Completeness

Announcements

- Problem Set 9 due **Friday, December 9** at 2:15PM.
  - Stop by OH with questions!
  - Email **cs103@cs.stanford.edu** with questions!
- Problem session tonight, 7-8PM in 370-370
  - Alternate time: tomorrow, 7-8PM in Gates 100
  - Covers nondeterminism, NP, and NP-completeness.

Please evaluate this course on Axess.

Your feedback really does make a difference.

Previously on CS103...

The Complexity Class P

- The complexity class P (**polynomial time**) contains all problems that can be solved in polynomial time by a single-tape, deterministic TM.
- Formally:

$$P = \bigcup_{k=0}^{\infty} TIME(n^k)$$

The Complexity Class NP

- The complexity class NP (**nondeterministic polynomial time**) contains all problems that can be solved in polynomial time by a single-tape NTM.
- Formally:

$$NP = \bigcup_{k=0}^{\infty} NTIME(n^k)$$

A Problem in NP

- Does a Sudoku grid have a solution?
  - M = “On input ⟨S⟩, an encoding of a Sudoku puzzle:
    - **Nondeterministically** guess how to fill in all the squares.
    - **Deterministically** check whether the guess is correct.
    - If so, accept; if not, reject.”

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

A Problem in NP

- Does a Sudoku grid have a solution?
  - M = “On input ⟨S⟩, an encoding of a Sudoku puzzle:
    - **Nondeterministically** guess how to fill in all the squares.
    - **Deterministically** check whether the guess is correct.
    - If so, accept; if not, reject.”

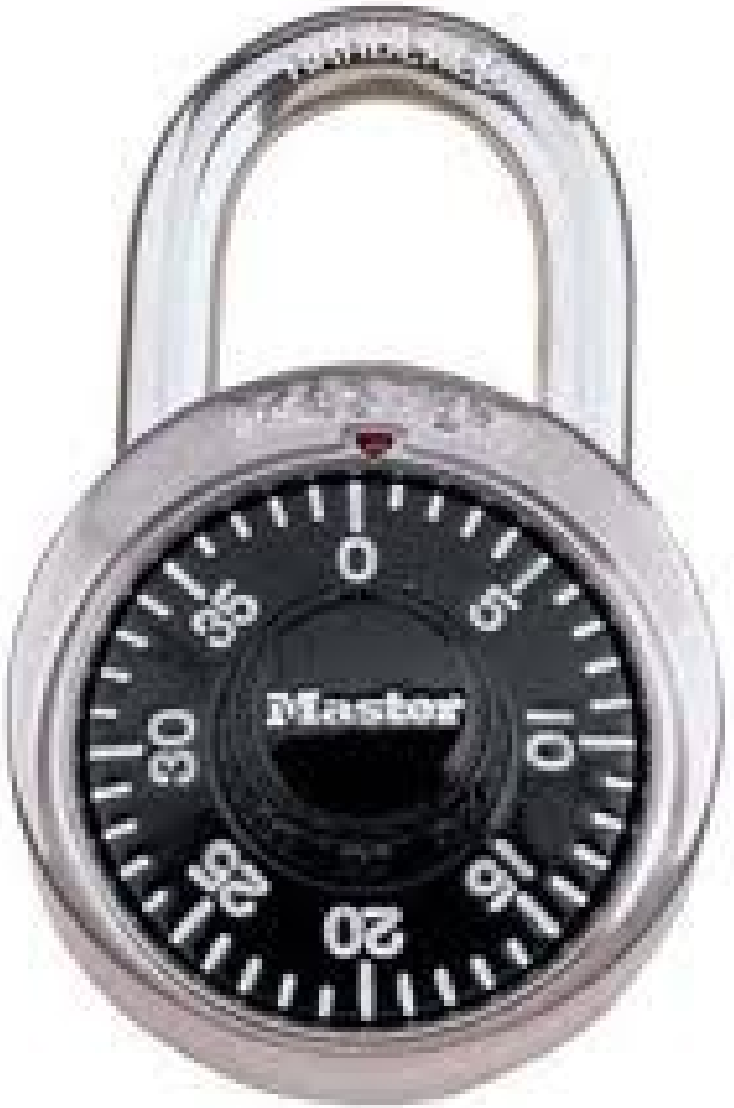
2	5	7	9	6	4	1	8	3
4	9	1	8	7	3	6	5	2
3	8	6	1	2	5	9	4	7
6	4	5	7	3	2	8	1	9
7	1	9	5	4	8	3	2	6
8	3	2	6	1	9	5	7	4
1	6	3	2	5	7	4	9	8
5	7	8	4	9	6	2	3	1
9	2	4	3	8	1	7	6	5

Verifiers

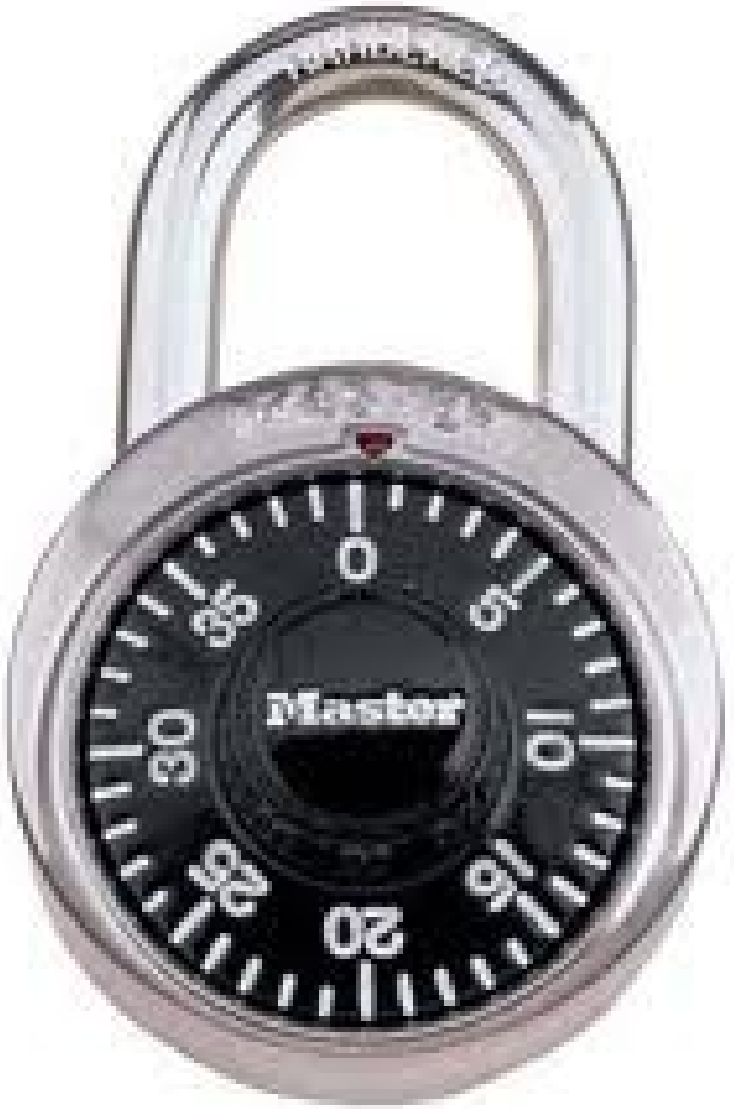
- A **verifier** is a TM V that confirms that a string *w* is contained in some language L.
- The verifier takes as input *w* and some string *x* called the **certificate**, **witness**, or **proof** of *w*.
- Using the information given in *x*, V checks whether *w* is contained in L.

An Efficiently Verifiable Puzzle

An Efficiently Verifiable Puzzle



An Efficiently Verifiable Puzzle



Question: Can this lock be opened?

Verifiers, Formally

- Formally, a **verifier** is a TM V such that

  $w \in L$ iff for some x, V accepts $\langle w, x \rangle$

- In other words

  $L = \{ w \mid \exists x.$ V accepts $\langle w, x \rangle \}$

- If $w \in L$, the verifier can check this easily if we know the proper x.
- If $w \notin L$, the verifier does not help much.
  - Just because V rejects $\langle w, x \rangle$ does not mean that $w \notin L$.

Verification is Powerful

- Many undecidable languages can still be verified.
- Here is a verifier for *HALT*:
  - V = “On input ⟨M, *w*, n⟩, where M is a TM, *w* is a string, and n is a natural number:
    - Run M on *w* for n steps.
    - If M halts *w* within that time , accept; otherwise reject.”
- V always halts on all inputs (even if M loops on *w*).
- If M halts on *w*, there is some choice of n for which H accepts (namely, the number of steps M takes before it halts on *w*).
- Thus *HALT* can be **verified** but not **decided**.

Polynomial-Time Verifiers

- A **polynomial-time verifier** is a verifier V whose runtime on $\langle w, x \rangle$ is a polynomial in $|w|$.
- Since the runtime is a polynomial in $|w|$, V can only read polynomially many characters of x.
- If there is **any** certificate x for w, then there is a certificate whose size is a polynomial in $|w|$.

A Problem in NP

- Does a Sudoku grid have a solution?
  - V = “On input ⟨S, A⟩, where S is a Sudoku grid and A is a proposed solution to the grid:
    - **Deterministically** check whether A is a legal solution to S.
    - If so, accept; if not, reject.”

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

A Problem in NP

- Does a Sudoku grid have a solution?
  - V = “On input ⟨S, A⟩, where S is a Sudoku grid and A is a proposed solution to the grid:
    - **Deterministically** check whether A is a legal solution to S.
    - If so, accept; if not, reject.”

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

A Problem in NP

- Does a Sudoku grid have a solution?
  - V = “On input ⟨S, A⟩, where S is a Sudoku grid and A is a proposed solution to the grid:
    - **Deterministically** check whether A is a legal solution to S.
    - If so, accept; if not, reject.”

In an m × m grid, there are O(m) rows, columns, and squares to check.

Each can be checked in polynomial time (only m elements in each need to be checked)

Collectively, a solution can be verified in polynomial time.

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

The Verifier Definition of NP

- **Theorem**: L ∈ NP iff there is a polynomial-time verifier for L.
  - See proof in Sipser for details; it's really interesting!
- P is the class of languages that can be **solved** efficiently.
- NP is the class of languages that can be **verified** efficiently.

P vs. NP

- The question of whether P = NP is the most important question in all of theoretical computer science.
- With the verifier definition of NP, one way of phrasing this question is

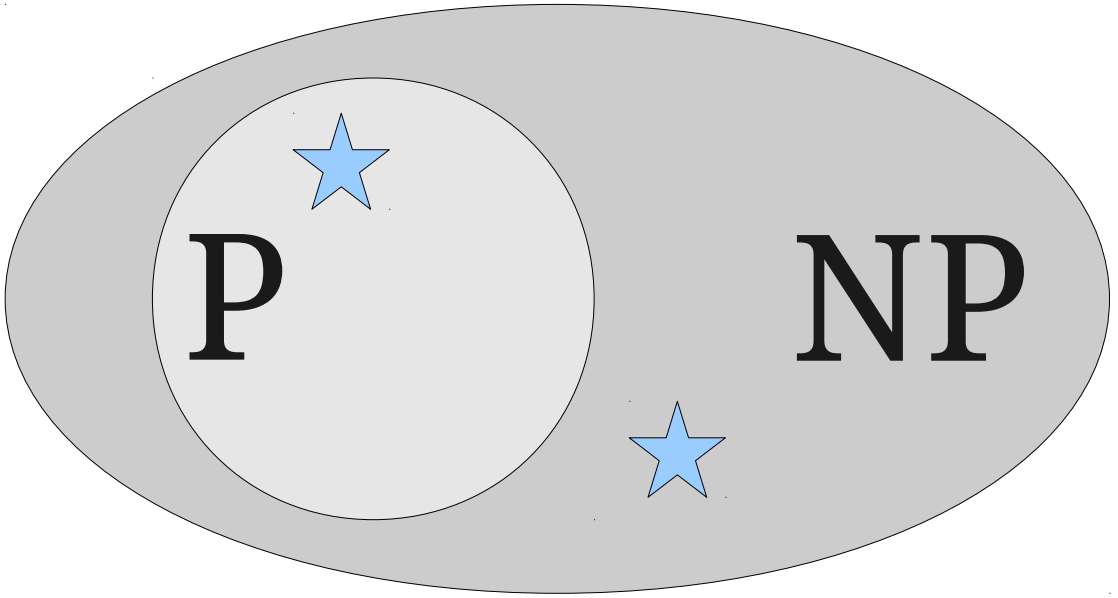
If a problem can be **verified** efficiently,
can it be **solved** efficiently?

- An answer either way will give fundamental insights into the nature of computation.

NP-Completeness

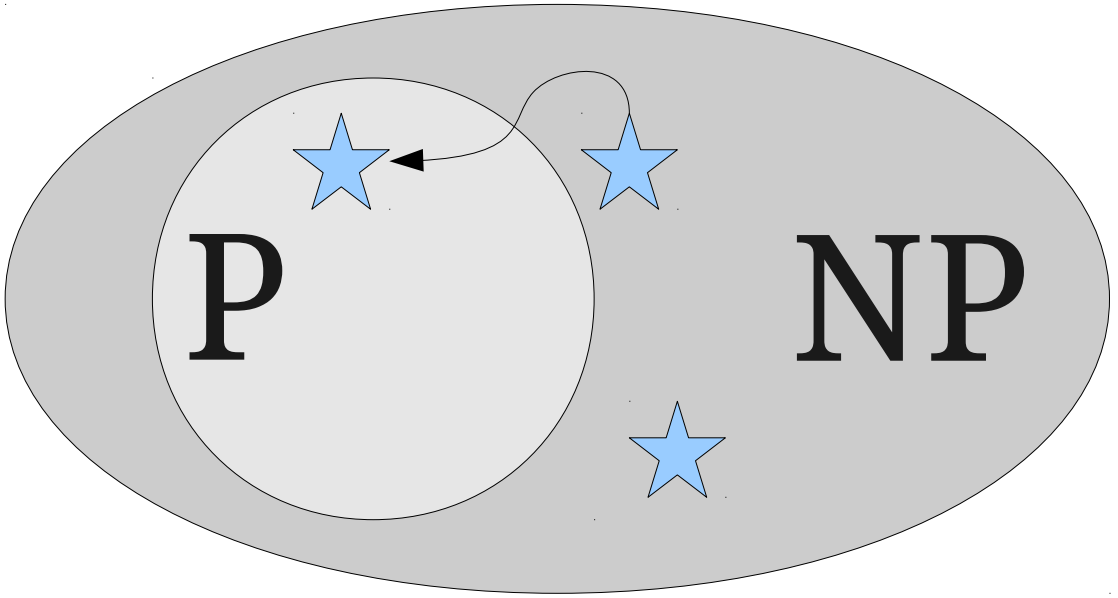
Polynomial-Time Reductions

- If a language L' is polynomial-time reducible to a problem L and L ∈ P, then L' ∈ P.



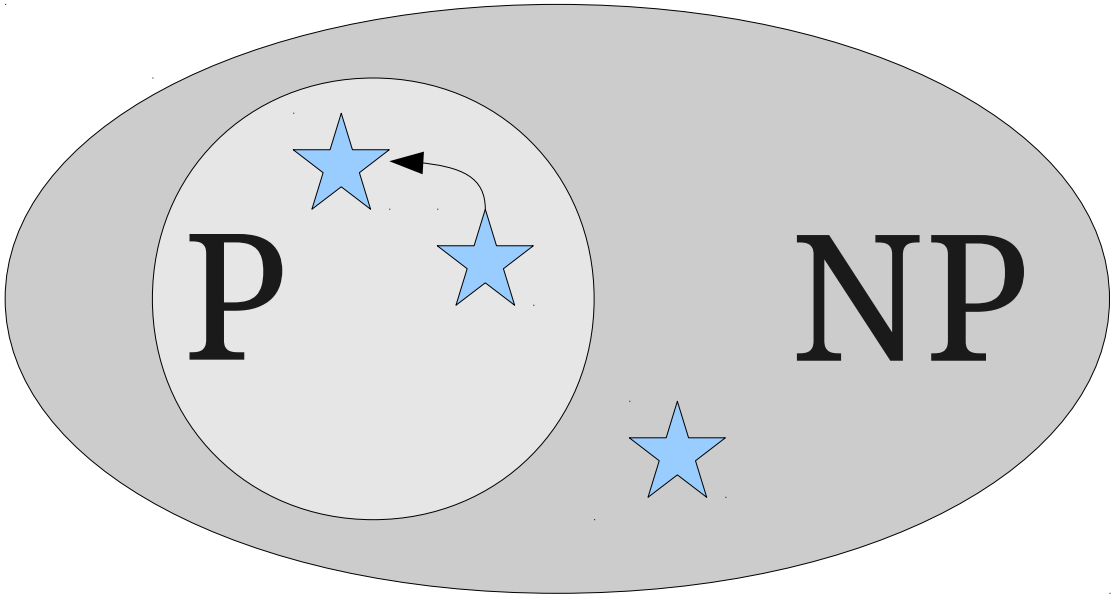
Polynomial-Time Reductions

- If a language L' is polynomial-time reducible to a problem L and L ∈ P, then L' ∈ P.



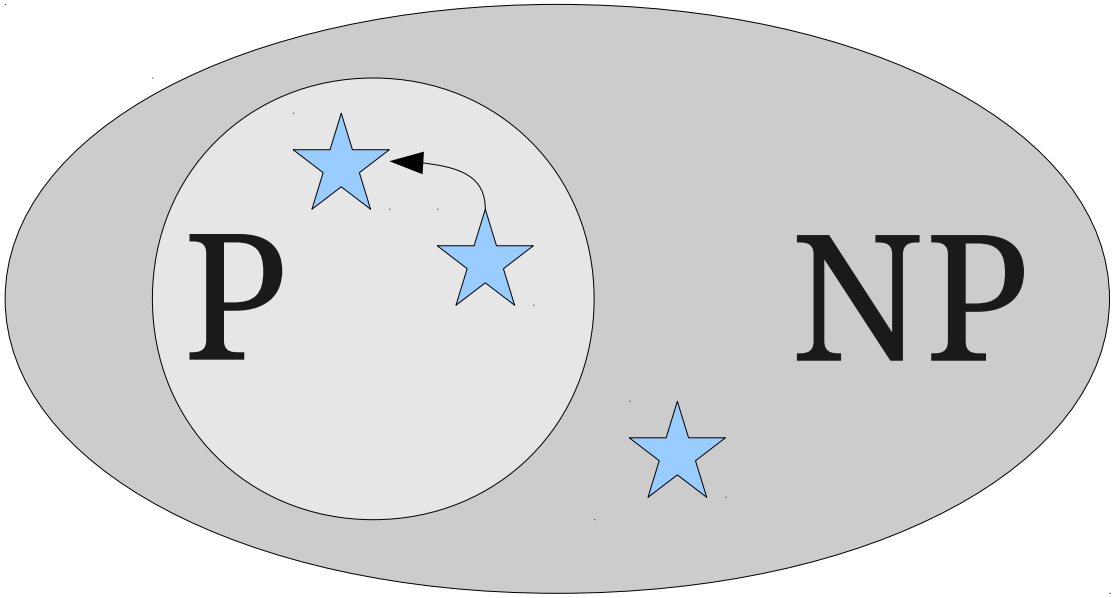
Polynomial-Time Reductions

- If a language L' is polynomial-time reducible to a problem L and L ∈ P, then L' ∈ P.



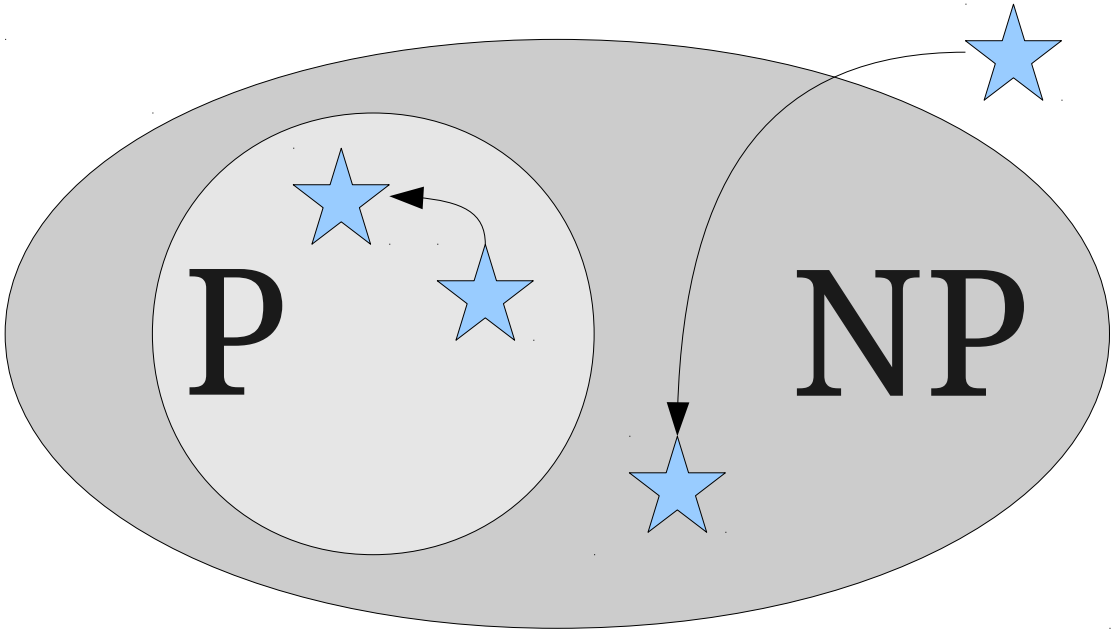
Polynomial-Time Reductions

- If a language L' is polynomial-time reducible to a problem L and L ∈ P, then L' ∈ P.
- If a problem L' is polynomial-time reducible to a problem L and L ∈ NP, then L' ∈ NP.



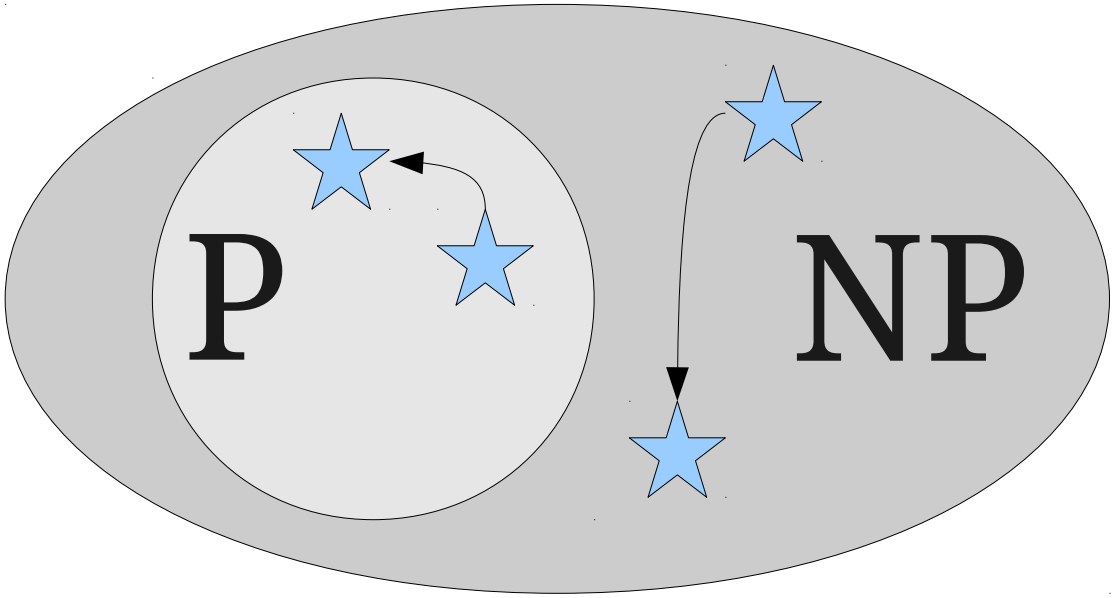
Polynomial-Time Reductions

- If a language L' is polynomial-time reducible to a problem L and L ∈ P, then L' ∈ P.
- If a problem L' is polynomial-time reducible to a problem L and L ∈ NP, then L' ∈ NP.



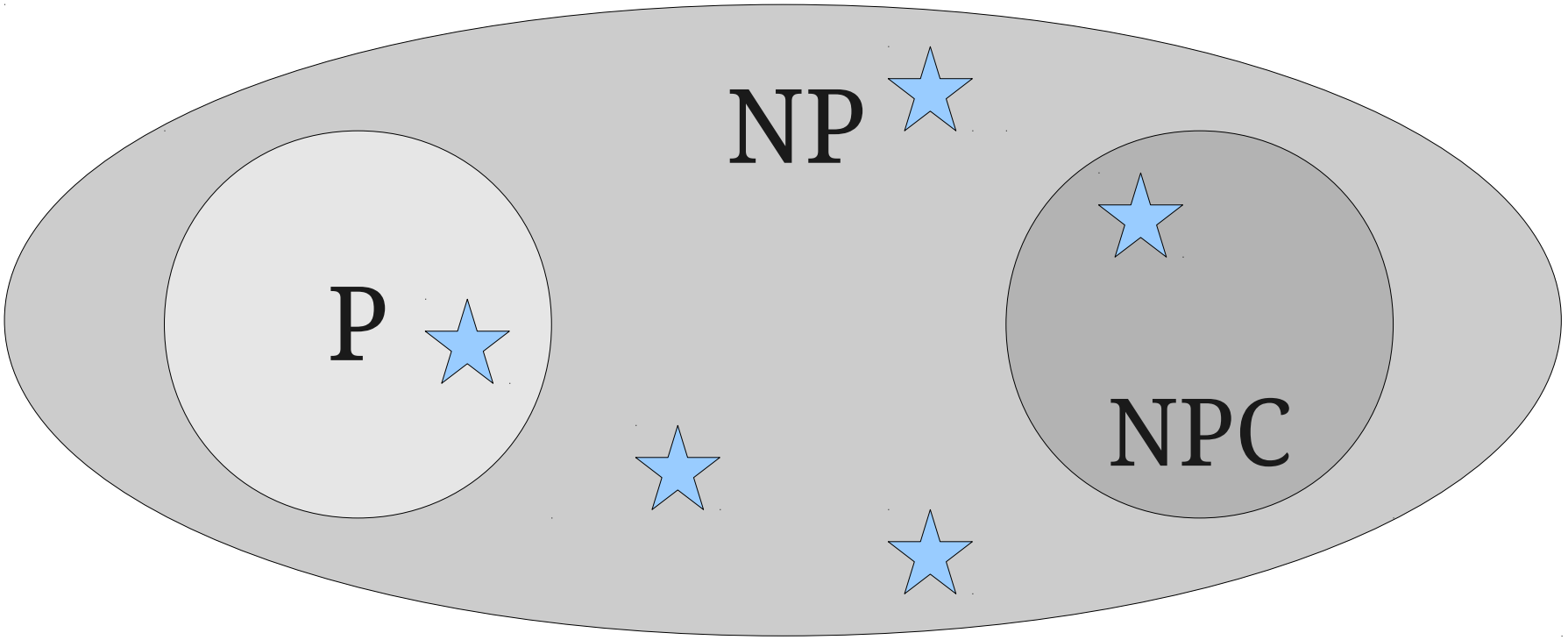
Polynomial-Time Reductions

- If a language L' is polynomial-time reducible to a problem L and L ∈ P, then L' ∈ P.
- If a problem L' is polynomial-time reducible to a problem L and L ∈ NP, then L' ∈ NP.



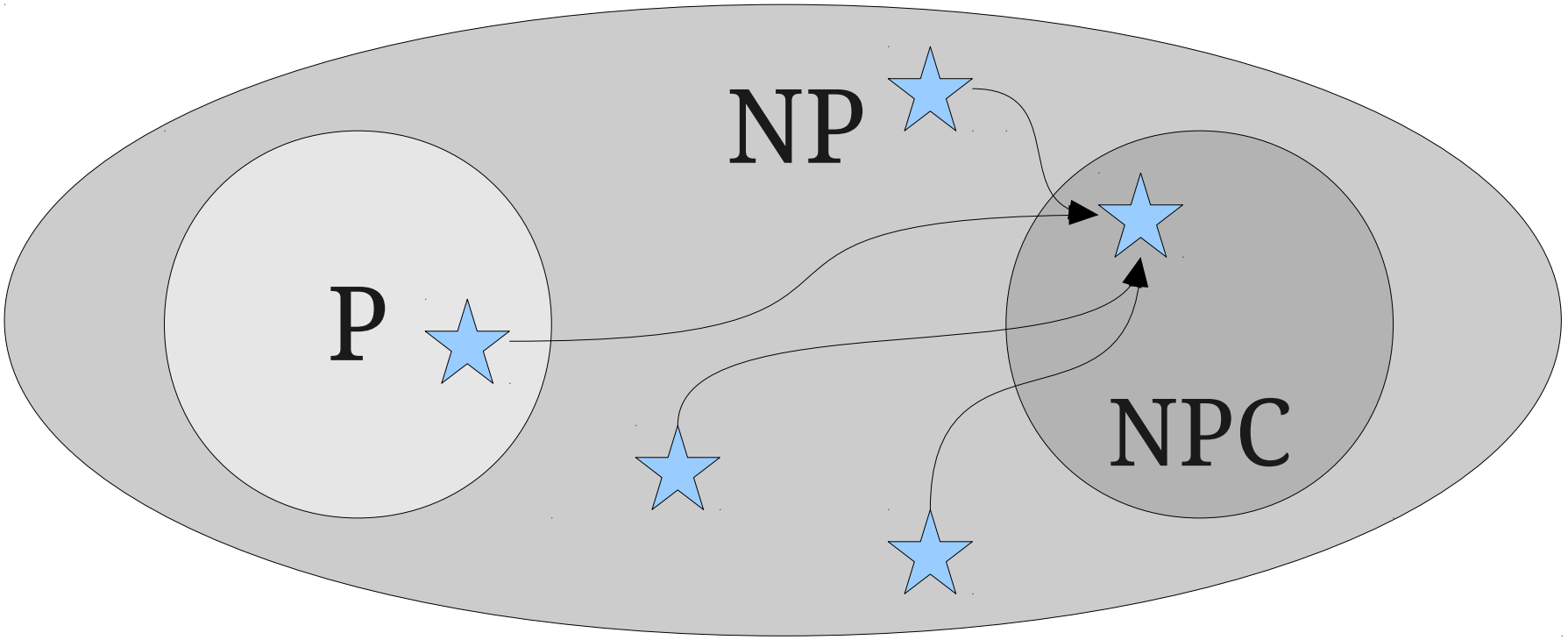
NP-Completeness

- A language L is called **NP-hard** if there is a polynomial-time reduction from **any** problem in NP to L.
- A language in L is called **NP-complete** if L ∈ NP and L is NP-hard.
- The class **NPC** is the set of NP-complete problems.



NP-Completeness

- A language L is called **NP-hard** if there is a polynomial-time reduction from **any** problem in NP to L.
- A language in L is called **NP-complete** if L ∈ NP and L is NP-hard.
- The class **NPC** is the set of NP-complete problems.

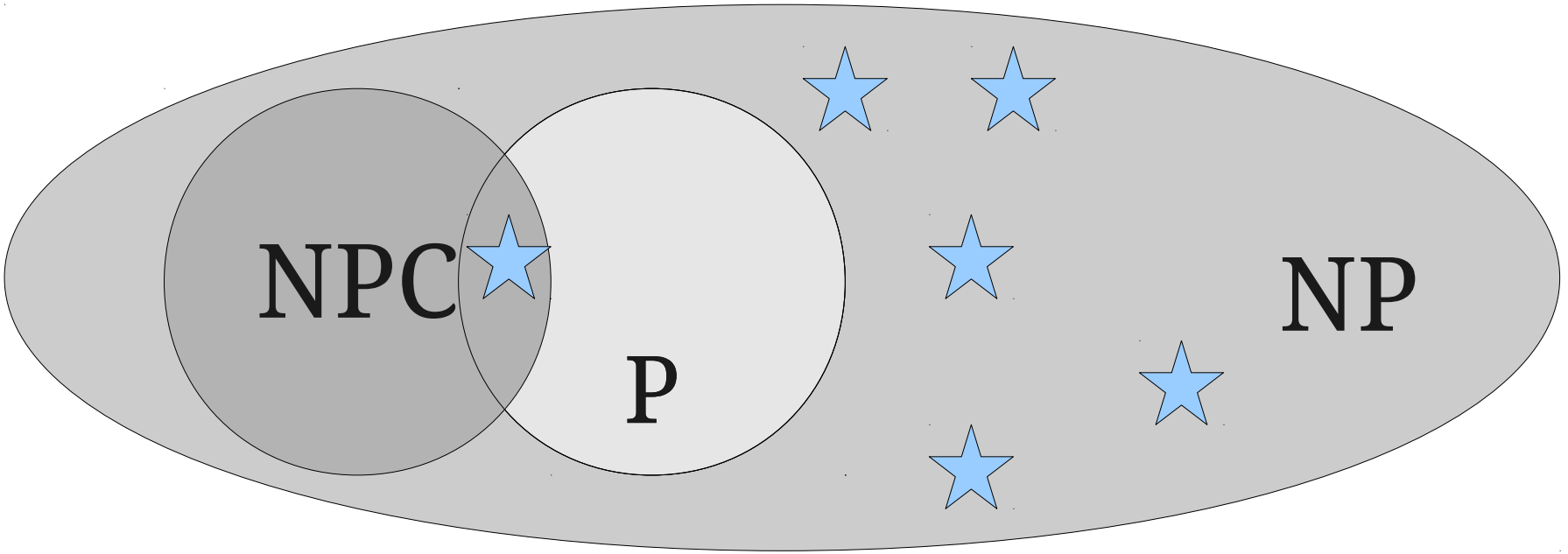


The Tantalizing Truth

- **Theorem:** If ***any*** NP-complete language is in P, then P = NP.
- **Proof Sketch**: If L ∈ NPC, all languages in NP are polynomial-time reducible to it. Since L ∈ P, any language in NP is also in P. Thus NP ⊆ P, so P = NP.

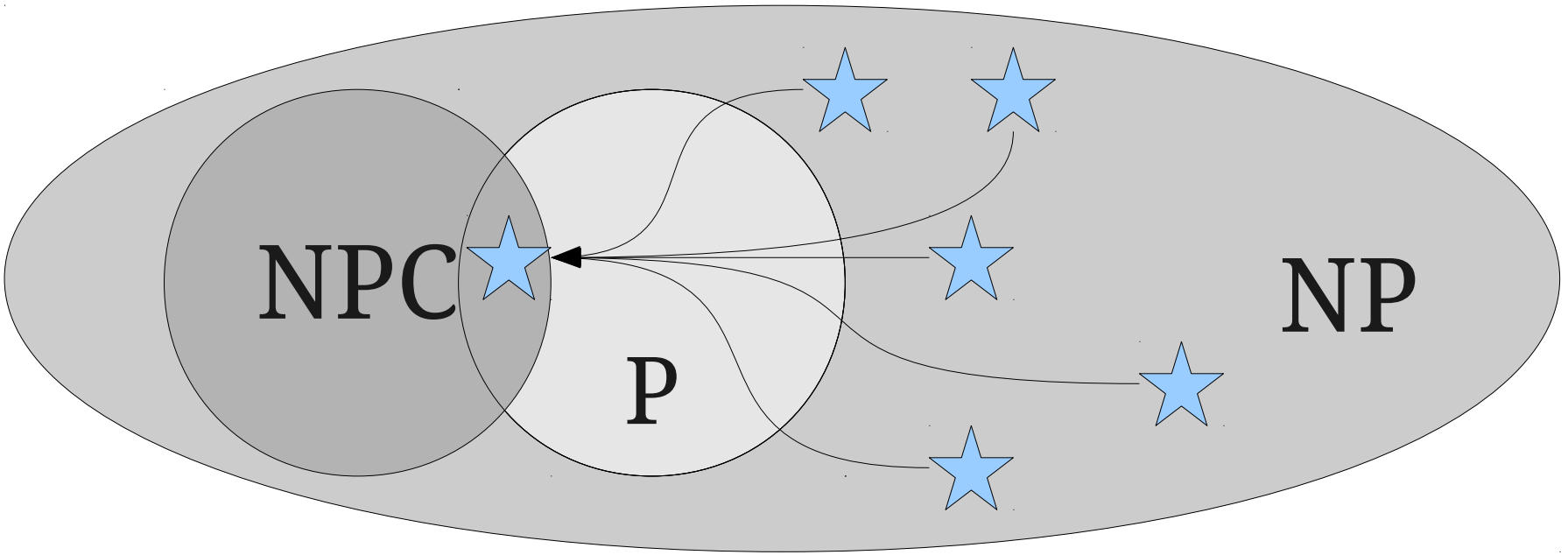
The Tantalizing Truth

- **Theorem:** If ***any*** NP-complete language is in P, then P = NP.
- **Proof Sketch**: If L ∈ NPC, all languages in NP are polynomial-time reducible to it. Since L ∈ P, any language in NP is also in P. Thus NP ⊆ P, so P = NP.



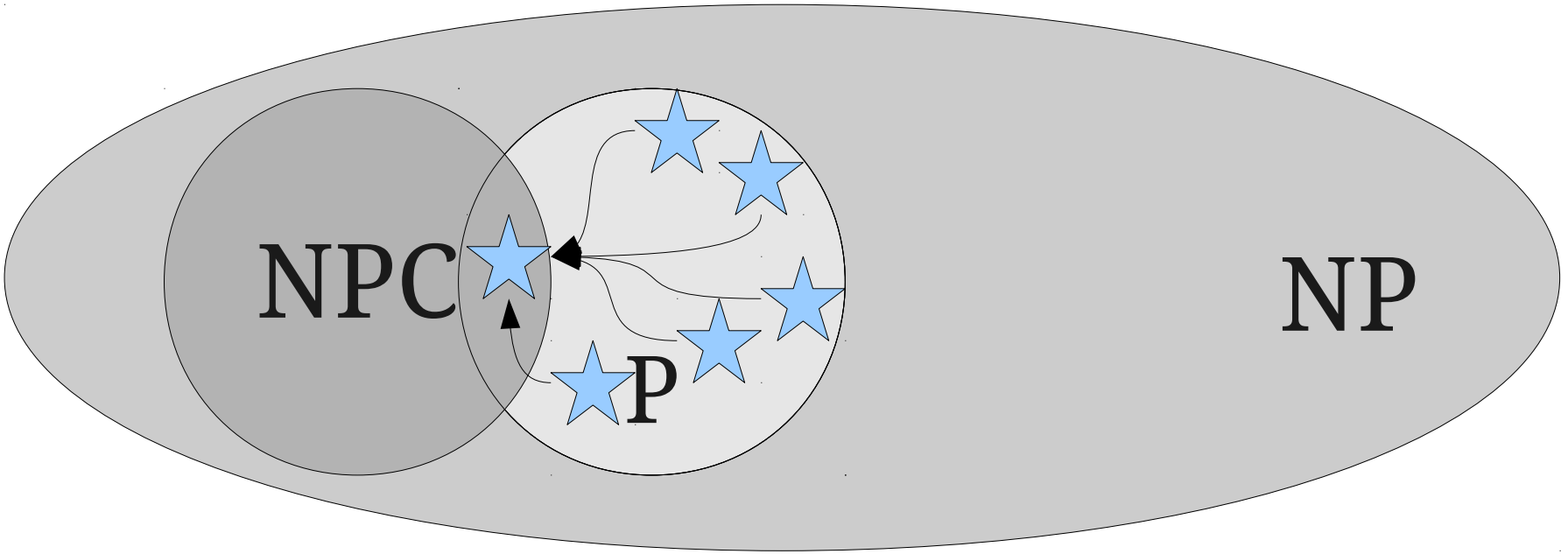
The Tantalizing Truth

- **Theorem:** If ***any*** NP-complete language is in P, then P = NP.
- **Proof Sketch**: If L ∈ NPC, all languages in NP are polynomial-time reducible to it. Since L ∈ P, any language in NP is also in P. Thus NP ⊆ P, so P = NP.



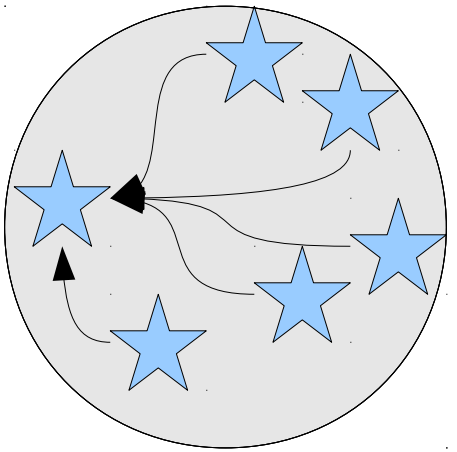
The Tantalizing Truth

- **Theorem:** If ***any*** NP-complete language is in P, then P = NP.
- **Proof Sketch**: If L ∈ NPC, all languages in NP are polynomial-time reducible to it. Since L ∈ P, any language in NP is also in P. Thus NP ⊆ P, so P = NP.



The Tantalizing Truth

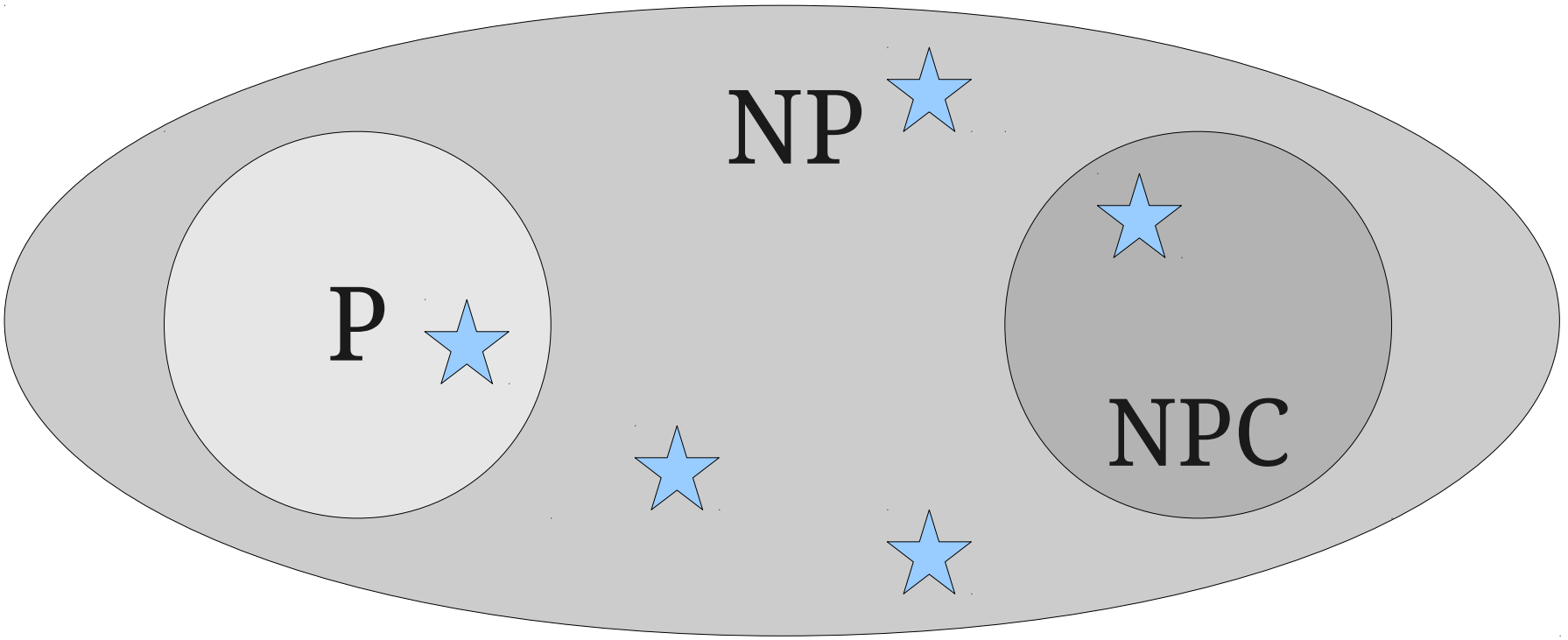
- **Theorem:** If ***any*** NP-complete language is in P, then P = NP.
- **Proof Sketch**: If L ∈ NPC, all languages in NP are polynomial-time reducible to it. Since L ∈ P, any language in NP is also in P. Thus NP ⊆ P, so P = NP.



