satisfying component

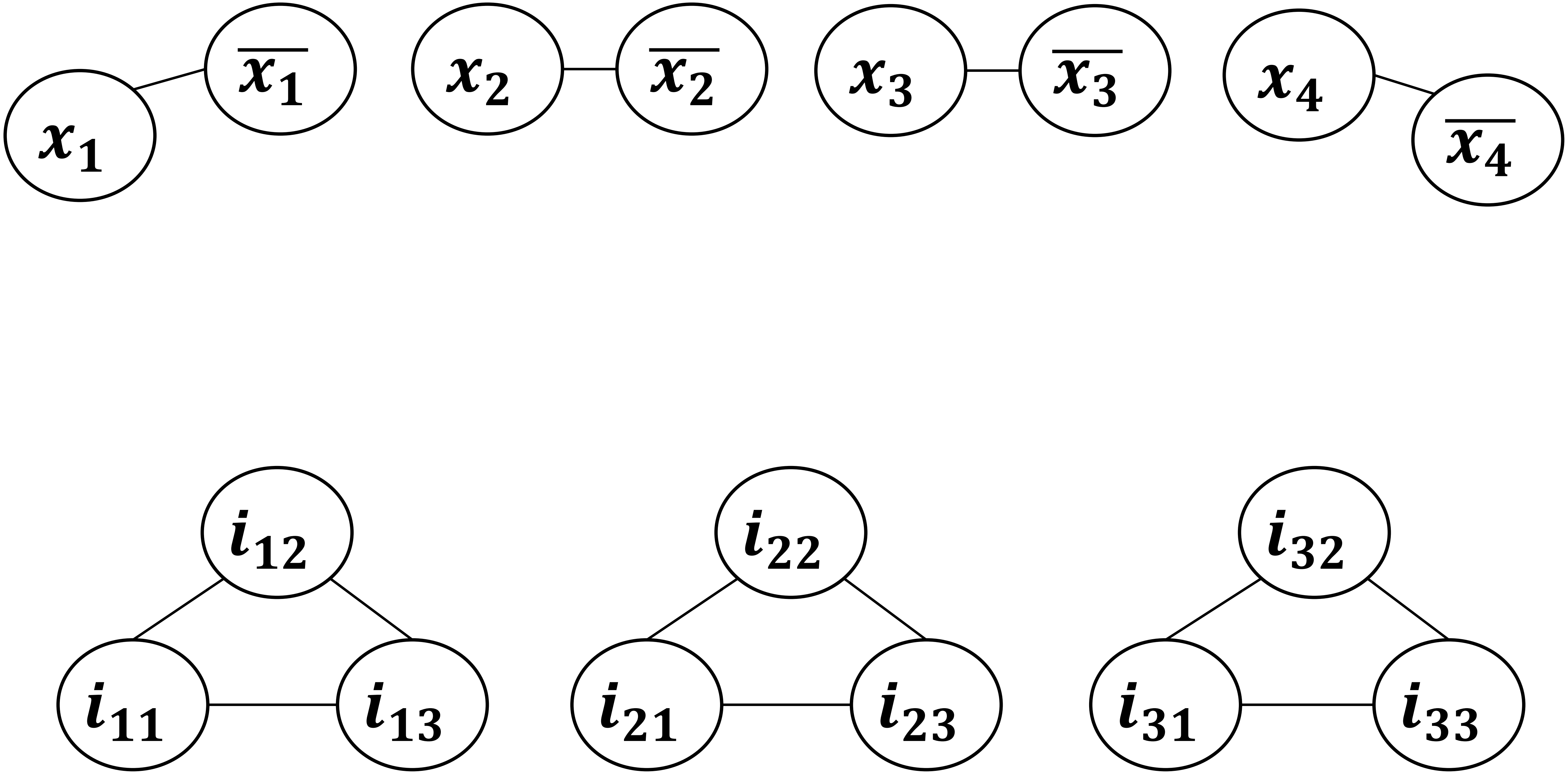
For each literal, create a pair of nodes

$$\phi = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_2} \vee \overline{x_3} \vee \overline{x_4})$$


$\boldsymbol{x_1}$
$\overline{\boldsymbol{x_1}}$
$\boldsymbol{x_2}$
$\overline{\boldsymbol{x_2}}$
$\boldsymbol{x_3}$
$\overline{\boldsymbol{x_3}}$
$\boldsymbol{x_4}$
$\overline{\boldsymbol{x_4}}$

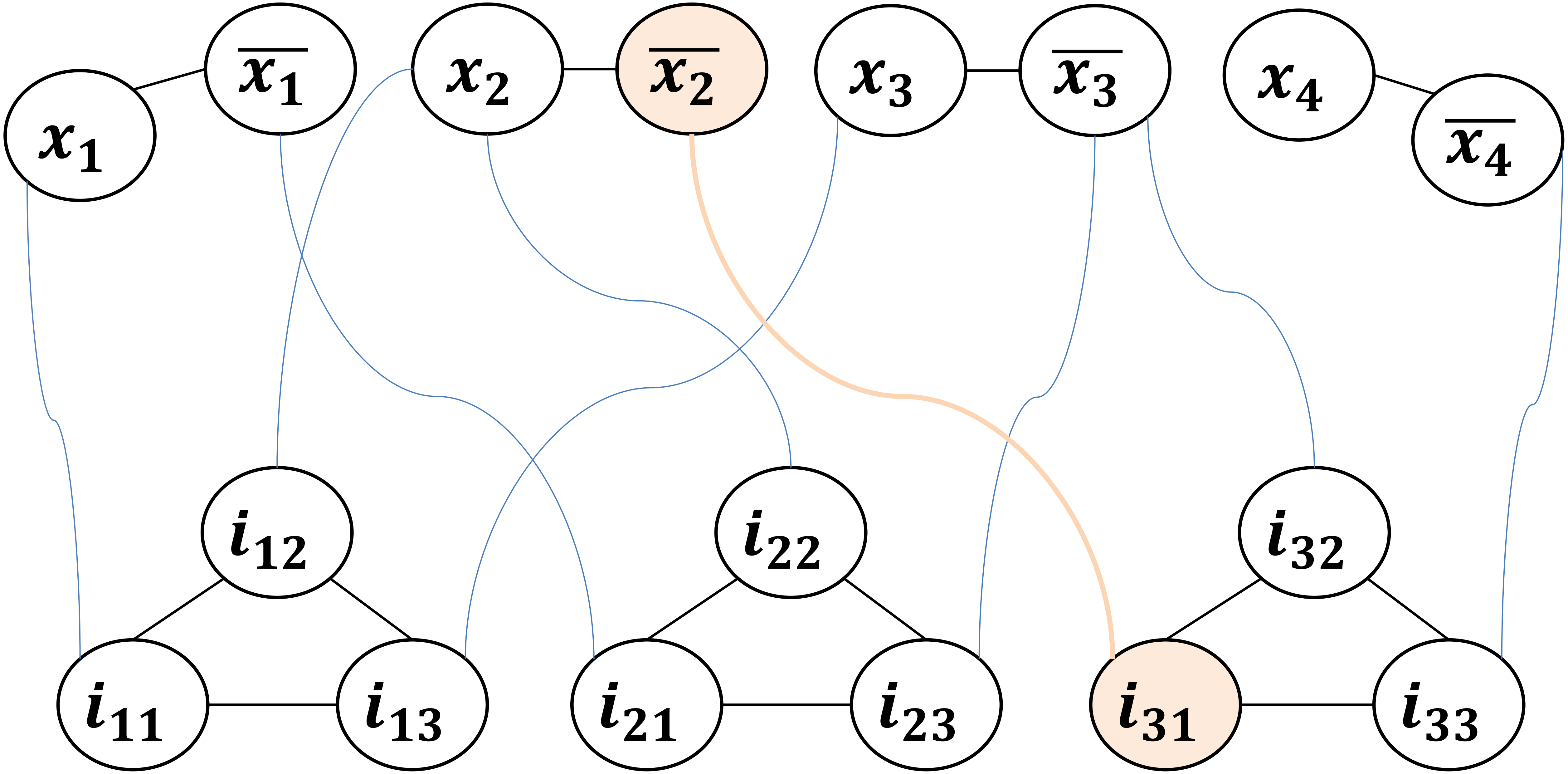
For each clause, create a triangle

$$\phi = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_2} \vee \overline{x_3} \vee \overline{x_4})$$



Connect each variable in the clause to the corresponding literal

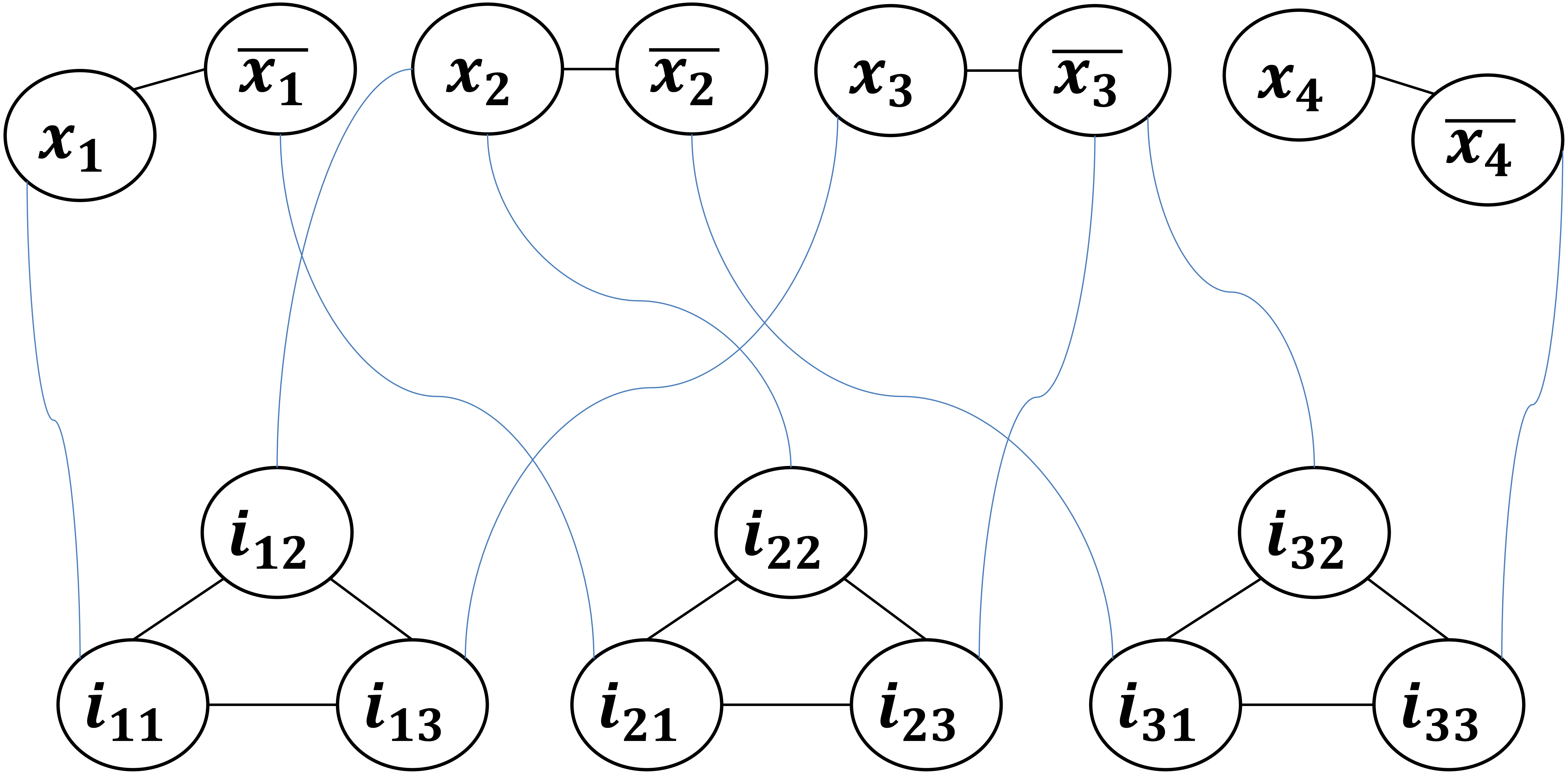
$$\phi = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_2} \vee \overline{x_3} \vee \overline{x_4})$$



This construction clearly runs in polynomial time

Connect each variable in the clause to the corresponding literal

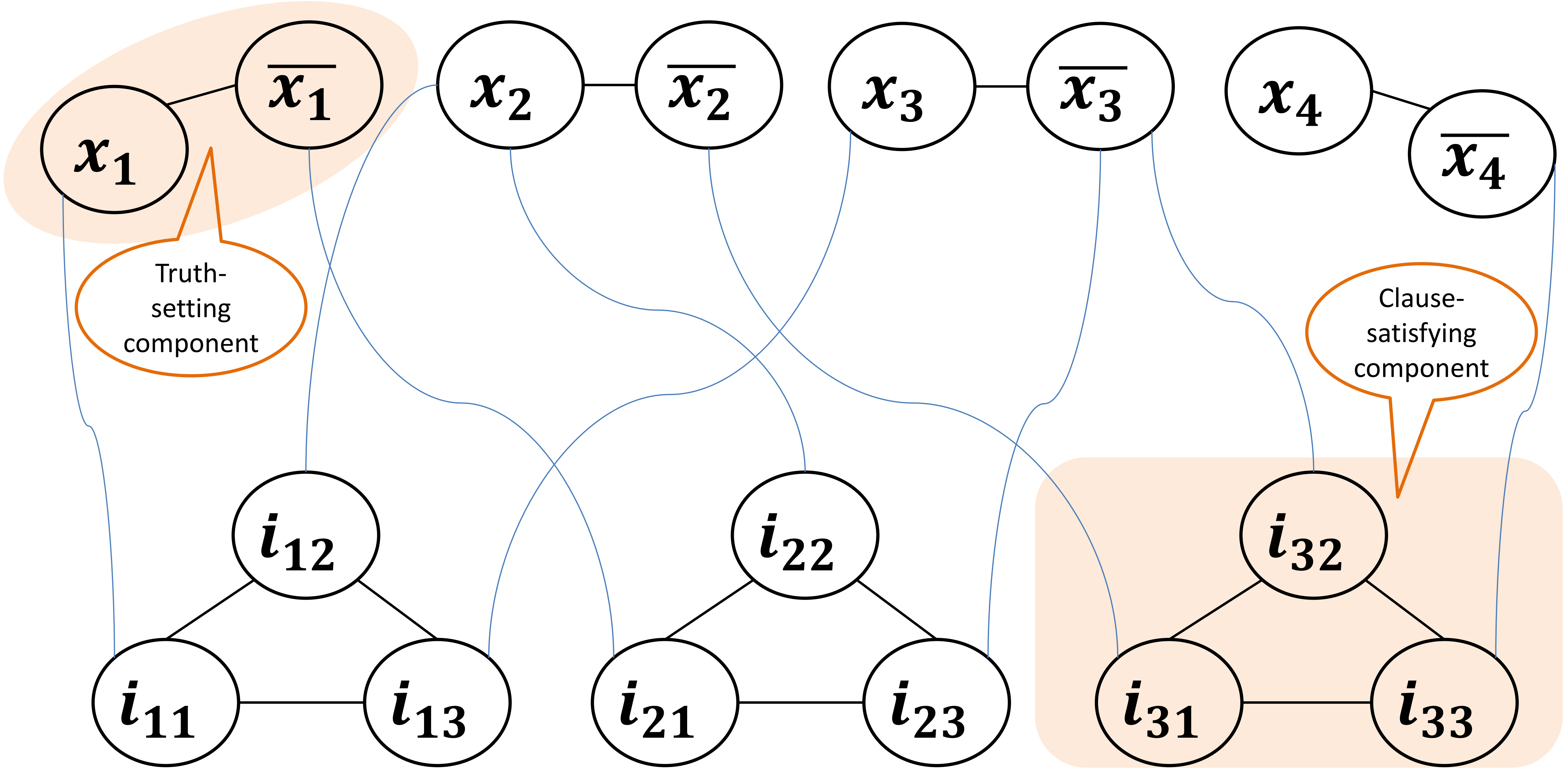
$$\phi = (x_1 \lor x_2 \lor x_3) \land (\overline{x_1} \lor x_2 \lor \overline{x_3}) \land (\overline{x_2} \lor \overline{x_3} \lor \overline{x_4})$$