P = NP

The Tantalizing Truth

- **Theorem:** If ***any*** NP-complete language is not in P, then P ≠ NP.
- **Proof**: If L is NP-complete, it is in NP. If it is in NP but not P, then P ≠ NP.



An Feel for NP-Completeness

- If a problem is NP-complete, under the (commonly-held) assumption that P ≠ NP, then there cannot be an efficient algorithm for it.
- In a sense, NP-complete problems are the hardest problems in NP.
- Because it is conjectured that P ≠ NP, NP-complete problems tend to be enormously difficult to solve.
  - All known algorithms for NP-complete problems run in exponential time.
  - All known algorithms for NP-complete problems are infeasible for any reasonably-sized inputs.

Why This Matters

- When discussing computability theory, we saw that these questions are undecidable:
  - Does M halt on w?
  - Is L(M) regular?
  - Is L(M) infinite?
  - Does M accept a string of length at most three?
  - Does M accept every string?
- These are all questions about TMs.
- Though these problems sometimes arise in practice, it's unlikely that you will bump into an undecidable problem unless you're asking questions about what a program does.

Why This Matters

- In contrast, NP-complete problems are **everywhere**.
  - Can you buy five cell phone towers to guarantee everyone in a city has service?
  - Is there a sequence of 100 genes that is common to some group of DNA strands?
  - Given a list of prices and a budget, can you buy over fifteen items?
  - Given a city layout, can you use 100,000 feet of piping to supply water to each building?
- **These are questions we want to solve on a daily basis**.

Why This Matters

- In contrast, NP-complete problems are **everywhere**.
  - Can you buy five cell phone towers to guarantee everyone in a city has service?
  - Is there a sequence of 100 genes that is common to some group of DNA strands?
  - Given a list of prices and a budget, can you buy over fifteen items?
  - Given a city layout, can you use 100,000 feet of piping to supply water to each building?
- **These are questions we need to solve on a daily basis**.

What Problems are NP-Complete?

- NP-complete problems give a promising approach for proving or disproving that P = NP:
  - If any NPC problem is in P, then P = NP.
  - If any NPC problem is not in P, then P ≠ NP.
- However, we haven't shown that any problems are NP-complete in the first place!
- How do we even know they exist?

Satisfiability

- A propositional logic formula is called **satisfiable** if there is an assignment to its variables that makes it evaluate to true.
- For example, the following are all satisfiable:
  - $p \wedge q$
  - $(p \rightarrow q) \wedge \neg p \wedge \neg q$
  - $(p \vee q) \wedge (\neg p \vee q) \wedge (p \vee \neg q)$
- The following are all unsatisfiable:
  - $p \wedge \neg p$
  - $(p \rightarrow \neg p) \wedge (\neg p \rightarrow p)$
  - $(p \vee q) \wedge (\neg p \vee q) \wedge (p \vee \neg q) \wedge (\neg p \vee \neg q)$

Conjunctive Normal Form

- A **literal** in propositional logic is a variable or its negation:
  - x
  - ¬y
  - But not x ∧ y.
- A **clause** is a many-way OR (*disjunction*) of literals.
  - ¬x ∨ y ∨ ¬z
  - x
  - But not x ∨ ¬(y ∨ z)
- A propositional logic formula φ is in **conjunctive normal form** (**CNF**) if it is the many-way AND (*conjunction*) of clauses.
  - (x ∨ y ∨ z) ∧ (¬x ∨ ¬y) ∧ (x ∨ y ∨ z ∨ ¬w)
  - x ∨ z
  - But not (x ∨ (y ∧ z)) ∨ (x ∨ y)

3-CNF

- A propositional formula is in **3-CNF** if
  - It is in CNF, and
  - Every clause has exactly three literals.
- For example:
  - (x ∨ y ∨ z) ∧ (¬x ∨ ¬y ∨ z) ∧ (x ∨ y ∨ ¬z)
  - (x ∨ x ∨ x) ∧ (y ∨ ¬y ∨ ¬x) ∧ (x ∨ y ∨ ¬y)
  - But not (x ∨ y ∨ z ∨ w) ∧ (x ∨ y)
- The language **3SAT** is defined as follows:

  3SAT = { φ | φ is a satisfiable 3-CNF formula }

The Cook-Levin Theorem

- The Cook-Levin Theorem states that

  $$\textbf{SAT} = \{ \varphi \mid \varphi \text{ is satisfiable} \}$$

  is NP-complete.
- A stronger version of this theorem is that

  $$\textbf{3SAT} = \{ \varphi \mid \varphi \text{ is a satisfiable 3-CNF formula} \}$$

  is also NP-complete.
- This is an **extremely important result** in complexity theory, as it gives a starting point for finding NP-complete problems.

Proving the Cook-Levin Theorem

- **Proof Sketch**:
  - Given any problem in NP, there is a polynomial-time NTM for it.
  - Build, in polynomial time and space, a boolean formula that simulates running the NTM on some string *w*.
  - The formula is satisfiable iff the NTM accepts *w*.
- In the interests of time, we'll skip the details for now; see Sipser for more information.

Using the Cook-Levin Theorem

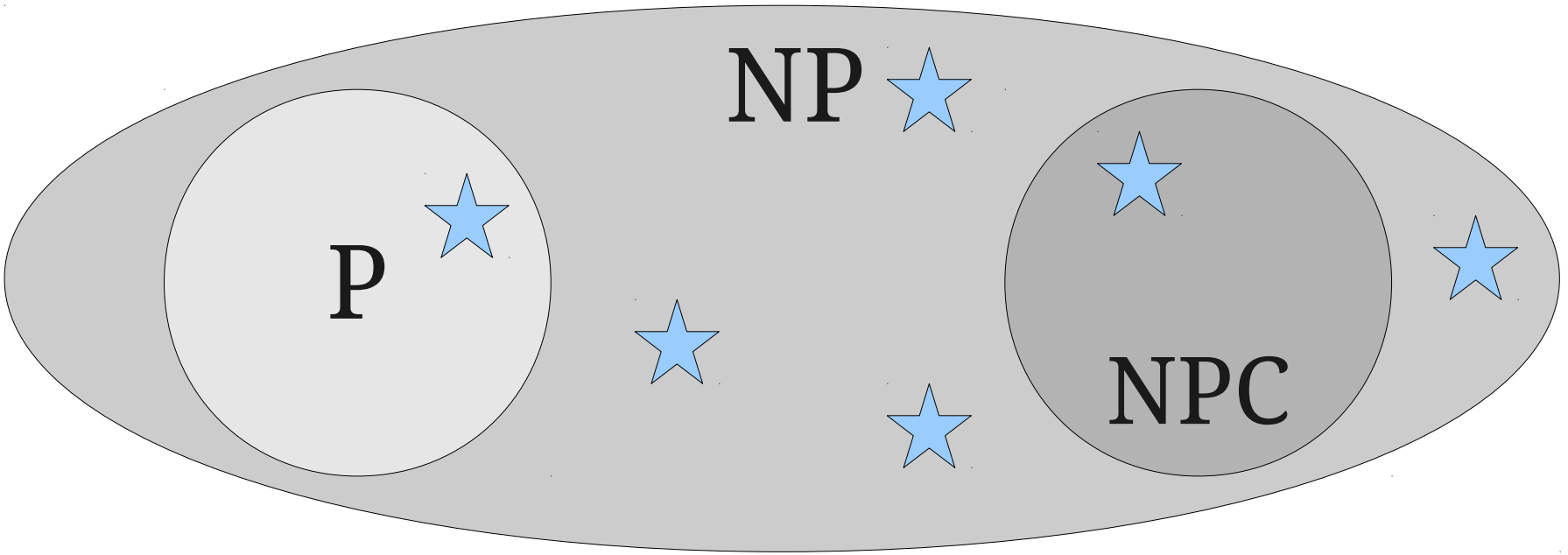
- When discussing decidability, we used the undecidability of *HALT* to prove that many other languages are undecidable.
  - **Idea**: Reduce *HALT* to some other language.
- Using the NP-completeness of 3SAT, we can show the NP-completeness of many other problems.
  - **Idea**: Reduce 3SAT to some other language.

NP-Completeness

- **Theorem**: If L' is NP-complete and L' is polynomial-time reducible to L ∈ NP, then L is NP-complete.
- **Proof Sketch**: Any language in NP has a polynomial-time reduction to L'; let that reduction be f. L' is polynomial-time reducible to L by a reduction f'. Then the composition of f' and f is a polynomial-time reduction from the original language to L.

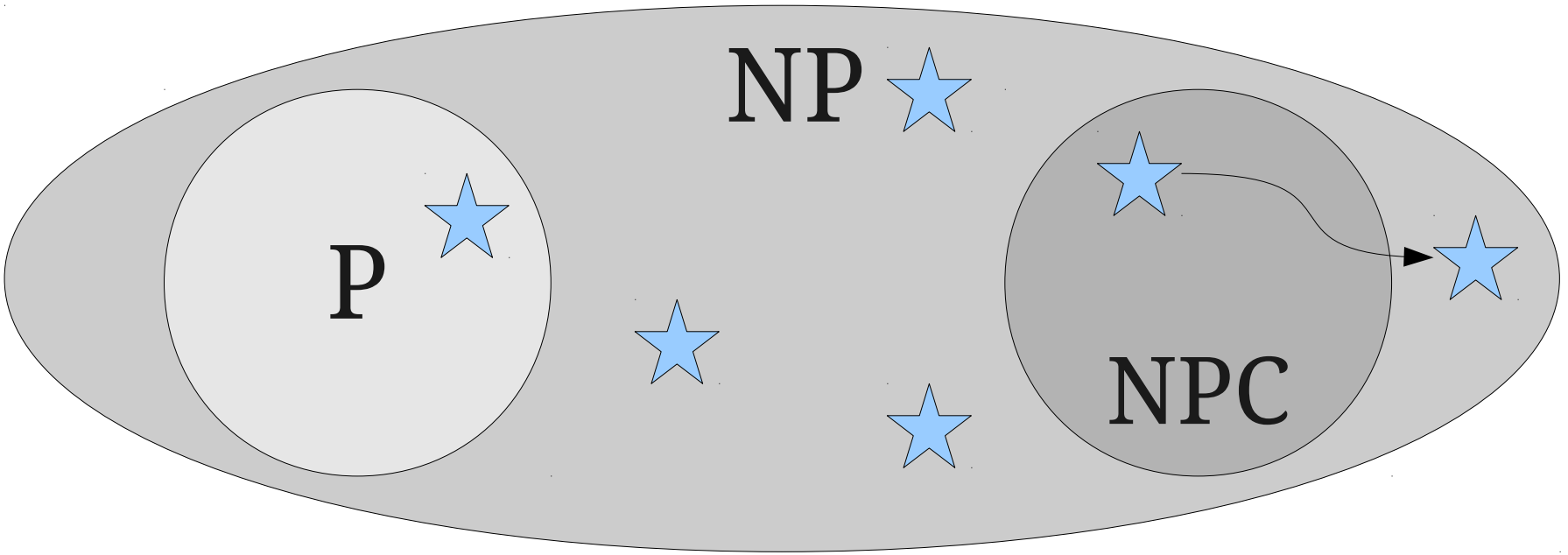
NP-Completeness

- **Theorem**: If L' is NP-complete and L' is polynomial-time reducible to L ∈ NP, then L is NP-complete.
- **Proof Sketch**: Any language in NP has a polynomial-time reduction to L'; let that reduction be f. L' is polynomial-time reducible to L by a reduction f'. Then the composition of f' and f is a polynomial-time reduction from the original language to L.



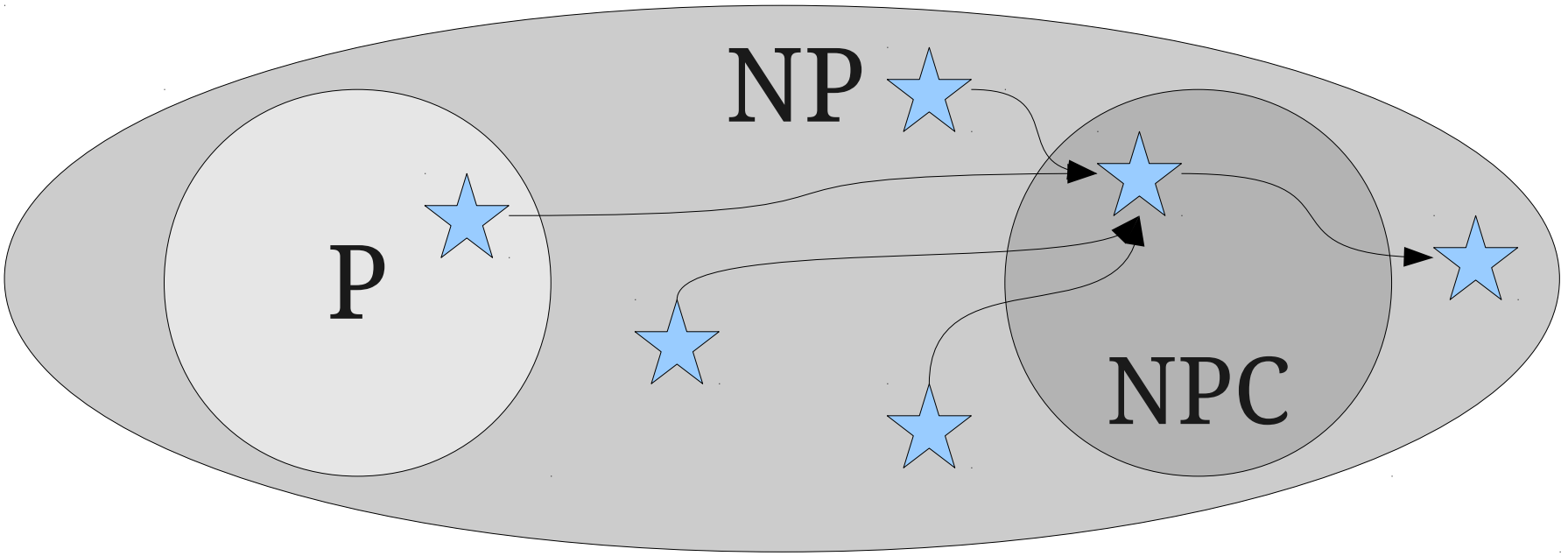
NP-Completeness

- **Theorem**: If L' is NP-complete and L' is polynomial-time reducible to L ∈ NP, then L is NP-complete.
- **Proof Sketch**: Any language in NP has a polynomial-time reduction to L'; let that reduction be f. L' is polynomial-time reducible to L by a reduction f'. Then the composition of f' and f is a polynomial-time reduction from the original language to L.



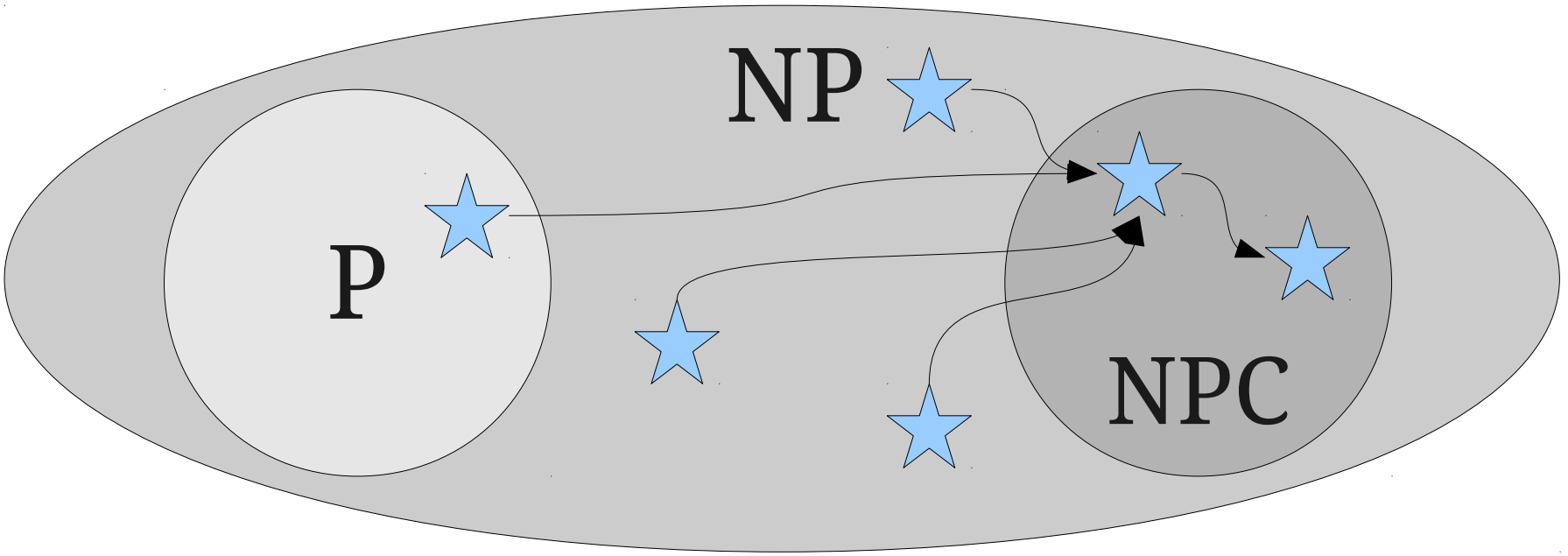
NP-Completeness

- **Theorem**: If L' is NP-complete and L' is polynomial-time reducible to L ∈ NP, then L is NP-complete.
- **Proof Sketch**: Any language in NP has a polynomial-time reduction to L'; let that reduction be f. L' is polynomial-time reducible to L by a reduction f'. Then the composition of f' and f is a polynomial-time reduction from the original language to L.



NP-Completeness

- **Theorem**: If L' is NP-complete and L' is polynomial-time reducible to L ∈ NP, then L is NP-complete.
- **Proof Sketch**: Any language in NP has a polynomial-time reduction to L'; let that reduction be f. L' is polynomial-time reducible to L by a reduction f'. Then the composition of f' and f is a polynomial-time reduction from the original language to L.

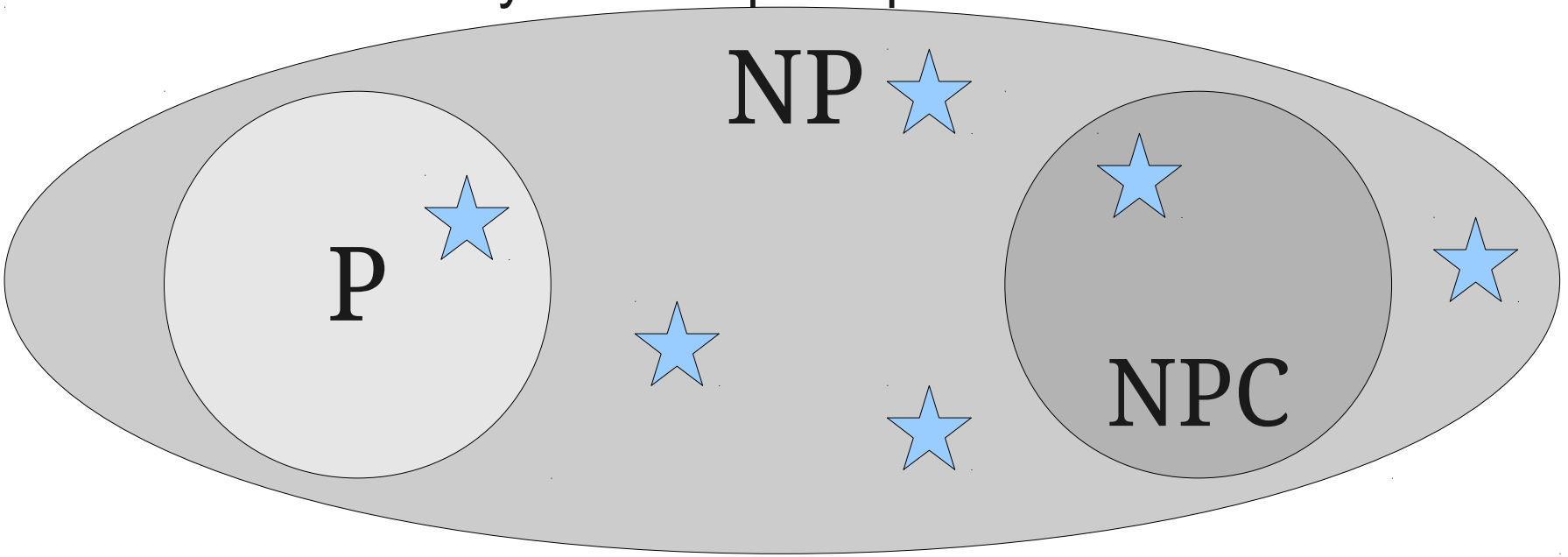


Be Careful!

- To show that a language is NP-complete, reduce a known NP-complete problem to it.
- **Do not** reduce the language to a known NP-complete problem!
  - We already knew that you could do this; **every** NP problem is reducible to any NP-complete problem!

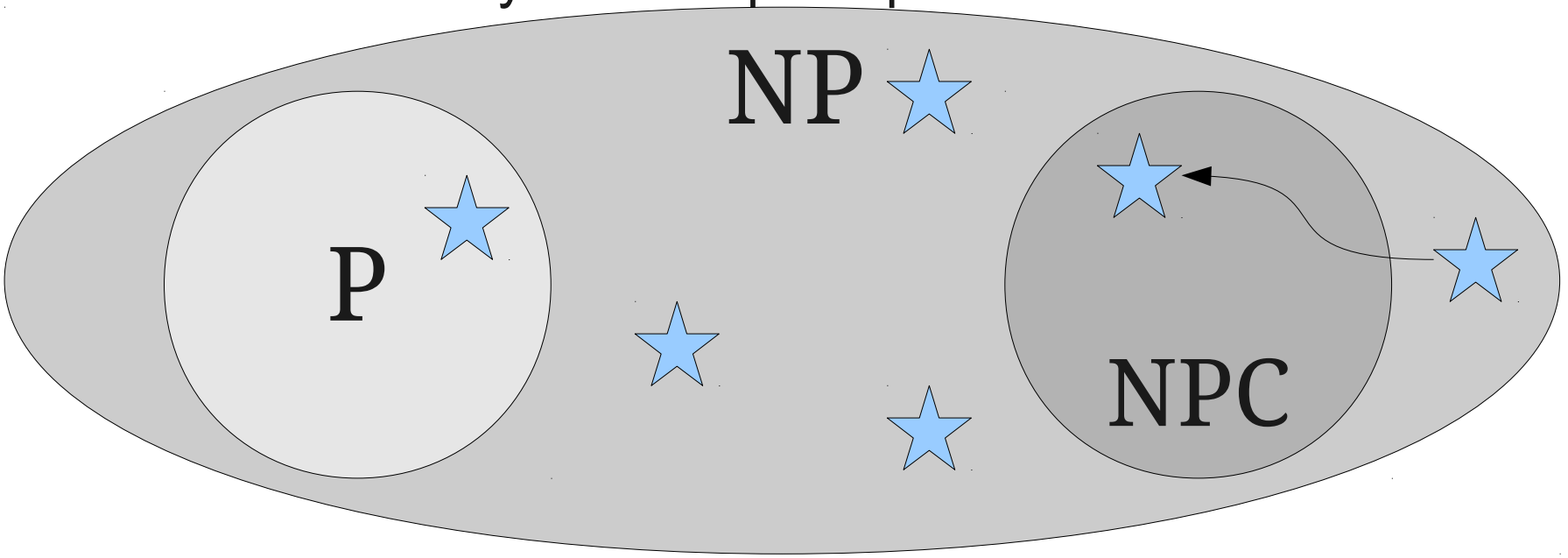
Be Careful!

- To show that a language is NP-complete, reduce a known NP-complete problem to it.
- **Do not** reduce the language to a known NP-complete problem!
  - We already knew that you could do this; **every** NP problem is reducible to any NP-complete problem!



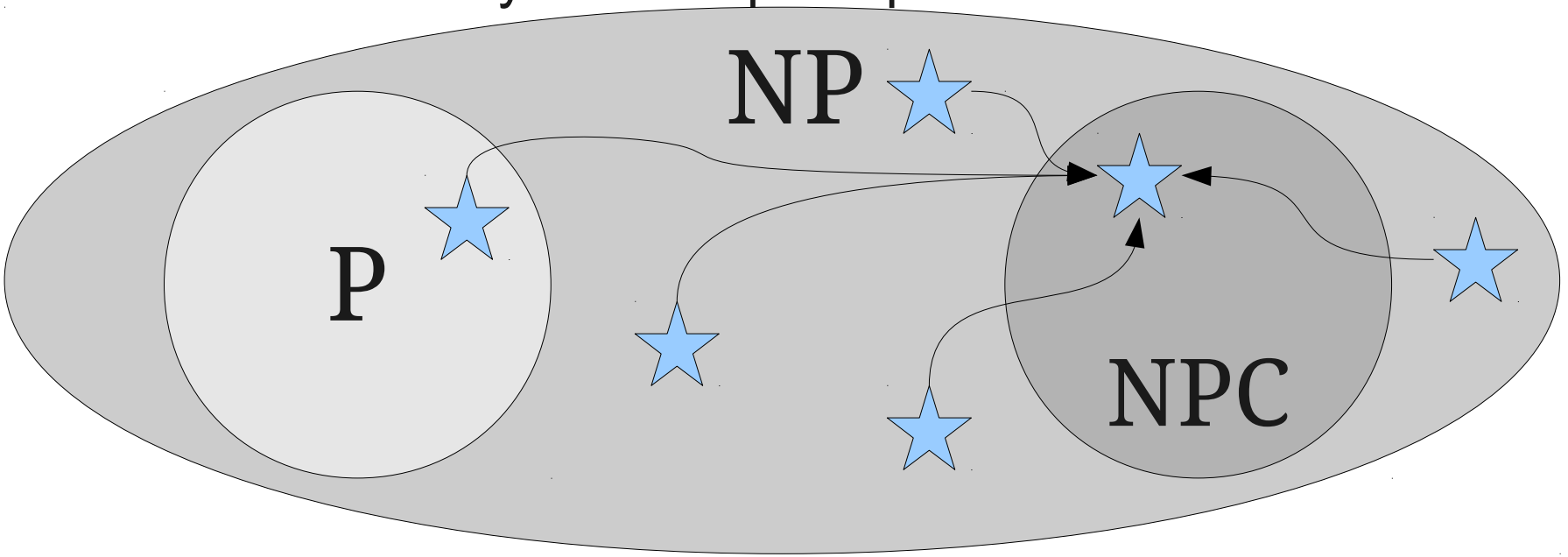
Be Careful!

- To show that a language is NP-complete, reduce a known NP-complete problem to it.
- **Do not** reduce the language to a known NP-complete problem!
  - We already knew that you could do this; **every** NP problem is reducible to any NP-complete problem!

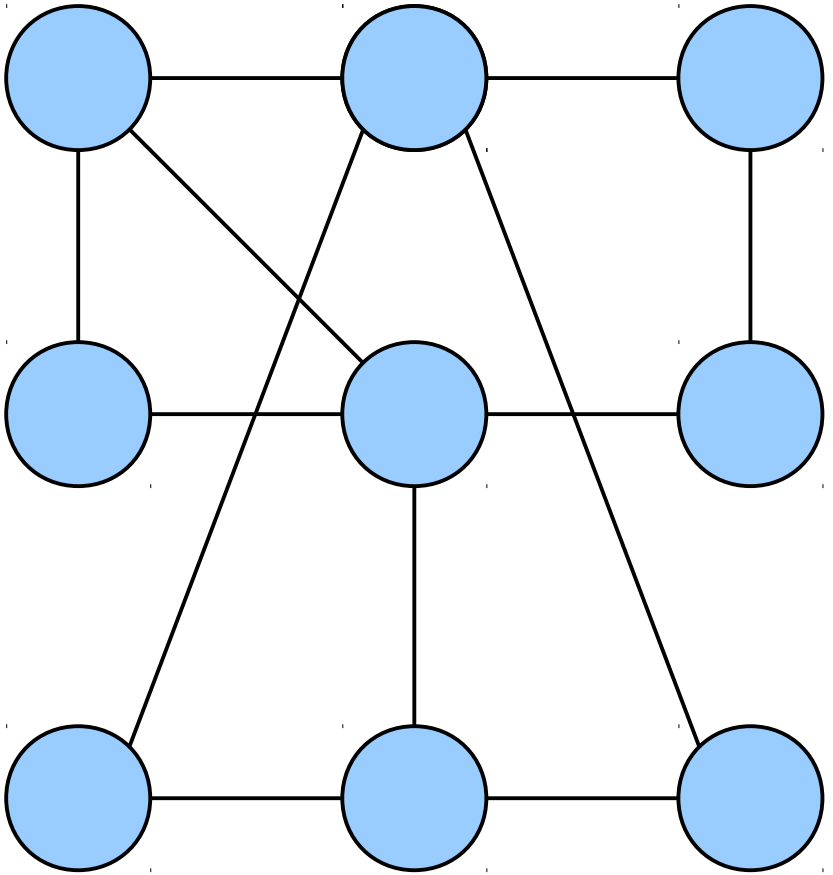


Be Careful!

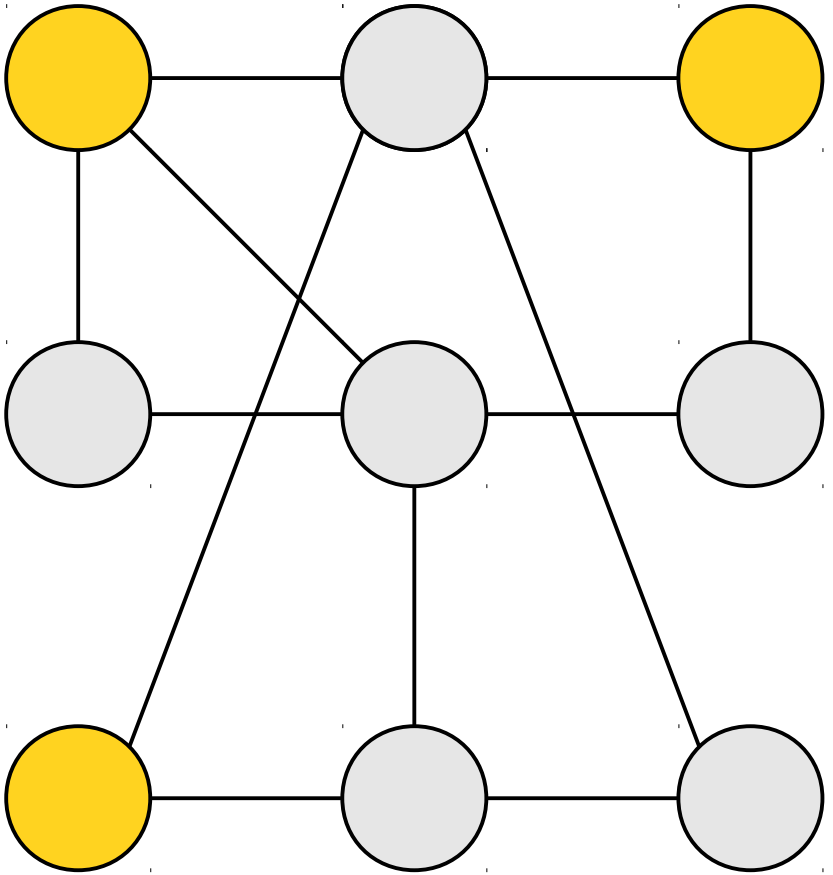
- To show that a language is NP-complete, reduce a known NP-complete problem to it.
- **Do not** reduce the language to a known NP-complete problem!
  - We already knew that you could do this; **every** NP problem is reducible to any NP-complete problem!



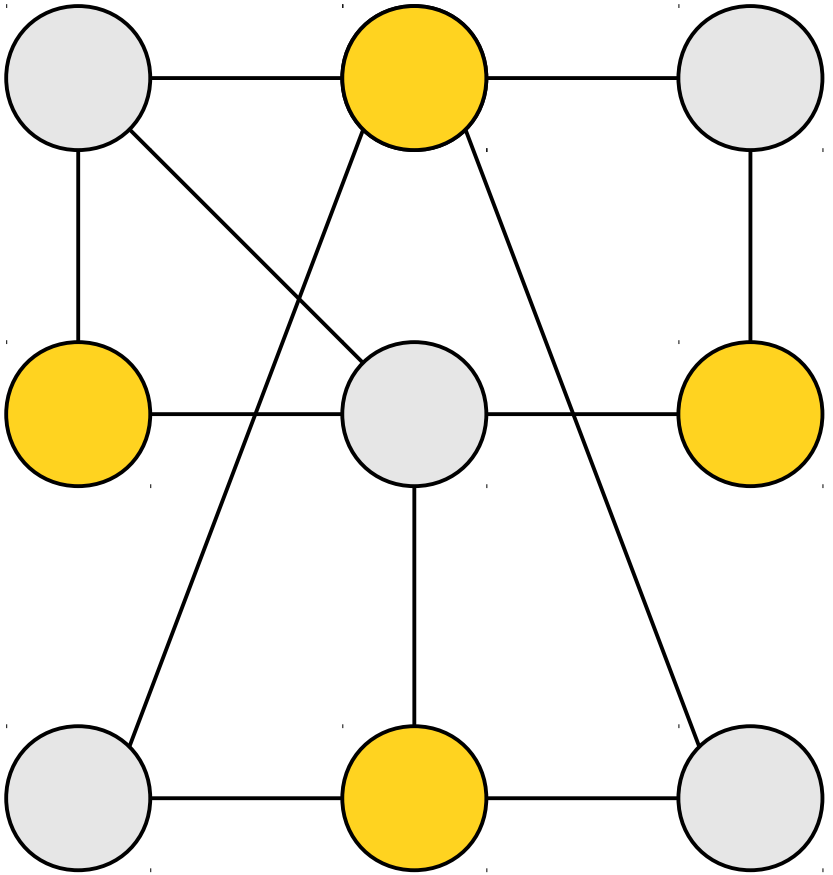
An **independent set** in an undirected graph
is a set of vertices that have no edges between them



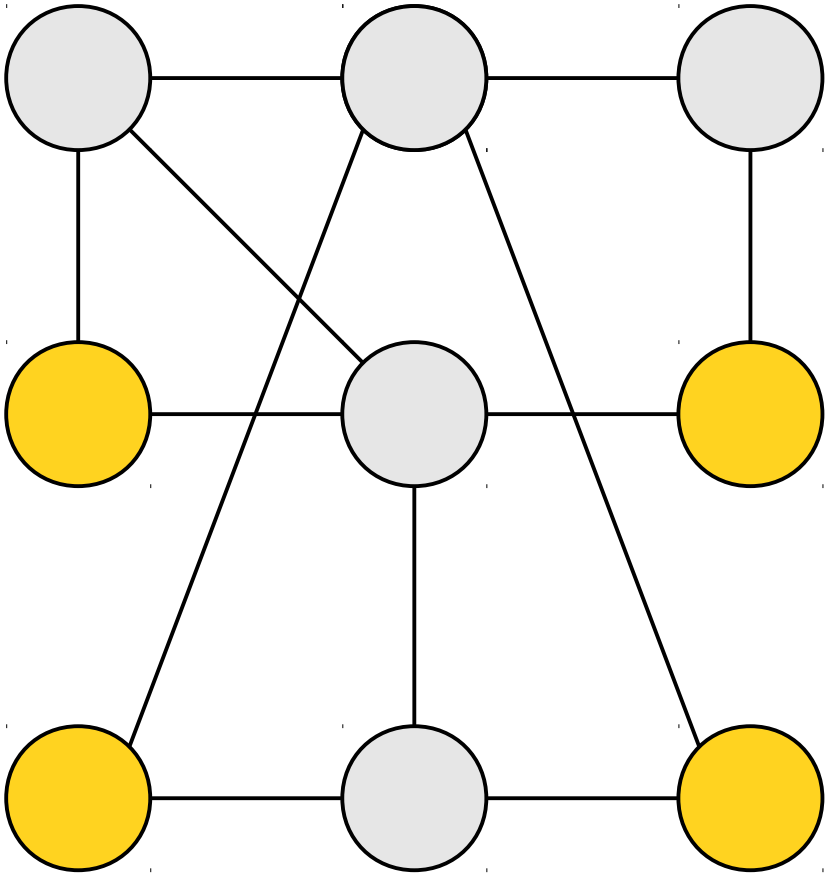
An **independent set** in an undirected graph
is a set of vertices that have no edges between them



An **independent set** in an undirected graph
is a set of vertices that have no edges between them



An **independent set** in an undirected graph
is a set of vertices that have no edges between them



An **independent set** in an undirected graph
is a set of vertices that have no edges between them

The Independent Set Problem

- Given an undirected graph G and a natural number n, the **independent set problem** is

  Does G contain an independent set of size at least n?

- As a formal language:

  INDSET = { ⟨G, n⟩ | G is an undirected graph with an independent set of size at least n }

INDSET $\in$ NP

- The independent set problem is in NP.
- Here is a polynomial-time verifier that checks whether S is an n-element independent set:
  - V = “On input ⟨G, n, S⟩:
    - If |S| < n, reject.
    - For each edge in G, if both endpoints are in S, reject.
    - Otherwise, accept.”

INDSET is NP-Complete

- The INDSET problem is NP-complete.
- Use a polynomial-time reduction from 3SAT:
  - Given a 3-CNF formula φ with n clauses, construct a graph G such that φ is satisfiable iff G has an independent set of size n.
- How can we accomplish this?

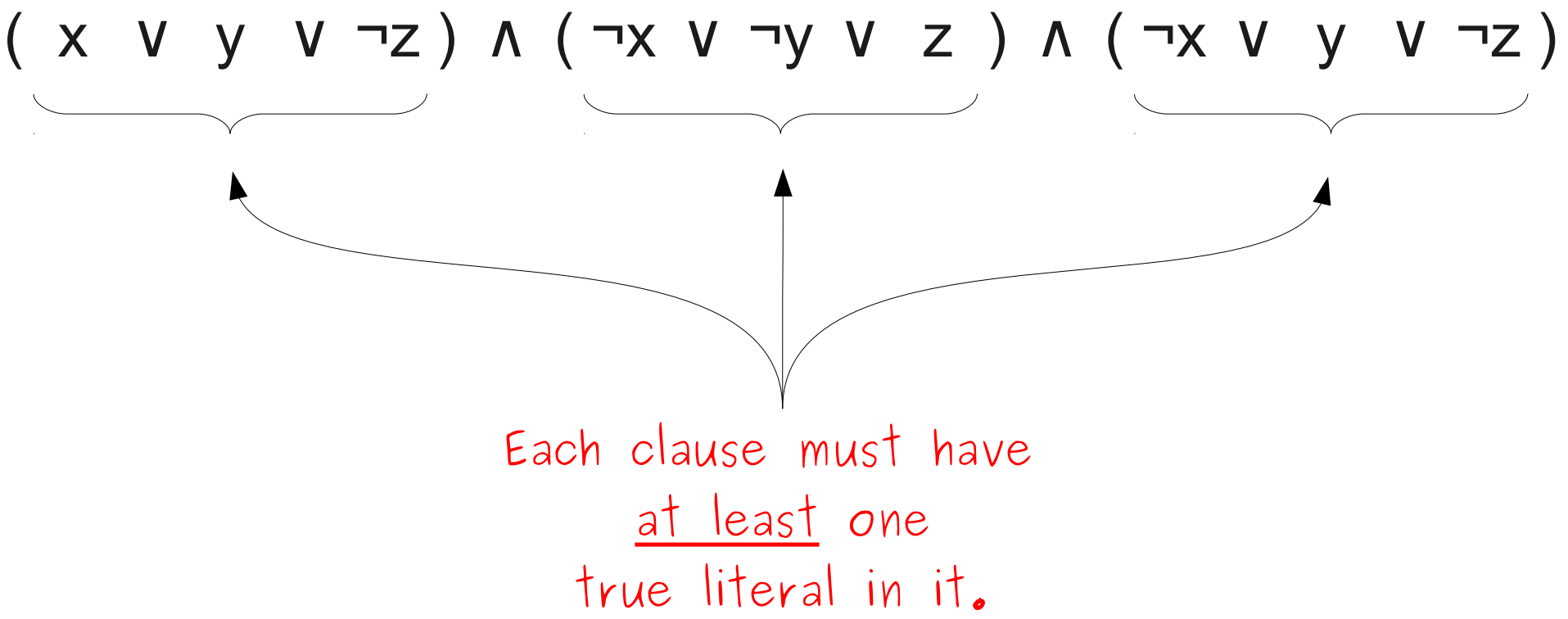
The Structure of 3SAT

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

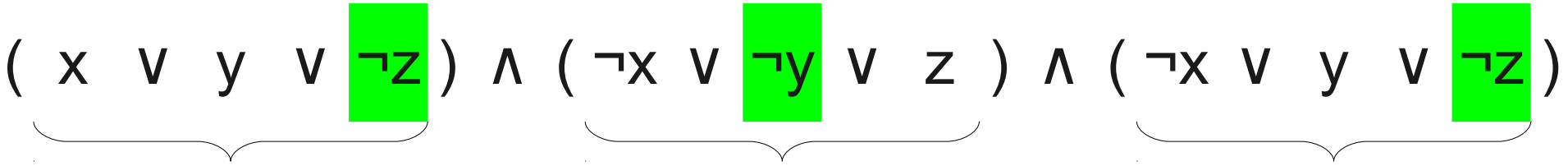
The Structure of 3SAT

$$(\underbrace{x \vee y \vee \neg z}) \wedge (\underbrace{\neg x \vee \neg y \vee z}) \wedge (\underbrace{\neg x \vee y \vee \neg z})$$

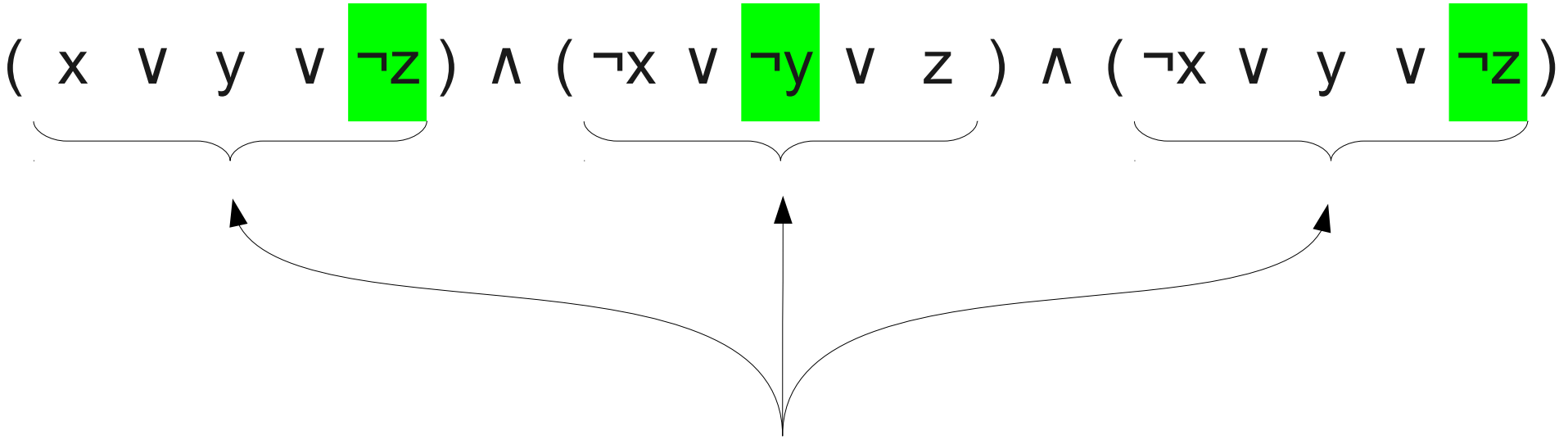
The Structure of 3SAT



The Structure of 3SAT

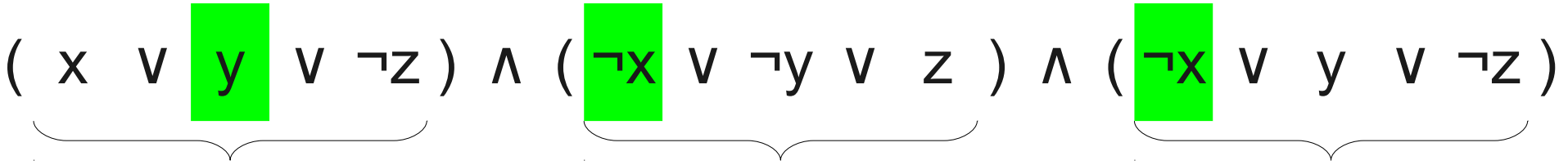


The Structure of 3SAT



One way to think about solving 3SAT is to pick which literals in each clause need to be made true.