This construction clearly runs in polynomial time

Connect each variable in the clause to the corresponding literal

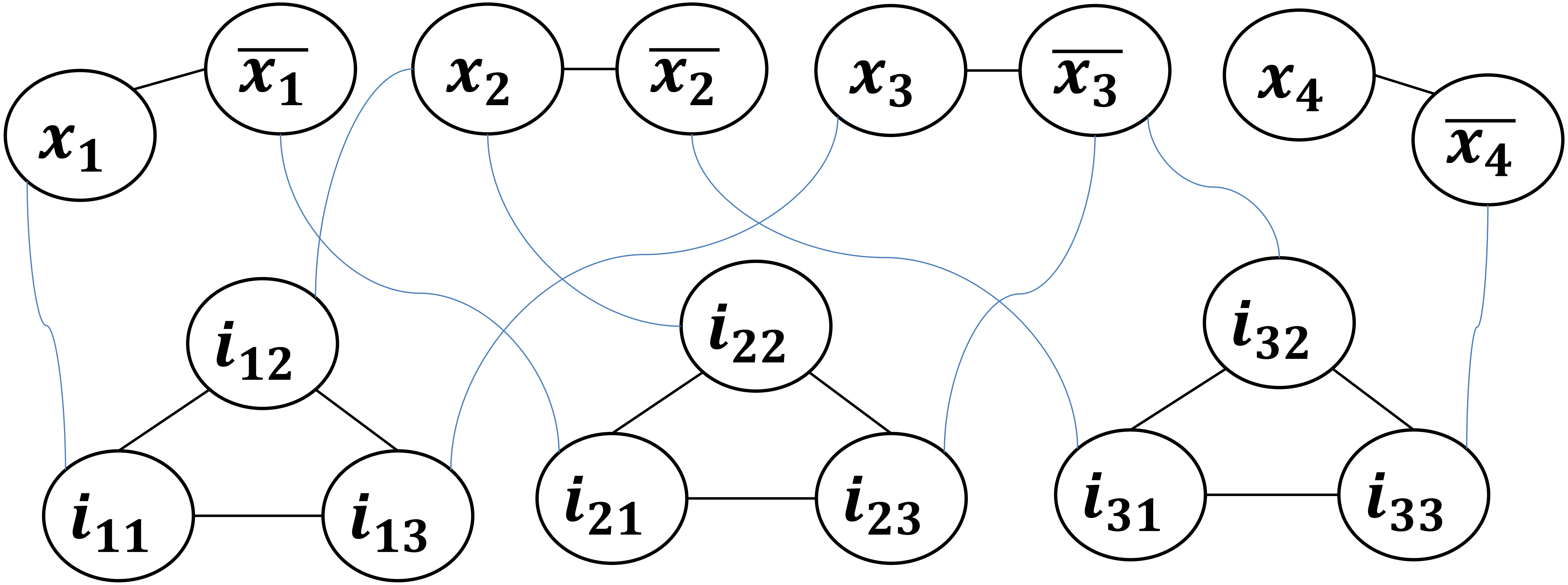
$$\phi = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_2} \vee \overline{x_3} \vee \overline{x_4})$$



Example graph G as an instance of the VERTEX-COVER problem constructed from the formula ϕ

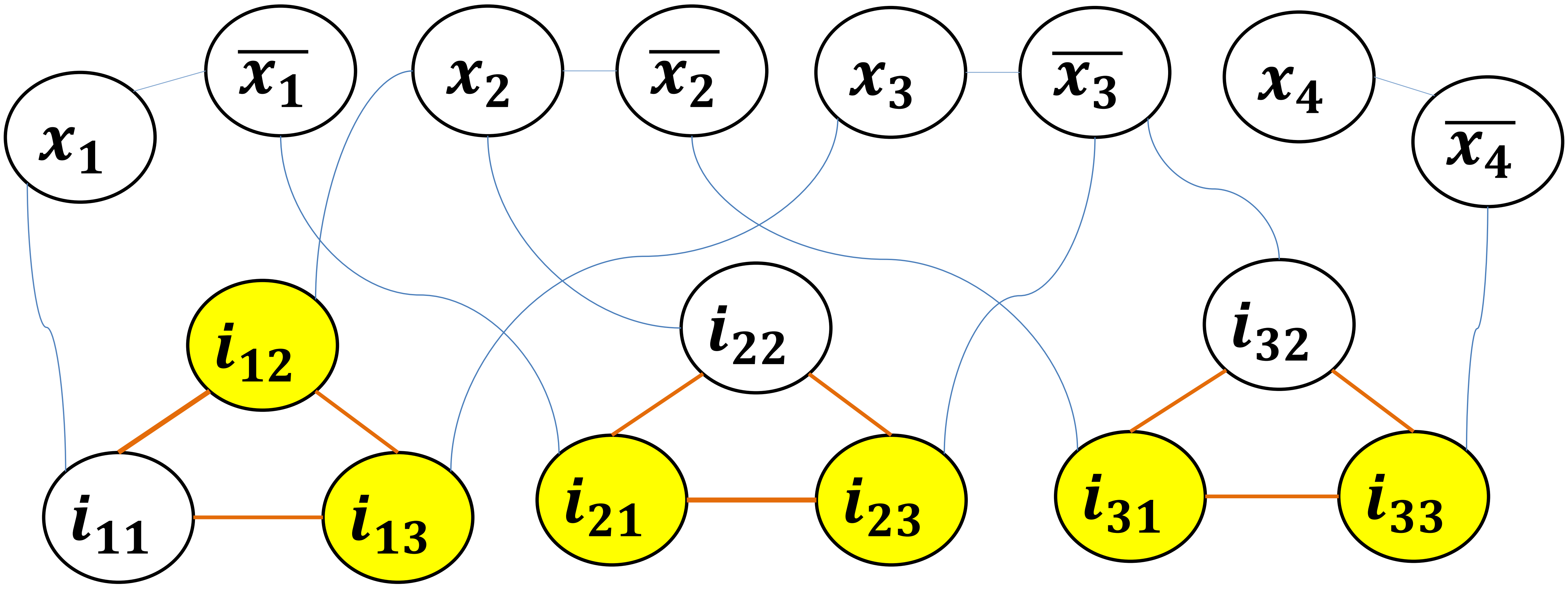
If we can find a vertex cover of size at most k=n + 2m (n-number of literals, m-number of clauses), then this vertex cover represents a truth assignment for 3-SAT problem

$$\phi = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_2} \vee \overline{x_3} \vee \overline{x_4})$$



Proof: 1/4

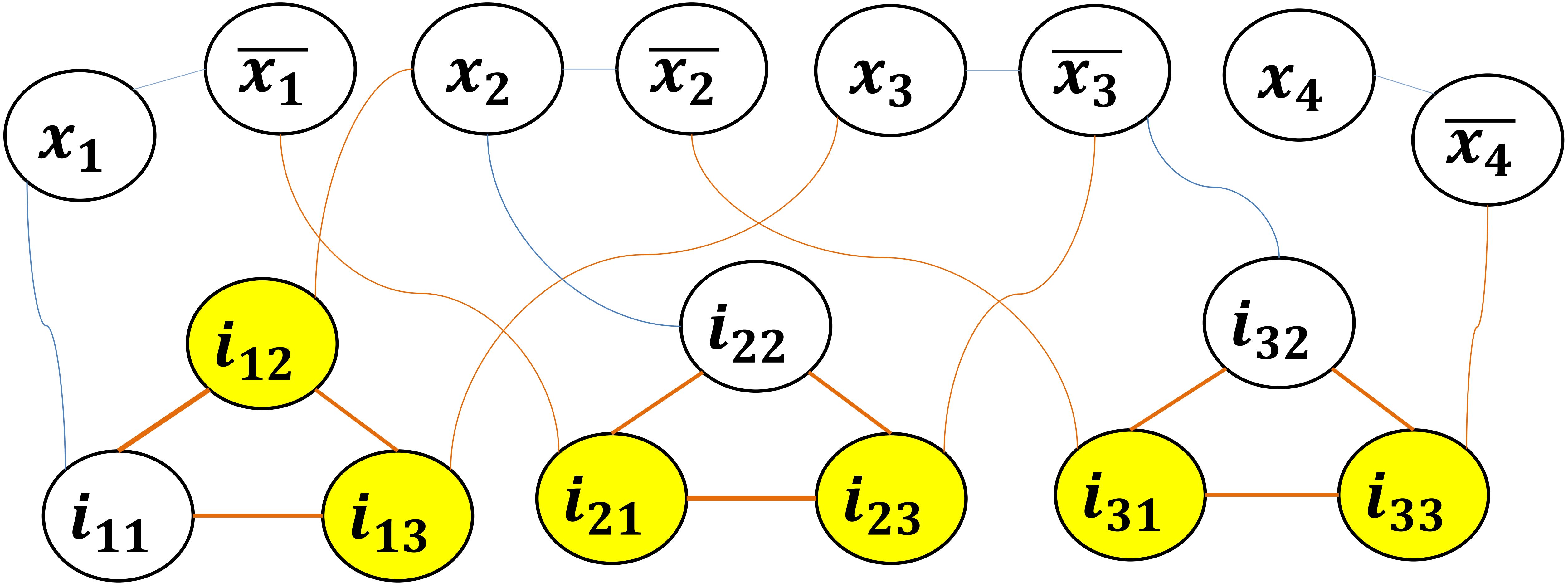
$$\phi = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_2} \vee \overline{x_3} \vee \overline{x_4})$$



The vertex cover must contain two vertices from each clause-satisfying component (to cover all edges of a triangle).

Proof: 2/4

$$\phi = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_2} \vee \overline{x_3} \vee \overline{x_4})$$

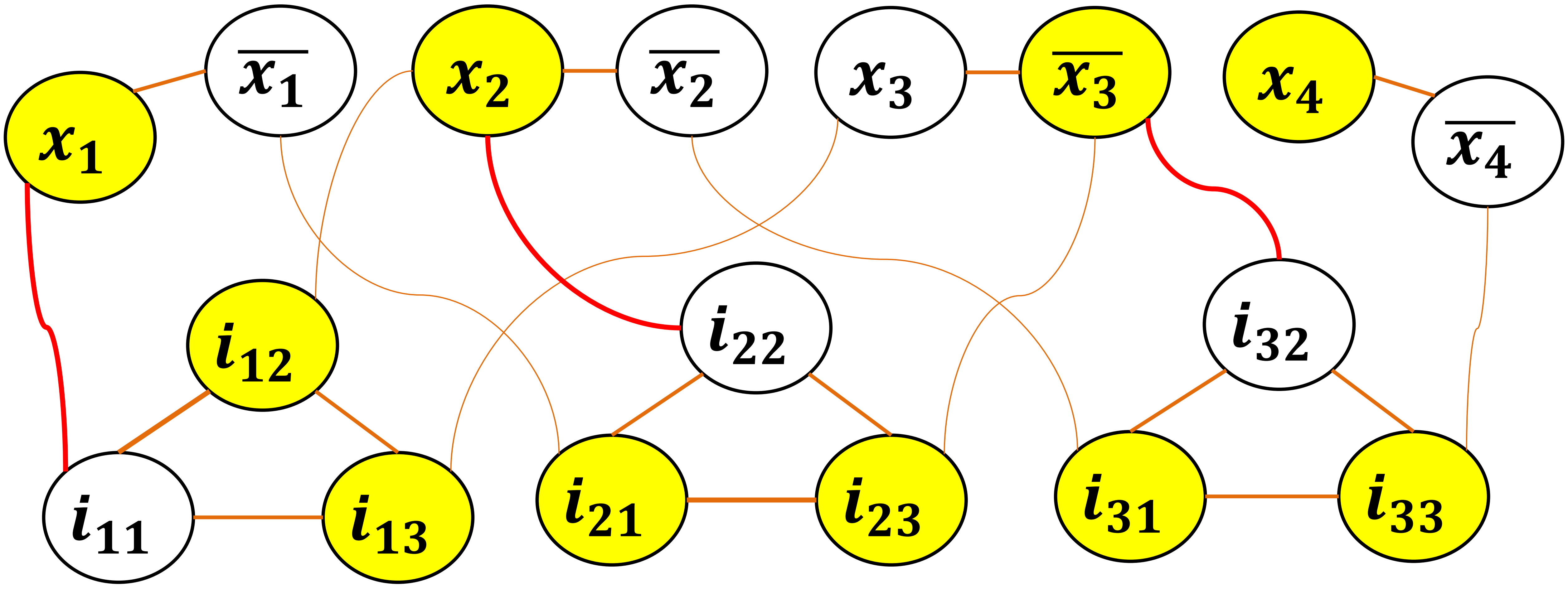


The vertex cover must contain two vertices from each clause-satisfying component (to cover all edges of a triangle).

Now all the outgoing edges from yellow vertices are covered too.

Proof: 3/4

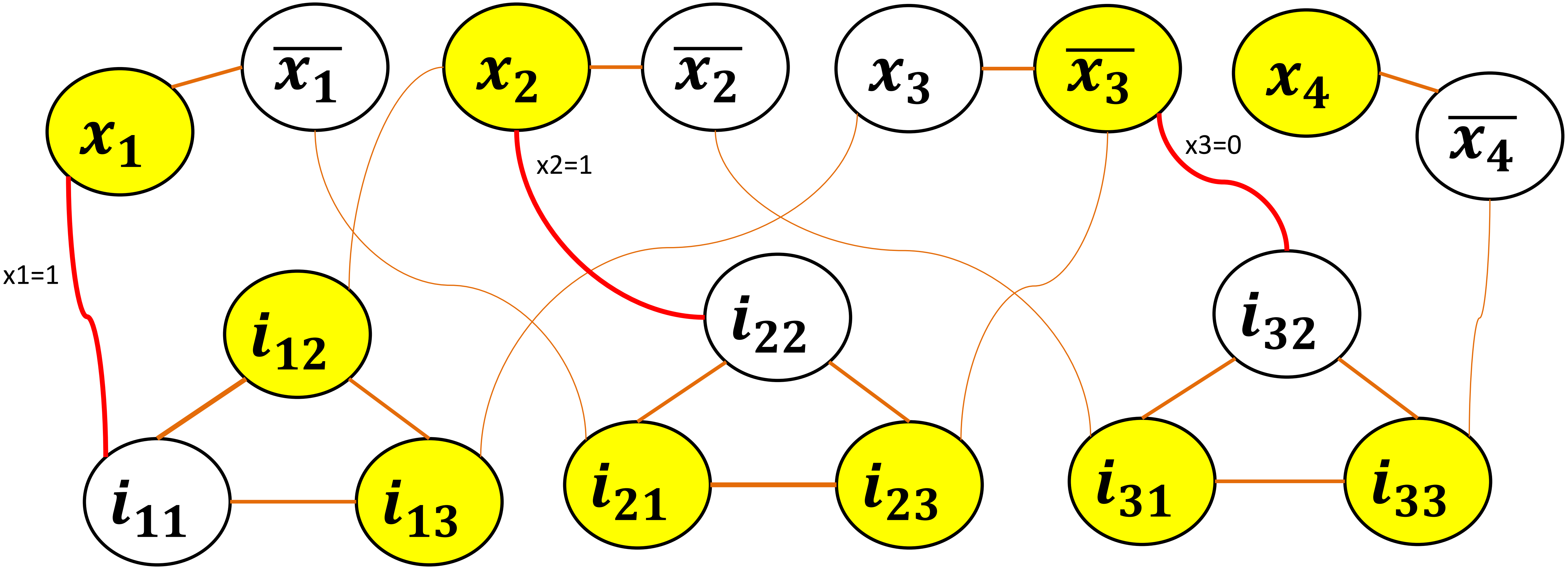
$$\phi = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_2} \vee \overline{x_3} \vee \overline{x_4})$$



This leaves one edge incident to a clause-satisfying component that is not covered by a vertex in the clause-satisfying component (colored red). Hence, each red edge must be covered by the other endpoint, which is labeled with a literal. This literal node will also cover an edge in the Truth-setting component

Proof: 4/4

$$\phi = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_2} \vee \overline{x_3} \vee \overline{x_4})$$



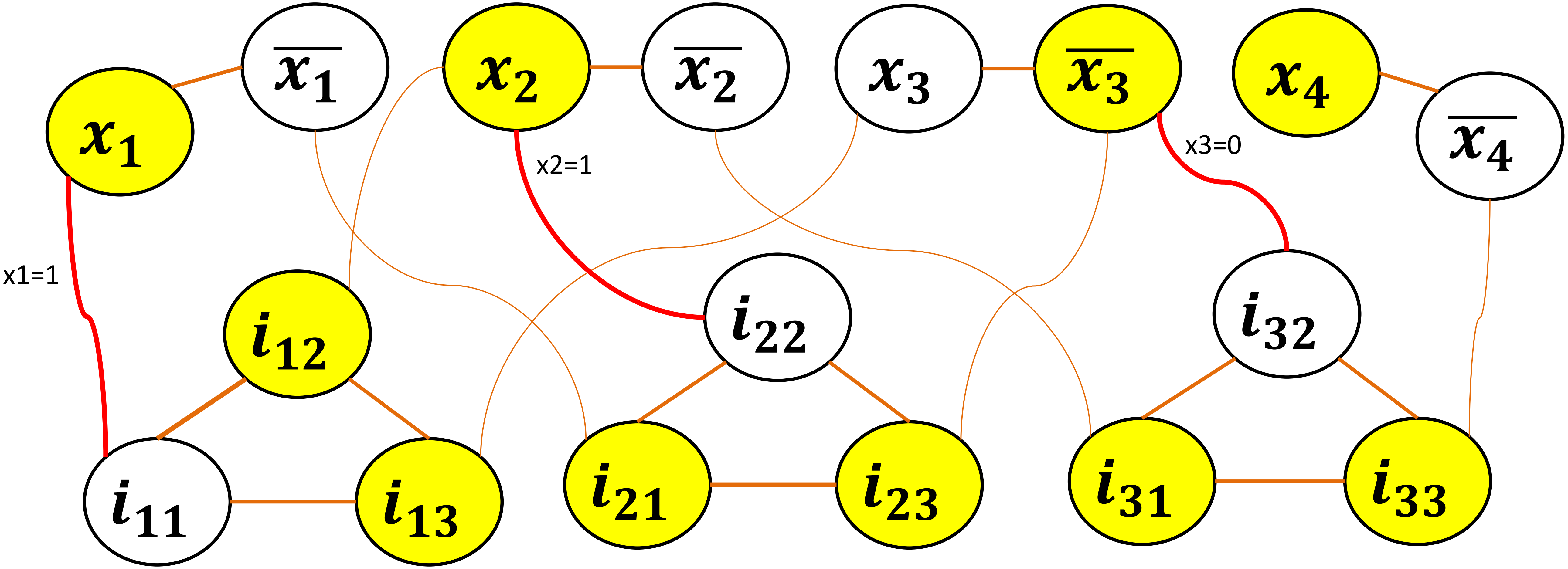
Thus, if we assign the literal in ϕ associated with this node 1, then each clause in ϕ will be satisfied (because each clause is a disjunction, and it is enough that at least one of the literals is True).