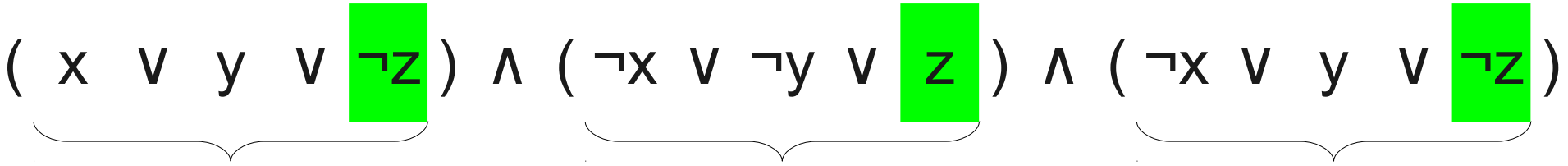
The Structure of 3SAT



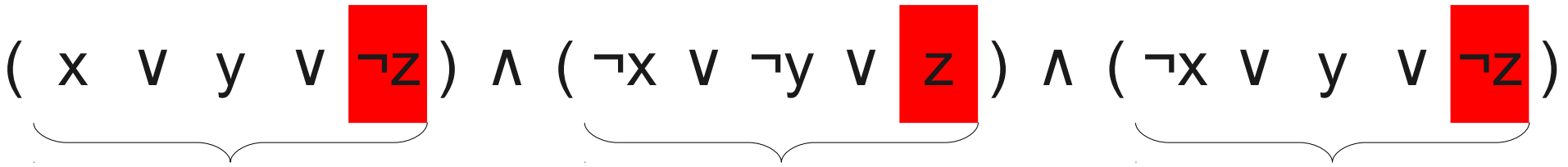
The Structure of 3SAT

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

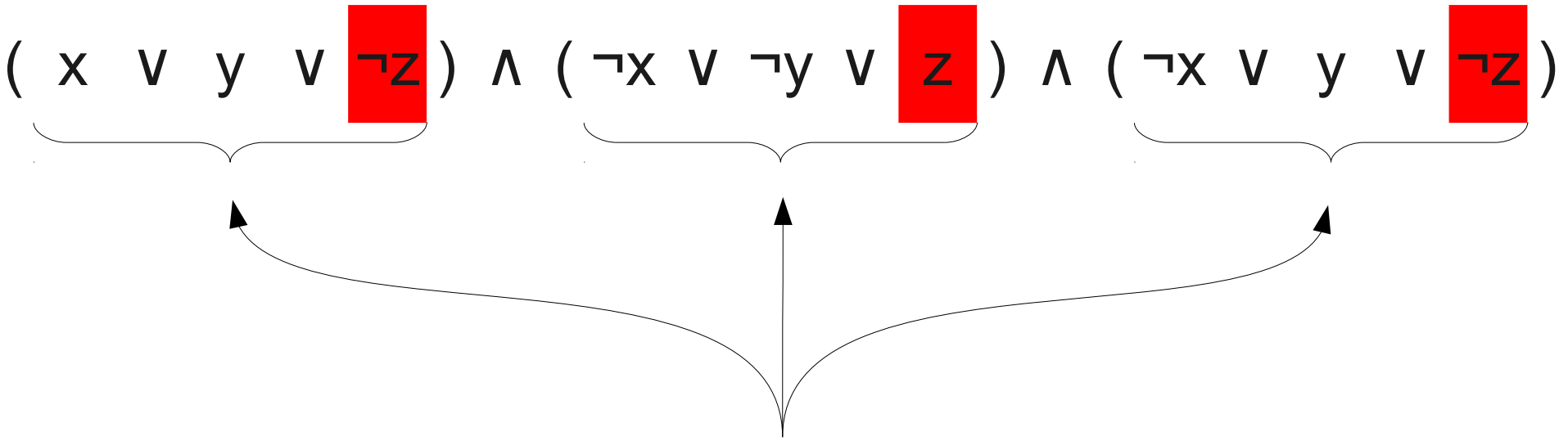
The Structure of 3SAT



The Structure of 3SAT



The Structure of 3SAT



… subject to the constraint that
we never choose a literal
and its negation

From 3SAT to INDSET

- To convert a 3SAT instance to INDSET, we need a graph such that an independent set in that graph
  - gives us a way to choose which literal in each clause should be true,
  - doesn't simultaneously choose a literal and its negation, and
  - has size polynomially large in the length of the original formula.

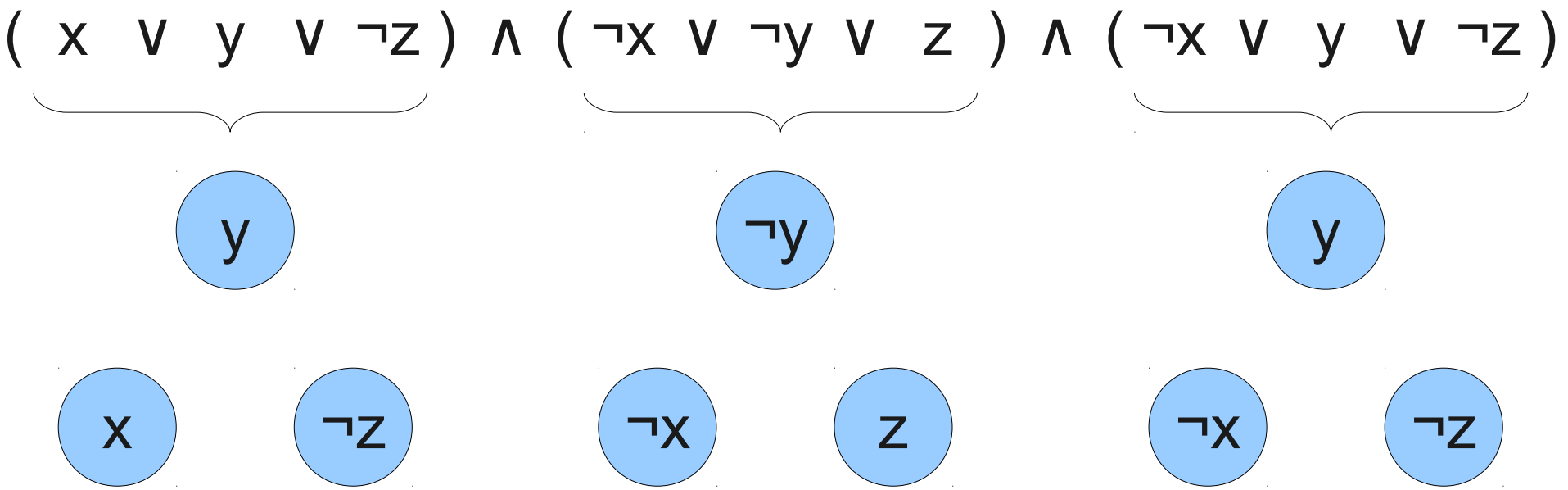
From 3SAT to INDSET

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

From 3SAT to INDSET

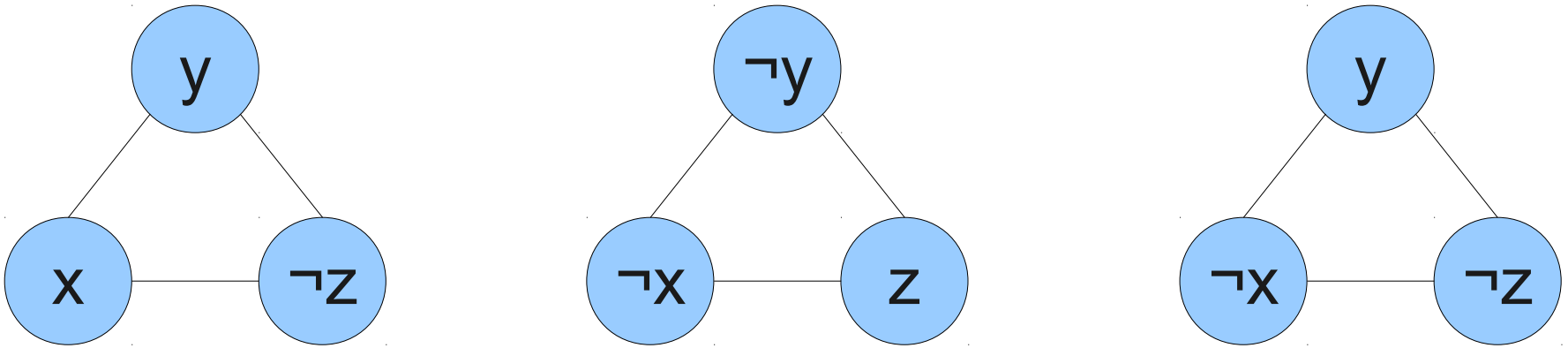
$$(\underbrace{x \vee y \vee \neg z}) \wedge (\underbrace{\neg x \vee \neg y \vee z}) \wedge (\underbrace{\neg x \vee y \vee \neg z})$$

From 3SAT to INDSET



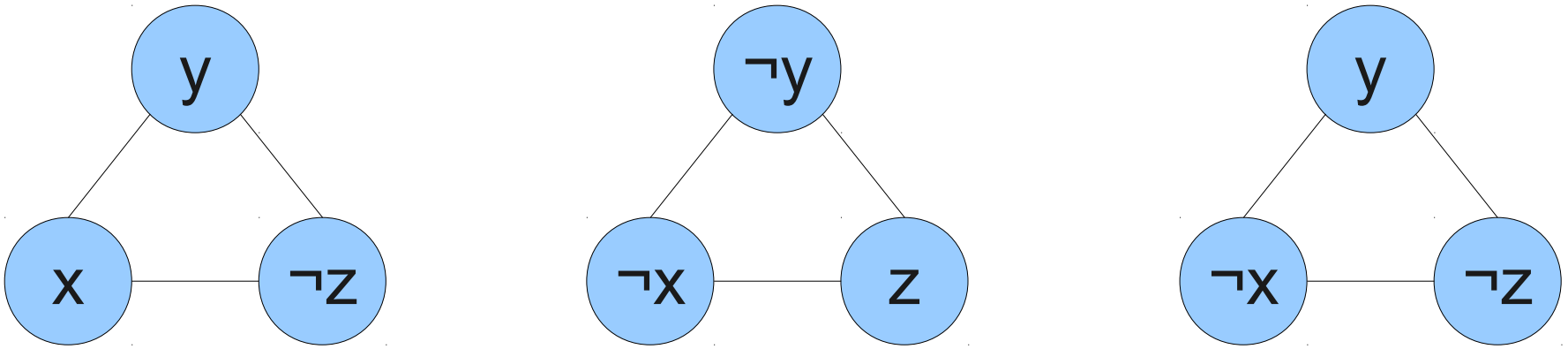
From 3SAT to INDSET

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



From 3SAT to INDSET

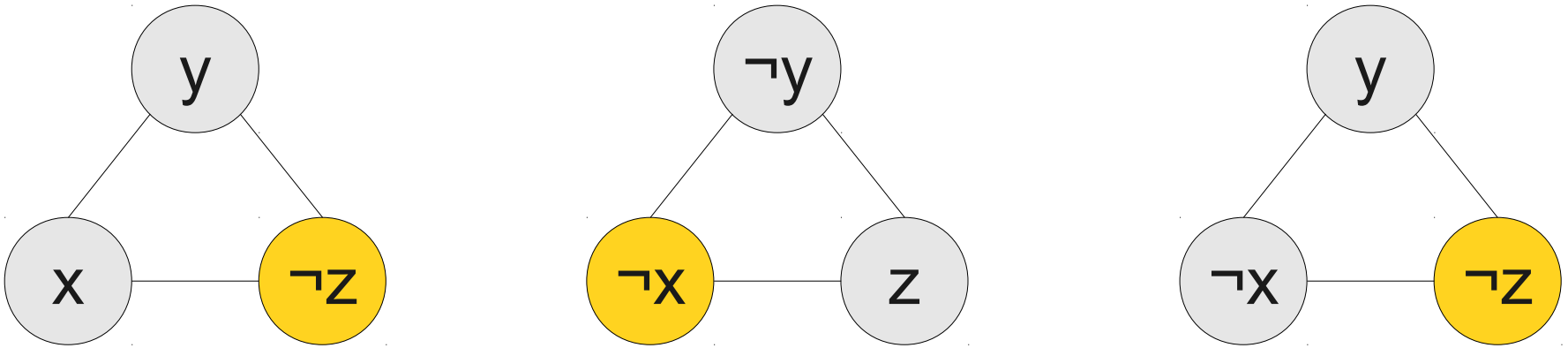
$$(x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee z) \wedge (\neg x \vee y \vee \neg z)$$



Any independent set in the graph chooses exactly one one literal from each clause to be true.

From 3SAT to INDSET

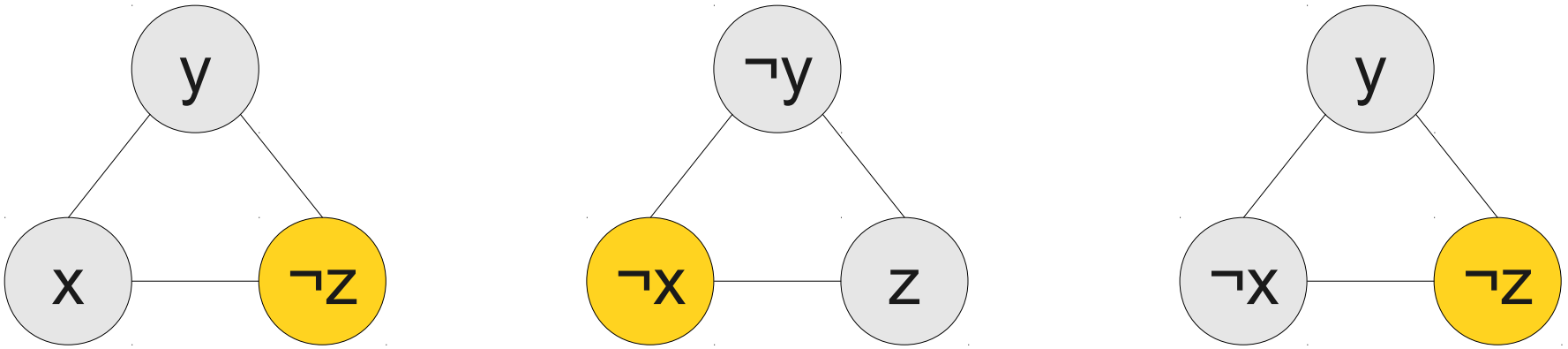
$$(x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee z) \wedge (\neg x \vee y \vee \neg z)$$



Any independent set in the graph chooses exactly one one literal from each clause to be true.

From 3SAT to INDSET

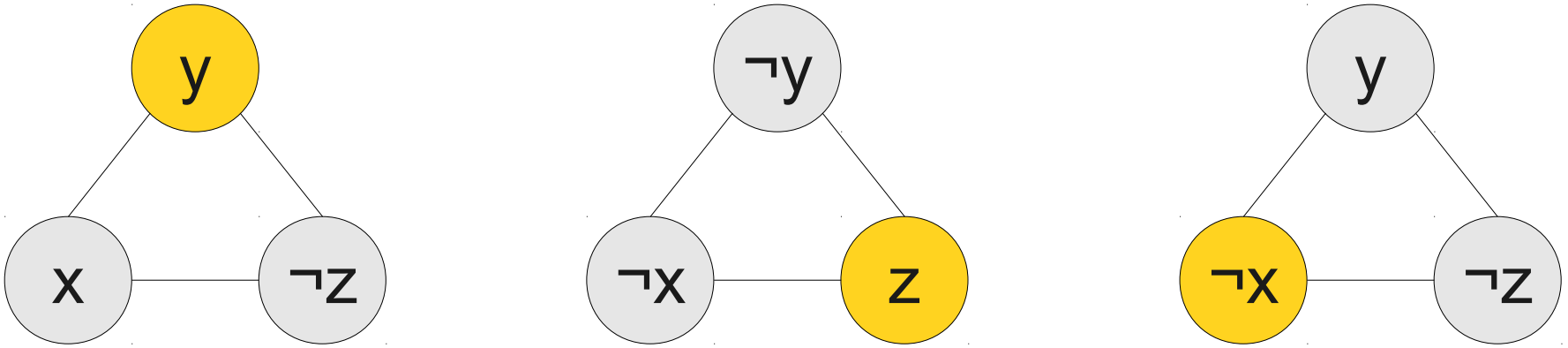
$$(x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee z) \wedge (\neg x \vee y \vee \neg z)$$



Any independent set in the graph chooses exactly one one literal from each clause to be true.

From 3SAT to INDSET

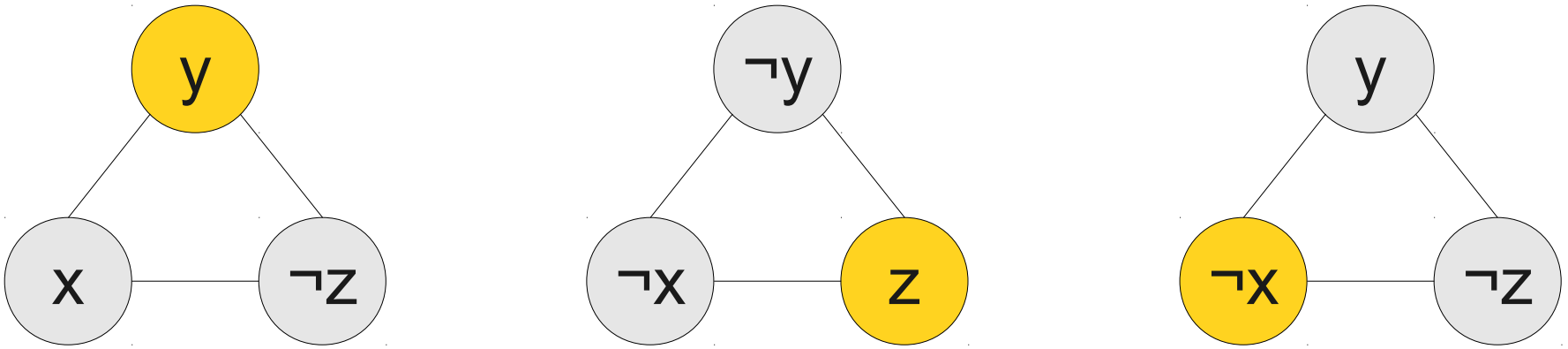
$$(x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee z) \wedge (\neg x \vee y \vee \neg z)$$



Any independent set in the graph chooses exactly one one literal from each clause to be true.

From 3SAT to INDSET

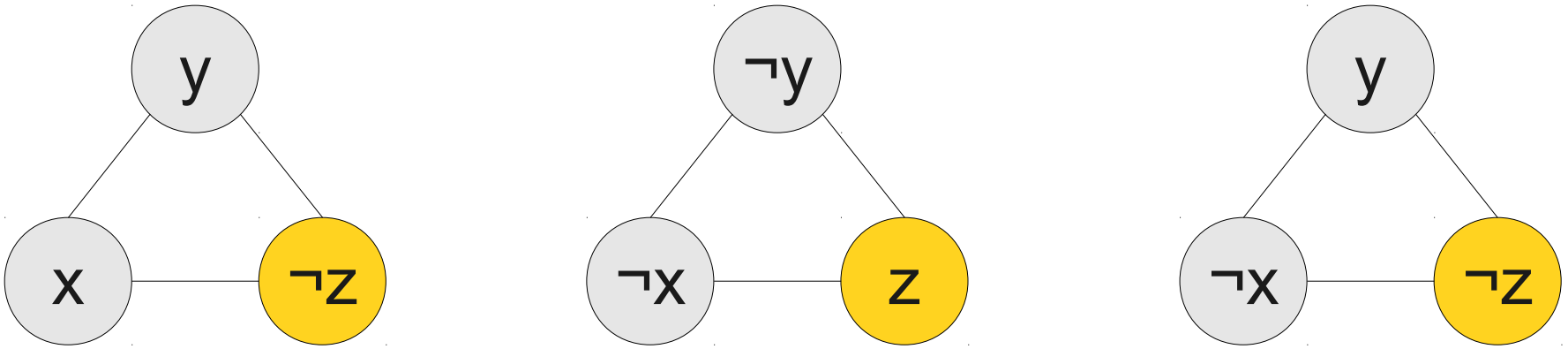
$$(x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee z) \wedge (\neg x \vee y \vee \neg z)$$



Any independent set in the graph chooses exactly one one literal from each clause to be true.

From 3SAT to INDSET

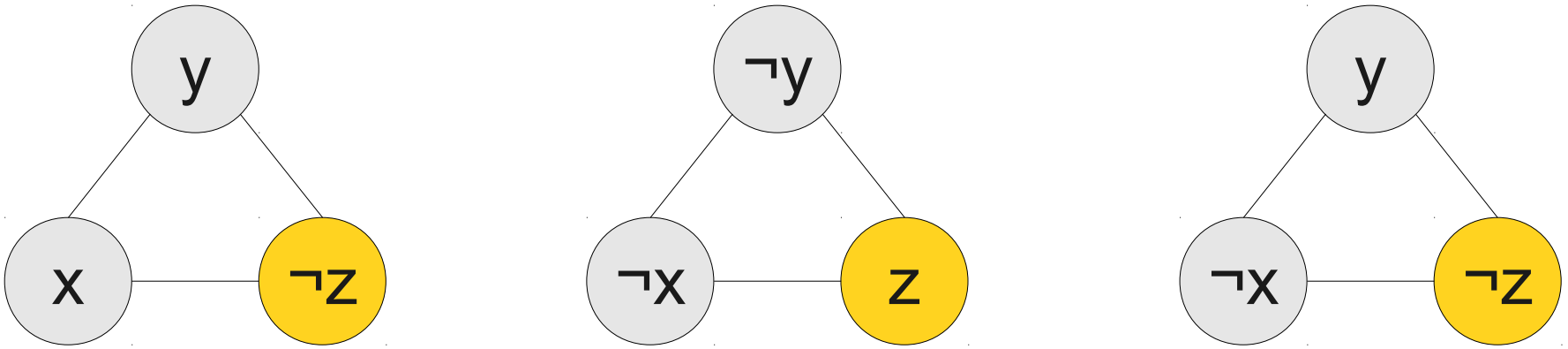
$$(x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee z) \wedge (\neg x \vee y \vee \neg z)$$



Any independent set in the graph chooses exactly one one literal from each clause to be true.

From 3SAT to INDSET

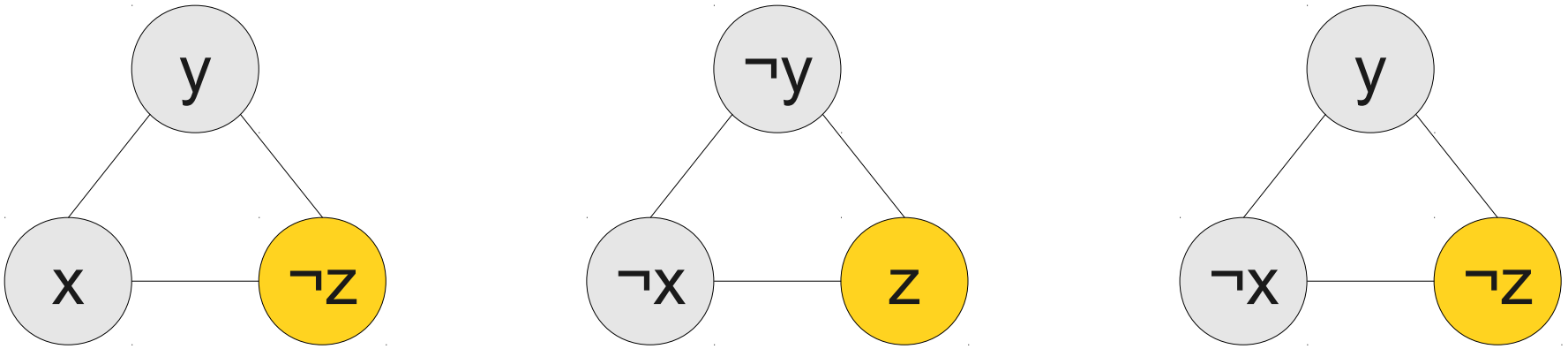
$$(x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee z) \wedge (\neg x \vee y \vee \neg z)$$



Any independent set in the graph chooses exactly one one literal from each clause to be true.

From 3SAT to INDSET

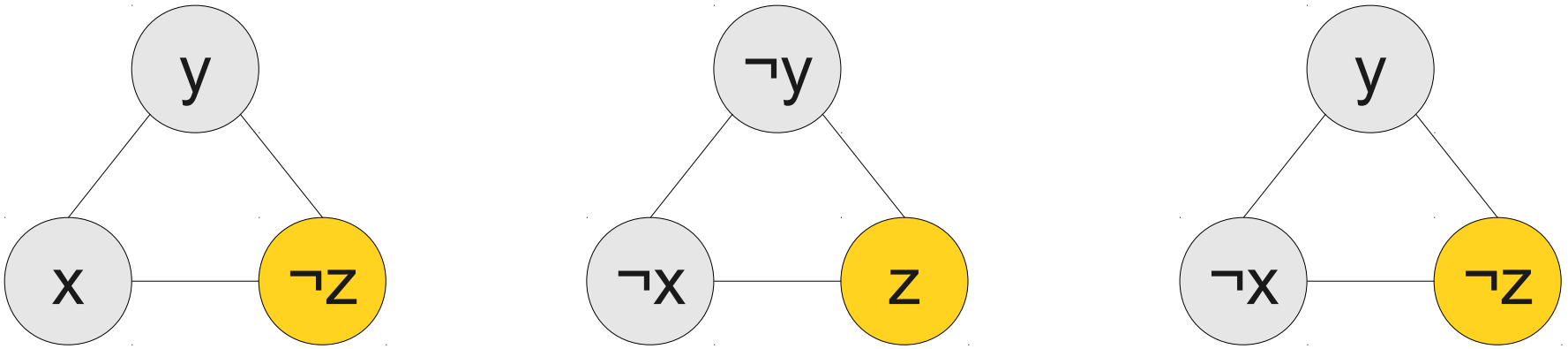
$$(x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee z) \wedge (\neg x \vee y \vee \neg z)$$



Any independent set in the graph chooses exactly one one literal from each clause to be true.

From 3SAT to INDSET

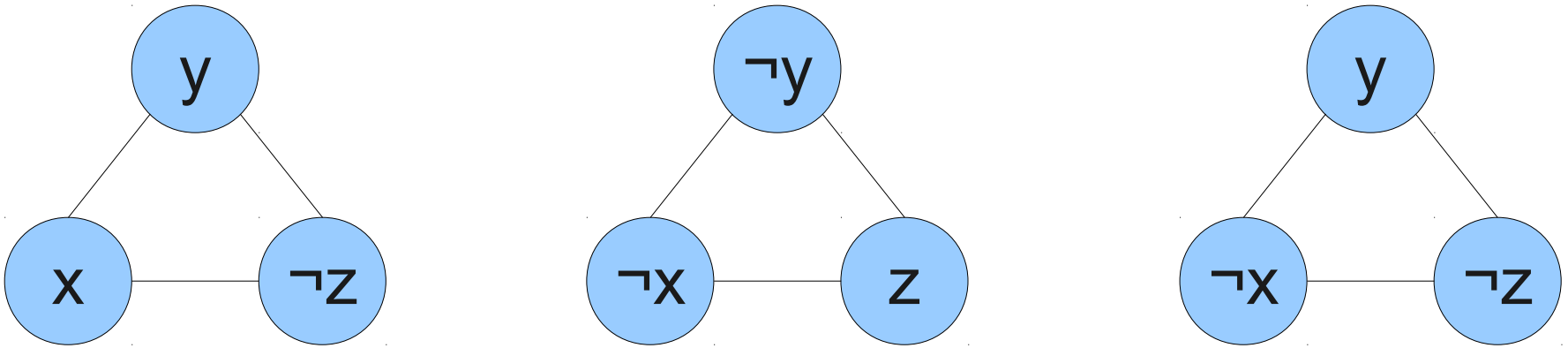
$$(x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee z) \wedge (\neg x \vee y \vee \neg z)$$



We need a way to ensure that
we don't pick a literal and its negation!

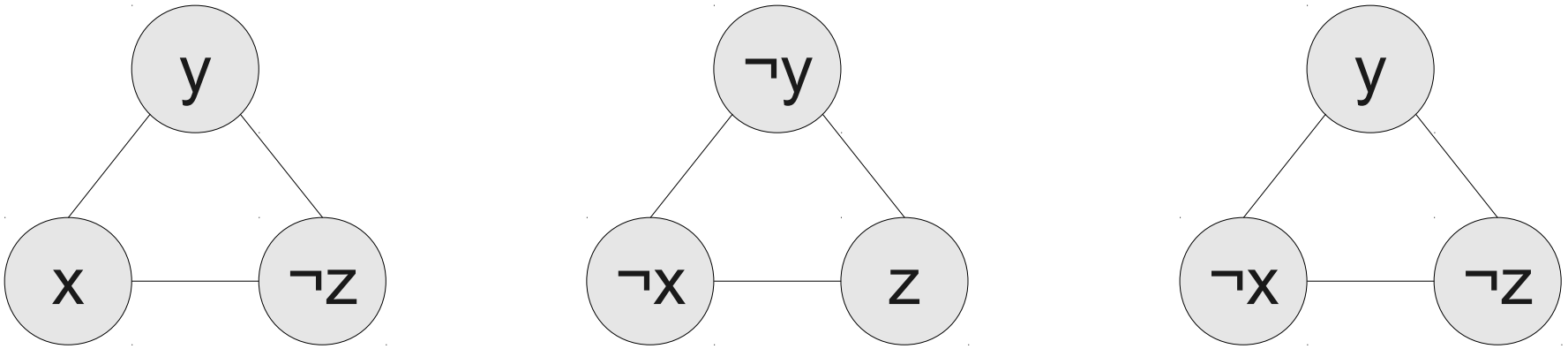
From 3SAT to INDSET

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



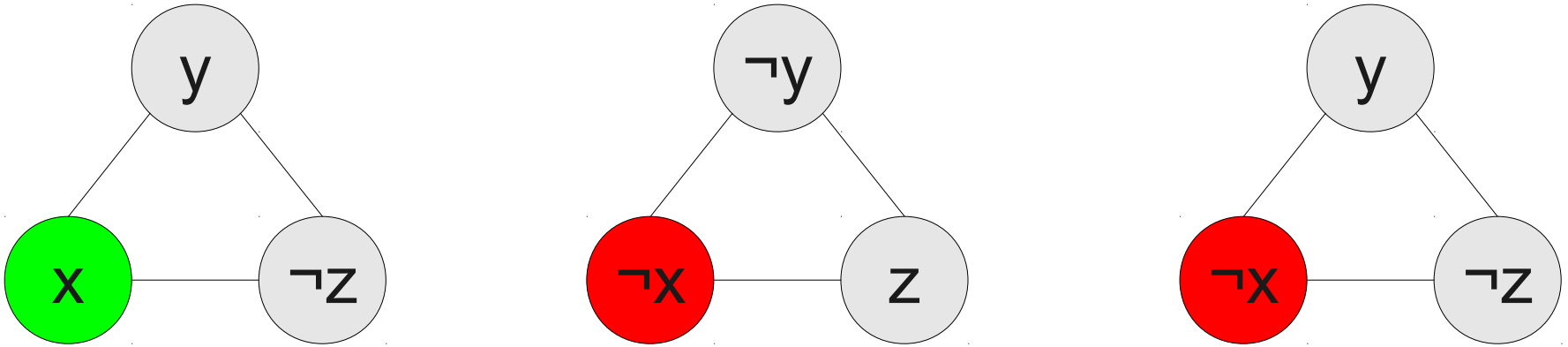
From 3SAT to INDSET

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

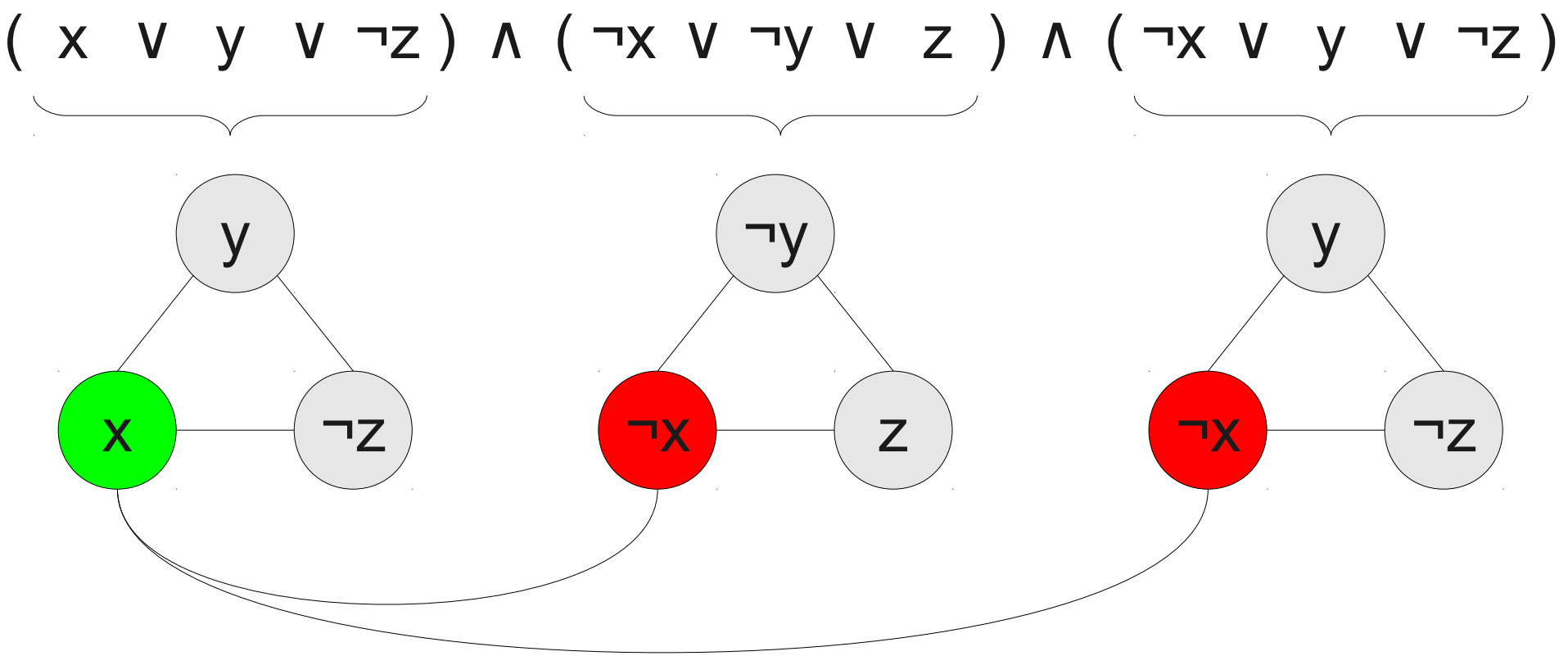


From 3SAT to INDSET

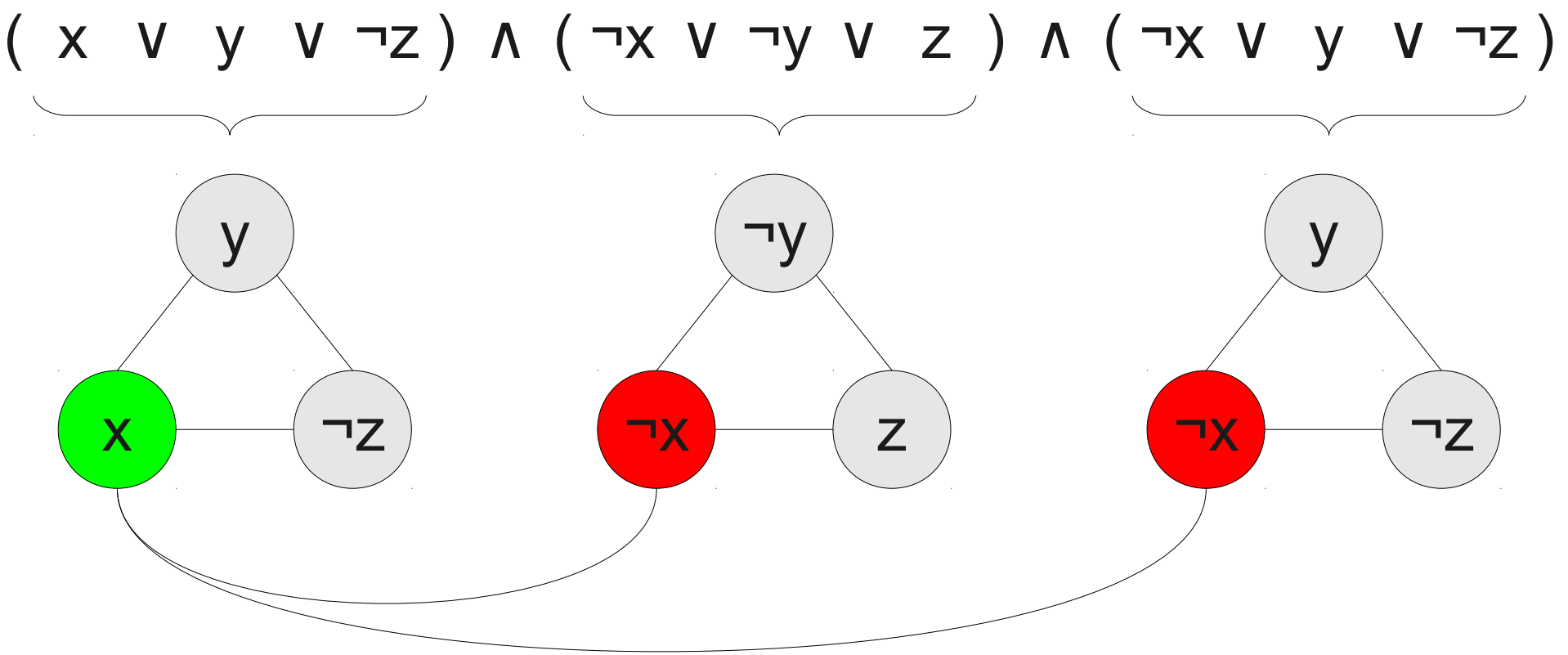
$$(\ x \ \vee \ y \ \vee \neg z\ ) \wedge (\ \neg x \vee \neg y \vee \ z\ ) \wedge (\ \neg x \vee \ y \ \vee \neg z\ )$$



From 3SAT to INDSET

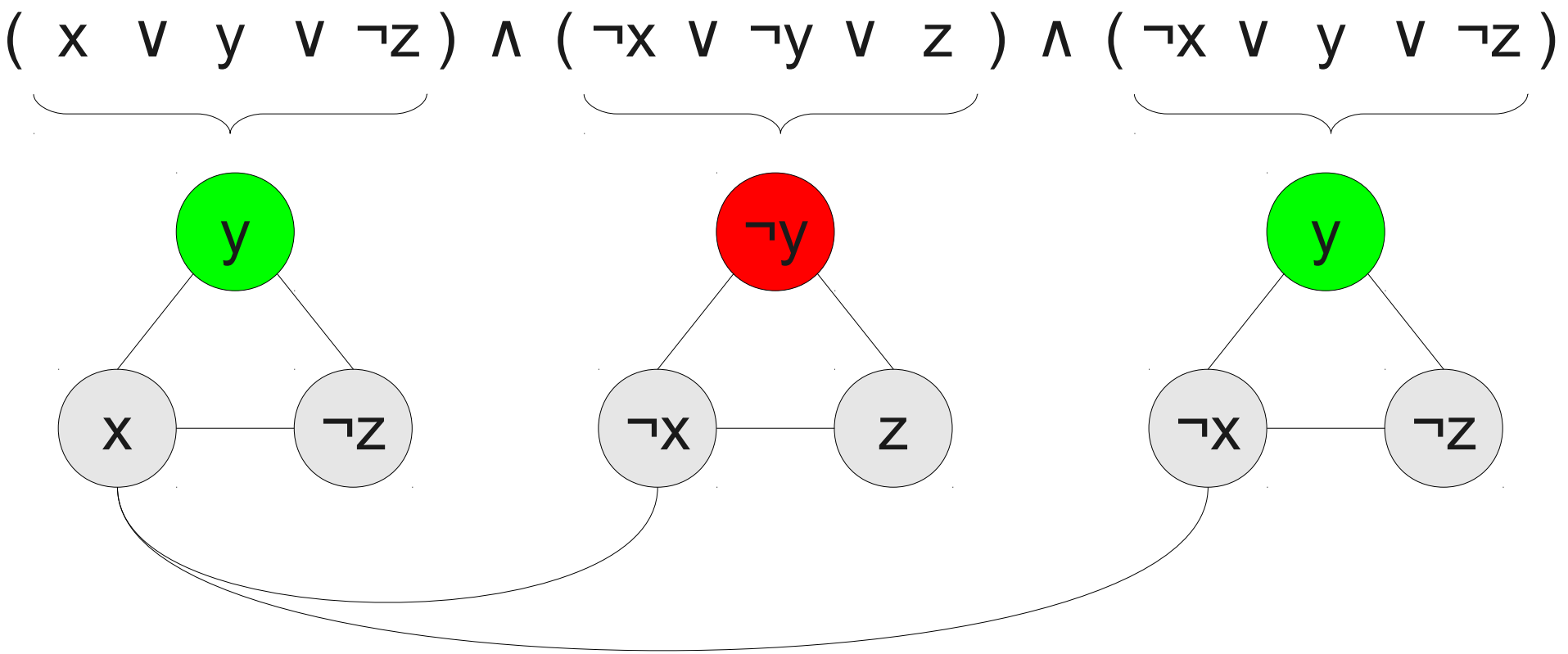


From 3SAT to INDSET

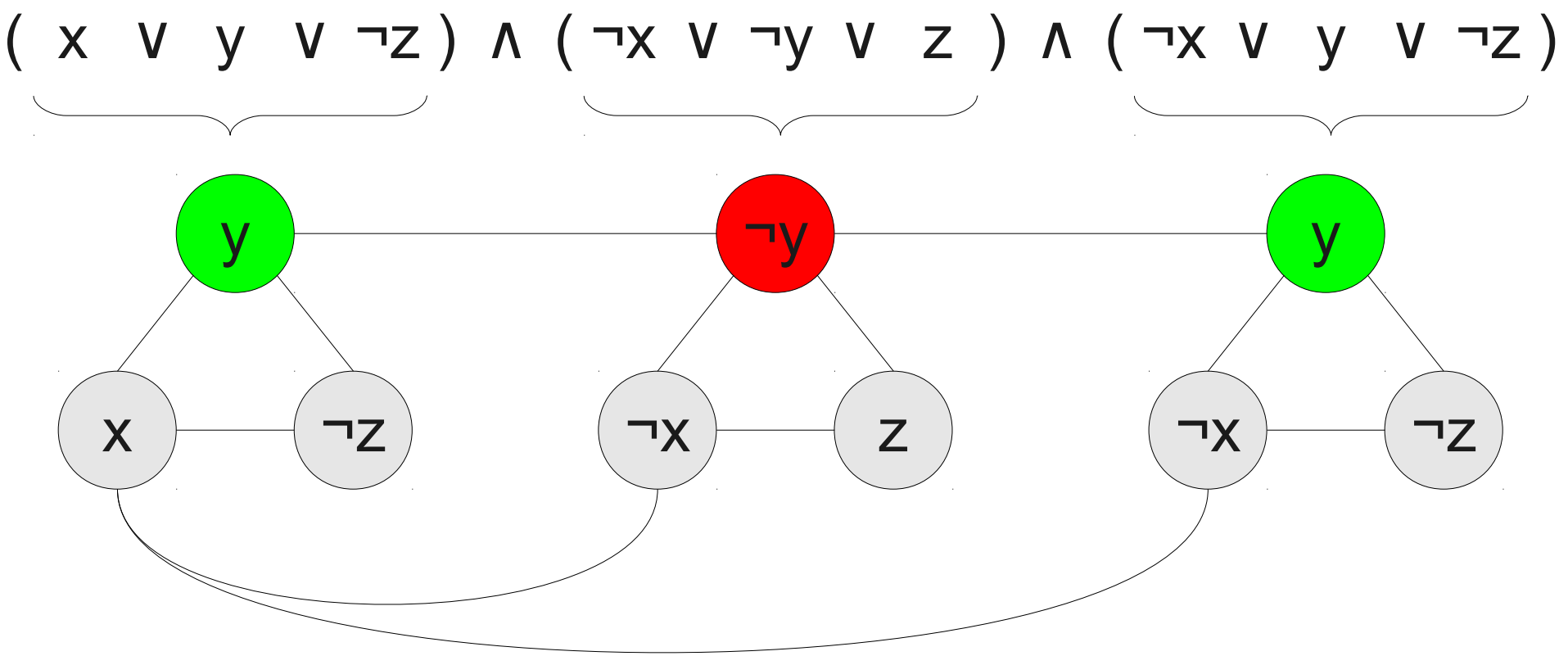


No independent set in this graph can choose any x and ¬x node at the same time!

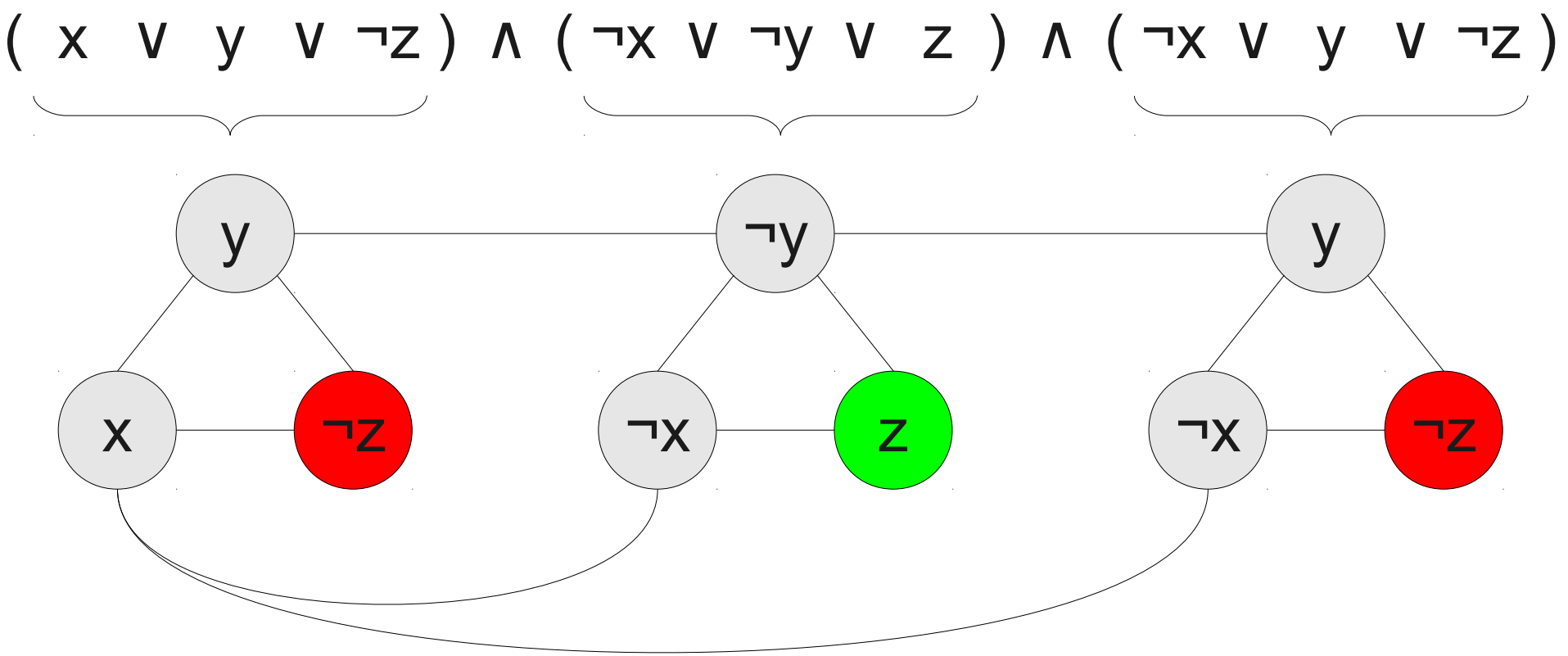
From 3SAT to INDSET



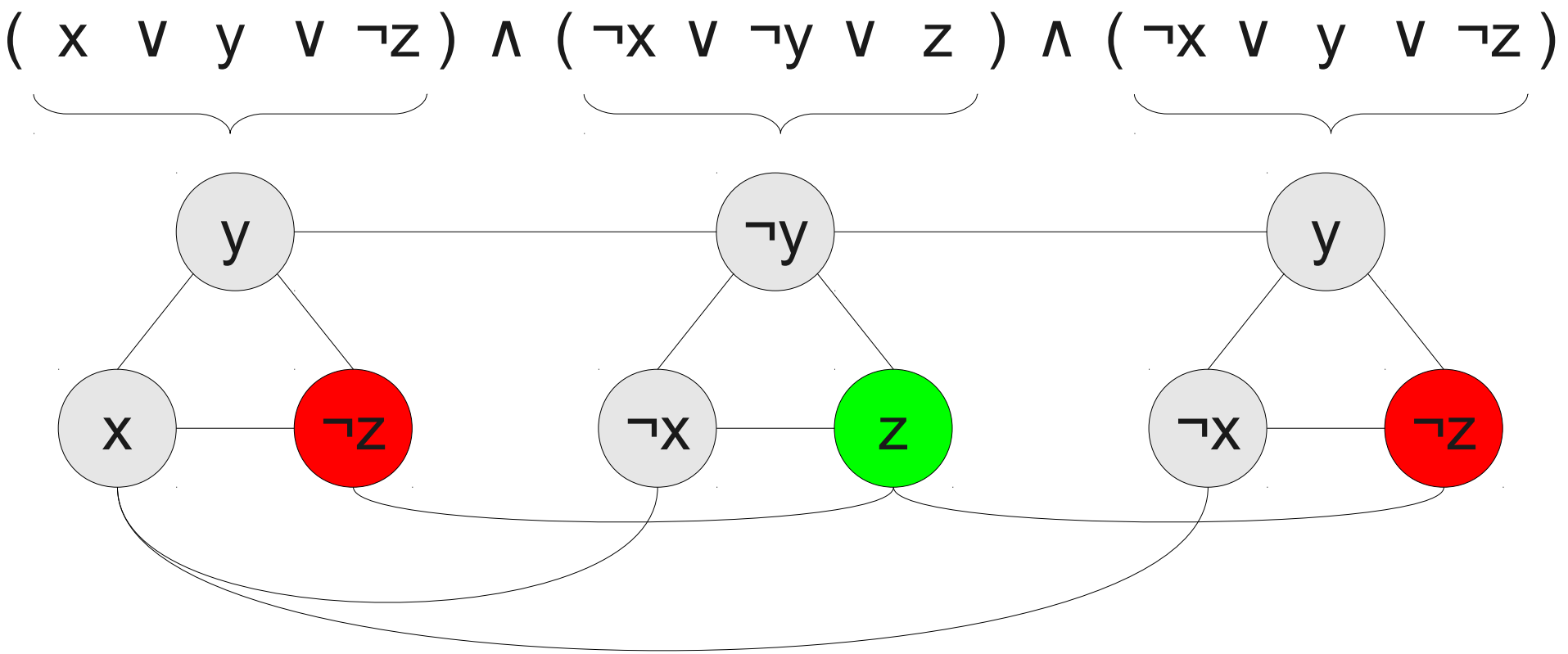
From 3SAT to INDSET



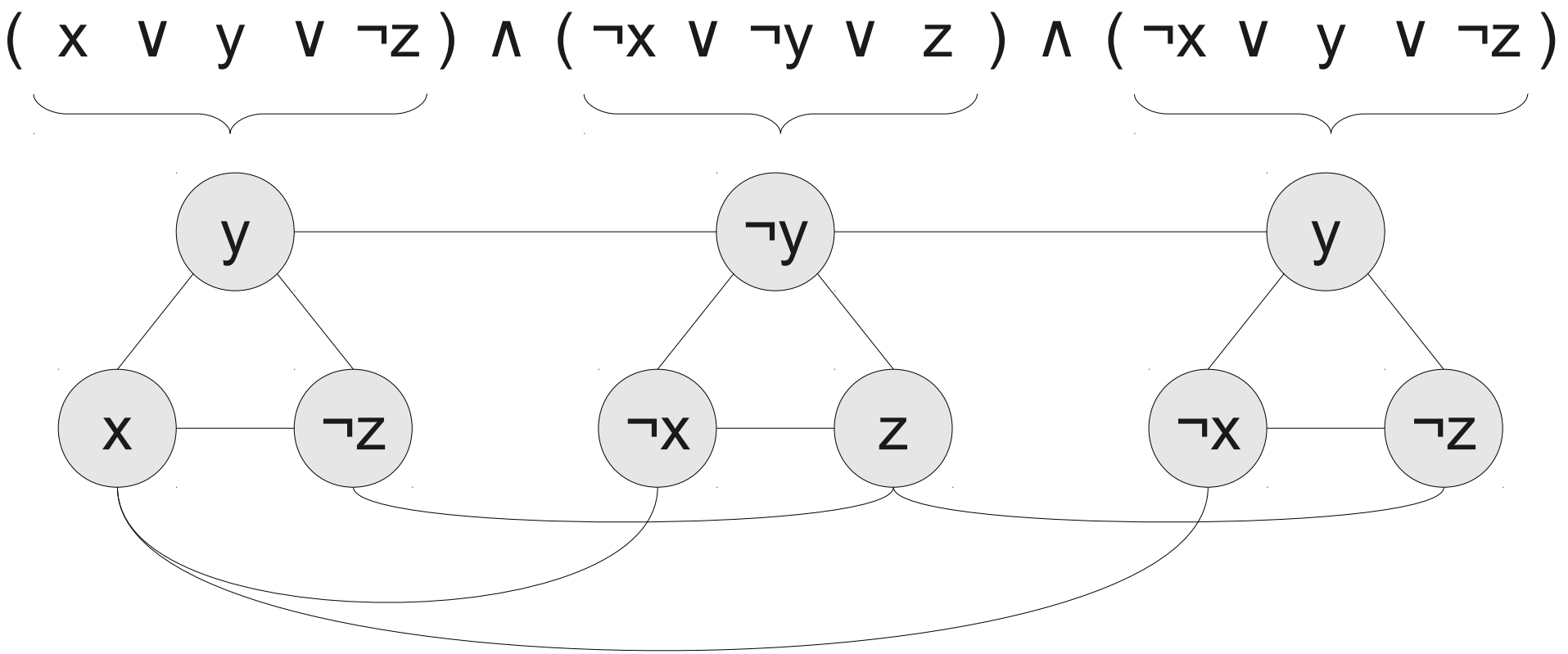
From 3SAT to INDSET



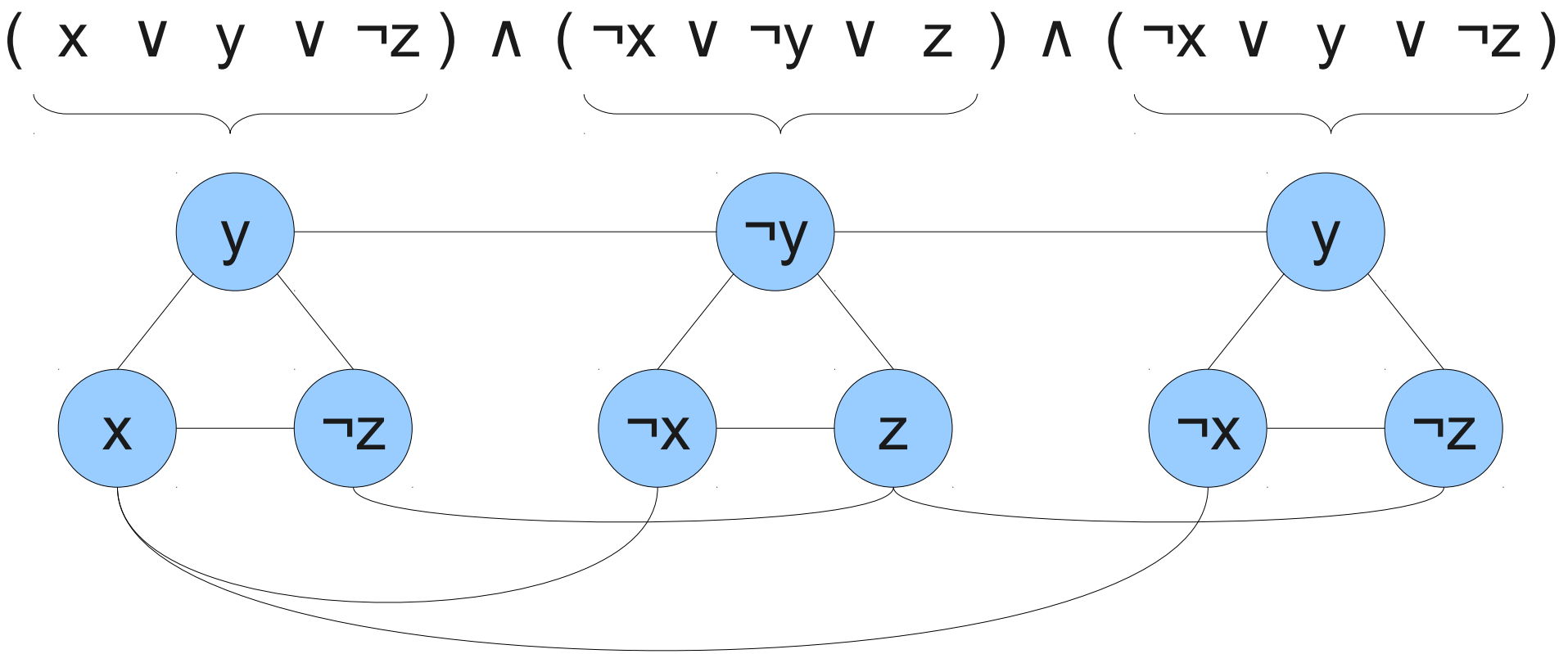
From 3SAT to INDSET



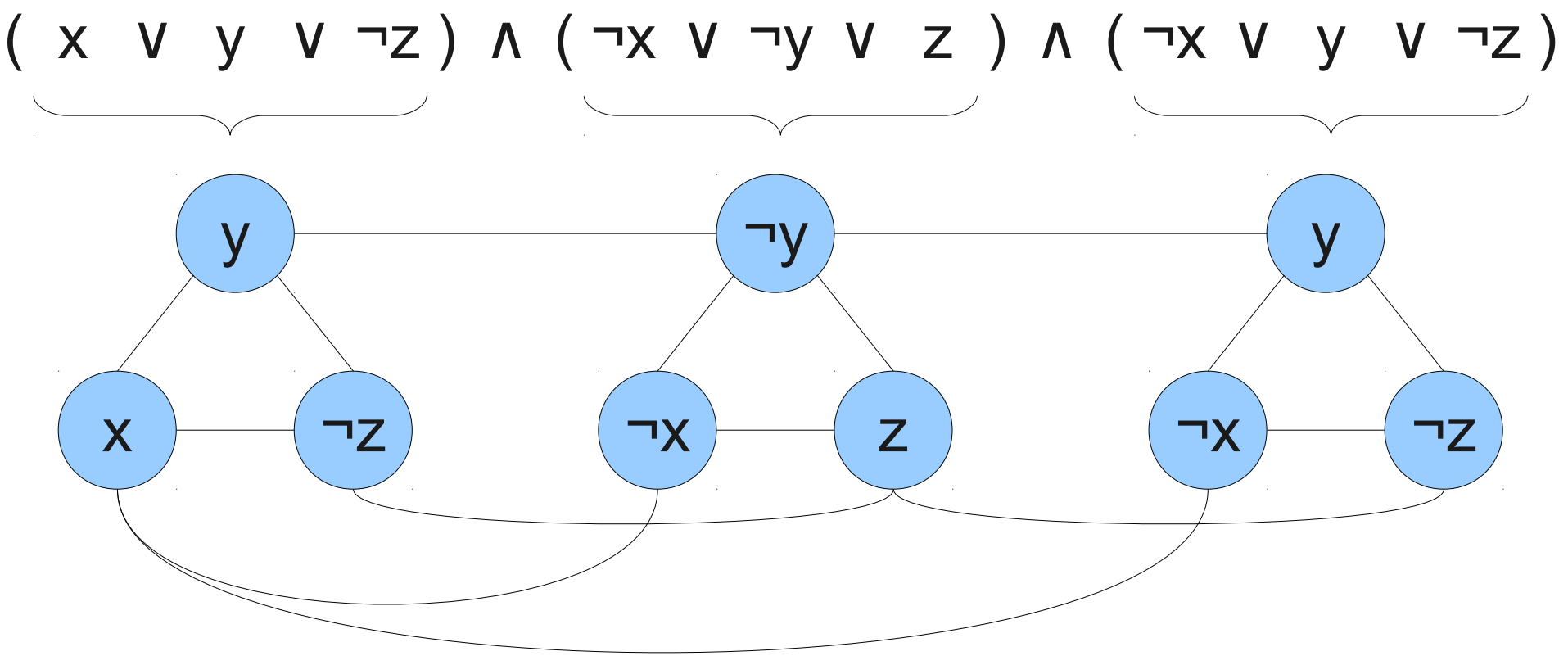
From 3SAT to INDSET



From 3SAT to INDSET

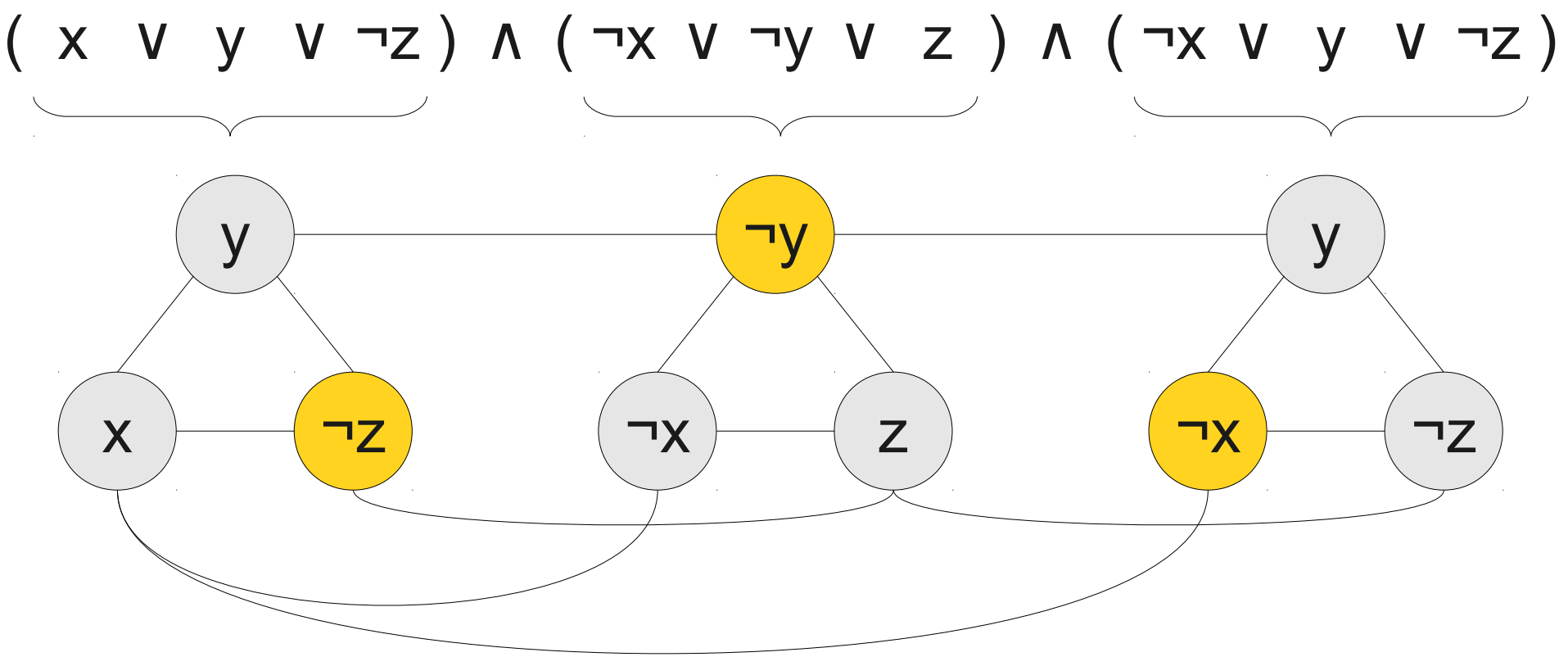


From 3SAT to INDSET



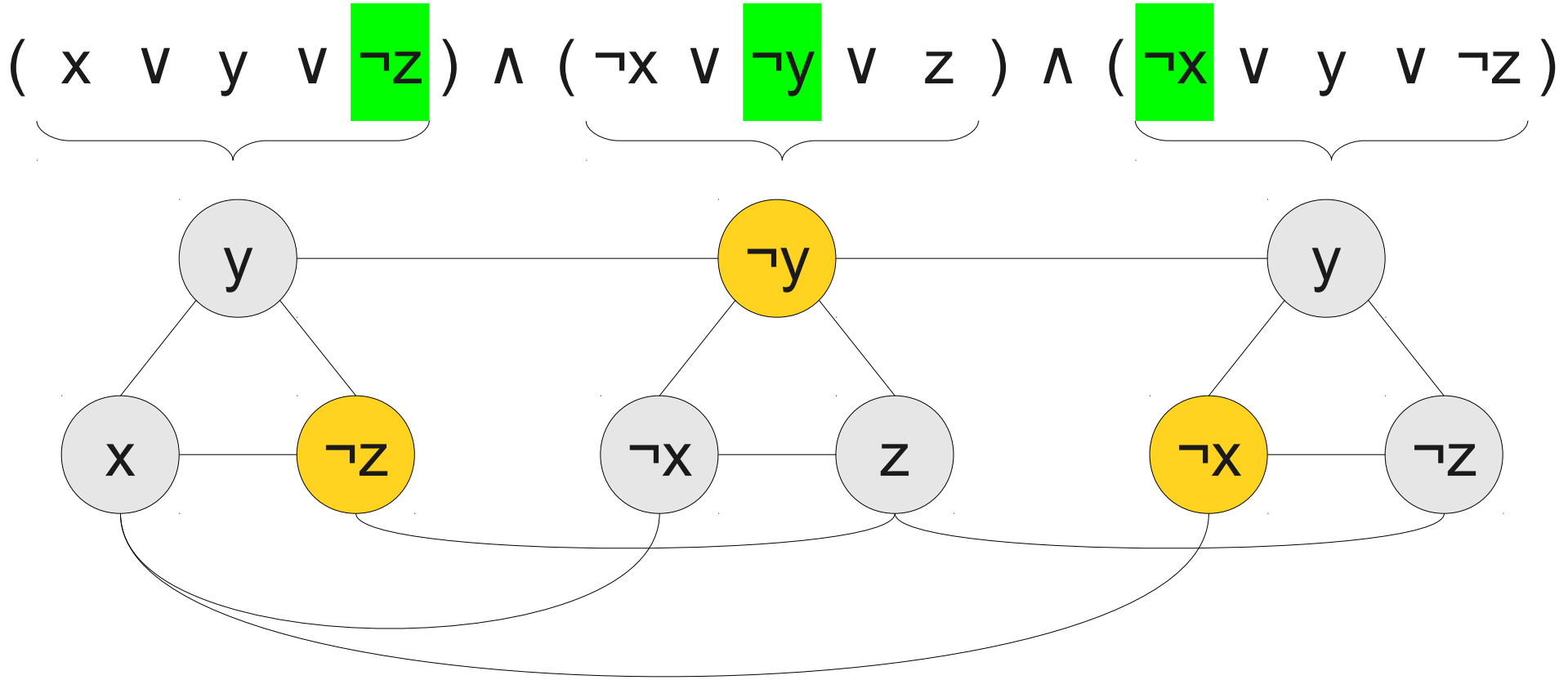
If this graph has an independent set of size three, the original formula is satisfiable.

From 3SAT to INDSET



If this graph has an independent set of size three, the original formula is satisfiable.

From 3SAT to INDSET

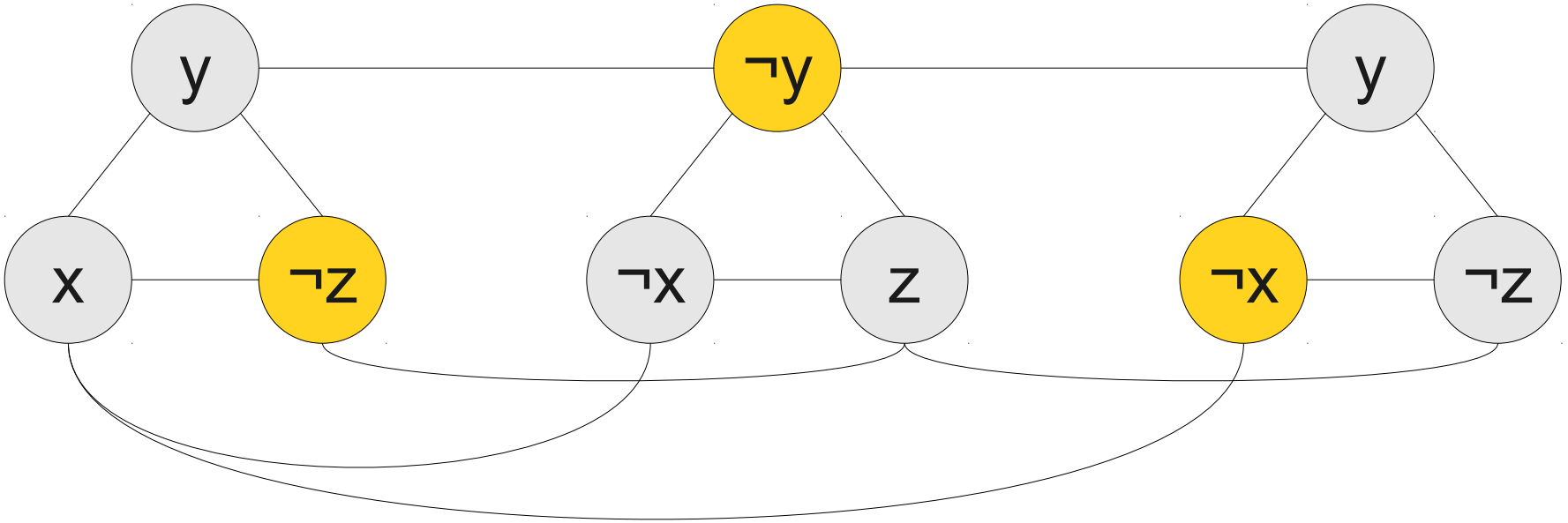


If this graph has an independent set of size three, the original formula is satisfiable.

From 3SAT to INDSET

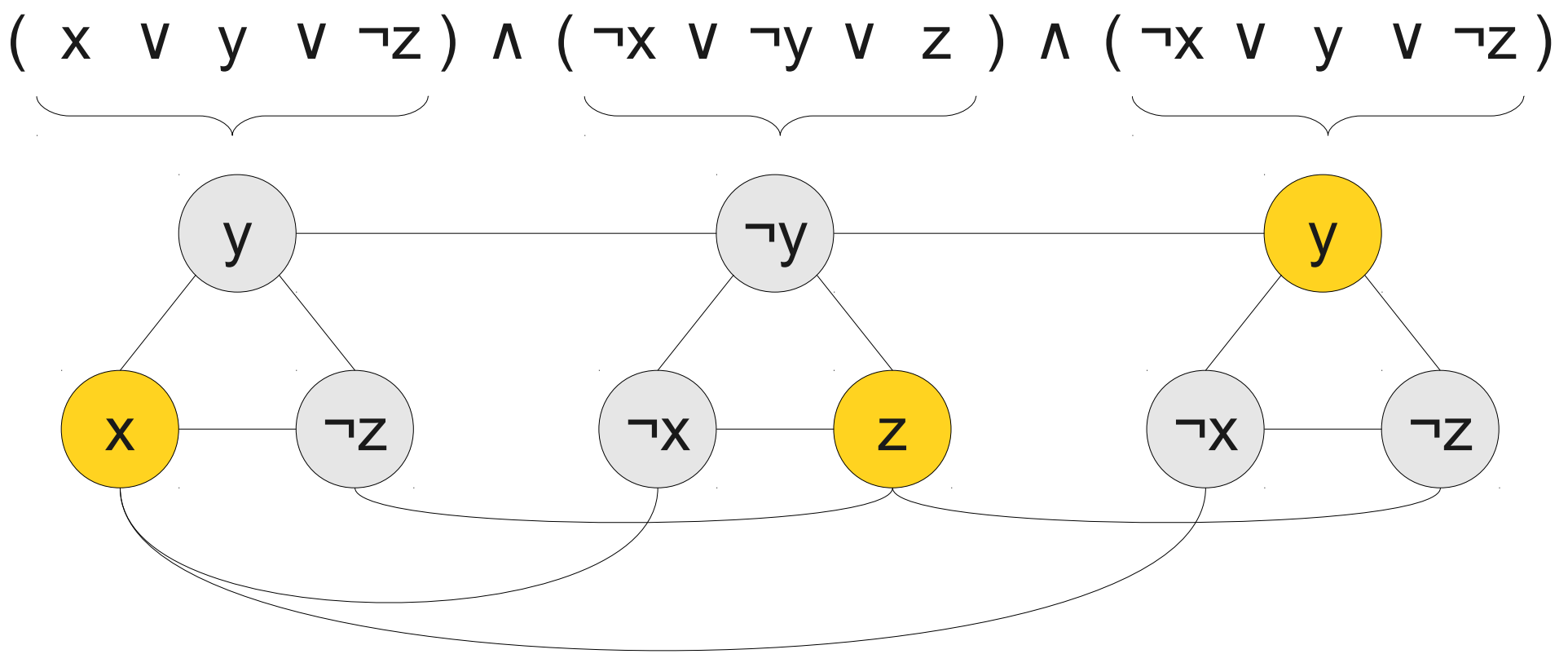
x = false, y = false, z = false.

(x ∨ y ∨ ¬z) ∧ (¬x ∨ ¬y ∨ z) ∧ (¬x ∨ y ∨ ¬z)



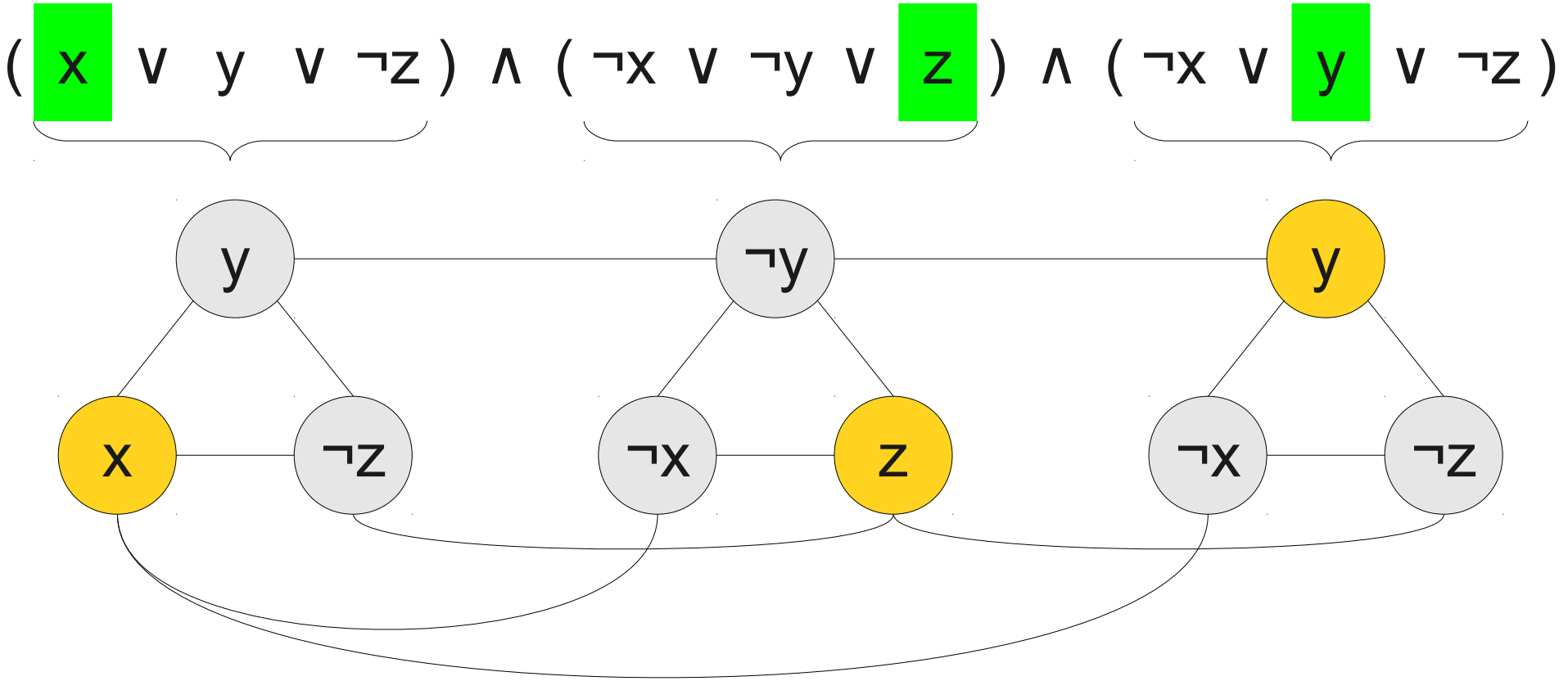
If this graph has an independent set of size three, the original formula is satisfiable.

From 3SAT to INDSET



If this graph has an independent set of size three, the original formula is satisfiable.

From 3SAT to INDSET

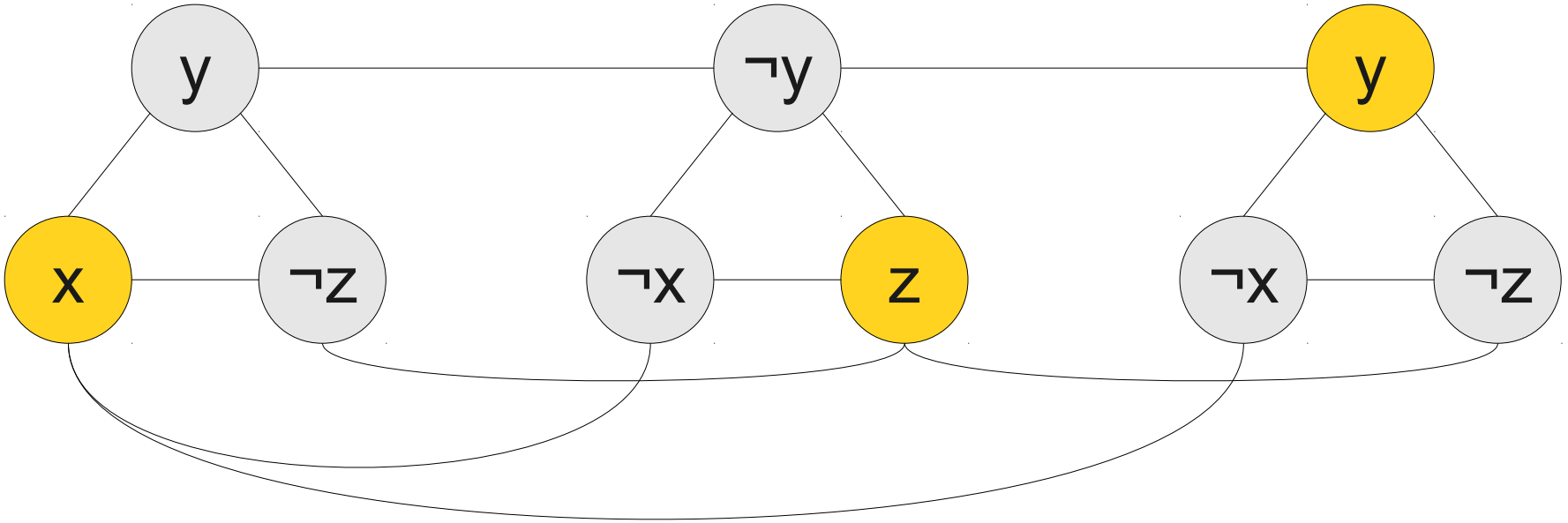


If this graph has an independent set of size three, the original formula is satisfiable.

From 3SAT to INDSET

x = true, y = true, z = true.

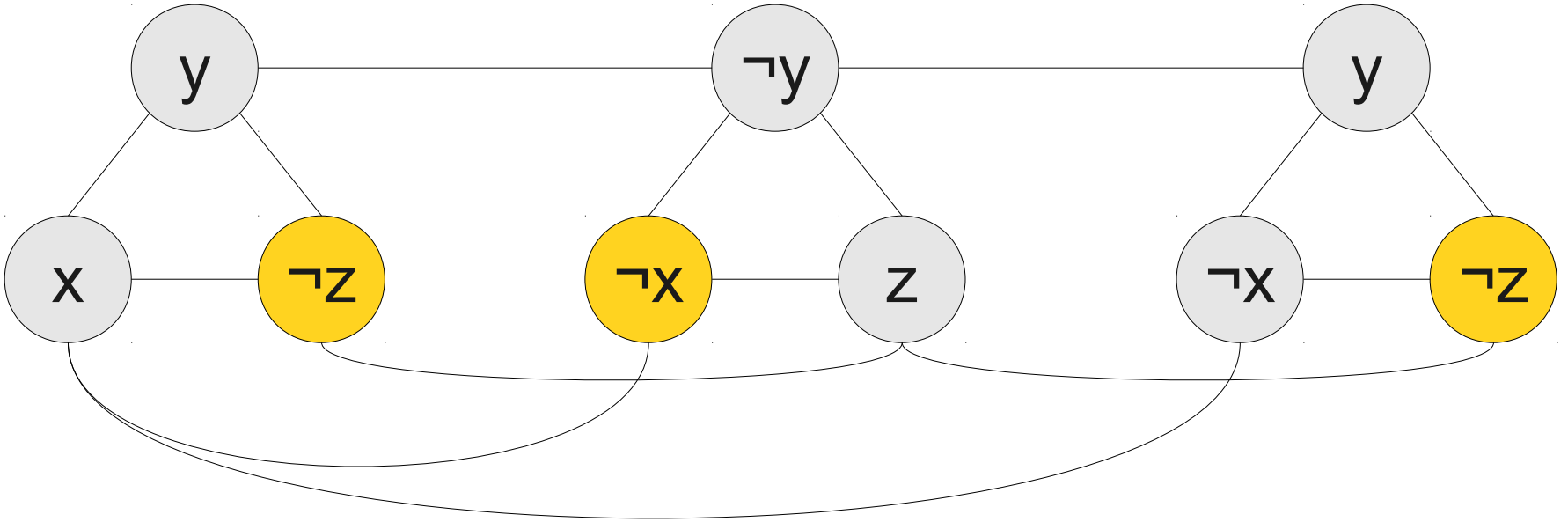
(x ∨ y ∨ ¬z) ∧ (¬x ∨ ¬y ∨ z) ∧ (¬x ∨ y ∨ ¬z)



If this graph has an independent set of size three, the original formula is satisfiable.