Assignment: x1=1, x2=1, x3=0, x4=1.

Hence, all of ϕ becomes satisfied.

Conclusion

$$\phi = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_2} \vee \overline{x_3} \vee \overline{x_4})$$



Thus, if we can find a vertex cover of size at most k=n+2m in this graph, then we can find a set of variables that satisfy the entire 3-SAT formula.

If we knew how to solve vertex-cover, we would be able to solve 3-SAT.

We have shown that:

1. **Vertex-Cover is in NP**
2. **3-SAT $\leq_p$Vertex-Cover**

⇩

VERTEX-COVER is NP-complete

This reduction uses gadgets (components)

Constructing them is a skill which requires a lot of practice.
You will get this practice in the course on the Theory of Computation

Example 2. CLIQUE

CLIQUE as a decision problem

Input: undirected graph G(V,E) and an integer k.
Output: Yes, if there is a subset C of k vertices such that, for every pair of vertices u,v in C, there is an edge (u,v) ∈ E. No, otherwise.

Intuitively, a clique is a subset of vertices that are all connected by a direct edge.

We will prove that CLIQUE is NP-complete

1. **CLIQUE is in NP**
2. **VERTEX-COVER $\leq_p$ CLIQUE**

1. CLIQUE is in NP

Can we check the solution to CLIQUE in polynomial time?

We are given an input graph G (V, E),|V| = n, and an integer k, and we have a proposed solution: subset C.

1. **We check whether the size of C is k**
2. **Then for every pair of vertices u,v in C we check whether edge (u,v) ∈ E.**

The first check can be completed in O(n) steps.
The second check in O(n^2) steps.

Thus the entire verification can be completed in poly-time O(n^2).

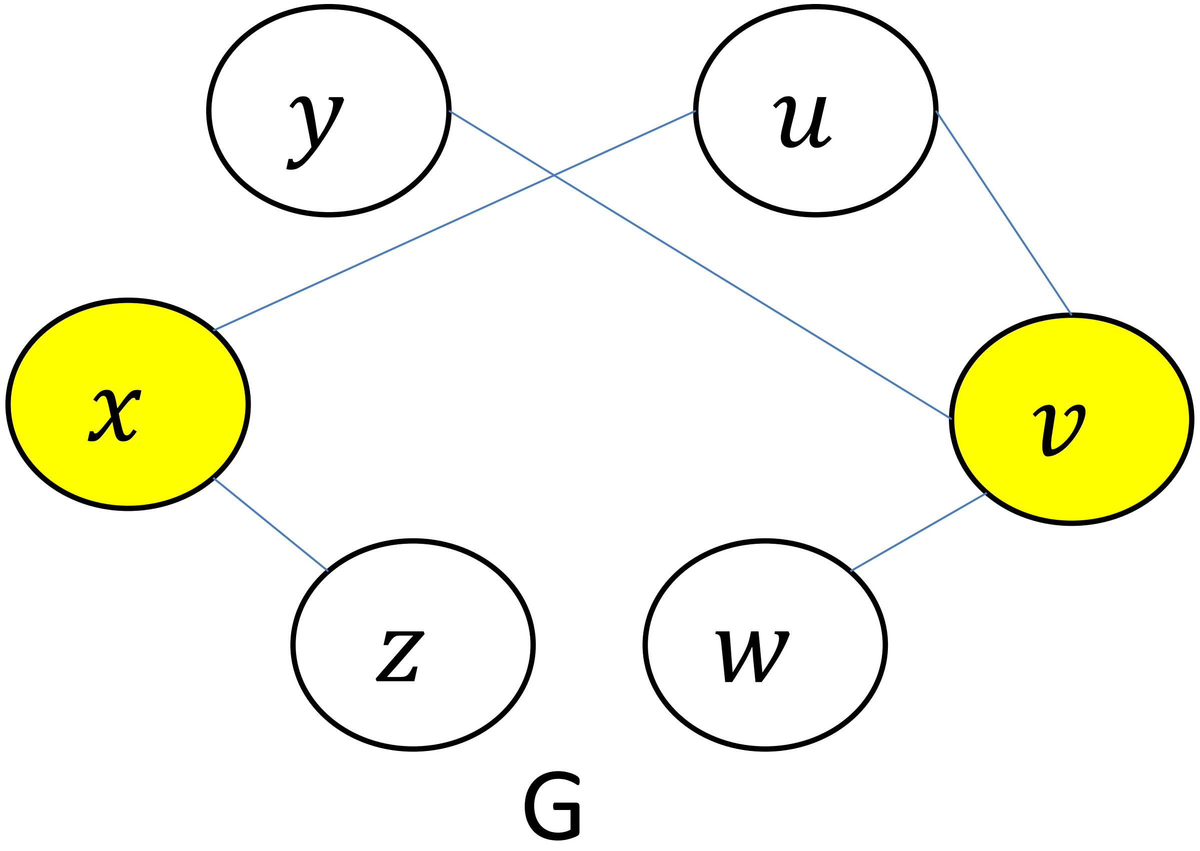
CLIQUE is in NP

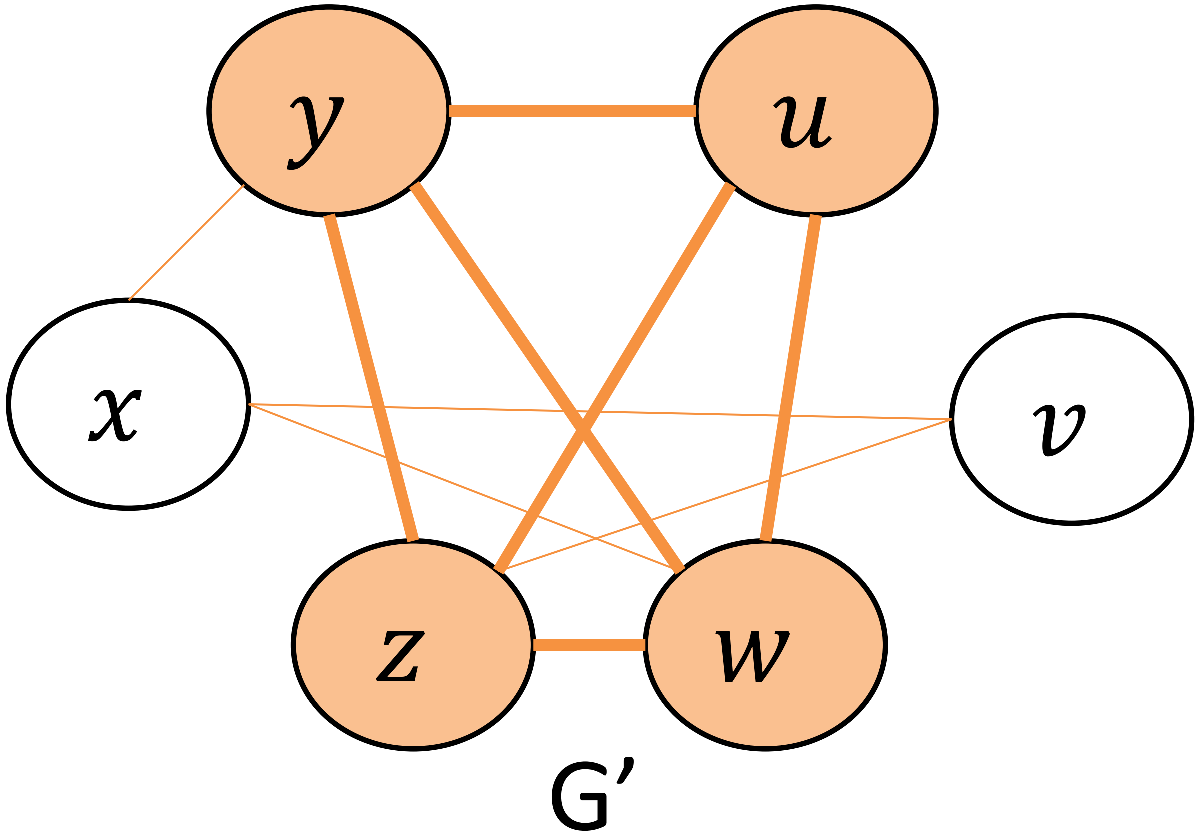
2. Reduction of Vertex-Cover to CLIQUE

Given an undirected graph G (V,E), we define the complement of G as G' = (V,E'), where E' contains all edges (u,v) such that (u,v) ∉ E

Essentially, G' has the same set of vertices as G, none of the edges in G, and all the edges that do not exist in G.