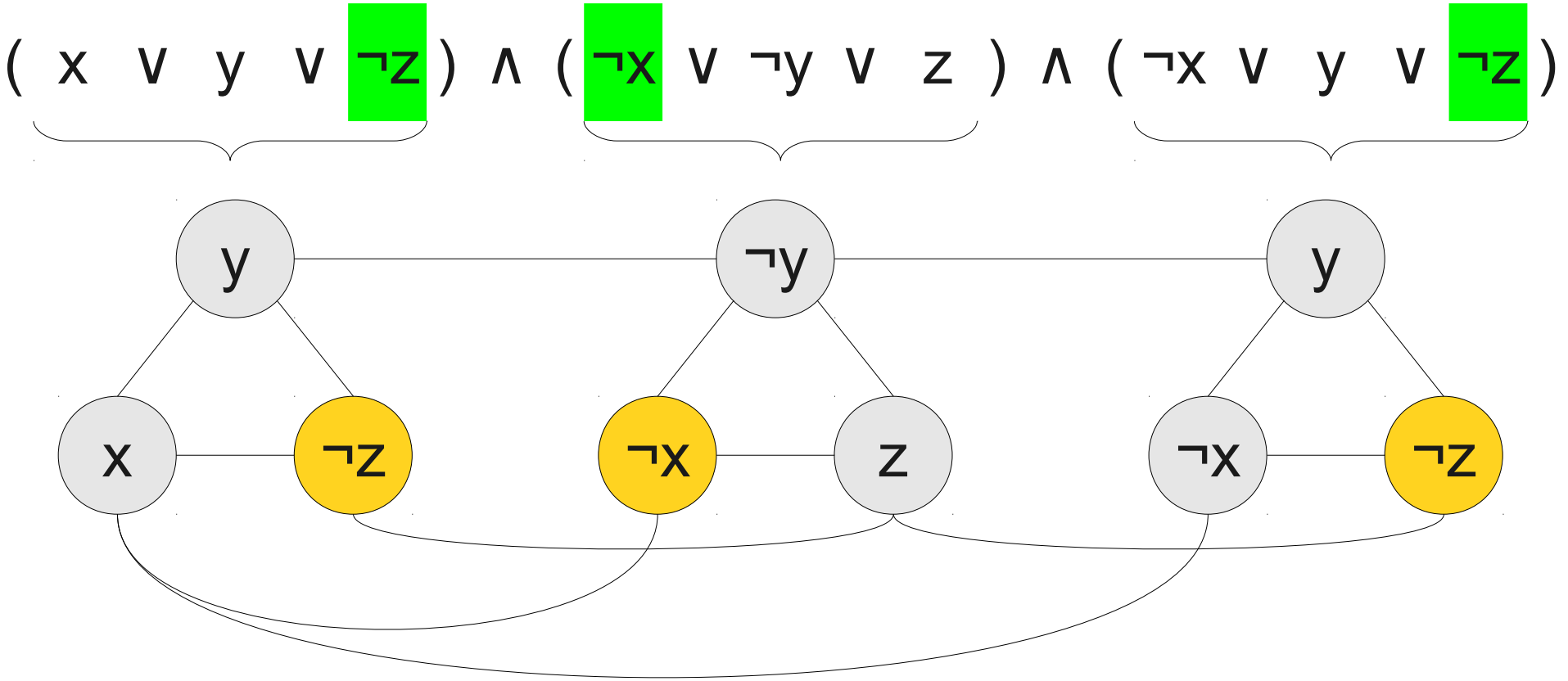
From 3SAT to INDSET

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



If this graph has an independent set of size three, the original formula is satisfiable.

From 3SAT to INDSET

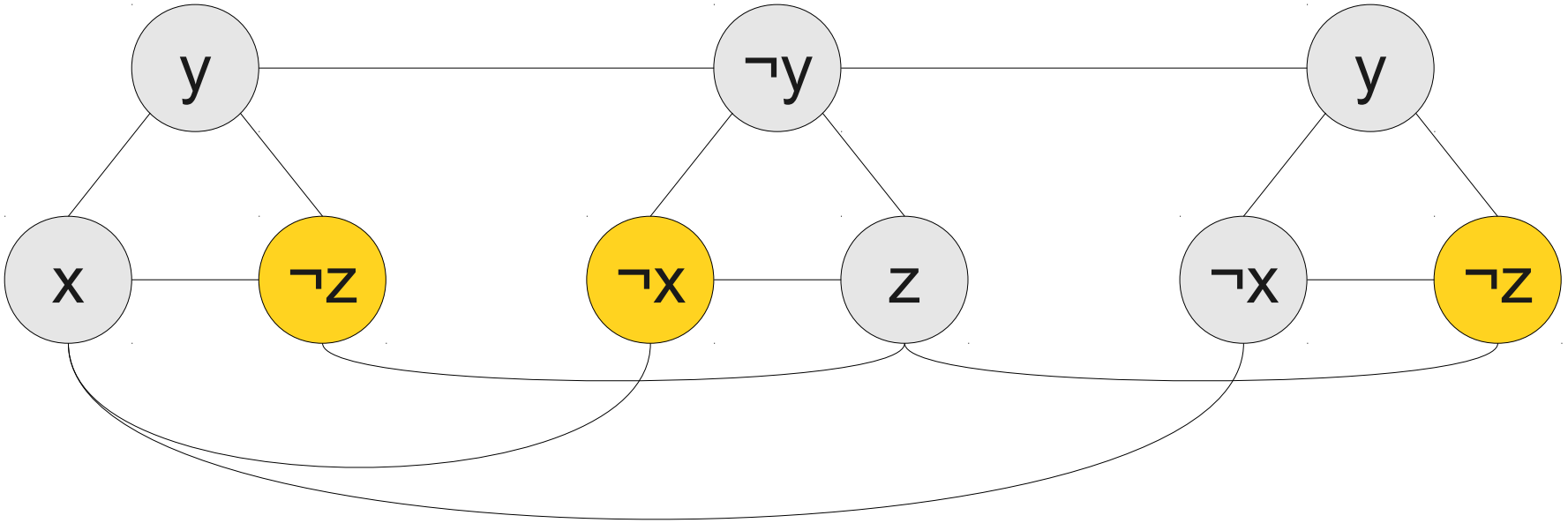


If this graph has an independent set of size three, the original formula is satisfiable.

From 3SAT to INDSET

x = false, y = ??, z = false.

(x ∨ y ∨ ¬z) ∧ (¬x ∨ ¬y ∨ z) ∧ (¬x ∨ y ∨ ¬z)

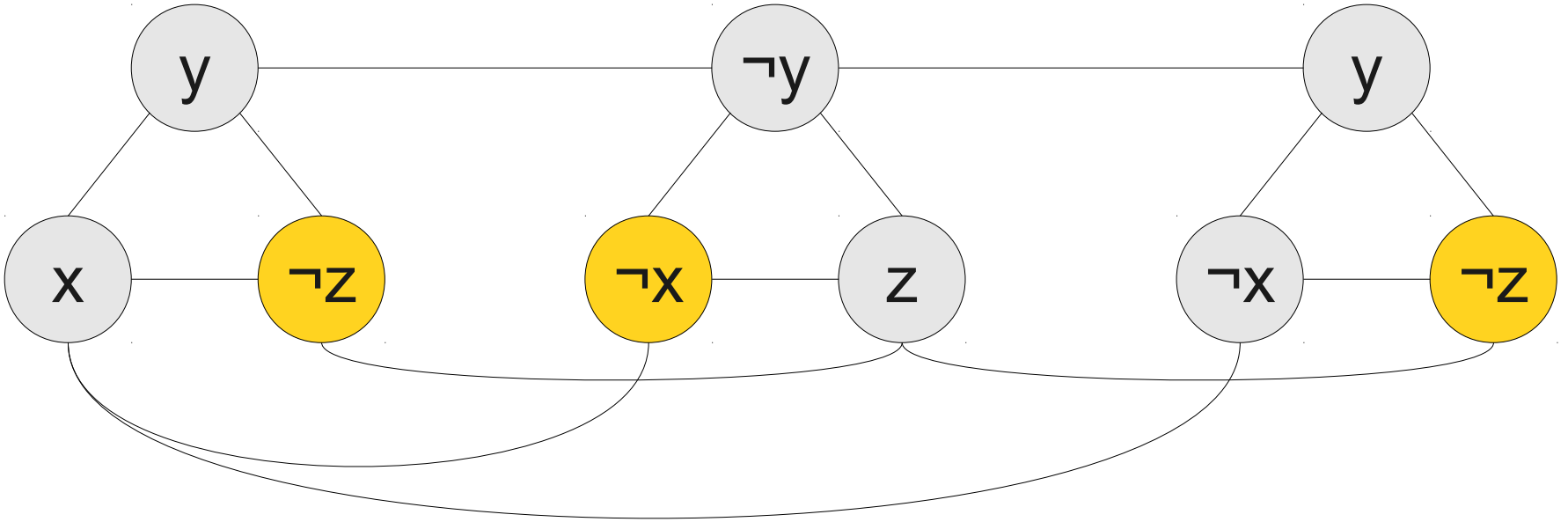


If this graph has an independent set of size three, the original formula is satisfiable.

From 3SAT to INDSET

x = false, y = true, z = false.

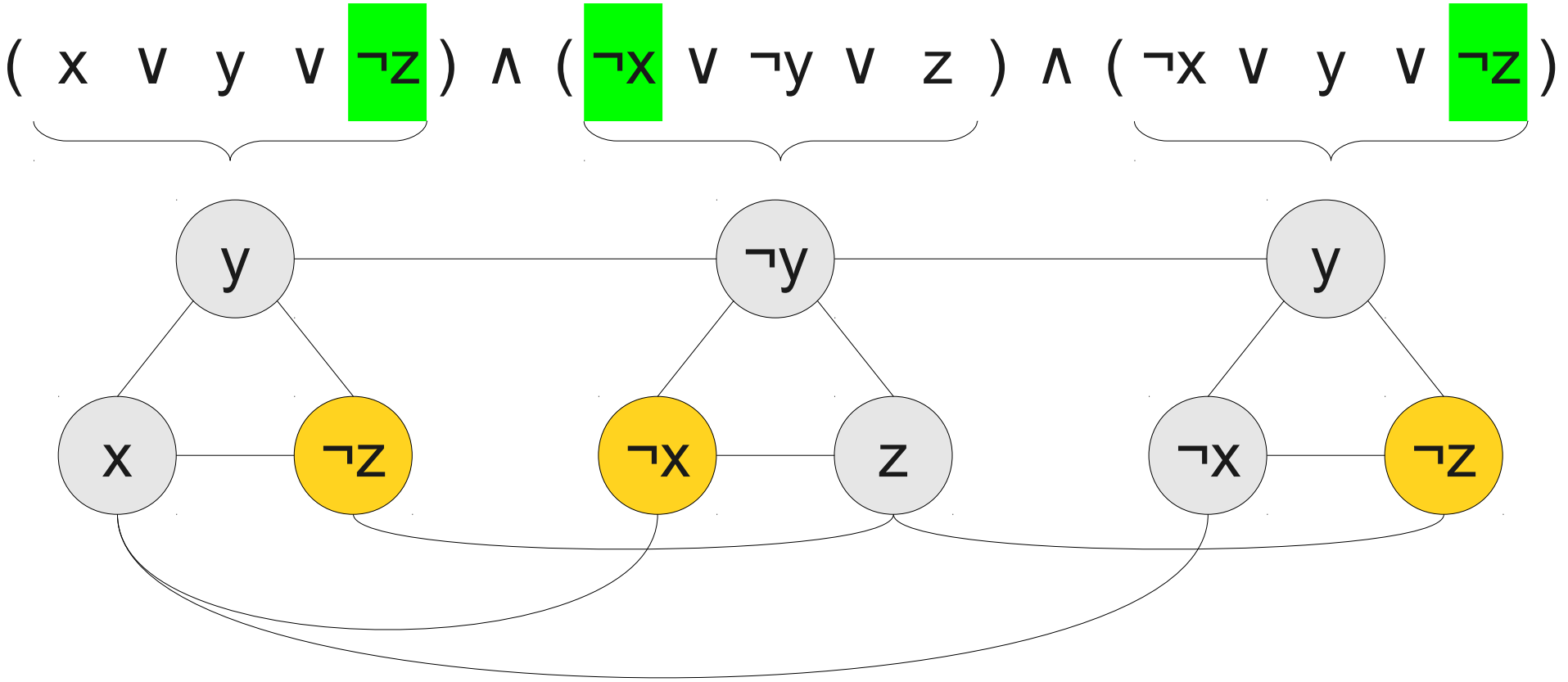
(x ∨ y ∨ ¬z) ∧ (¬x ∨ ¬y ∨ z) ∧ (¬x ∨ y ∨ ¬z)



If this graph has an independent set of size three, the original formula is satisfiable.

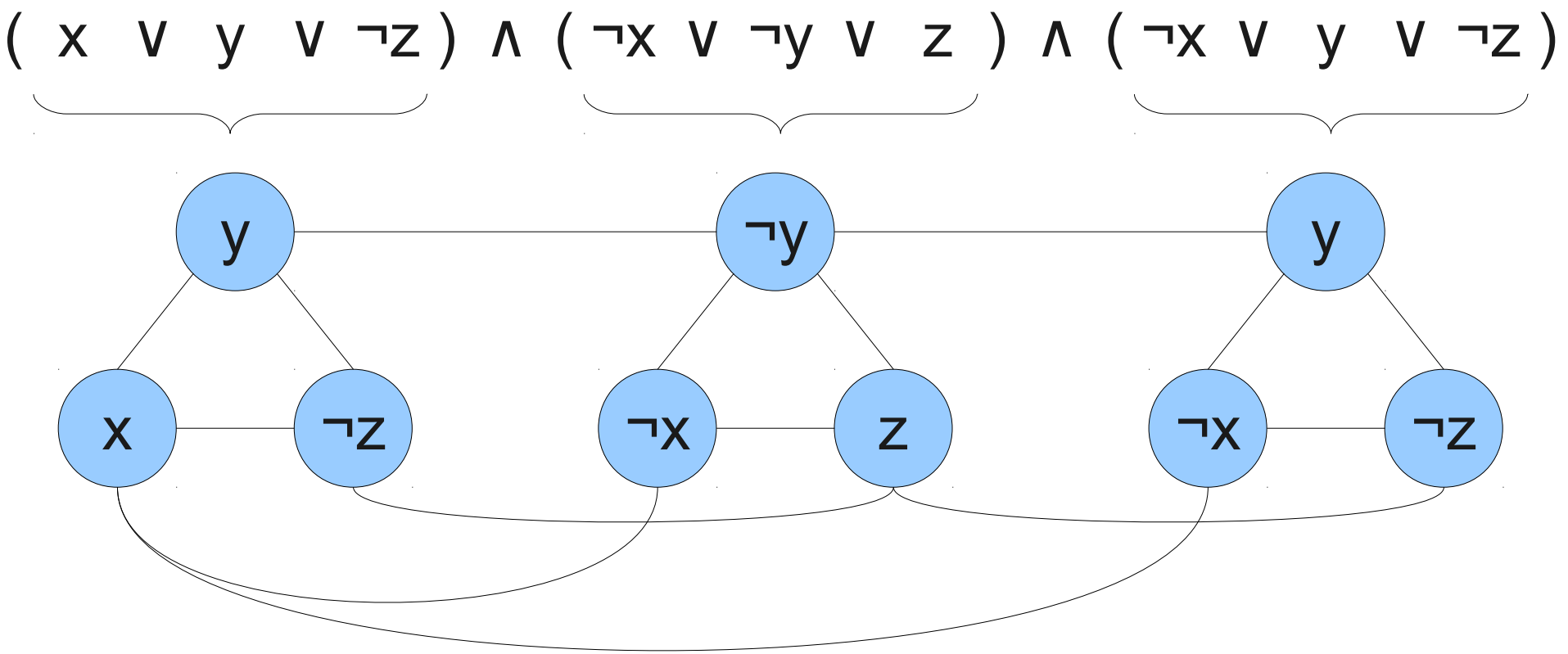
From 3SAT to INDSET

x = false, y = false, z = false.

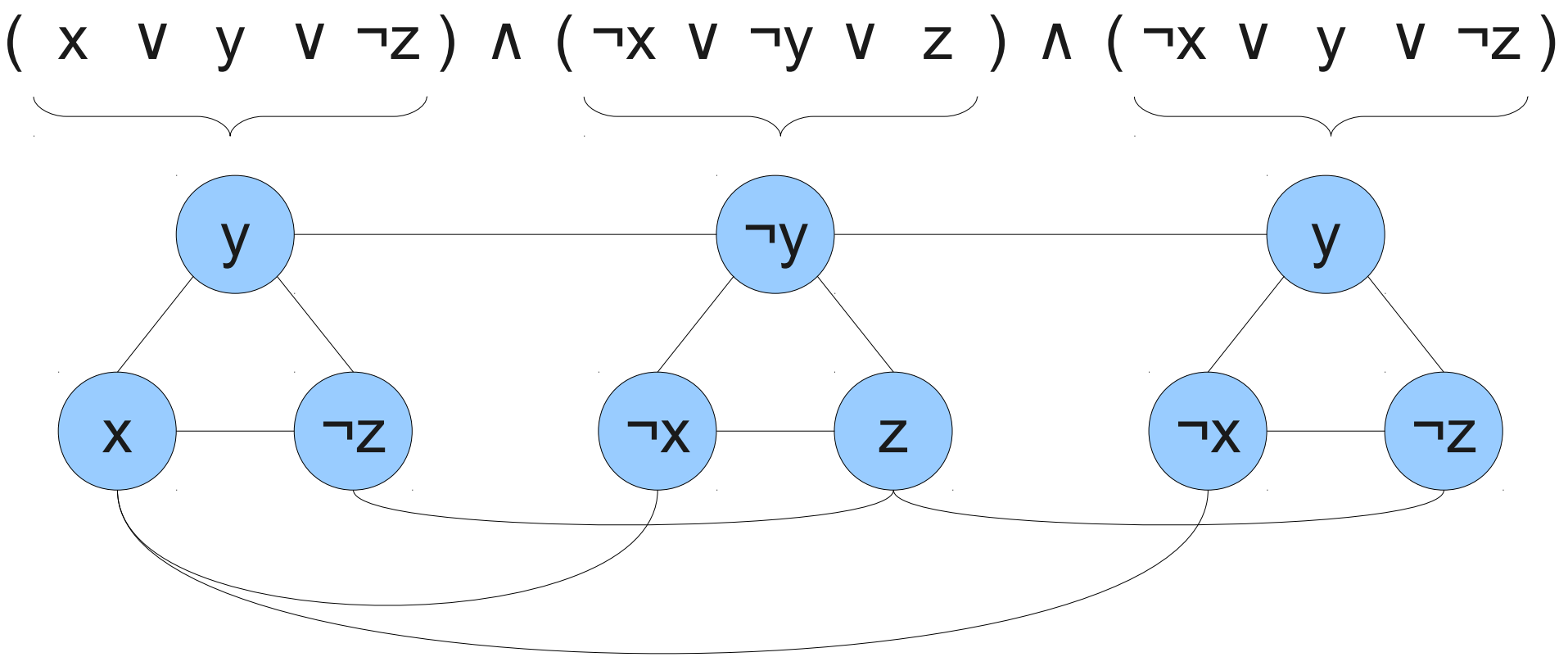


If this graph has an independent set of size three, the original formula is satisfiable.

From 3SAT to INDSET



From 3SAT to INDSET

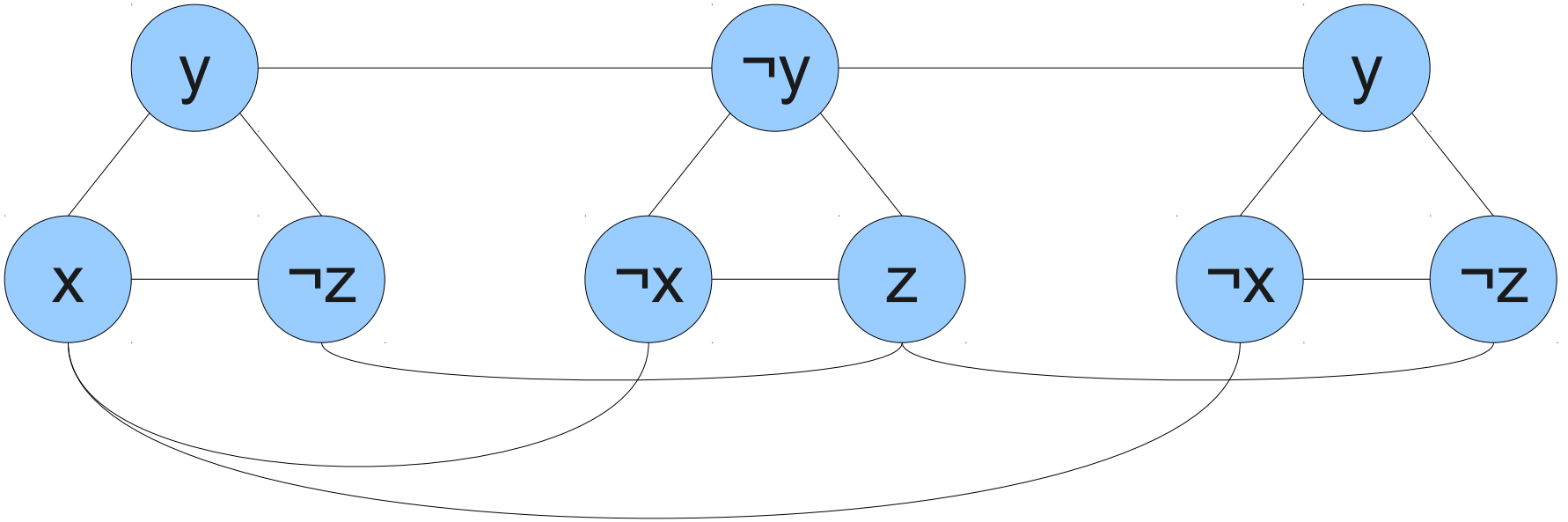


If the original formula is satisfiable,
this graph has an independent set of size three.

From 3SAT to INDSET

x = false, y = true, z = false.

(x ∨ y ∨ ¬z) ∧ (¬x ∨ ¬y ∨ z) ∧ (¬x ∨ y ∨ ¬z)

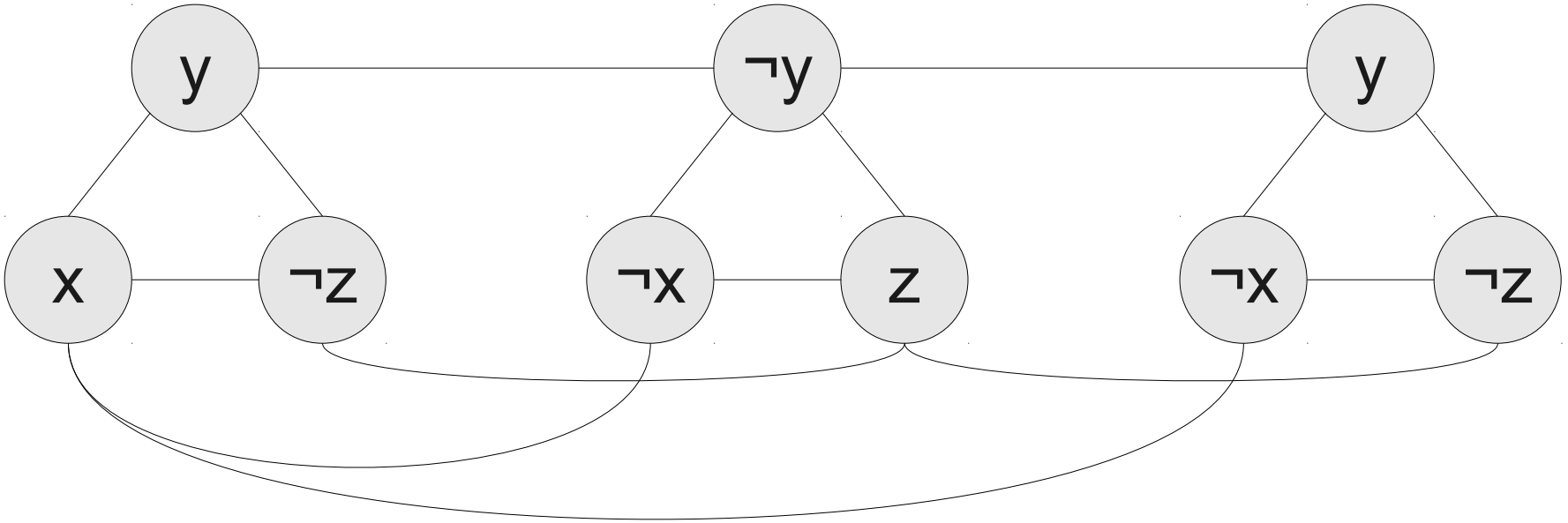


If the original formula is satisfiable,
this graph has an independent set of size three.

From 3SAT to INDSET

x = false, y = true, z = false.

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

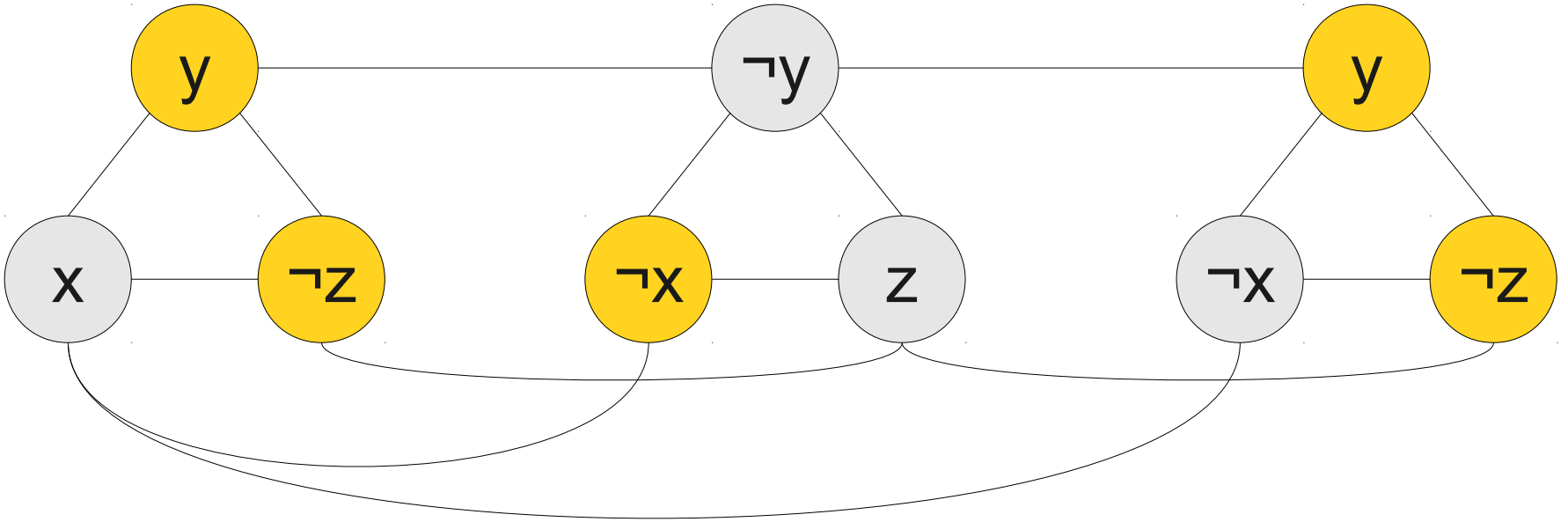


If the original formula is satisfiable,
this graph has an independent set of size three.

From 3SAT to INDSET

x = false, y = true, z = false.

(x ∨ y ∨ ¬z) ∧ (¬x ∨ ¬y ∨ z) ∧ (¬x ∨ y ∨ ¬z)

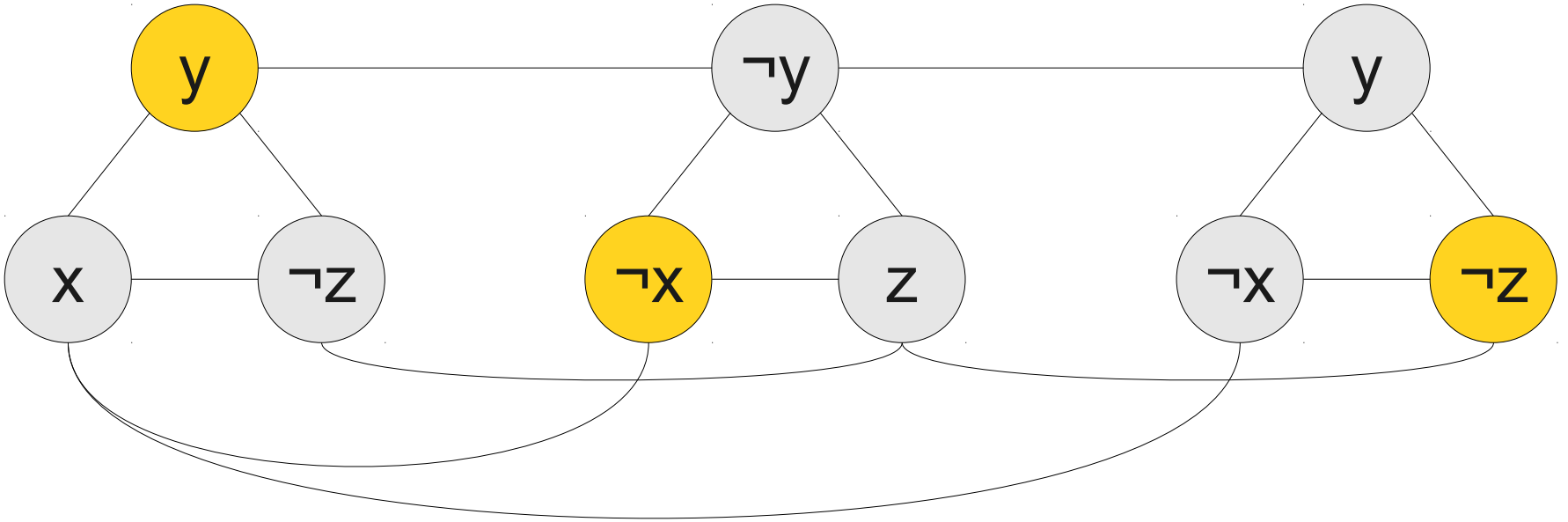


If the original formula is satisfiable,
this graph has an independent set of size three.

From 3SAT to INDSET

x = false, y = true, z = false.

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



If the original formula is satisfiable,
this graph has an independent set of size three.

From 3SAT to INDSET

- Let $\varphi = C_1 \wedge C_2 \wedge \ldots \wedge C_n$ be a 3-CNF formula.
- Construct the graph G as follows:
  - For each clause $C_i = x_1 \vee x_2 \vee x_3$, where x_1, x_2, and x_3 are literals, add three new nodes into G with edges connecting them.
  - For each pair of nodes v_i and $\neg v_i$, where v_i is some variable, add an edge connecting v_i and $\neg v_i$. (Note that there are multiple copies of these nodes)
- **Claim One:** This reduction can be computed in polynomial time.
- **Claim**: G has an independent set of size n iff φ is satisfiable.

A Polynomial-Time Reduction

Lemma: This reduction can be computed in polynomial time.

A Polynomial-Time Reduction

Lemma: This reduction can be computed in polynomial time.

Proof: Suppose that the original 3-CNF formula φ has n clauses, each of which has three literals.

A Polynomial-Time Reduction

Lemma: This reduction can be computed in polynomial time.

Proof: Suppose that the original 3-CNF formula φ has n clauses, each of which has three literals. Then we construct 3n nodes in our graph.

A Polynomial-Time Reduction

Lemma: This reduction can be computed in polynomial time.

Proof: Suppose that the original 3-CNF formula φ has n clauses, each of which has three literals. Then we construct 3n nodes in our graph. Each clause contributes 3 edges, so there are O(n) edges added from clauses.

A Polynomial-Time Reduction

Lemma: This reduction can be computed in polynomial time.

Proof: Suppose that the original 3-CNF formula φ has n clauses, each of which has three literals. Then we construct 3n nodes in our graph. Each clause contributes 3 edges, so there are O(n) edges added from clauses. For each pair of nodes representing opposite literals, we introduce one edge.

A Polynomial-Time Reduction

Lemma: This reduction can be computed in polynomial time.

Proof: Suppose that the original 3-CNF formula φ has n clauses, each of which has three literals. Then we construct 3n nodes in our graph. Each clause contributes 3 edges, so there are O(n) edges added from clauses. For each pair of nodes representing opposite literals, we introduce one edge. Since there are $O(n^2)$ pairs of literals, this introduces at most $O(n^2)$ new edges.

A Polynomial-Time Reduction

Lemma: This reduction can be computed in polynomial time.

Proof: Suppose that the original 3-CNF formula φ has n clauses, each of which has three literals. Then we construct 3n nodes in our graph. Each clause contributes 3 edges, so there are O(n) edges added from clauses. For each pair of nodes representing opposite literals, we introduce one edge. Since there are $O(n^2)$ pairs of literals, this introduces at most $O(n^2)$ new edges. This gives a graph with O(n) nodes and $O(n^2)$ edges.

A Polynomial-Time Reduction

Lemma: This reduction can be computed in polynomial time.

Proof: Suppose that the original 3-CNF formula φ has n clauses, each of which has three literals. Then we construct 3n nodes in our graph. Each clause contributes 3 edges, so there are O(n) edges added from clauses. For each pair of nodes representing opposite literals, we introduce one edge. Since there are $O(n^2)$ pairs of literals, this introduces at most $O(n^2)$ new edges. This gives a graph with O(n) nodes and $O(n^2)$ edges. Each node and edge can be constructed in polynomial time, so overall this reduction can be computed in polynomial time.

A Polynomial-Time Reduction

Lemma: This reduction can be computed in polynomial time.

Proof: Suppose that the original 3-CNF formula φ has n clauses, each of which has three literals. Then we construct 3n nodes in our graph. Each clause contributes 3 edges, so there are O(n) edges added from clauses. For each pair of nodes representing opposite literals, we introduce one edge. Since there are $O(n^2)$ pairs of literals, this introduces at most $O(n^2)$ new edges. This gives a graph with O(n) nodes and $O(n^2)$ edges. Each node and edge can be constructed in polynomial time, so overall this reduction can be computed in polynomial time. ■

One Direction of Implication

Lemma: If the graph G has an independent set of size n (where n is the number of clauses) in φ, then φ is satisfiable.

One Direction of Implication

Lemma: If the graph G has an independent set of size n (where n is the number of clauses) in φ, then φ is satisfiable.

Proof: Suppose G has an independent set of size n, call if S.

One Direction of Implication

Lemma: If the graph G has an independent set of size n (where n is the number of clauses) in φ, then φ is satisfiable.

Proof: Suppose G has an independent set of size n, call if S. No two nodes in S can correspond to v and ¬v for any variable v, because there is an edge between all nodes with this property.

One Direction of Implication

Lemma: If the graph G has an independent set of size n (where n is the number of clauses) in φ, then φ is satisfiable.

Proof: Suppose G has an independent set of size n, call if S. No two nodes in S can correspond to v and ¬v for any variable v, because there is an edge between all nodes with this property. Thus for each variable v, either there is a node in S with label v, or there is a node in S with label ¬v, or no node in S has either label.

One Direction of Implication

Lemma: If the graph G has an independent set of size n (where n is the number of clauses) in φ, then φ is satisfiable.

Proof: Suppose G has an independent set of size n, call if S. No two nodes in S can correspond to v and ¬v for any variable v, because there is an edge between all nodes with this property. Thus for each variable v, either there is a node in S with label v, or there is a node in S with label ¬v, or no node in S has either label. In the first case, set v to true; in the second case, set v to false; in the third case, choose a value for v arbitrarily.

One Direction of Implication

Lemma: If the graph G has an independent set of size n (where n is the number of clauses) in φ, then φ is satisfiable.

Proof: Suppose G has an independent set of size n, call if S. No two nodes in S can correspond to v and ¬v for any variable v, because there is an edge between all nodes with this property. Thus for each variable v, either there is a node in S with label v, or there is a node in S with label ¬v, or no node in S has either label. In the first case, set v to true; in the second case, set v to false; in the third case, choose a value for v arbitrarily. We claim that this gives a satisfying assignment for φ.

One Direction of Implication

Lemma: If the graph G has an independent set of size n (where n is the number of clauses) in φ, then φ is satisfiable.

Proof: Suppose G has an independent set of size n, call if S. No two nodes in S can correspond to v and ¬v for any variable v, because there is an edge between all nodes with this property. Thus for each variable v, either there is a node in S with label v, or there is a node in S with label ¬v, or no node in S has either label. In the first case, set v to true; in the second case, set v to false; in the third case, choose a value for v arbitrarily. We claim that this gives a satisfying assignment for φ.

To see this, we show that each clause C in φ is satisfied.

One Direction of Implication

Lemma: If the graph G has an independent set of size n (where n is the number of clauses) in φ, then φ is satisfiable.

Proof: Suppose G has an independent set of size n, call if S. No two nodes in S can correspond to v and ¬v for any variable v, because there is an edge between all nodes with this property. Thus for each variable v, either there is a node in S with label v, or there is a node in S with label ¬v, or no node in S has either label. In the first case, set v to true; in the second case, set v to false; in the third case, choose a value for v arbitrarily. We claim that this gives a satisfying assignment for φ.

To see this, we show that each clause C in φ is satisfied. By construction, no two nodes in S can come from nodes added by C, because each has an edge to the other.

One Direction of Implication

Lemma: If the graph G has an independent set of size n (where n is the number of clauses) in φ, then φ is satisfiable.

Proof: Suppose G has an independent set of size n, call if S. No two nodes in S can correspond to v and ¬v for any variable v, because there is an edge between all nodes with this property. Thus for each variable v, either there is a node in S with label v, or there is a node in S with label ¬v, or no node in S has either label. In the first case, set v to true; in the second case, set v to false; in the third case, choose a value for v arbitrarily. We claim that this gives a satisfying assignment for φ.

To see this, we show that each clause C in φ is satisfied. By construction, no two nodes in S can come from nodes added by C, because each has an edge to the other. Since there are n nodes in S and n clauses in φ, for any clause in φ some node corresponding to a literal from that clause is in S.

One Direction of Implication

Lemma: If the graph G has an independent set of size n (where n is the number of clauses) in φ, then φ is satisfiable.

Proof: Suppose G has an independent set of size n, call if S. No two nodes in S can correspond to v and ¬v for any variable v, because there is an edge between all nodes with this property. Thus for each variable v, either there is a node in S with label v, or there is a node in S with label ¬v, or no node in S has either label. In the first case, set v to true; in the second case, set v to false; in the third case, choose a value for v arbitrarily. We claim that this gives a satisfying assignment for φ.

To see this, we show that each clause C in φ is satisfied. By construction, no two nodes in S can come from nodes added by C, because each has an edge to the other. Since there are n nodes in S and n clauses in φ, for any clause in φ some node corresponding to a literal from that clause is in S. If that node has the form x, then C contains x, and since we set x to true, C is satisfied.

One Direction of Implication

Lemma: If the graph G has an independent set of size n (where n is the number of clauses) in φ, then φ is satisfiable.

Proof: Suppose G has an independent set of size n, call if S. No two nodes in S can correspond to v and ¬v for any variable v, because there is an edge between all nodes with this property. Thus for each variable v, either there is a node in S with label v, or there is a node in S with label ¬v, or no node in S has either label. In the first case, set v to true; in the second case, set v to false; in the third case, choose a value for v arbitrarily. We claim that this gives a satisfying assignment for φ.

To see this, we show that each clause C in φ is satisfied. By construction, no two nodes in S can come from nodes added by C, because each has an edge to the other. Since there are n nodes in S and n clauses in φ, for any clause in φ some node corresponding to a literal from that clause is in S. If that node has the form x, then C contains x, and since we set x to true, C is satisfied. If that node has the form ¬x, then C contains ¬x, and since we set x to false, C is satisfied.

One Direction of Implication

Lemma: If the graph G has an independent set of size n (where n is the number of clauses) in φ, then φ is satisfiable.

Proof: Suppose G has an independent set of size n, call if S. No two nodes in S can correspond to v and ¬v for any variable v, because there is an edge between all nodes with this property. Thus for each variable v, either there is a node in S with label v, or there is a node in S with label ¬v, or no node in S has either label. In the first case, set v to true; in the second case, set v to false; in the third case, choose a value for v arbitrarily. We claim that this gives a satisfying assignment for φ.

To see this, we show that each clause C in φ is satisfied. By construction, no two nodes in S can come from nodes added by C, because each has an edge to the other. Since there are n nodes in S and n clauses in φ, for any clause in φ some node corresponding to a literal from that clause is in S. If that node has the form x, then C contains x, and since we set x to true, C is satisfied. If that node has the form ¬x, then C contains ¬x, and since we set x to false, C is satisfied. Thus all clauses in φ are satisfied, so φ is satisfied by this assignment.

One Direction of Implication

Lemma: If the graph G has an independent set of size n (where n is the number of clauses) in φ, then φ is satisfiable.

Proof: Suppose G has an independent set of size n, call if S. No two nodes in S can correspond to v and ¬v for any variable v, because there is an edge between all nodes with this property. Thus for each variable v, either there is a node in S with label v, or there is a node in S with label ¬v, or no node in S has either label. In the first case, set v to true; in the second case, set v to false; in the third case, choose a value for v arbitrarily. We claim that this gives a satisfying assignment for φ.

To see this, we show that each clause C in φ is satisfied. By construction, no two nodes in S can come from nodes added by C, because each has an edge to the other. Since there are n nodes in S and n clauses in φ, for any clause in φ some node corresponding to a literal from that clause is in S. If that node has the form x, then C contains x, and since we set x to true, C is satisfied. If that node has the form ¬x, then C contains ¬x, and since we set x to false, C is satisfied. Thus all clauses in φ are satisfied, so φ is satisfied by this assignment. ■

The Other Direction

Lemma: If φ is satisfiable and has n clauses, then G has an independent set of size n.

The Other Direction

Lemma: If φ is satisfiable and has n clauses, then G has an independent set of size n.