Observation: if there is a Vertex-Cover C of size k in G, then V - C is a Clique of G' of size n - k.

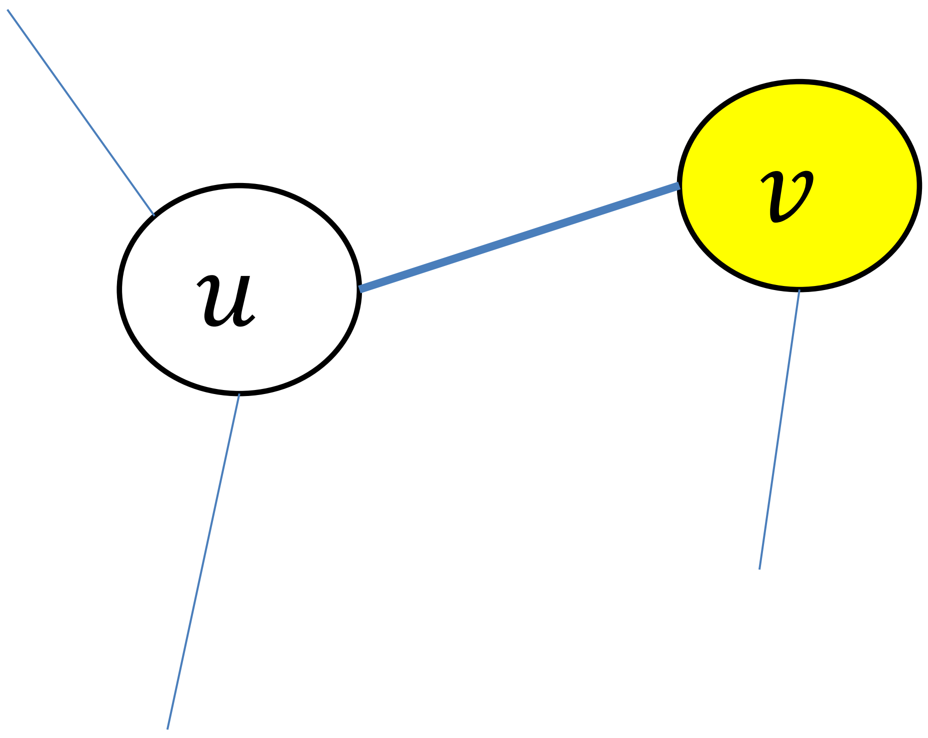




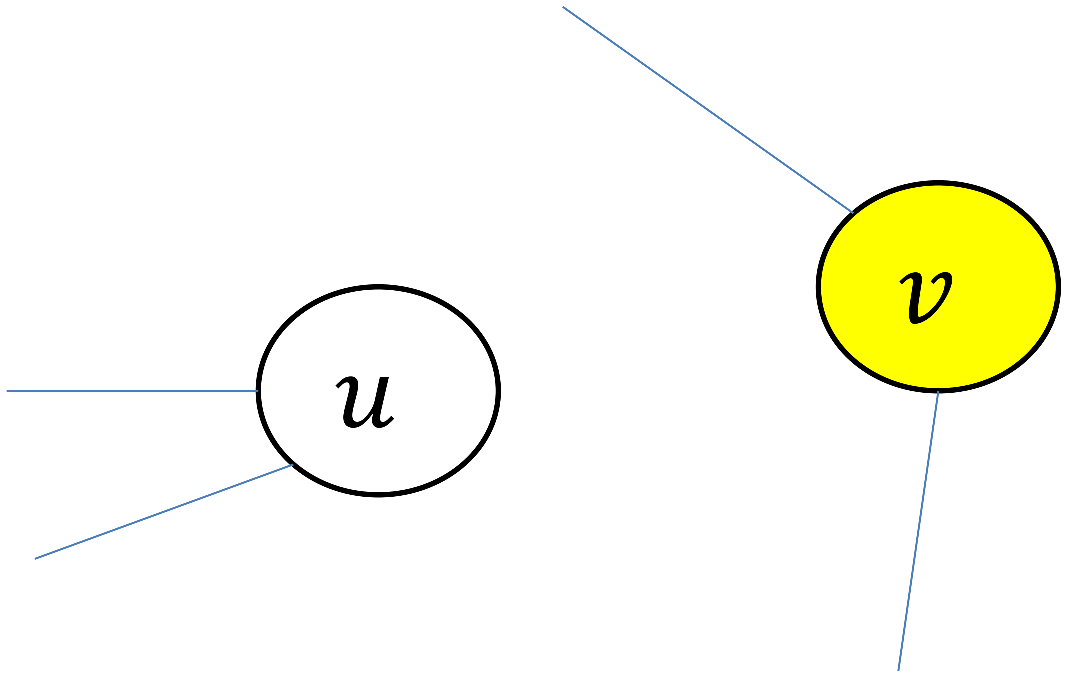
VERTEX-COVER ≤p CLIQUE: Proof 1/3

Observation: if there is a Vertex-Cover C of size k in G, then V - C is a Clique of G' of size n - k.

Let (u,v) be an arbitrary edge in E.
Then, by construction, (u,v) ∉ E', which implies that at least one of v or u does not belong to Clique in G' (both cannot be a part of a clique because there is no edge between them).



G



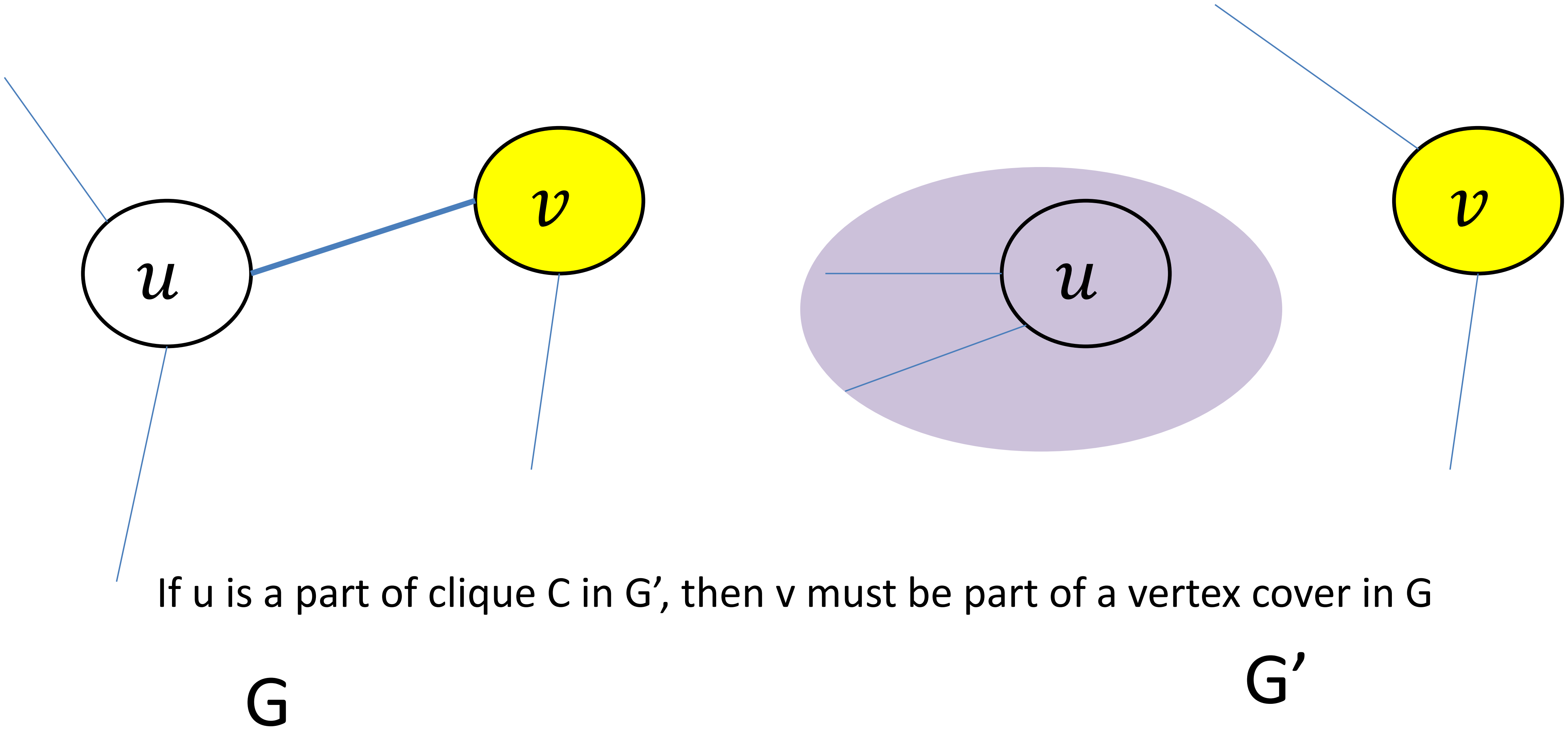
Both u and v cannot be part of a clique – not connected by an edge

G'

VERTEX-COVER ≤p CLIQUE: Proof 2/3

Let’s assume, without lost of generality, that the vertex which does not belong to Clique is vertex v. Then v must belong to a Vertex-Cover of G to cover the edge (u,v) in E.