Proof: Suppose that φ is satisfiable and consider any satisfying assignment for it.

The Other Direction

Lemma: If φ is satisfiable and has n clauses, then G has an independent set of size n.

Proof: Suppose that φ is satisfiable and consider any satisfying assignment for it. Thus under that assignment, for each clause C, there is some literal that evaluates to true.

The Other Direction

Lemma: If φ is satisfiable and has n clauses, then G has an independent set of size n.

Proof: Suppose that φ is satisfiable and consider any satisfying assignment for it. Thus under that assignment, for each clause C, there is some literal that evaluates to true. For each clause C, choose some literal that evaluates to true and add the corresponding node in G to a set S.

The Other Direction

Lemma: If φ is satisfiable and has n clauses, then G has an independent set of size n.

Proof: Suppose that φ is satisfiable and consider any satisfying assignment for it. Thus under that assignment, for each clause C, there is some literal that evaluates to true. For each clause C, choose some literal that evaluates to true and add the corresponding node in G to a set S. Then S has size n, since it contains one node per clause.

The Other Direction

Lemma: If φ is satisfiable and has n clauses, then G has an independent set of size n.

Proof: Suppose that φ is satisfiable and consider any satisfying assignment for it. Thus under that assignment, for each clause C, there is some literal that evaluates to true. For each clause C, choose some literal that evaluates to true and add the corresponding node in G to a set S. Then S has size n, since it contains one node per clause. We claim moreover that S is an independent set in G.

The Other Direction

Lemma: If φ is satisfiable and has n clauses, then G has an independent set of size n.

Proof: Suppose that φ is satisfiable and consider any satisfying assignment for it. Thus under that assignment, for each clause C, there is some literal that evaluates to true. For each clause C, choose some literal that evaluates to true and add the corresponding node in G to a set S. Then S has size n, since it contains one node per clause. We claim moreover that S is an independent set in G. To see this, note that there are two types of edges in G: edges between nodes representing literals in the same clause, and edges between variables and their negations.

The Other Direction

Lemma: If φ is satisfiable and has n clauses, then G has an independent set of size n.

Proof: Suppose that φ is satisfiable and consider any satisfying assignment for it. Thus under that assignment, for each clause C, there is some literal that evaluates to true. For each clause C, choose some literal that evaluates to true and add the corresponding node in G to a set S. Then S has size n, since it contains one node per clause. We claim moreover that S is an independent set in G. To see this, note that there are two types of edges in G: edges between nodes representing literals in the same clause, and edges between variables and their negations. No two nodes joined by edges within a clause are in S, because we explicitly picked one node per clause.

The Other Direction

Lemma: If φ is satisfiable and has n clauses, then G has an independent set of size n.

Proof: Suppose that φ is satisfiable and consider any satisfying assignment for it. Thus under that assignment, for each clause C, there is some literal that evaluates to true. For each clause C, choose some literal that evaluates to true and add the corresponding node in G to a set S. Then S has size n, since it contains one node per clause. We claim moreover that S is an independent set in G. To see this, note that there are two types of edges in G: edges between nodes representing literals in the same clause, and edges between variables and their negations. No two nodes joined by edges within a clause are in S, because we explicitly picked one node per clause. Moreover, no two nodes jointed by edges between opposite literals are in S, because in a satisfying assignment both of the two could not be true.

The Other Direction

Lemma: If φ is satisfiable and has n clauses, then G has an independent set of size n.

Proof: Suppose that φ is satisfiable and consider any satisfying assignment for it. Thus under that assignment, for each clause C, there is some literal that evaluates to true. For each clause C, choose some literal that evaluates to true and add the corresponding node in G to a set S. Then S has size n, since it contains one node per clause. We claim moreover that S is an independent set in G. To see this, note that there are two types of edges in G: edges between nodes representing literals in the same clause, and edges between variables and their negations. No two nodes joined by edges within a clause are in S, because we explicitly picked one node per clause. Moreover, no two nodes jointed by edges between opposite literals are in S, because in a satisfying assignment both of the two could not be true. Thus no nodes in S are joined by edges, so S is an independent set.

The Other Direction

Lemma: If φ is satisfiable and has n clauses, then G has an independent set of size n.

Proof: Suppose that φ is satisfiable and consider any satisfying assignment for it. Thus under that assignment, for each clause C, there is some literal that evaluates to true. For each clause C, choose some literal that evaluates to true and add the corresponding node in G to a set S. Then S has size n, since it contains one node per clause. We claim moreover that S is an independent set in G. To see this, note that there are two types of edges in G: edges between nodes representing literals in the same clause, and edges between variables and their negations. No two nodes joined by edges within a clause are in S, because we explicitly picked one node per clause. Moreover, no two nodes jointed by edges between opposite literals are in S, because in a satisfying assignment both of the two could not be true. Thus no nodes in S are joined by edges, so S is an independent set. ■

Putting it All Together

Theorem: INDSET is NP-complete.

Putting it All Together

Theorem: INDSET is NP-complete.
Proof: We know that INDSET $\in$ NP, because we constructed a polynomial-time verifier for it.

Putting it All Together

Theorem: INDSET is NP-complete.
Proof: We know that INDSET $\in$ NP, because we constructed a polynomial-time verifier for it. So all we need to show is that every problem in NP is polynomial-time reducible to INDSET.

Putting it All Together

Theorem: INDSET is NP-complete.
Proof: We know that INDSET $\in$ NP, because we constructed a polynomial-time verifier for it. So all we need to show is that every problem in NP is polynomial-time reducible to INDSET. To do this, we use the polynomial-time reduction from 3SAT to INDSET that we just gave.

Putting it All Together

Theorem: INDSET is NP-complete.
Proof: We know that INDSET $\in$ NP, because we constructed a polynomial-time verifier for it. So all we need to show is that every problem in NP is polynomial-time reducible to INDSET. To do this, we use the polynomial-time reduction from 3SAT to INDSET that we just gave. As we proved, $\varphi \in$ 3SAT iff $\langle G, n \rangle \in$ INDSET, and this reduction can be computed in polynomial time.

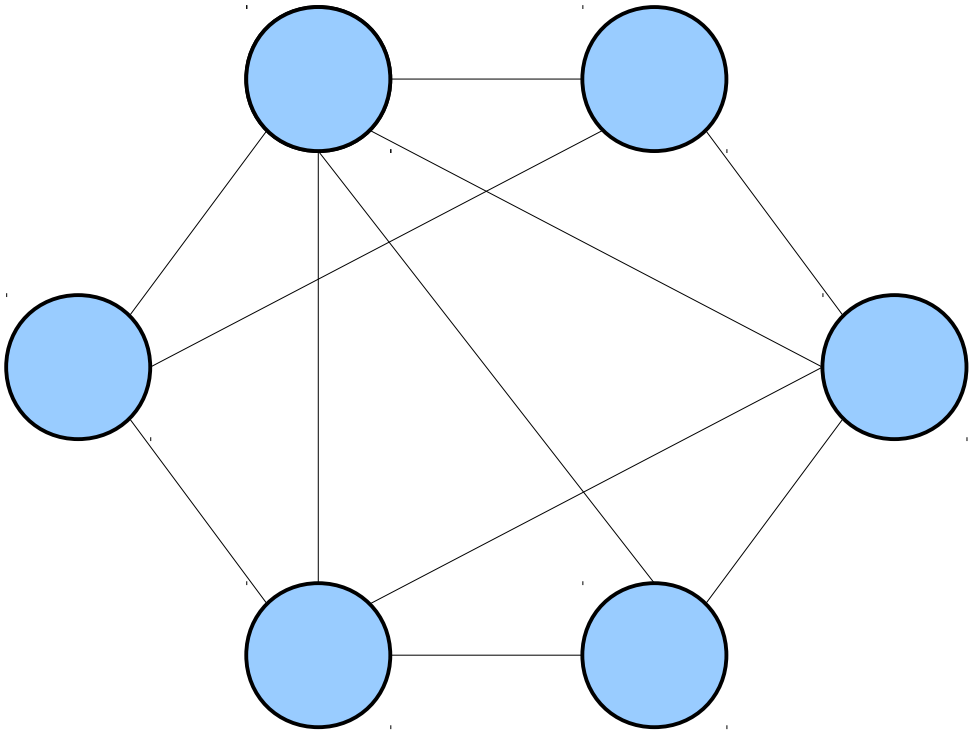
Putting it All Together

Theorem: INDSET is NP-complete.
Proof: We know that INDSET $\in$ NP, because we constructed a polynomial-time verifier for it. So all we need to show is that every problem in NP is polynomial-time reducible to INDSET. To do this, we use the polynomial-time reduction from 3SAT to INDSET that we just gave. As we proved, $\varphi \in$ 3SAT iff $\langle G, n \rangle \in$ INDSET, and this reduction can be computed in polynomial time. Thus 3SAT is polynomial-time reducible to INDSET, so INDSET is NP-complete.

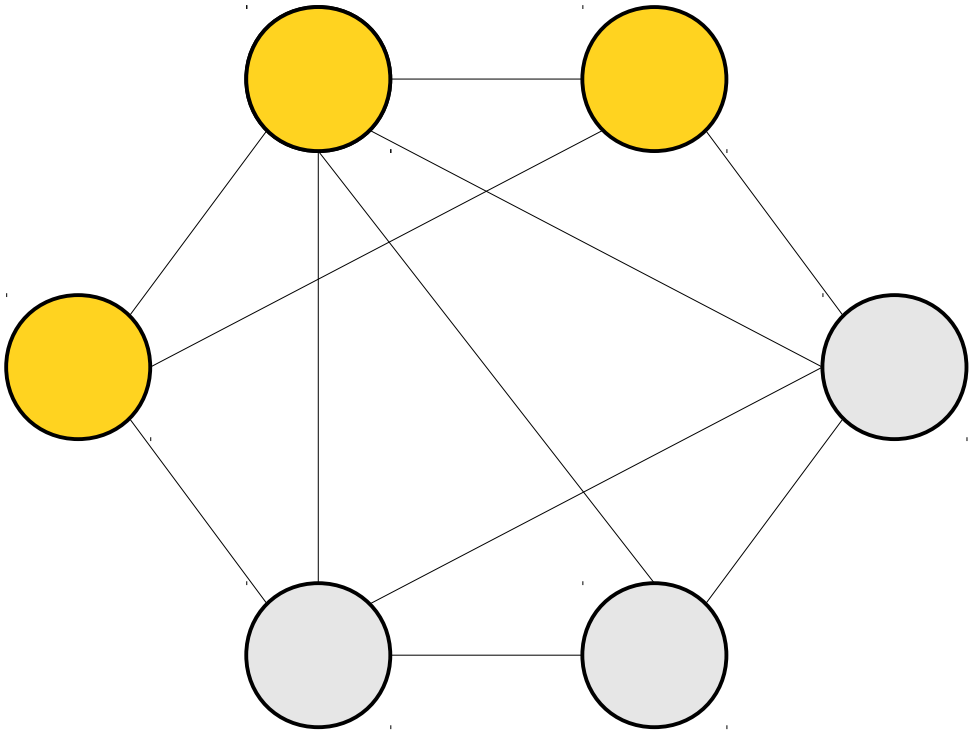
Putting it All Together

Theorem: INDSET is NP-complete.
Proof: We know that INDSET $\in$ NP, because we constructed a polynomial-time verifier for it. So all we need to show is that every problem in NP is polynomial-time reducible to INDSET. To do this, we use the polynomial-time reduction from 3SAT to INDSET that we just gave. As we proved, $\varphi \in$ 3SAT iff $\langle G, n \rangle \in$ INDSET, and this reduction can be computed in polynomial time. Thus 3SAT is polynomial-time reducible to INDSET, so INDSET is NP-complete. ■

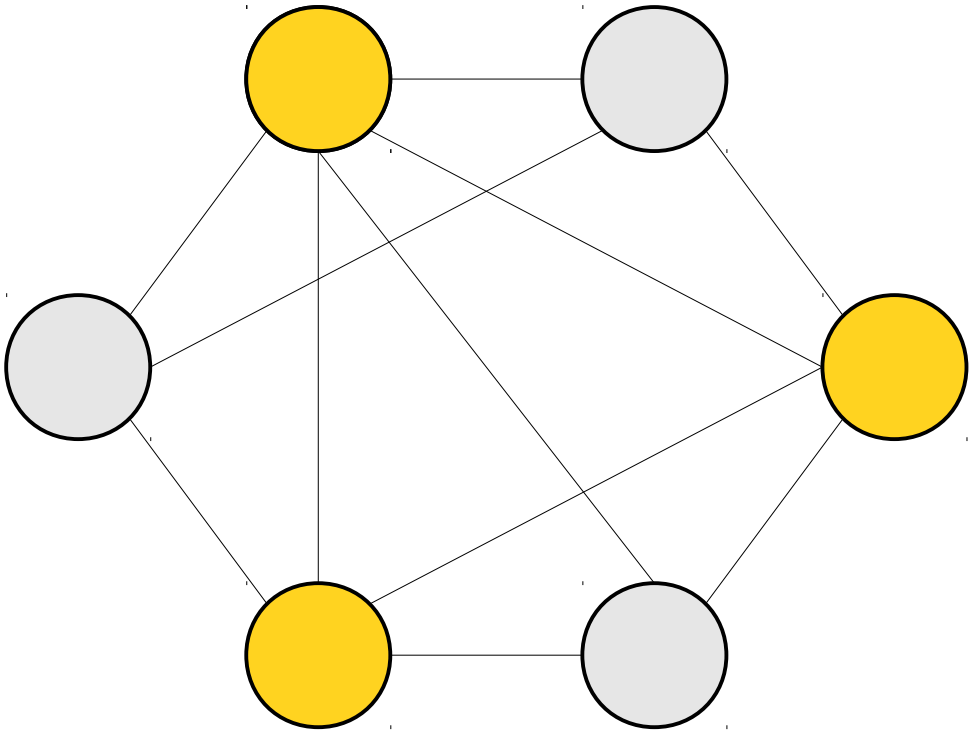
A **k-clique** in an undirected graph is a set of nodes where each pair of nodes in the set has an edge between them.



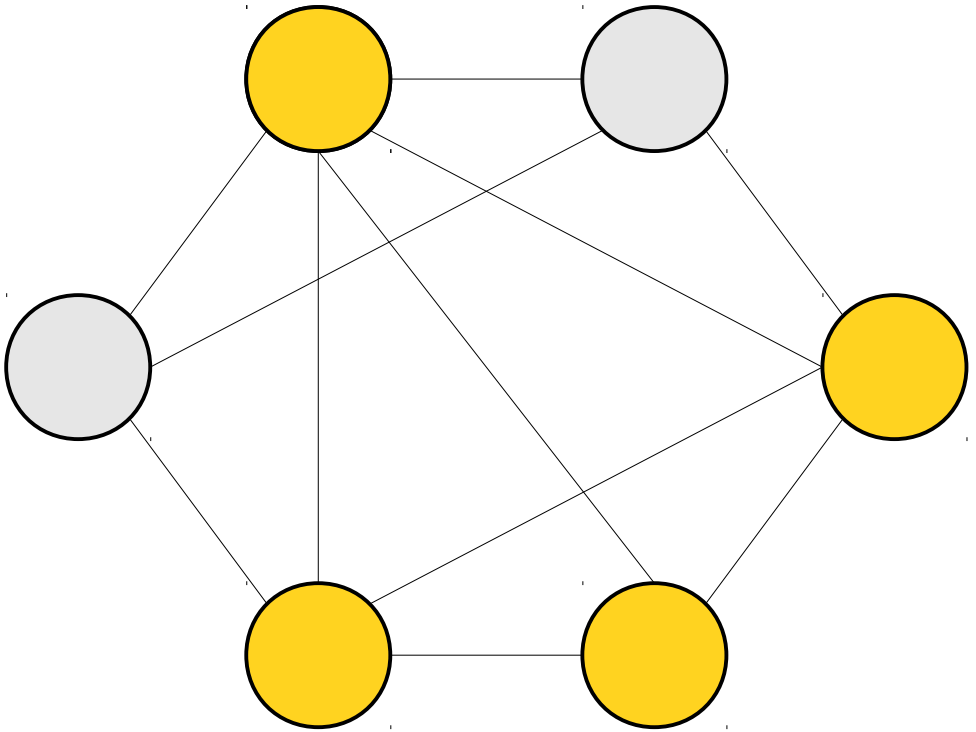
A **k-clique** in an undirected graph is a set of nodes where each pair of nodes in the set has an edge between them.



A **k-clique** in an undirected graph is a set of nodes where each pair of nodes in the set has an edge between them.



A **k-clique** in an undirected graph is a set of nodes where each pair of nodes in the set has an edge between them.



A **k-clique** in an undirected graph is a set of nodes where each pair of nodes in the set has an edge between them.

The Clique Problem

- The **clique problem** is

  Given an undirected graph G and a number k, does G contain a k-clique?

- As a formal language:

  CLIQUE = { ⟨G, k⟩ | G is an undirected graph that contains a k-clique }

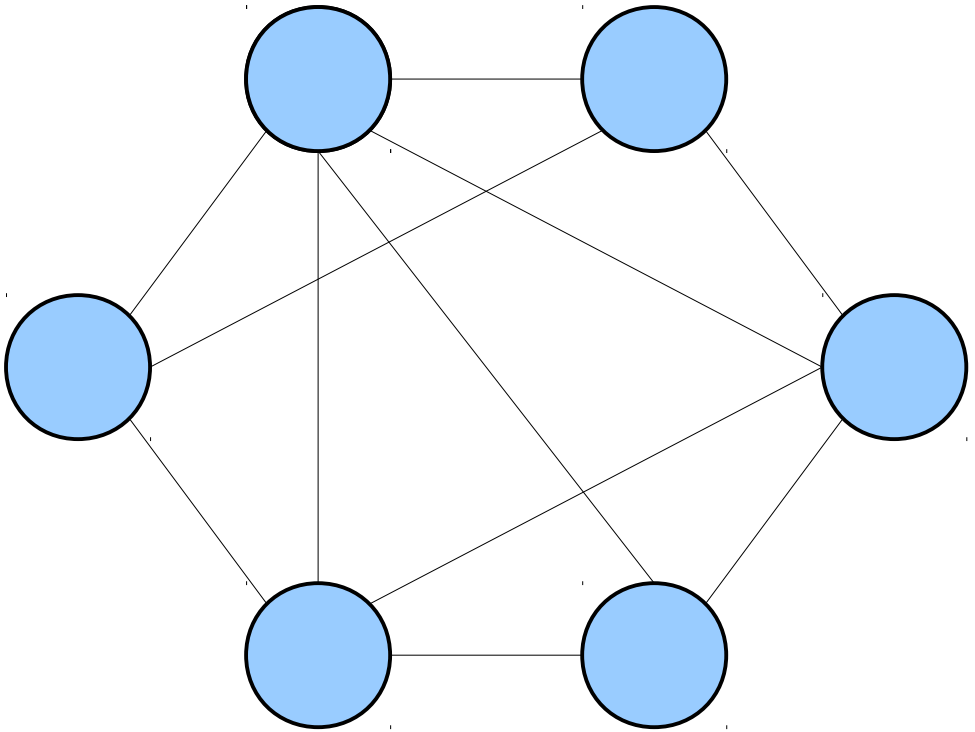
- We will show that CLIQUE is NP-complete.

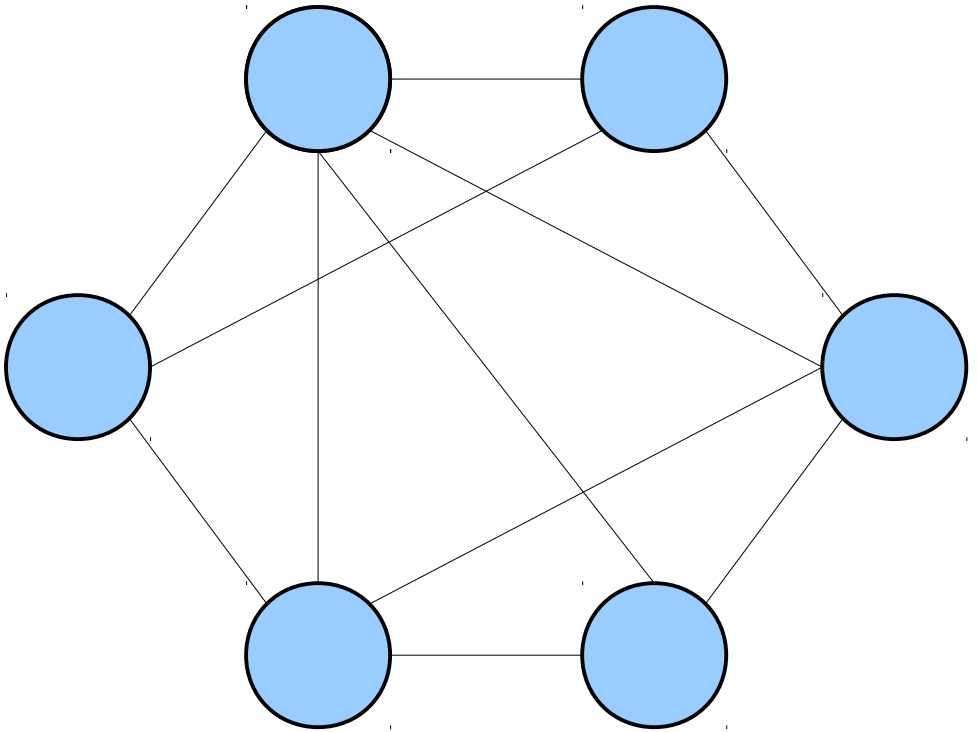
CLIQUE $\in$ NP

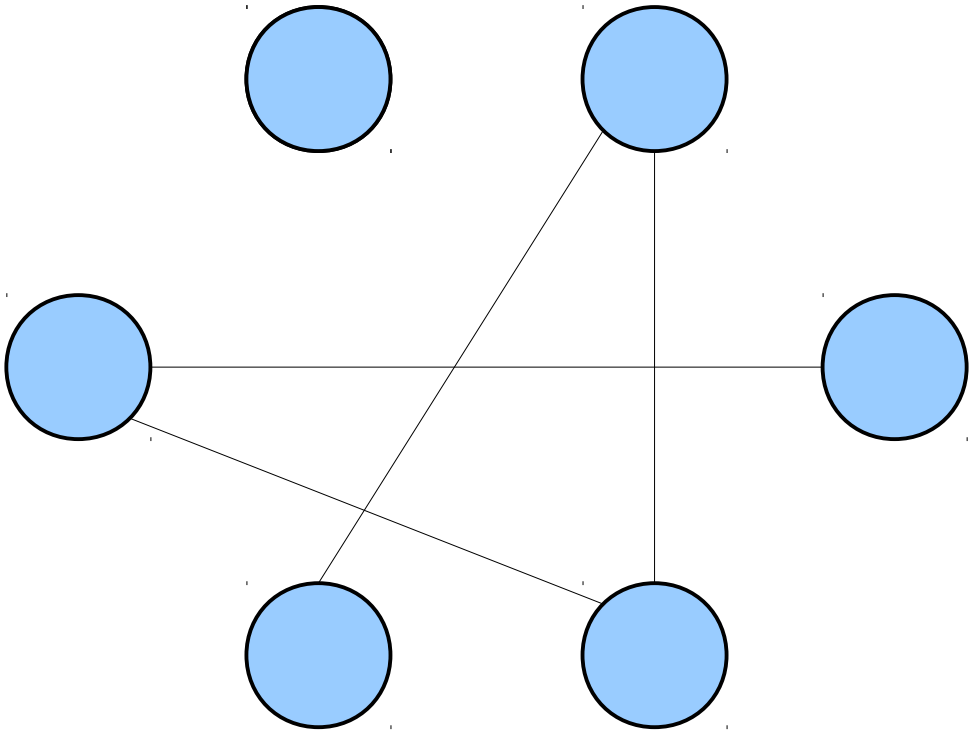
- To show that CLIQUE is NP-complete, we first need to show that CLIQUE $\in$ NP.
- Here is a polynomial-time NTM for CLIQUE:
- M = “On input $\langle G, k \rangle$:
  - **Nondeterministically** guess k nodes.
  - **Deterministically** check whether all k nodes have edges to one another.
  - If so, accept; otherwise, reject.”

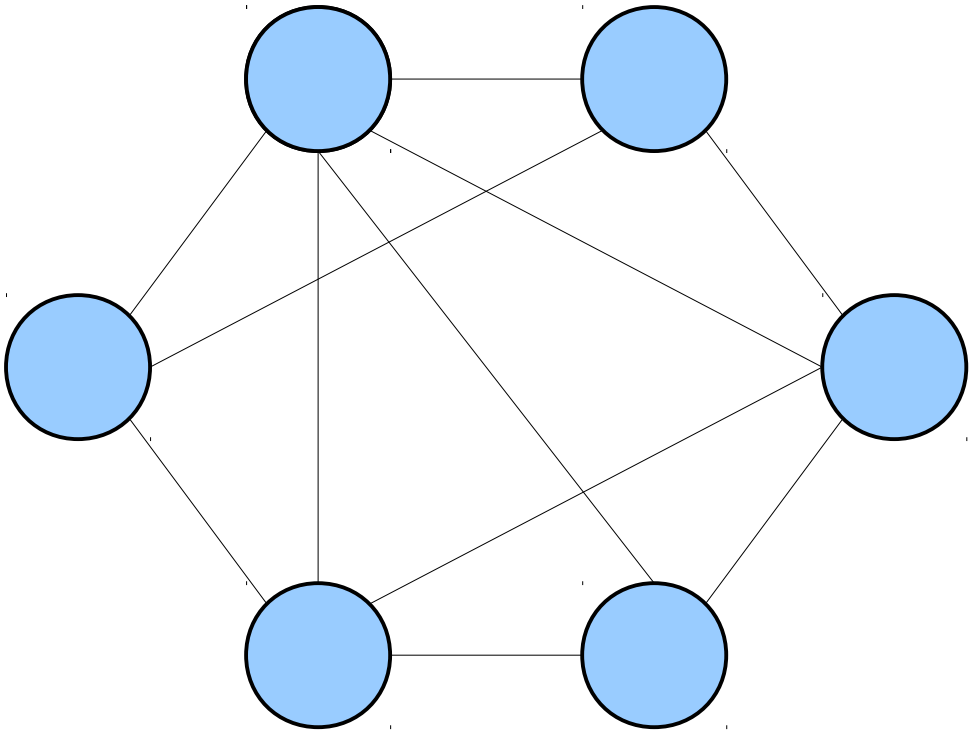
CLIQUE is NP-Complete

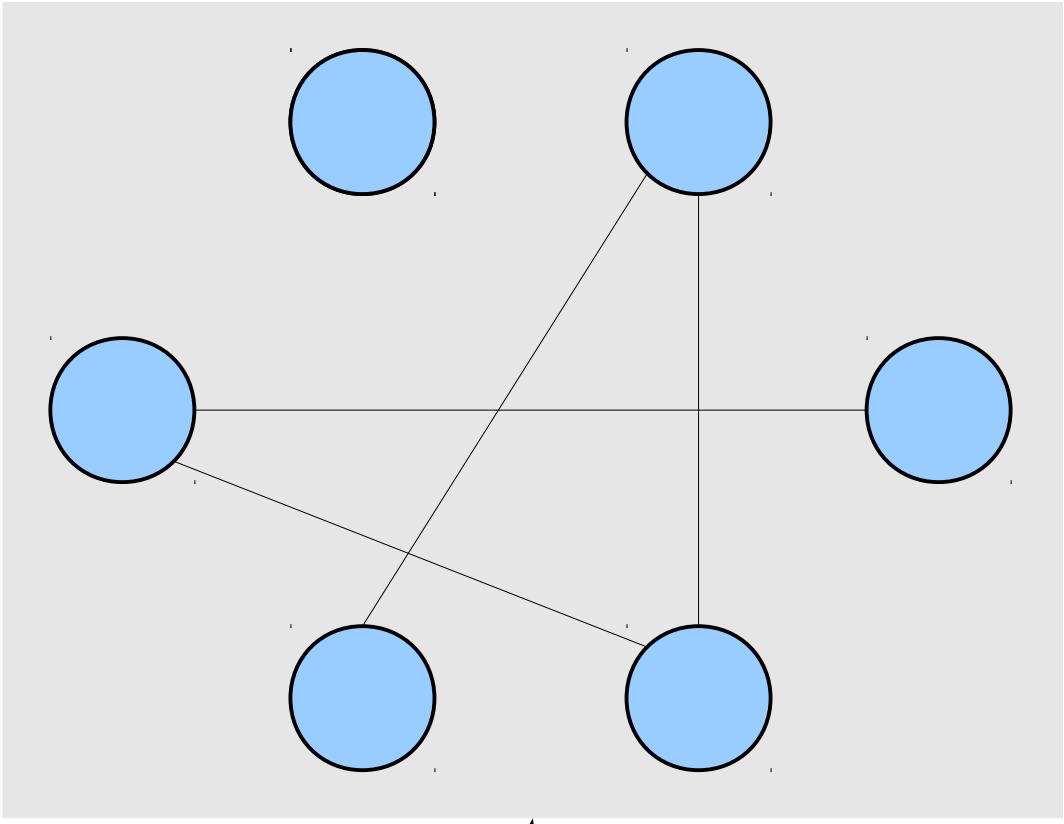
- To prove that CLIQUE is NP-complete, we need to show that any problem in NP is polynomial-time reducible to it.
- Rather than reducing from 3SAT, we'll reduce INDSET to CLIQUE.
- Since INDSET is NP-complete, this proves that CLIQUE is NP-complete as well.



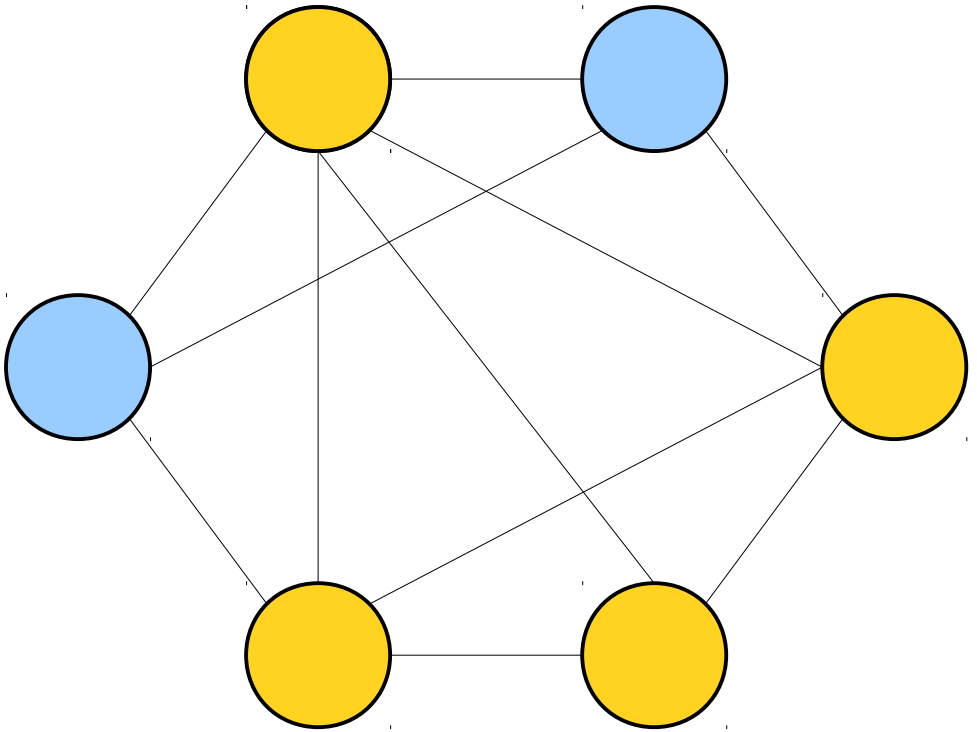


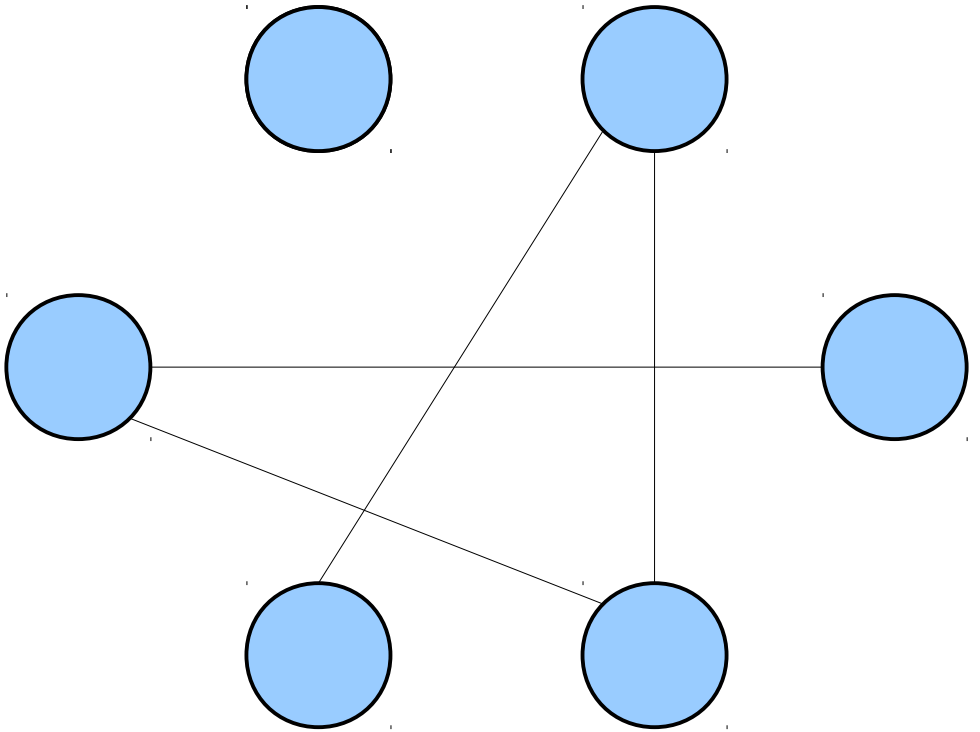


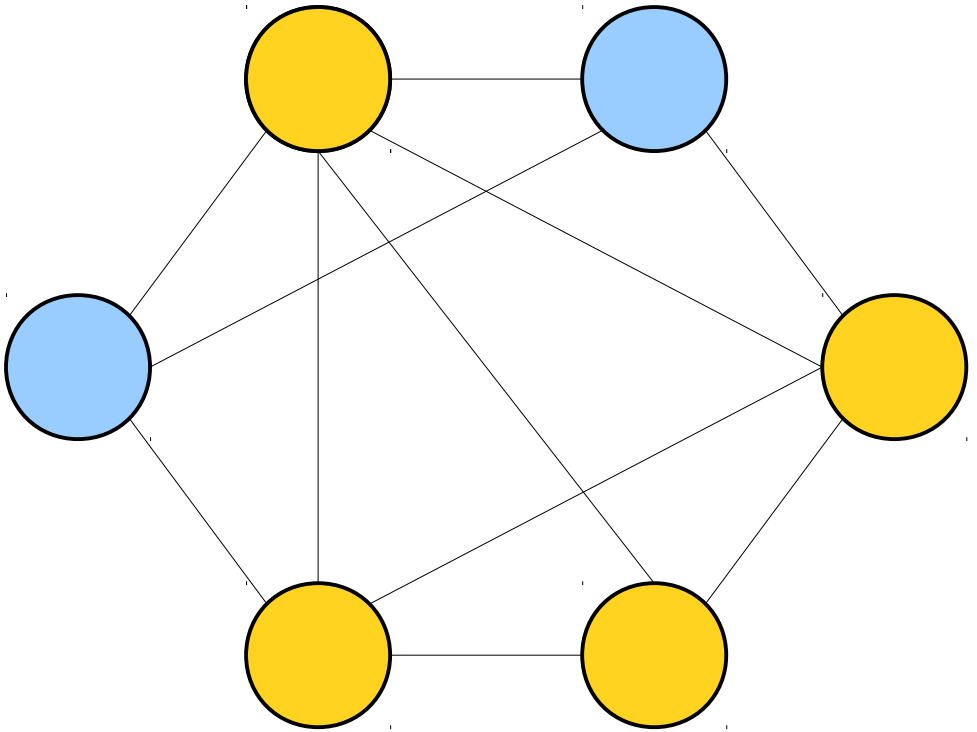


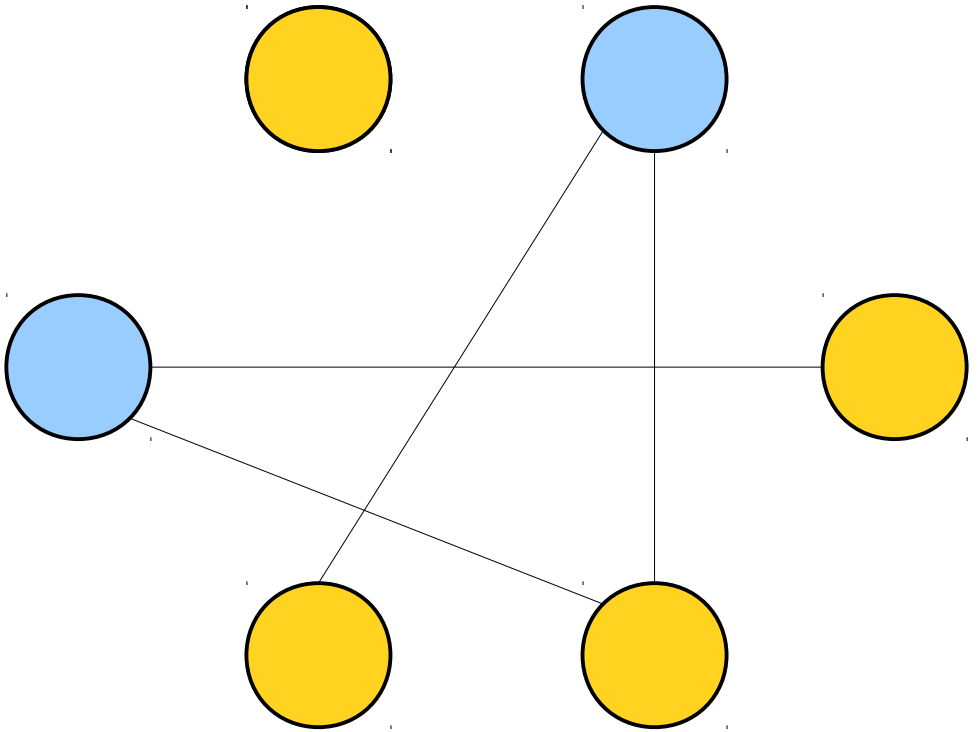


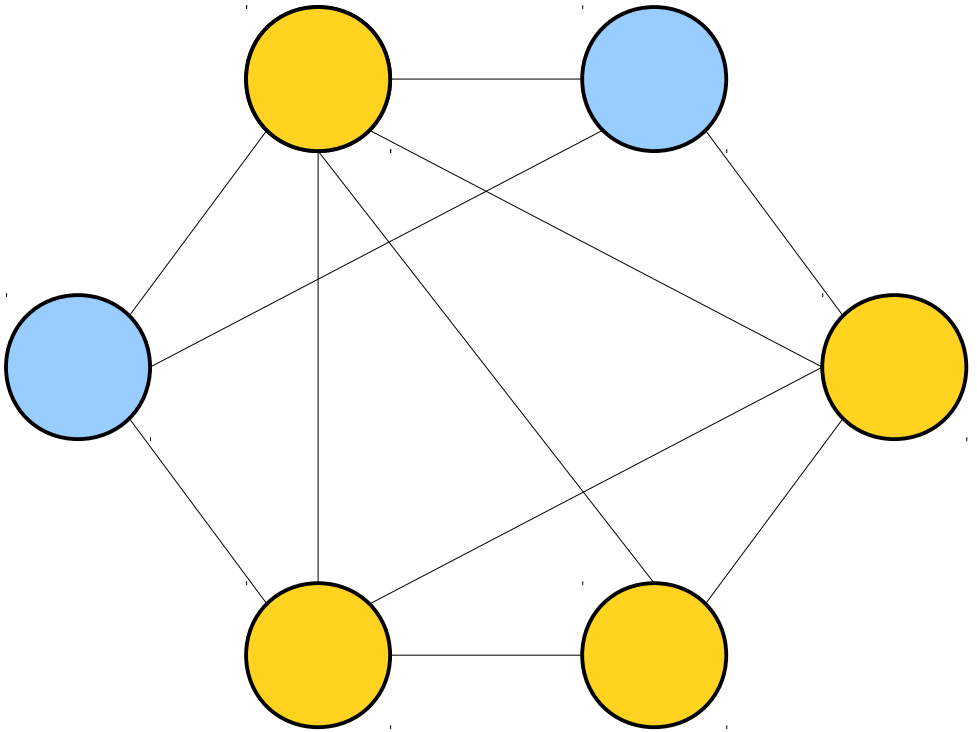
This graph is the complement graph of the left-hand graph. It has the same nodes, but contains all edges missing from the original graph. For simplicity, we omit self-loops.

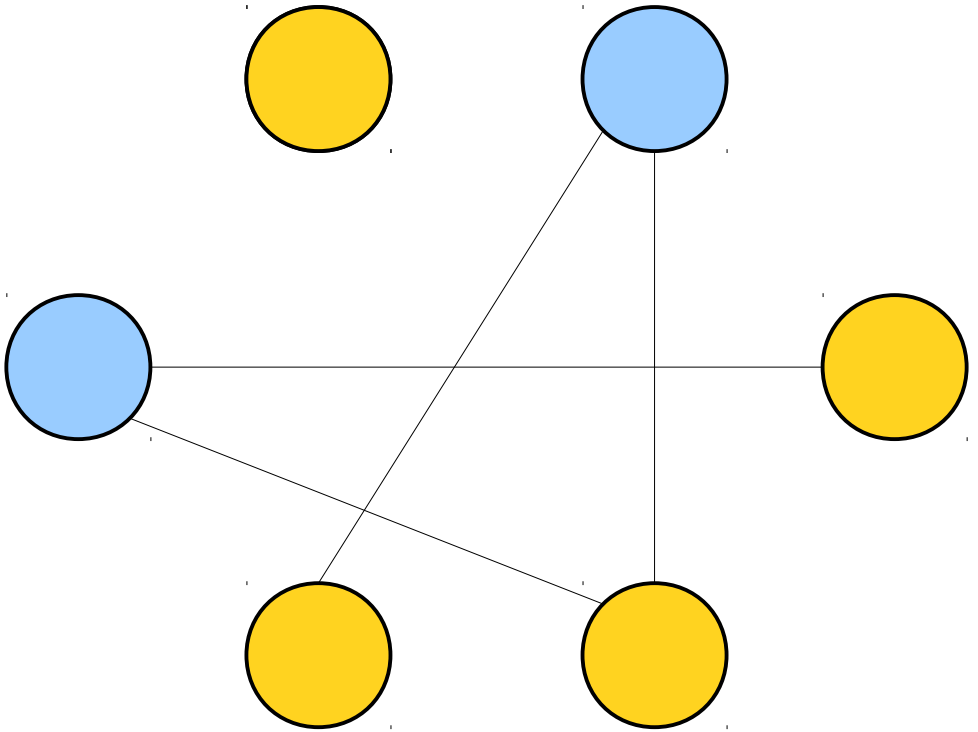












Any clique in the original graph is an independent set in the complement graph!

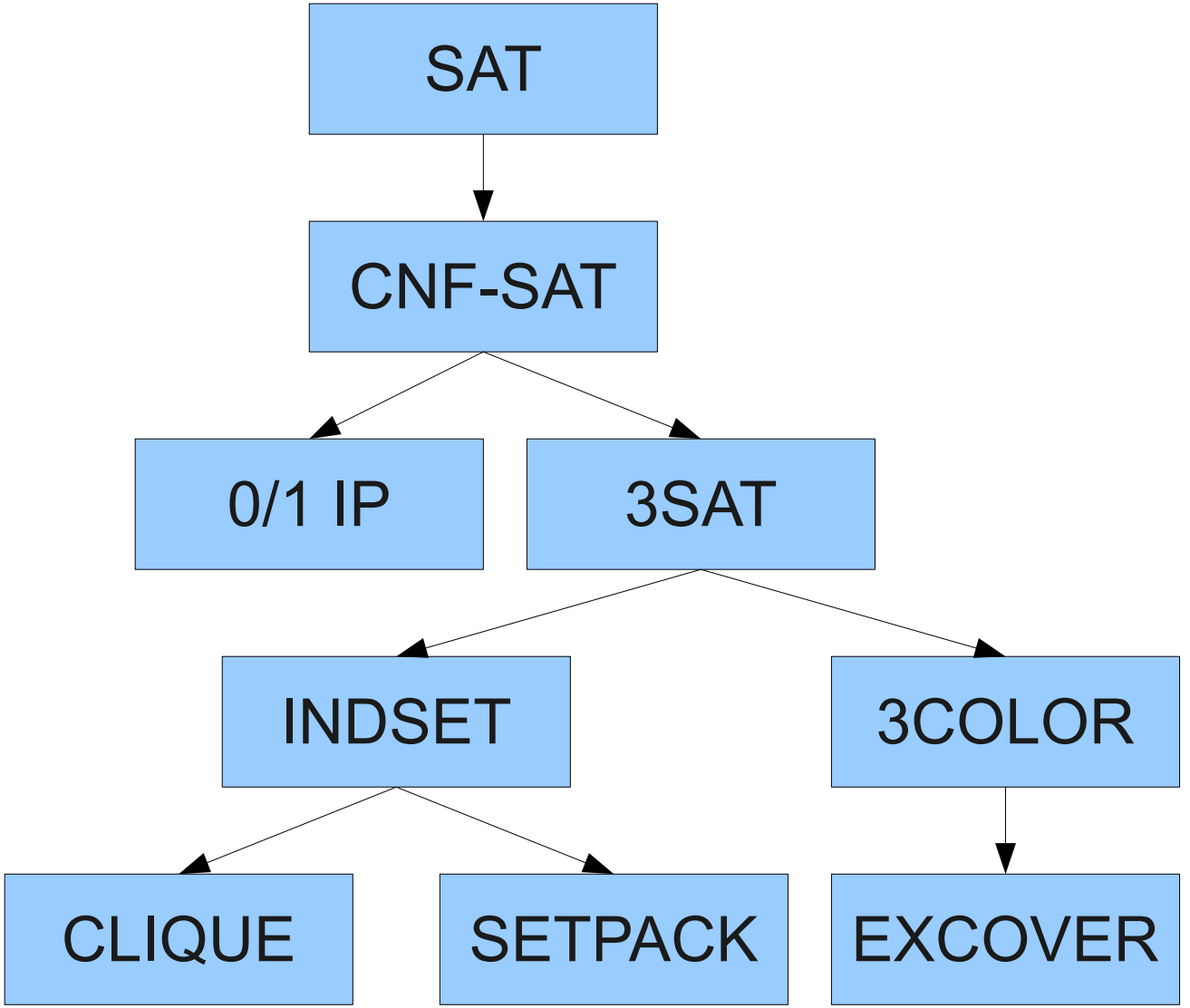
CLIQUE is NP-Complete

- **Proof sketch**:
  - Given as input ⟨G, k⟩, construct ⟨G', k⟩, where G' is the complement graph of G.
  - There is a k-clique in G iff there is an independent set of size k in G'.
  - Since we can construct the complement graph in polynomial time, this is a polynomial-time reduction from INDSET to CLIQUE.

A Genealogy of NP-Completeness

- The Cook-Levin theorem establishes that SAT (and 3SAT) are NP-complete.
- From SAT and 3SAT, we can prove that many other problems are NP-complete as well.
- We can visualize the chain of reductions to any problem as a tree.

A Genealogy of NP-Completeness



A Genealogy of NP-Completeness

- As of now, there are **thousands** of problems known to be NP-complete.
- For a fun list, check the Wikipedia article “List of NP-complete problems.”
- A polynomial-time solution to **any** of these problems proves that P = NP.
- Proving that **any** of these problems admits no polynomial-time solution proves that P ≠ NP.
- **Yet no one has done either of these yet!**

A More Complex Reduction

The Shape of a Reduction

- Polynomial-time reductions work by solving one problem with a solver for a different problem.
- Most problems in NP have different pieces that must be solved simultaneously.
- For example, in 3SAT:
  - Each clause must be made true,
  - but no literal and its complement may be picked.

Reductions and Gadgets

- Many reductions used to show NP-completeness work by using **gadgets**.
- Each piece of the original problem is translated into a “gadget” that handles some particular detail of the problem.
- These gadgets are then connected together to solve the overall problem.

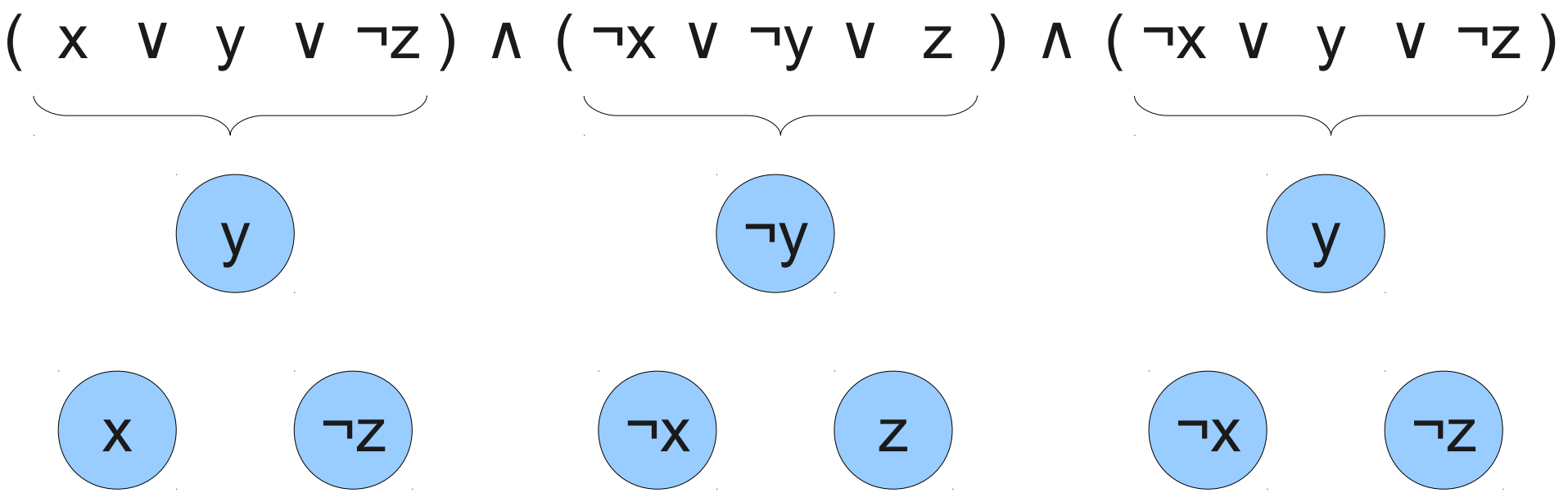
Gadgets in INDSET

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

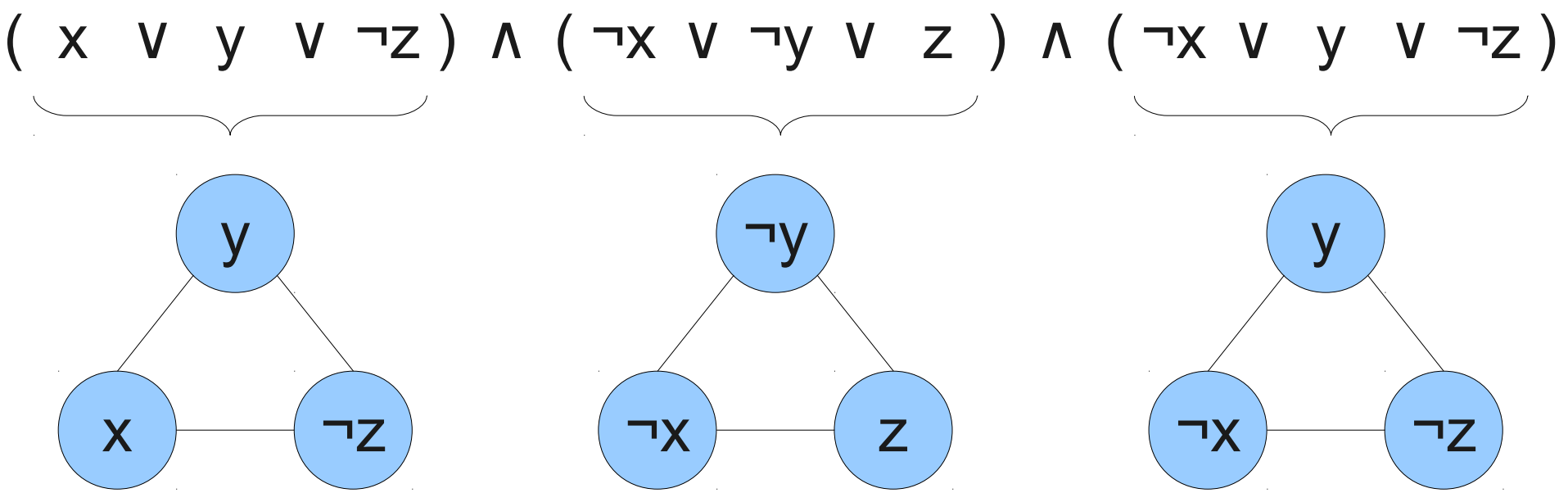
Gadgets in INDSET

$$(\underbrace{x \vee y \vee \neg z}) \wedge (\underbrace{\neg x \vee \neg y \vee z}) \wedge (\underbrace{\neg x \vee y \vee \neg z})$$

Gadgets in INDSET

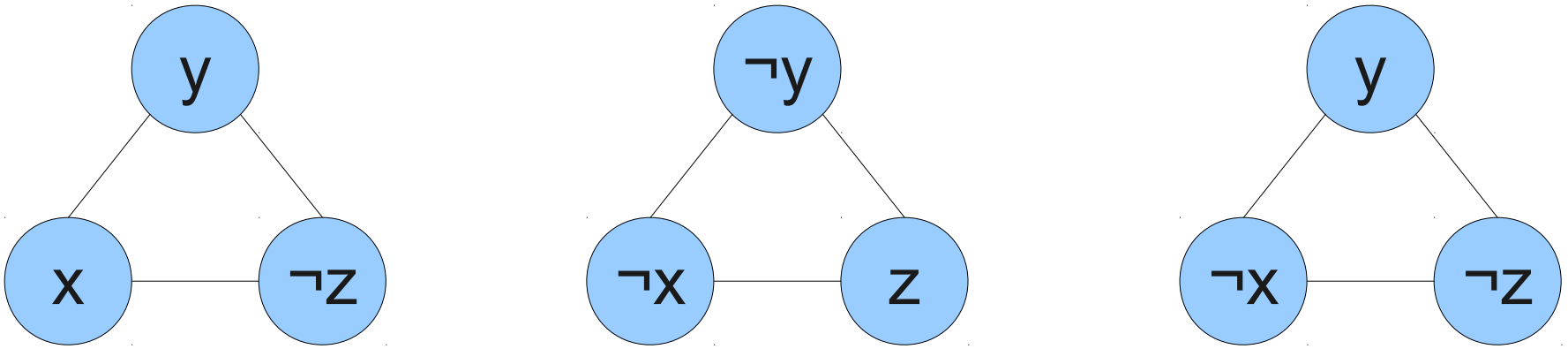


Gadgets in INDSET



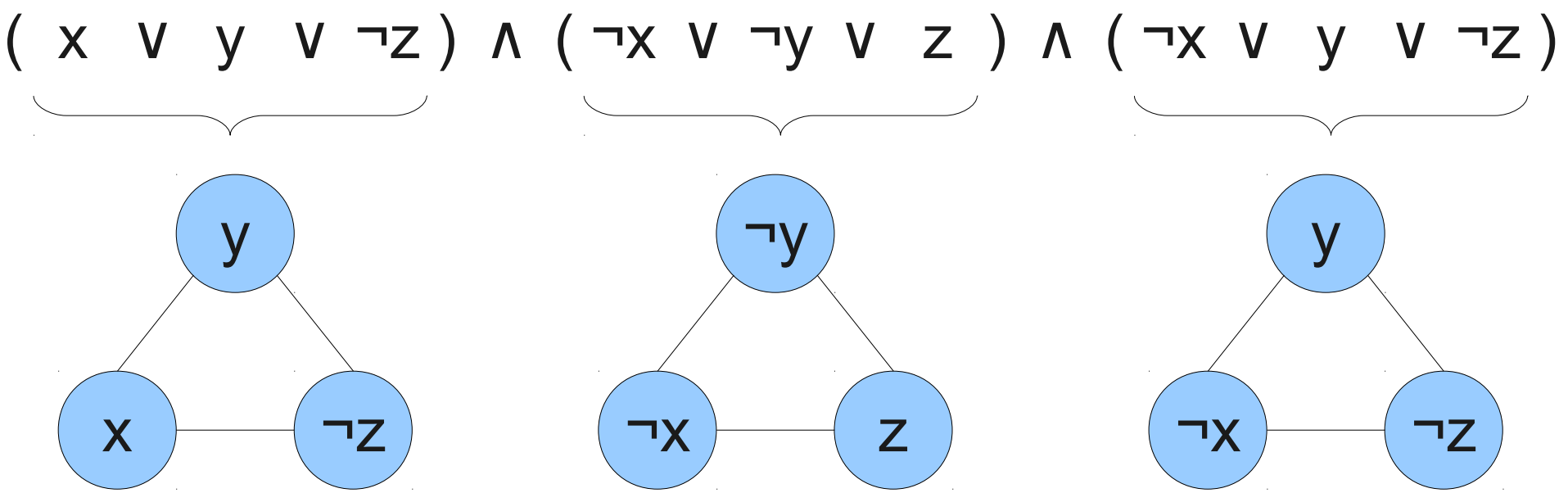
Gadgets in INDSET

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

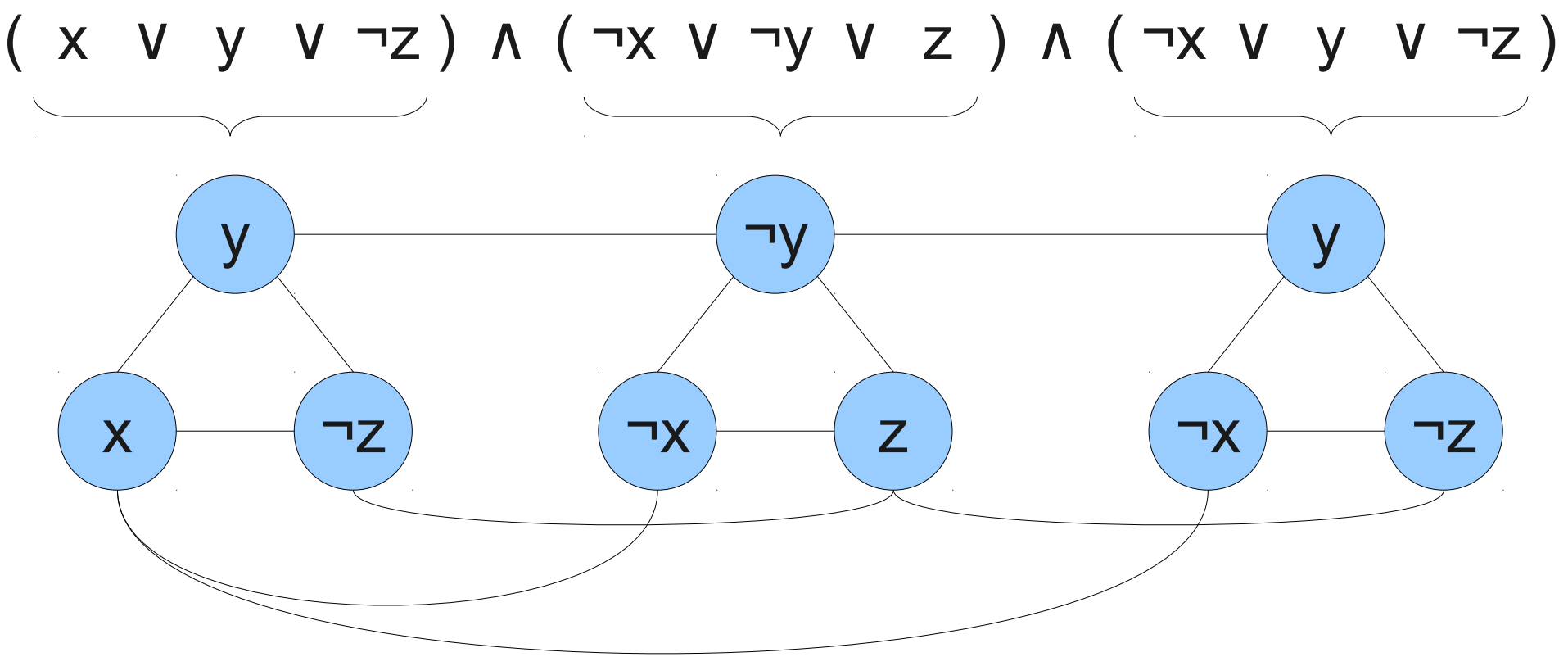


Each of these gadgets is designed to solve one part of the problem: ensuring each clause is satisfied.

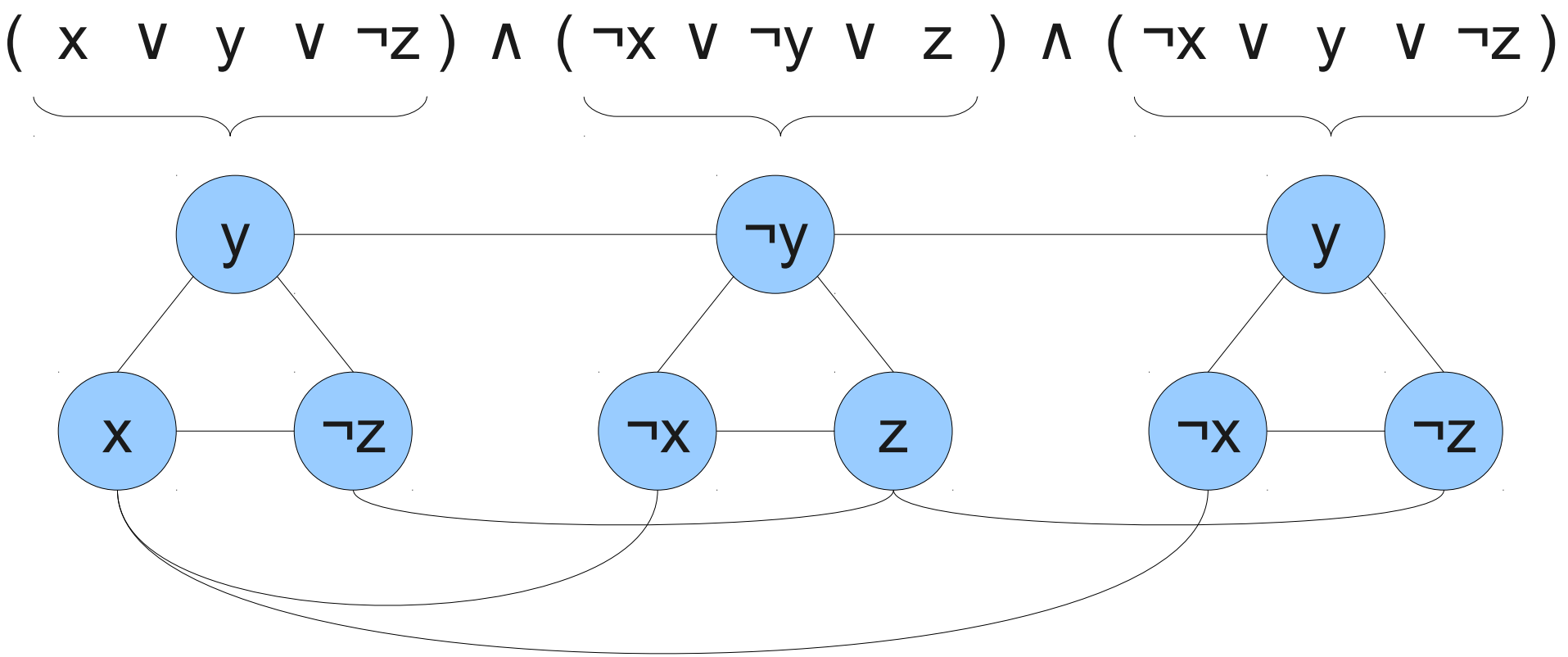
Gadgets in INDSET



Gadgets in INDSET

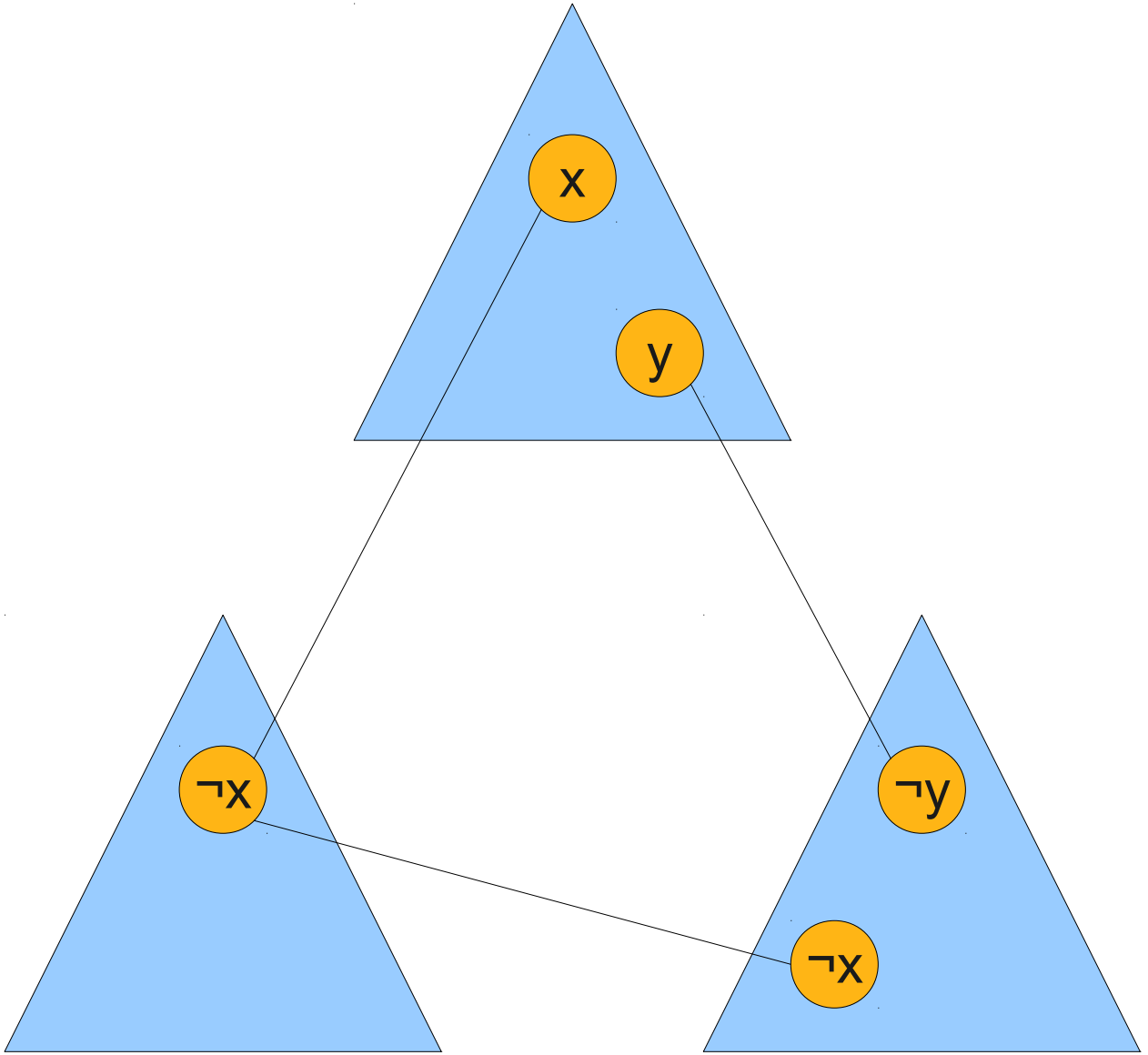


Gadgets in INDSET

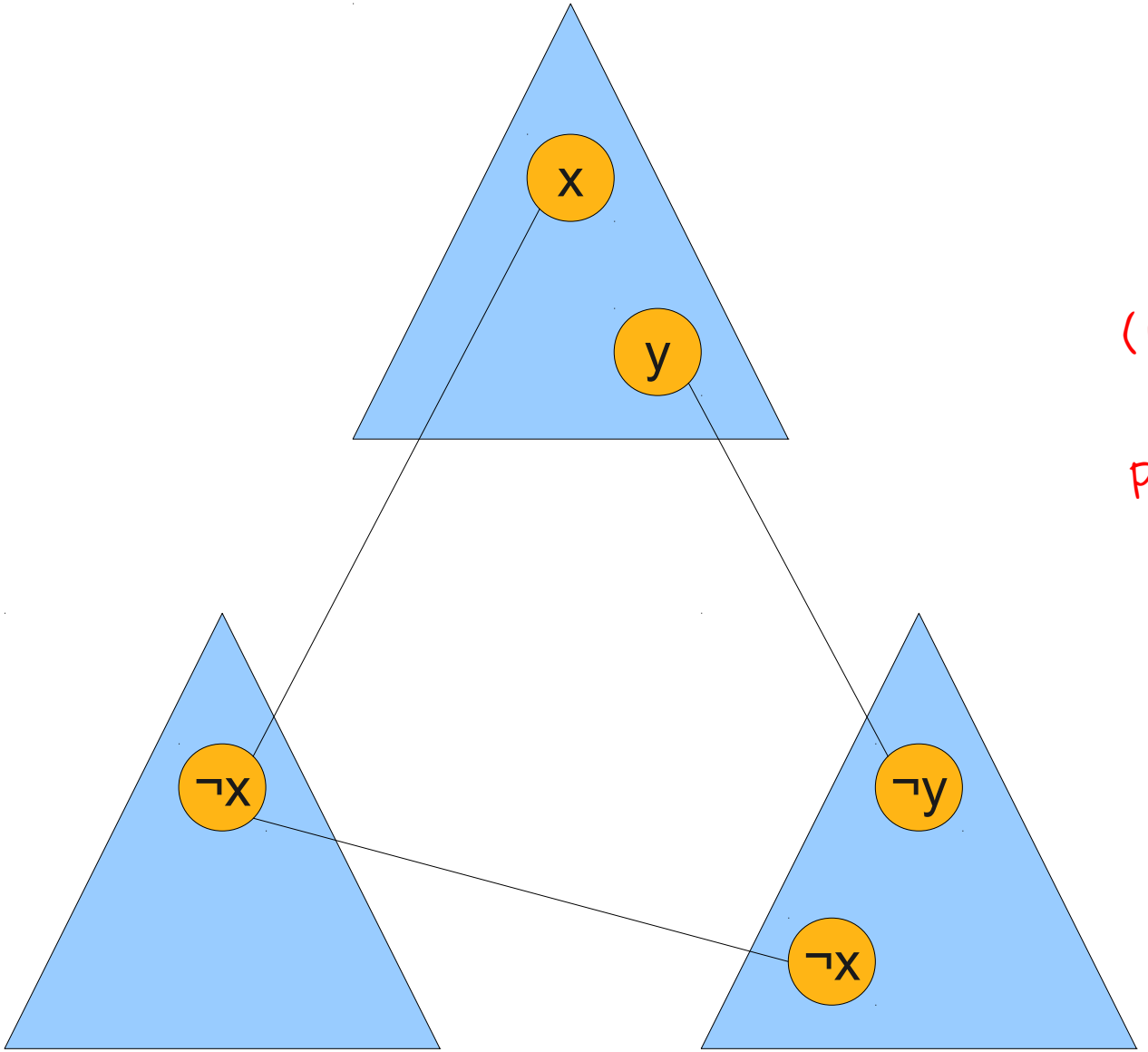


These connections ensure that the solutions to each gadget are linked to one another.

Gadgets in INDSET

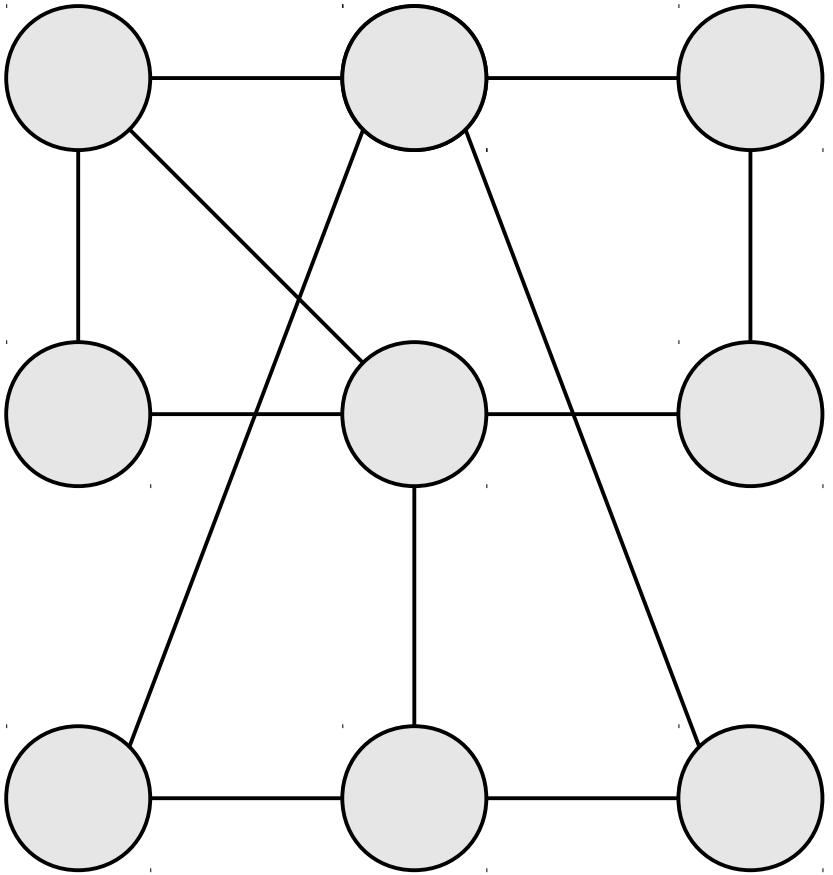


Gadgets in INDSET

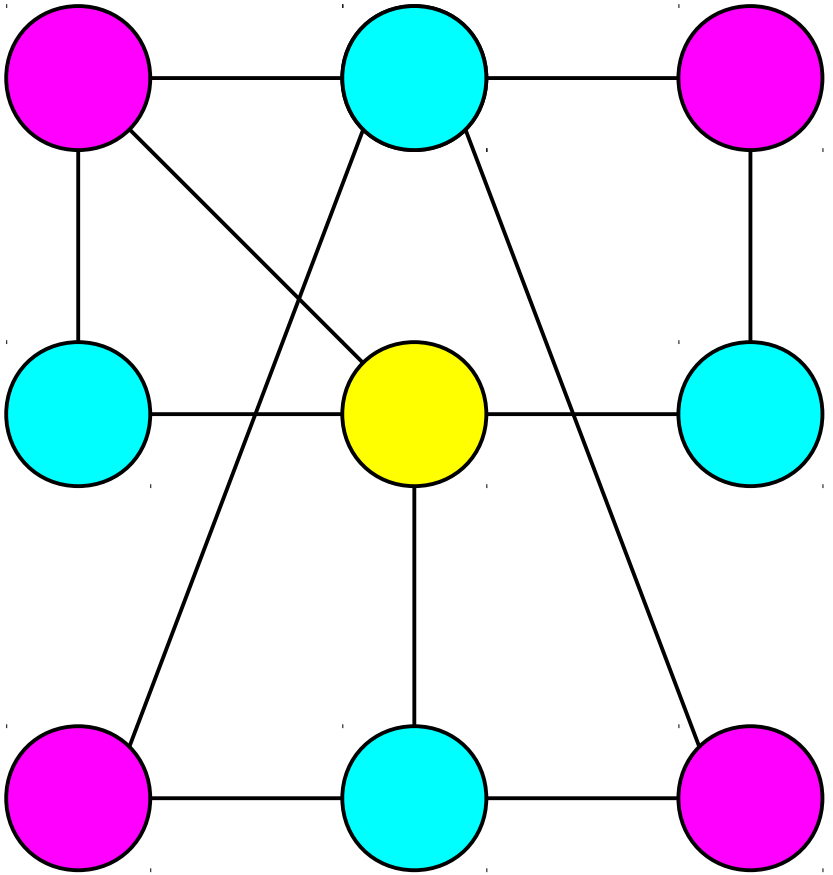


This schematic shows (roughly) how the gadgets are assembled. Many proofs or descriptions of reductions will use drawings like these.

A **3-coloring** of a graph is a way of coloring its nodes one of three colors such that no two connected nodes have the same color.



A **3-coloring** of a graph is a way of coloring its nodes one of three colors such that no two connected nodes have the same color.



A **3-coloring** of a graph is a way of coloring its nodes one of three colors such that no two connected nodes have the same color.

The 3-Coloring Problem

- The **3-coloring problem** is

  Given an undirected graph G,
  is there a legal 3-coloring of its nodes?

- As a formal language:

  3COLOR = { ⟨G⟩ | G is an undirected graph with a legal 3-coloring. }

- This problem is known to be NP-complete by a reduction from 3SAT.

3COLOR $\in$ NP

- We can prove that 3COLOR $\in$ NP by designing a polynomial-time nondeterministic TM for 3COLOR.
- M = “On input ⟨G⟩:
  - **Nondeterministically** guess an assignment of colors to the nodes.
  - **Deterministically** check whether it is a 3-coloring.
  - If so, accept; otherwise reject.”

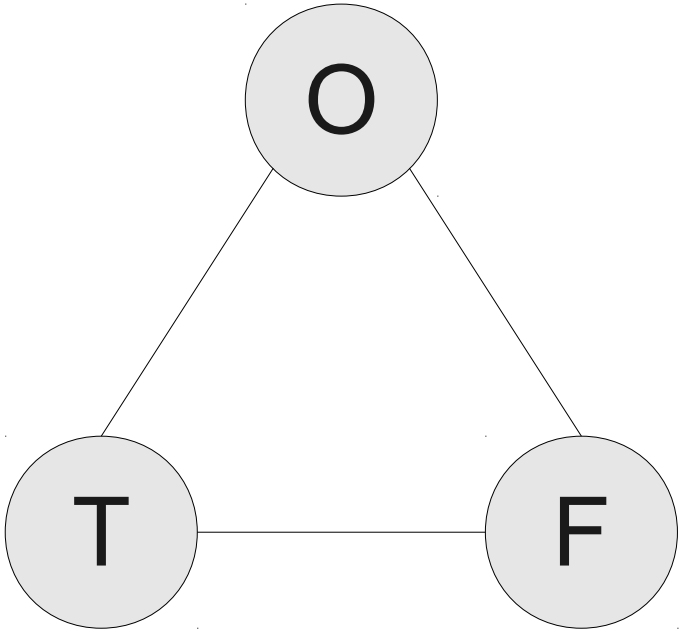
Why Not Two Colors?

- Although 3COLOR and 3SAT both have 3 in their names, the two are very different problems.
  - 3SAT deals with boolean variables, which can be either true or false.
  - 3COLOR deals with nodes, which can have one of three colors.
- It would seem that 2COLOR (whether a graph has a 2-coloring) would be a better fit.
- Interestingly, 2COLOR is known to be in P and is conjectured not to be NP-complete.
  - Though, if you can prove that it is, you've just won $1,000,000!

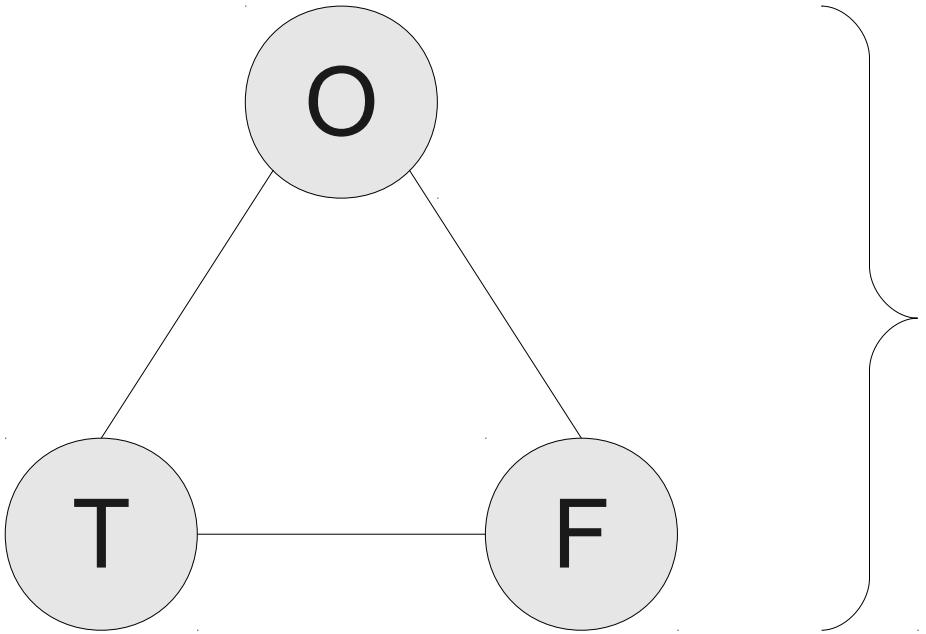
From 3SAT to 3COLOR

- In order to reduce 3SAT to 3COLOR, we need to somehow make a graph that is 3-colorable iff some 3-CNF formula φ is satisfiable.
- **Idea**: Use a collection of gadgets to solve the problem.
  - Build a gadget to assign two of the colors the labels “true” and “false.”
  - Build a gadget to force each variable to be either true or false.
  - Build a series of gadgets to force those variable assignments to satisfy each clause.

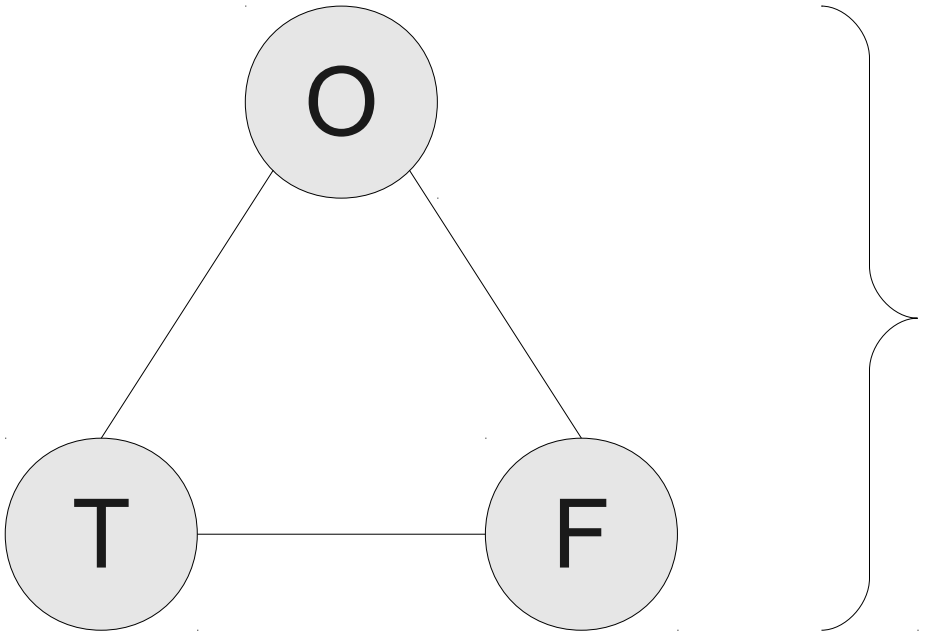
Gadget One: Assigning Meanings



Gadget One: Assigning Meanings

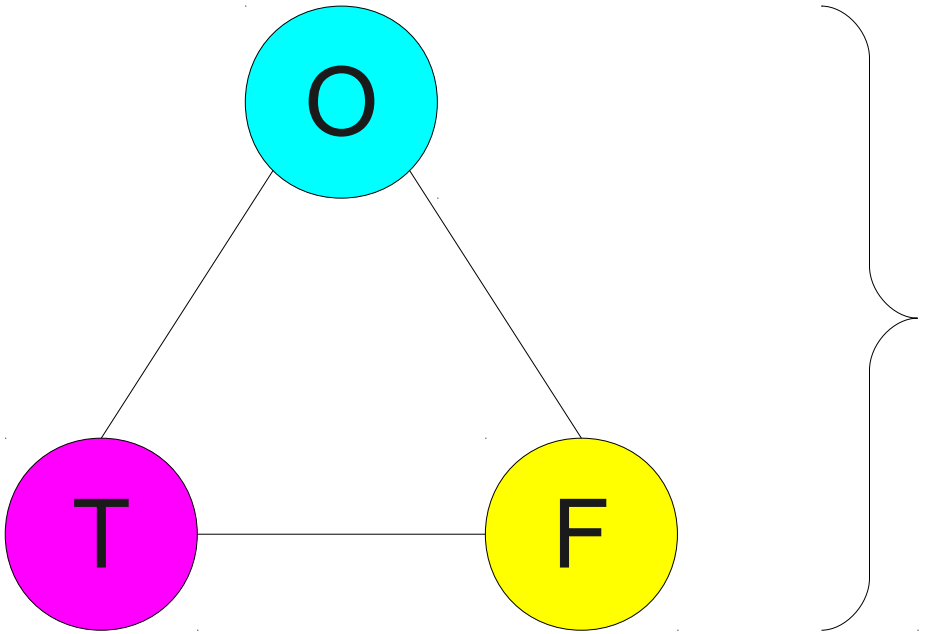


Gadget One: Assigning Meanings



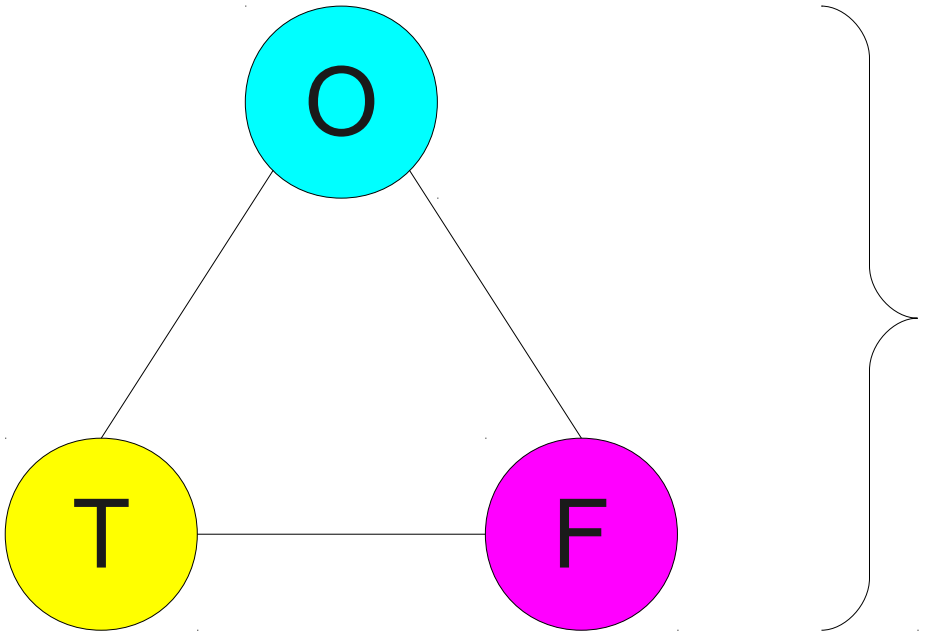
The color assigned to T will be interpreted as "true."
The color assigned to F will be interpreted as "false."
We do not associate any special meaning with O.