Since we have chosen the edge (u,v) from E arbitrarily, every such edge must be covered by one of vertices in C.

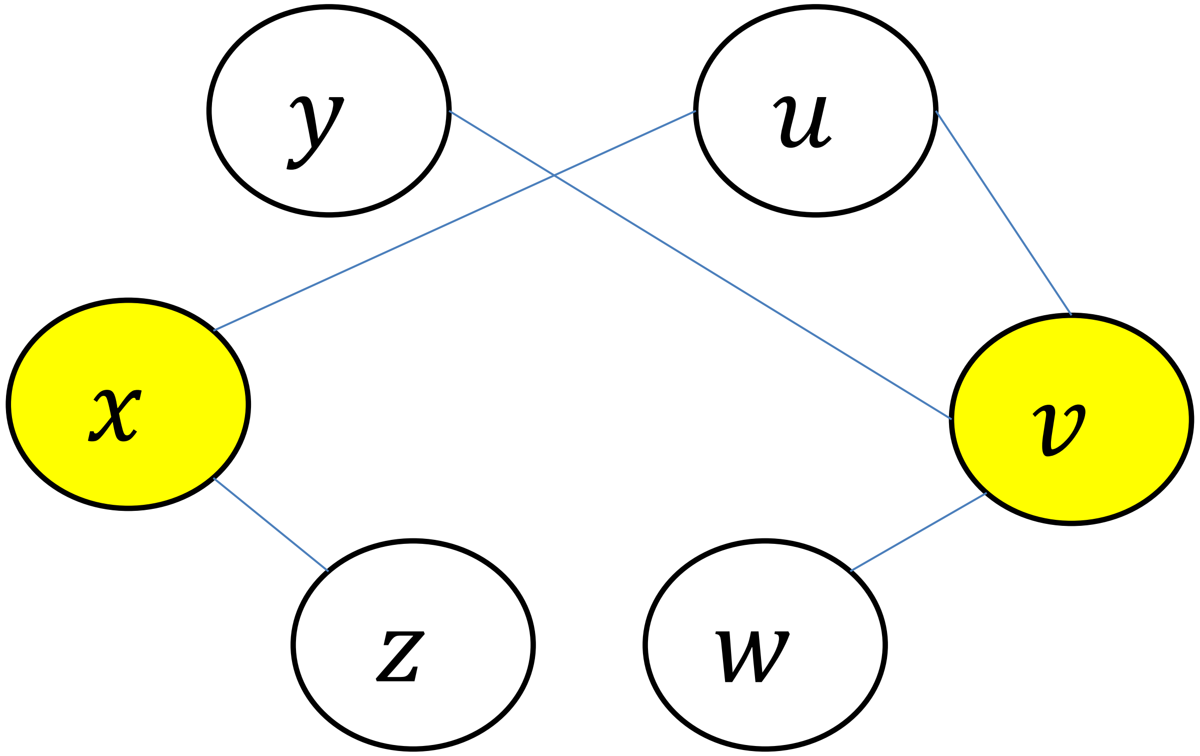


If u is a part of clique C in G’, then v must be part of a vertex cover in G

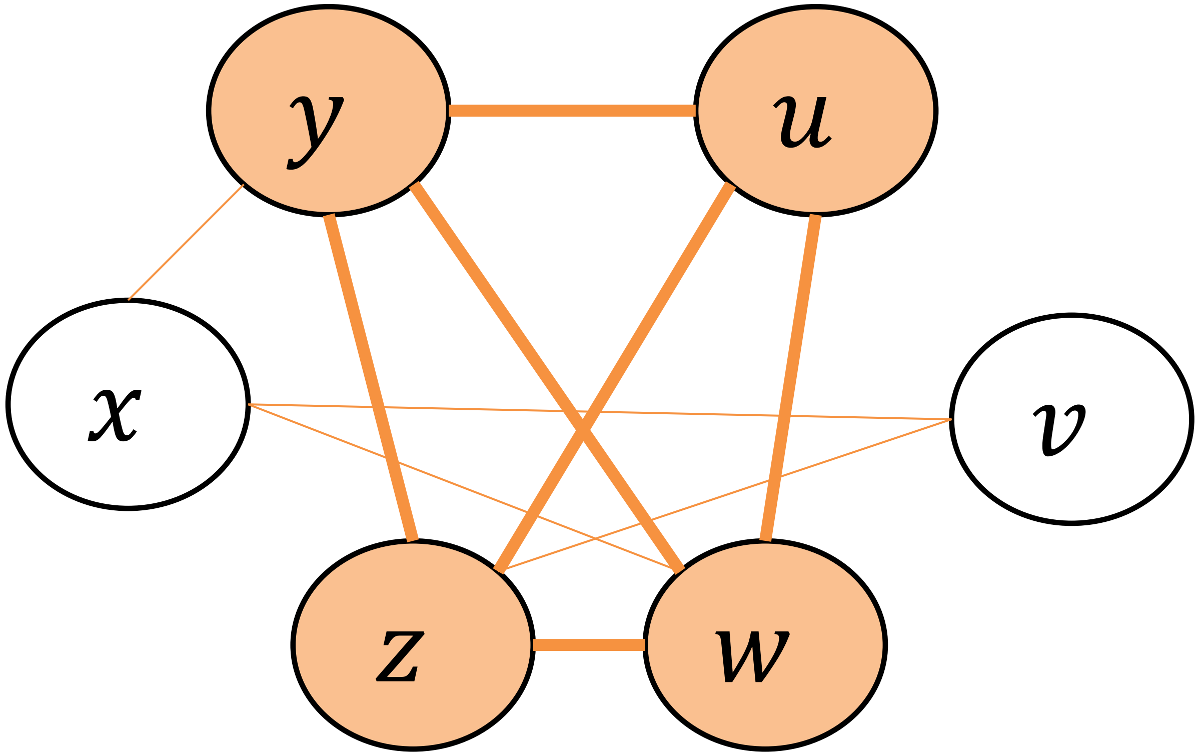
VERTEX-COVER ≤p CLIQUE: Proof 3/3

Therefore, each vertex that not a part of clique in G', must be in a vertex cover of G to cover its adjacent edge.

Hence to obtain an answer whether a given graph G has a VERTEX-COVER of size k, it is enough to answer whether the complement graph G' contains a CLIQUE of size n-k



Vertex–Cover of size 2 in G



Clique of size 4 in G'

We have shown that:

1. **Clique is in NP**
2. **Vertex-Cover $\leq_p$ Clique (the reduction runs in poly-time)**

⇓

CLIQUE is NP-complete

This reduction uses properties of graph complements.

User’s Guide for NP-complete problems

- If you suspect a problem you're looking at is NP-complete, the first step is to look for it in the catalogue of known NP-complete problems.

- If it is not there - find as similar an NP-complete problem as you can, and prove a reduction showing that a similar NP-complete problem is reducible to the one you want to solve.

- If neither of these works, you probably should continue to try to find an efficient algorithm…

Example: Longest-Path

Suppose you want to solve the Longest path problem (unweighted version).

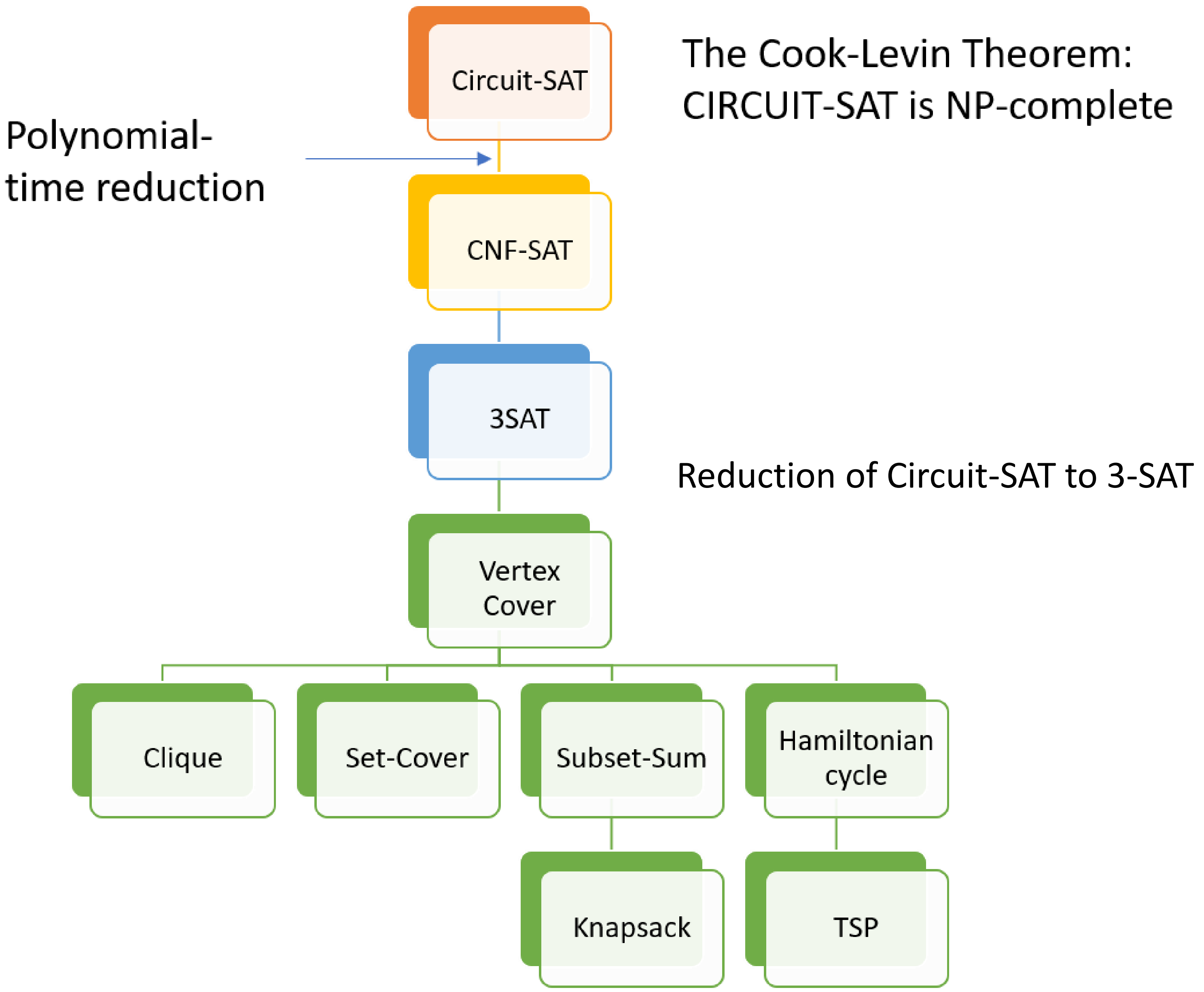
This problem can be formulated as a decision problem. If the path with k edges exists, then check if there is a path with (k+1) edges etc., until you find the max number of edges between vertices s and t.

Longest (unweighted) path between 2 vertices

Input: Undirected graph G(V,E), integer k and two vertices s and t.
Output: Yes, if there is a (simple) path of length k (edges) from s to t.
No, otherwise.

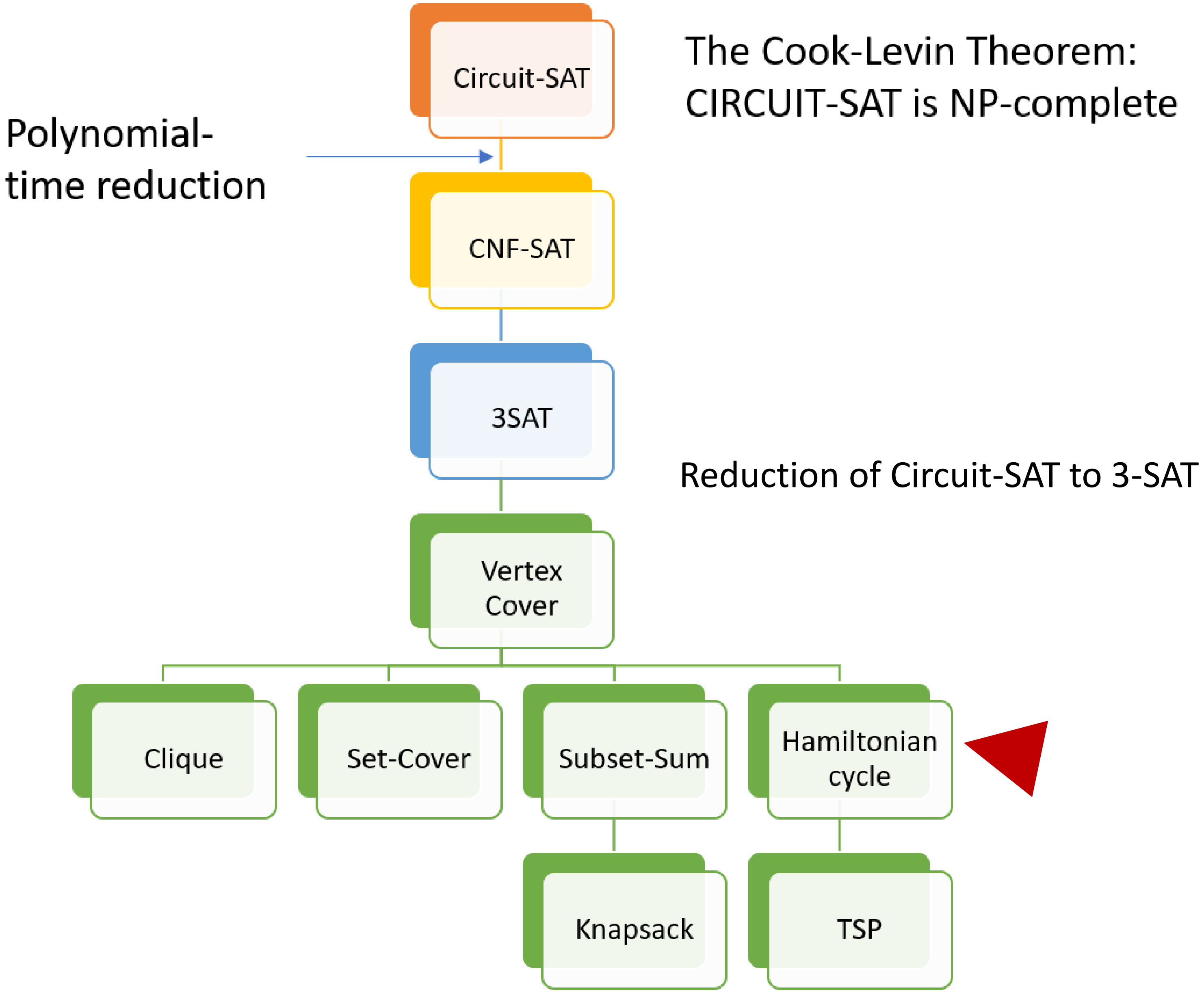
Is this problem NP-complete?

Look at known NP-complete problems



Which problem seems most similar to the Longest-Path?

Look at known NP-complete problems



Which problem seems most similar to the Longest-Path?

Reduce HAMILTONIAN-CYCLE to LONGEST-PATH

Hamiltonian-Cycle problem:

Does a given graph have a cycle visiting each vertex exactly once?

Here's a solution, using longest path as a subroutine:

Algorithm *hamiltonian_cycle* (G)

```
    for each edge (u,v) of G:
        if longest_path (G, k=n-1, u , v):
            return Yes     # path + edge form a cycle
    return No
```

We have shown that if we had a poly-time solution to the Longest-Path problem, then we could solve the Hamiltonian-Path problem with m invocations of this solution (in total polynomial time).

This is however impossible, because we know that Hamiltonian-Path is NP-complete.

Conclusion: Longest-Path must also be NP-complete.

Dealing with NP-complete problems

- Choose a better abstraction. Maybe the real-life problem you are trying to solve can be modeled differently.

  Example: sequence assembly problem which uses Eulerian path instead of a Hamiltonian path

- Solve the problem approximately instead of exactly. A lot of the time it is possible to come up with a provably fast algorithm, that doesn't solve the problem exactly but comes up with a solution you can prove is close to right.

- Use an exponential time solution anyway. If you really have to solve the problem exactly, you can implement an exponential algorithm. In many cases you can design an exponential algorithm which is still better than the Brute-Force.

Sample algorithms for NP-complete problems

- **Solve the problem approximately.**

Example: approximate solution to knapsack problem using greedy and dynamic programming heuristics:
video links (Stanford course):

  - https://youtu.be/FE413JeEBts
  - https://youtu.be/QFZ7E3qgNwM
  - https://youtu.be/KB-ueY1VNTU
  - https://youtu.be/GVrltG08knU
  - https://youtu.be/tOuAvsCvPvg
  - https://youtu.be/5heXe_tMSi8

- **Improve exponential-time solution.**

Example: better algorithm for Vertex Cover:
video links (Stanford course):

  - https://youtu.be/9eLvyM0gTWo
  - https://youtu.be/aj7WT49y-qE
  - https://youtu.be/yy3meMHpk10