Gadget One: Assigning Meanings



The color assigned to T will be interpreted as "true."
The color assigned to F will be interpreted as "false."
We do not associate any special meaning with O.

Gadget One: Assigning Meanings



The color assigned to T will be interpreted as "true."
The color assigned to F will be interpreted as "false."
We do not associate any special meaning with O.

Gadget Two: Forcing a Choice

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

Gadget Two: Forcing a Choice

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

Gadget Two: Forcing a Choice

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

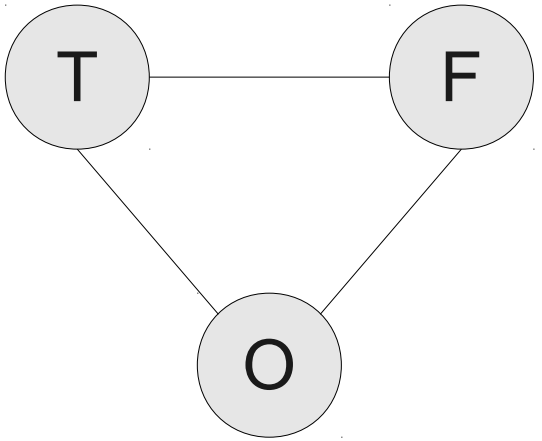
Gadget Two: Forcing a Choice

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

We need to ensure that no literal and its complement become true at the same time.

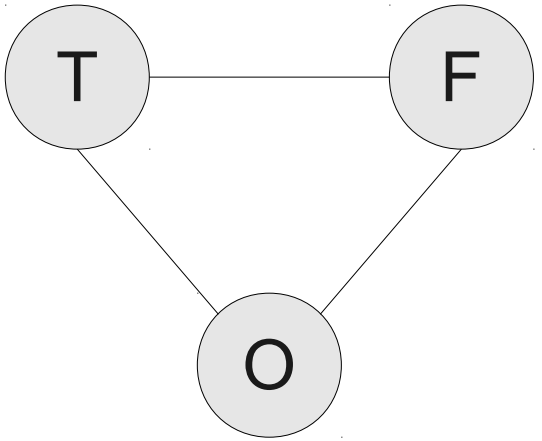
Gadget Two: Forcing a Choice

(x ∨ y ∨ ¬z) ∧ (¬x ∨ ¬y ∨ z) ∧ (¬x ∨ y ∨ ¬z)



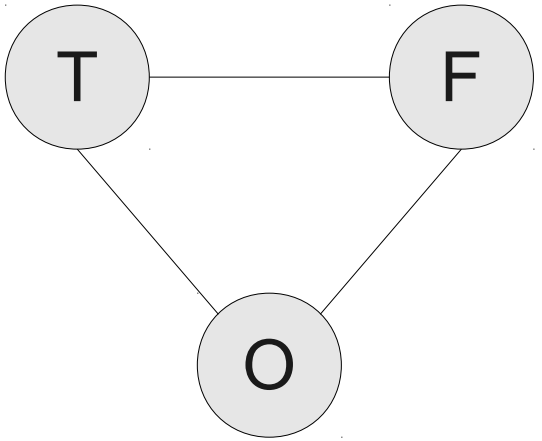
Gadget Two: Forcing a Choice

(x ∨ y ∨ ¬z) ∧ (¬x ∨ ¬y ∨ z) ∧ (¬x ∨ y ∨ ¬z)



Gadget Two: Forcing a Choice

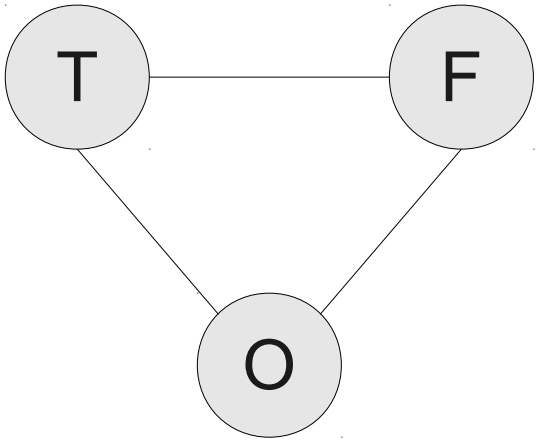
(x ∨ y ∨ ¬z) ∧ (¬x ∨ ¬y ∨ z) ∧ (¬x ∨ y ∨ ¬z)



These gadgets ensure that any variable and its negation don't both get assigned the "true" color.

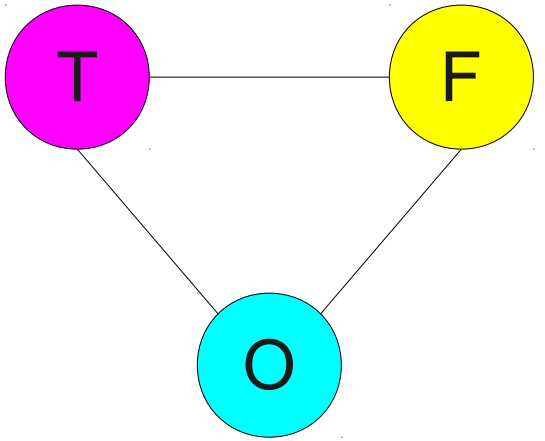
Gadget Two: Forcing a Choice

(x ∨ y ∨ ¬z) ∧ (¬x ∨ ¬y ∨ z) ∧ (¬x ∨ y ∨ ¬z)



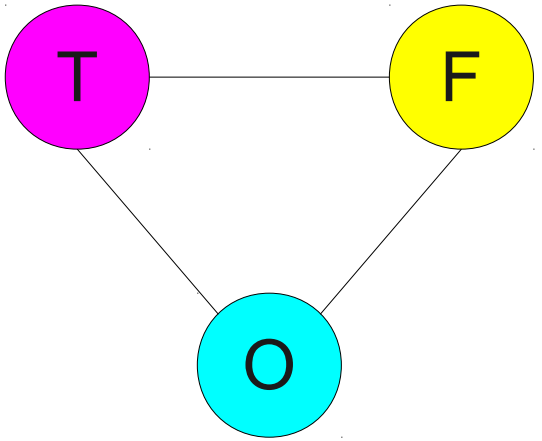
Gadget Two: Forcing a Choice

(x ∨ y ∨ ¬z) ∧ (¬x ∨ ¬y ∨ z) ∧ (¬x ∨ y ∨ ¬z)



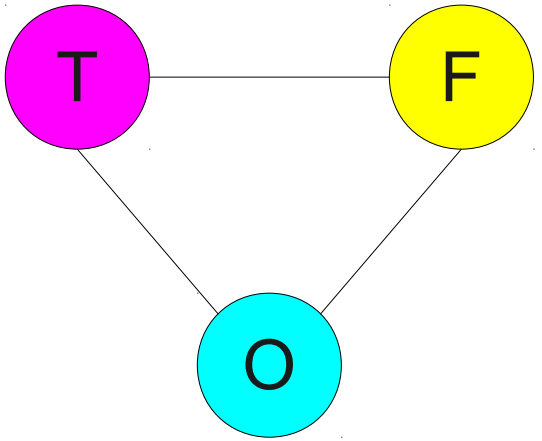
Gadget Two: Forcing a Choice

(x ∨ y ∨ ¬z) ∧ (¬x ∨ ¬y ∨ z) ∧ (¬x ∨ y ∨ ¬z)



Gadget Two: Forcing a Choice

(x ∨ y ∨ ¬z) ∧ (¬x ∨ ¬y ∨ z) ∧ (¬x ∨ y ∨ ¬z)



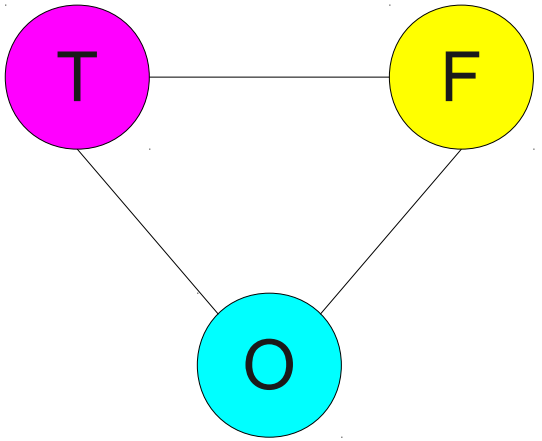
x is true

¬y is true

¬z is true

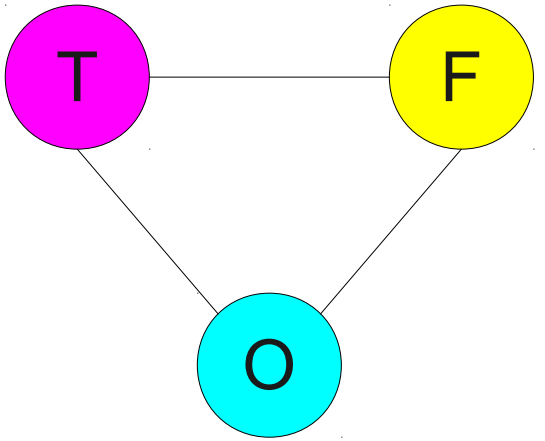
Gadget Two: Forcing a Choice

(x ∨ y ∨ ¬z) ∧ (¬x ∨ ¬y ∨ z) ∧ (¬x ∨ y ∨ ¬z)



Gadget Two: Forcing a Choice

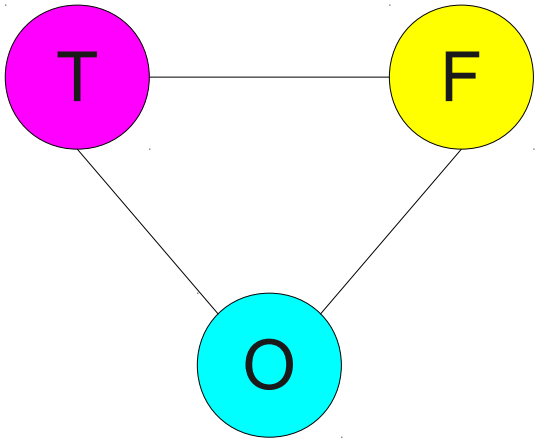
(x ∨ y ∨ ¬z) ∧ (¬x ∨ ¬y ∨ z) ∧ (¬x ∨ y ∨ ¬z)



We need to ensure that none of these nodes get colored with the "other" color!

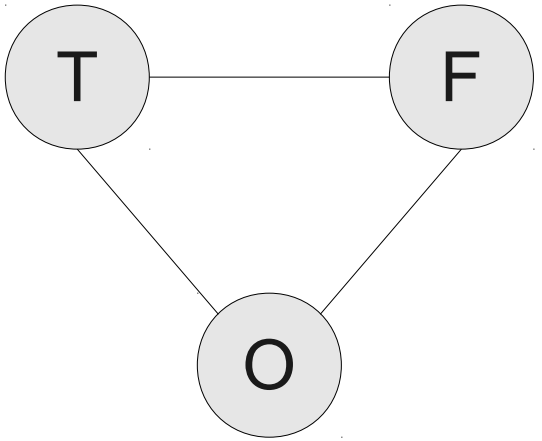
Gadget Two: Forcing a Choice

(x ∨ y ∨ ¬z) ∧ (¬x ∨ ¬y ∨ z) ∧ (¬x ∨ y ∨ ¬z)



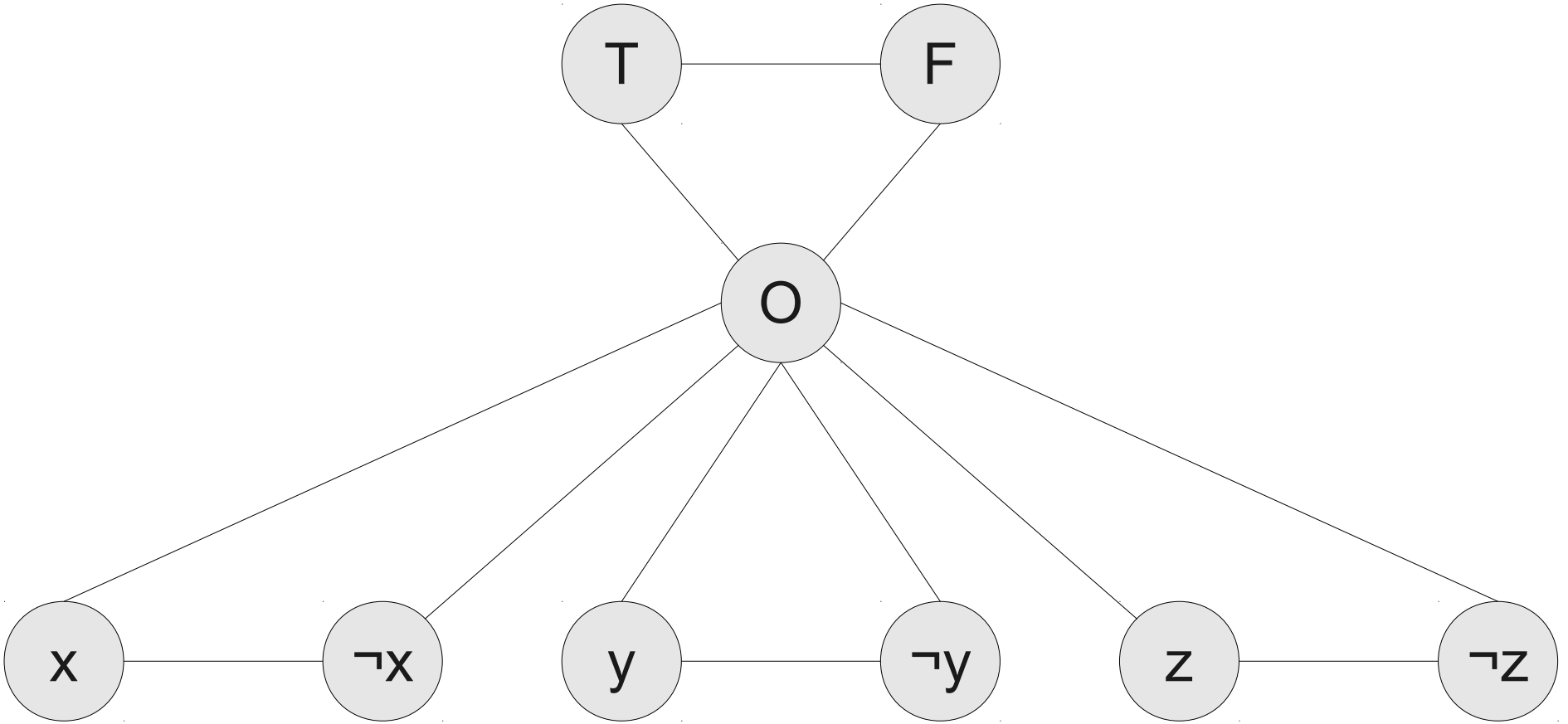
Gadget Two: Forcing a Choice

(x ∨ y ∨ ¬z) ∧ (¬x ∨ ¬y ∨ z) ∧ (¬x ∨ y ∨ ¬z)



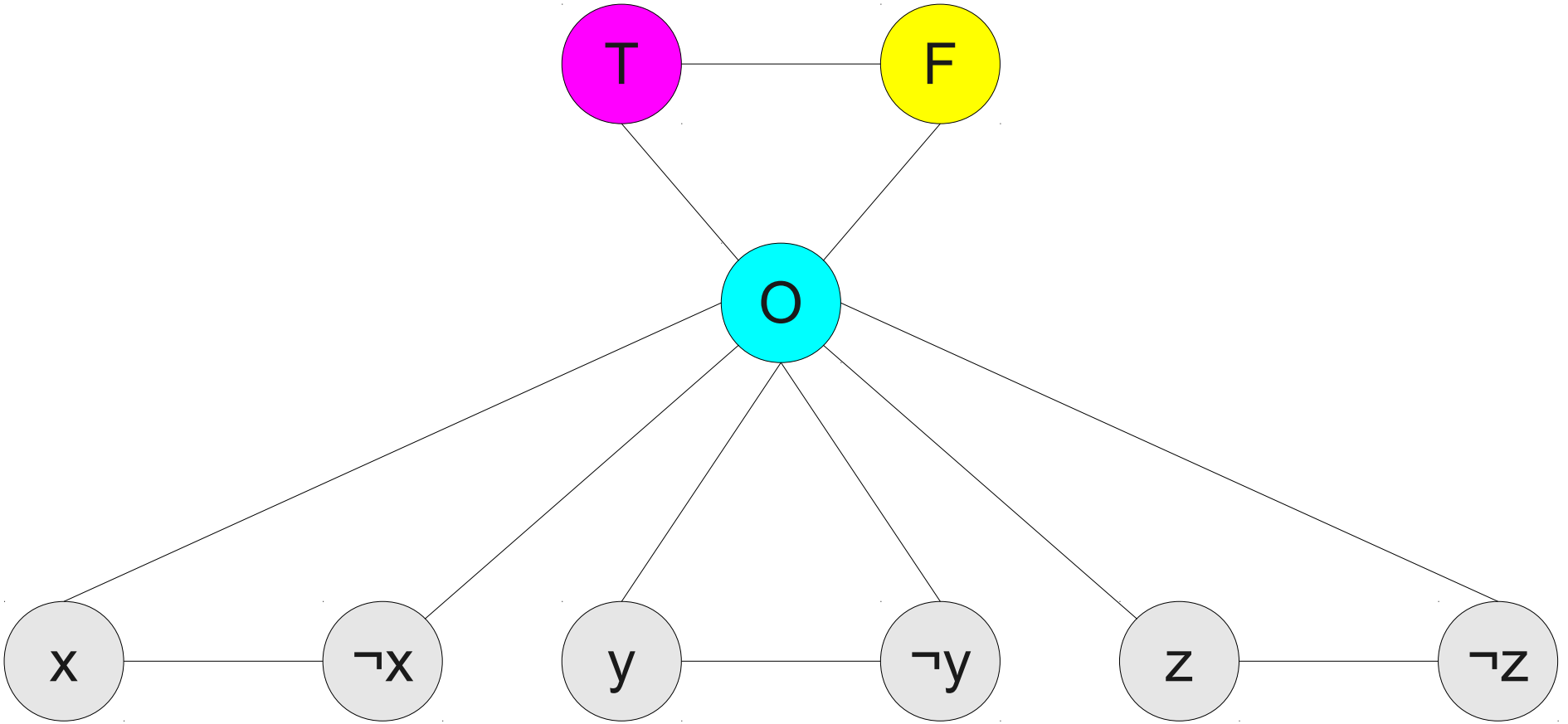
Gadget Two: Forcing a Choice

(x ∨ y ∨ ¬z) ∧ (¬x ∨ ¬y ∨ z) ∧ (¬x ∨ y ∨ ¬z)



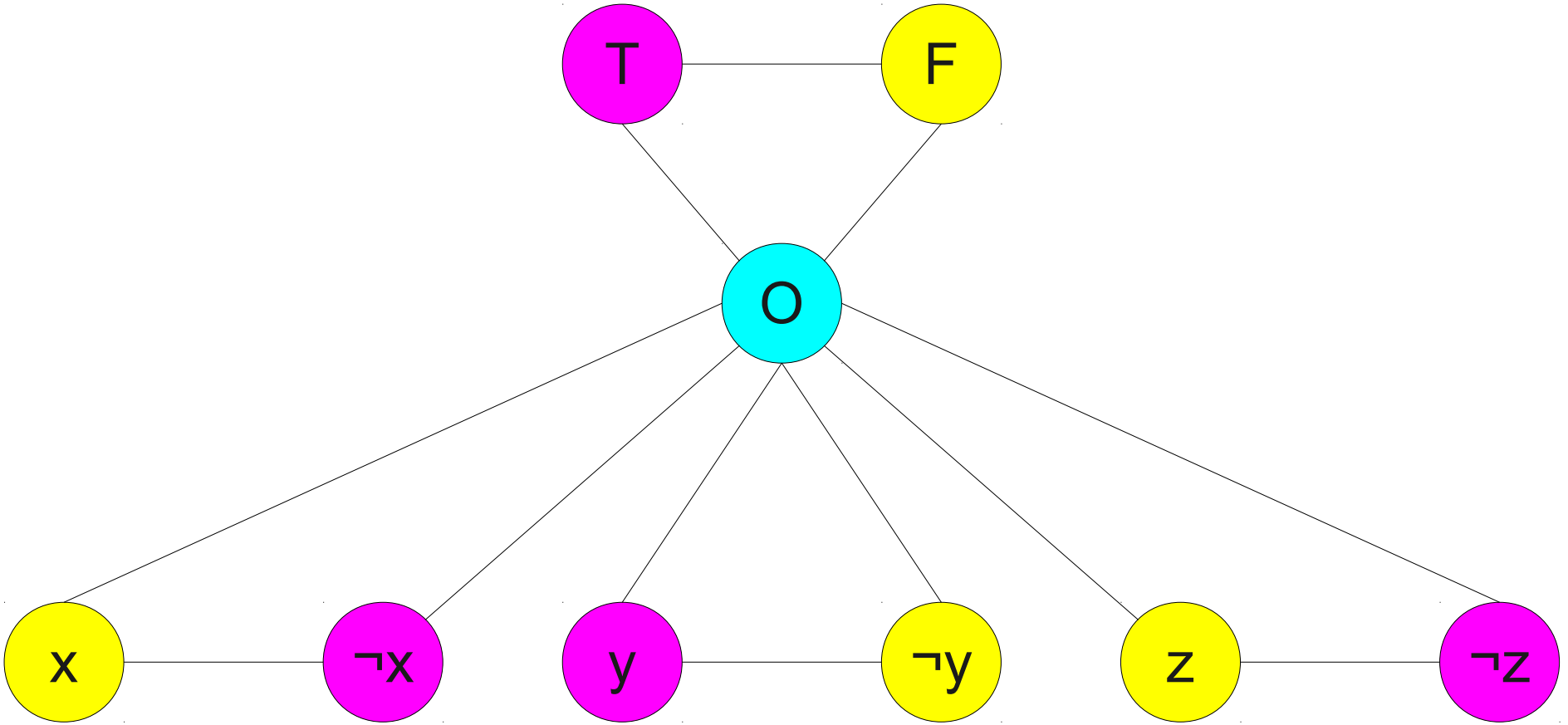
Gadget Two: Forcing a Choice

(x ∨ y ∨ ¬z) ∧ (¬x ∨ ¬y ∨ z) ∧ (¬x ∨ y ∨ ¬z)



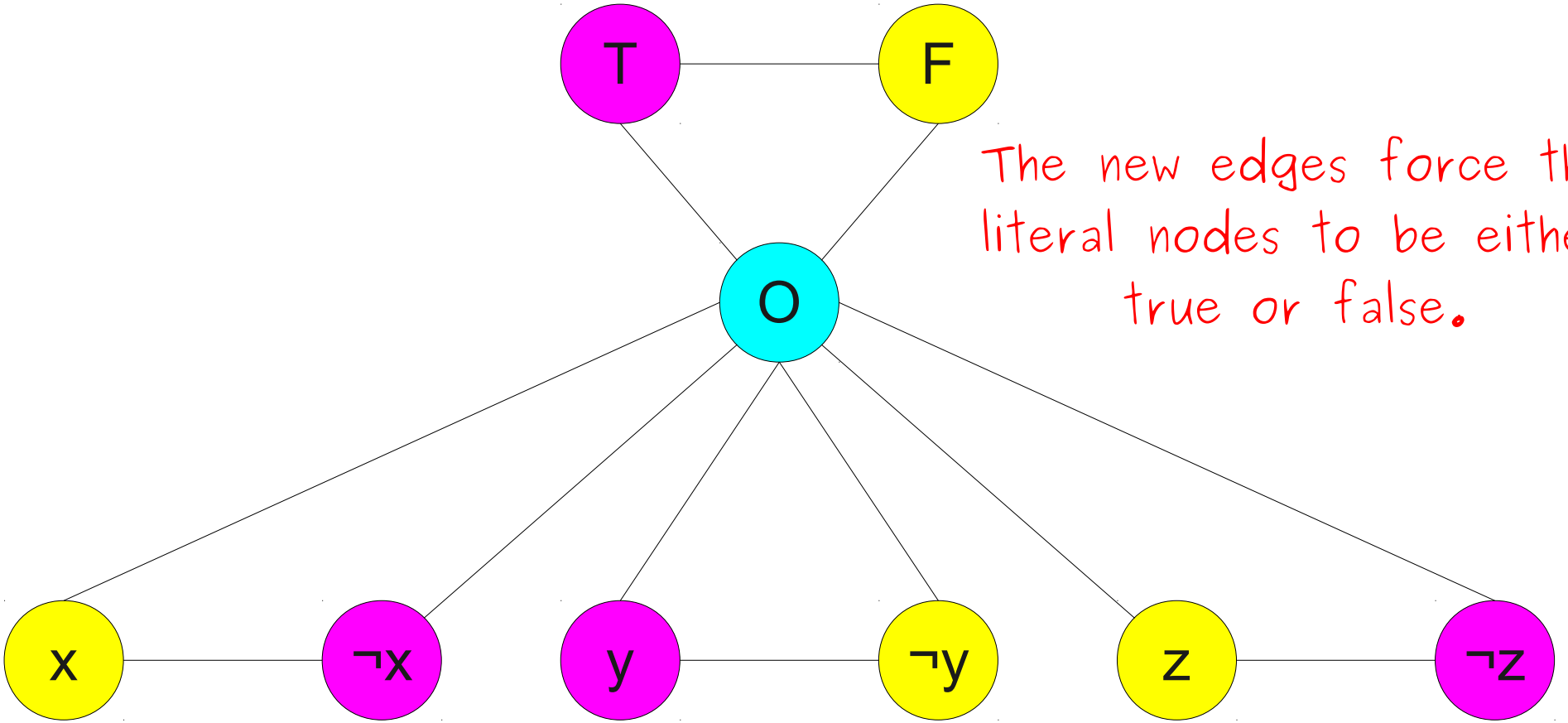
Gadget Two: Forcing a Choice

(x ∨ y ∨ ¬z) ∧ (¬x ∨ ¬y ∨ z) ∧ (¬x ∨ y ∨ ¬z)



Gadget Two: Forcing a Choice

(x ∨ y ∨ ¬z) ∧ (¬x ∨ ¬y ∨ z) ∧ (¬x ∨ y ∨ ¬z)

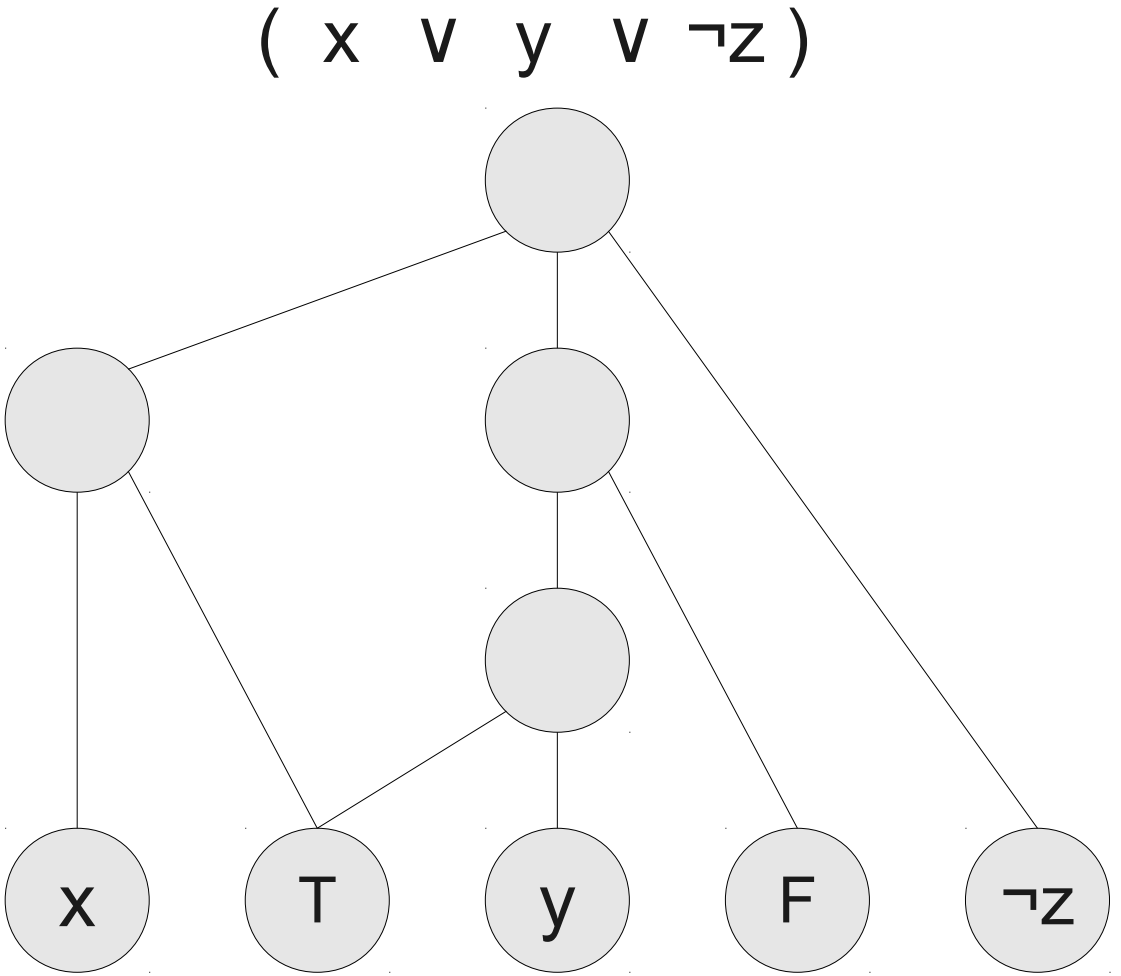


The new edges force the literal nodes to be either true or false.

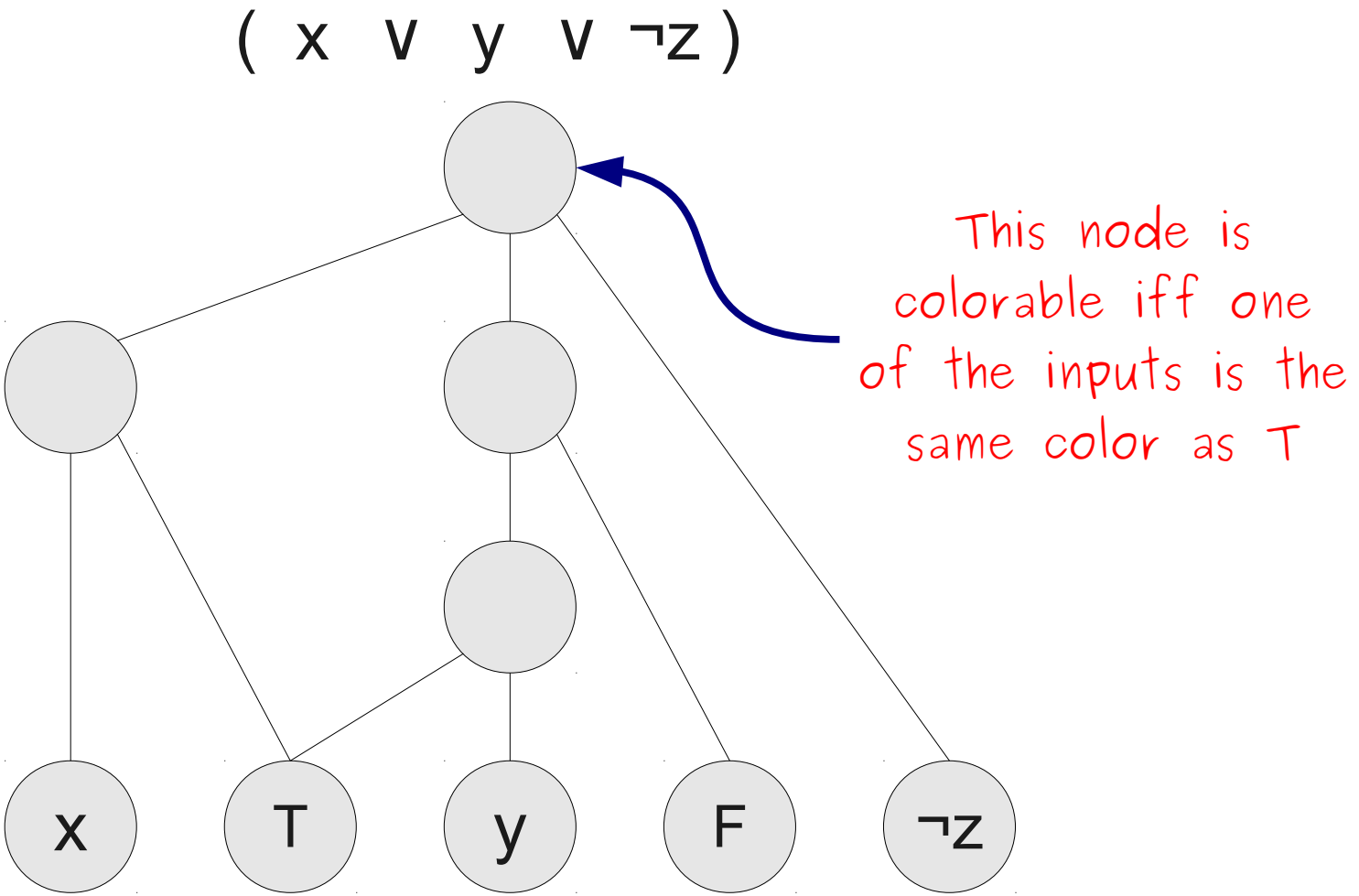
Gadget Three: Clause Satisfiability

(x ∨ y ∨ ¬z)

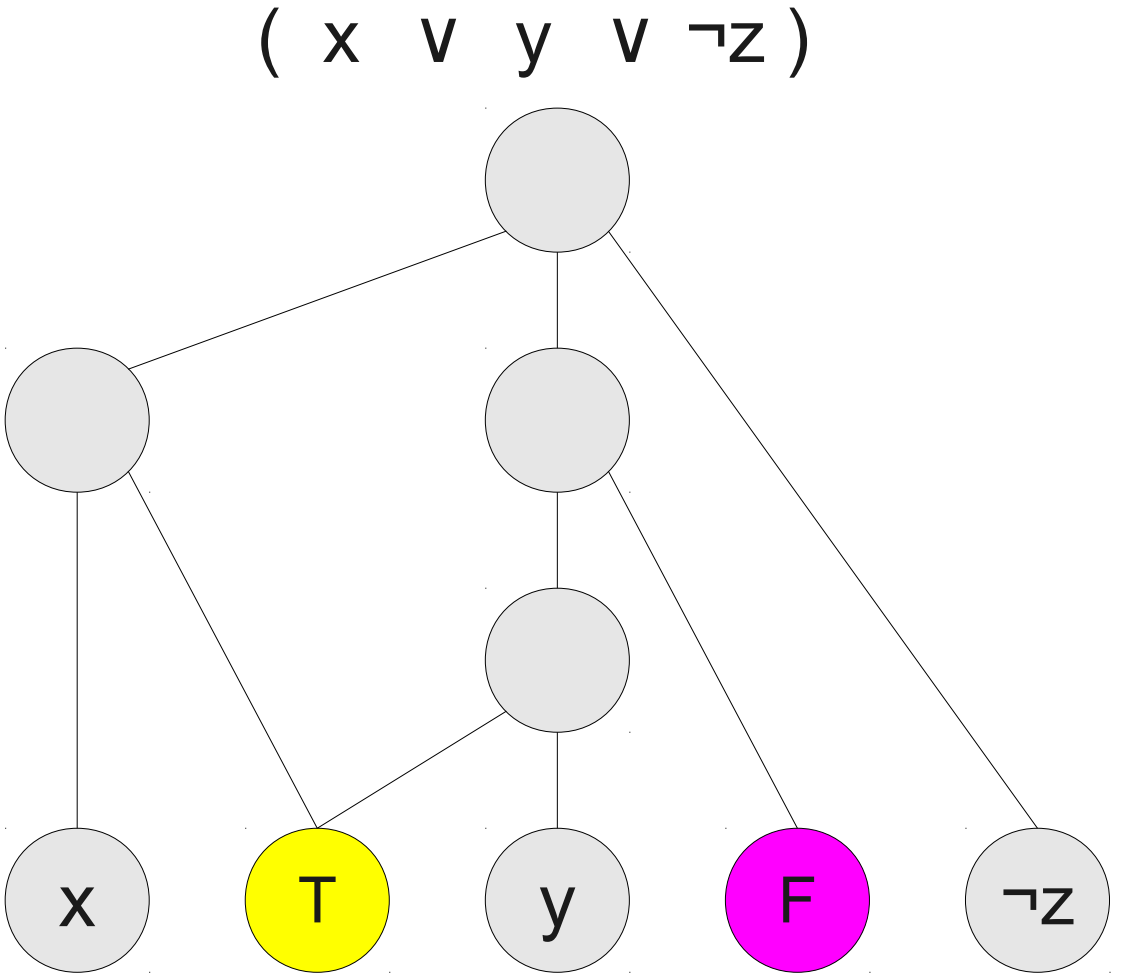
Gadget Three: Clause Satisfiability



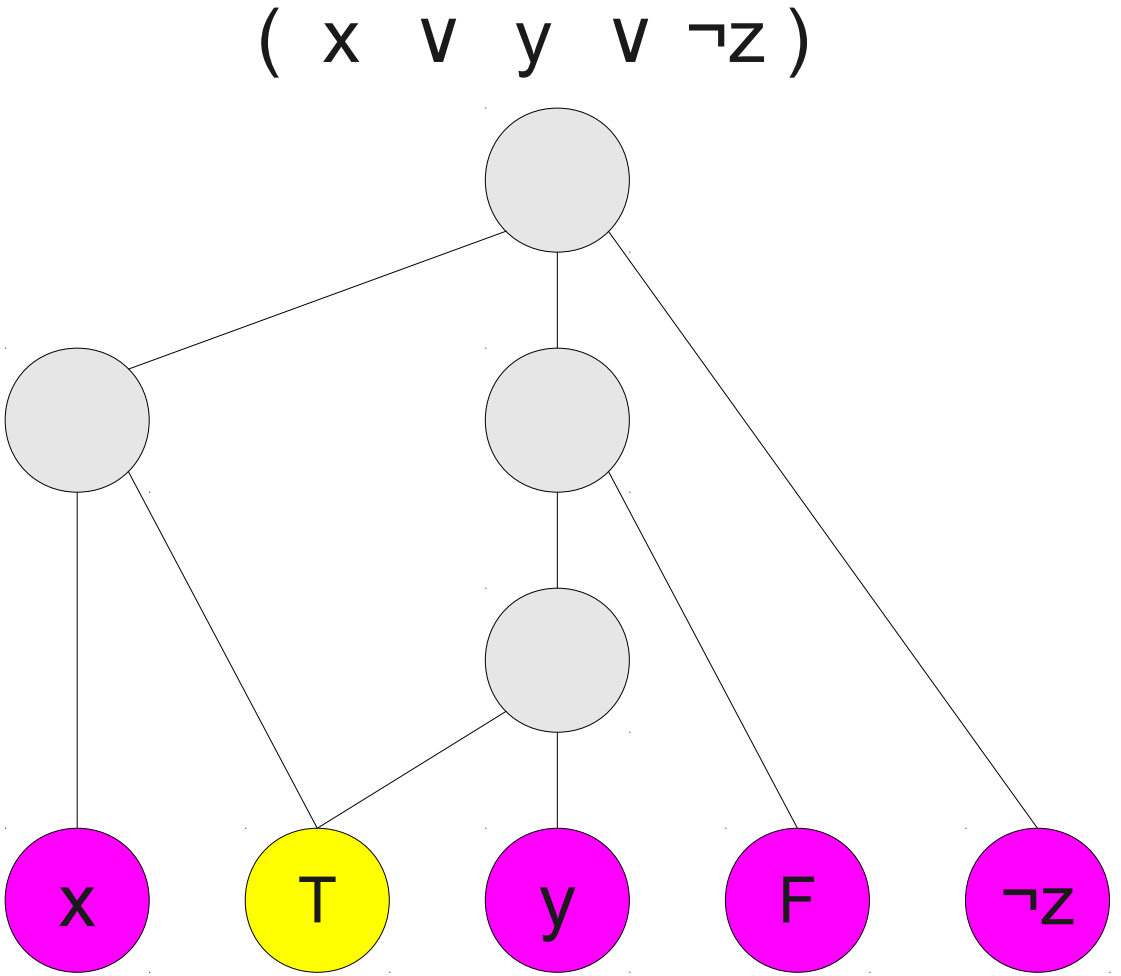
Gadget Three: Clause Satisfiability



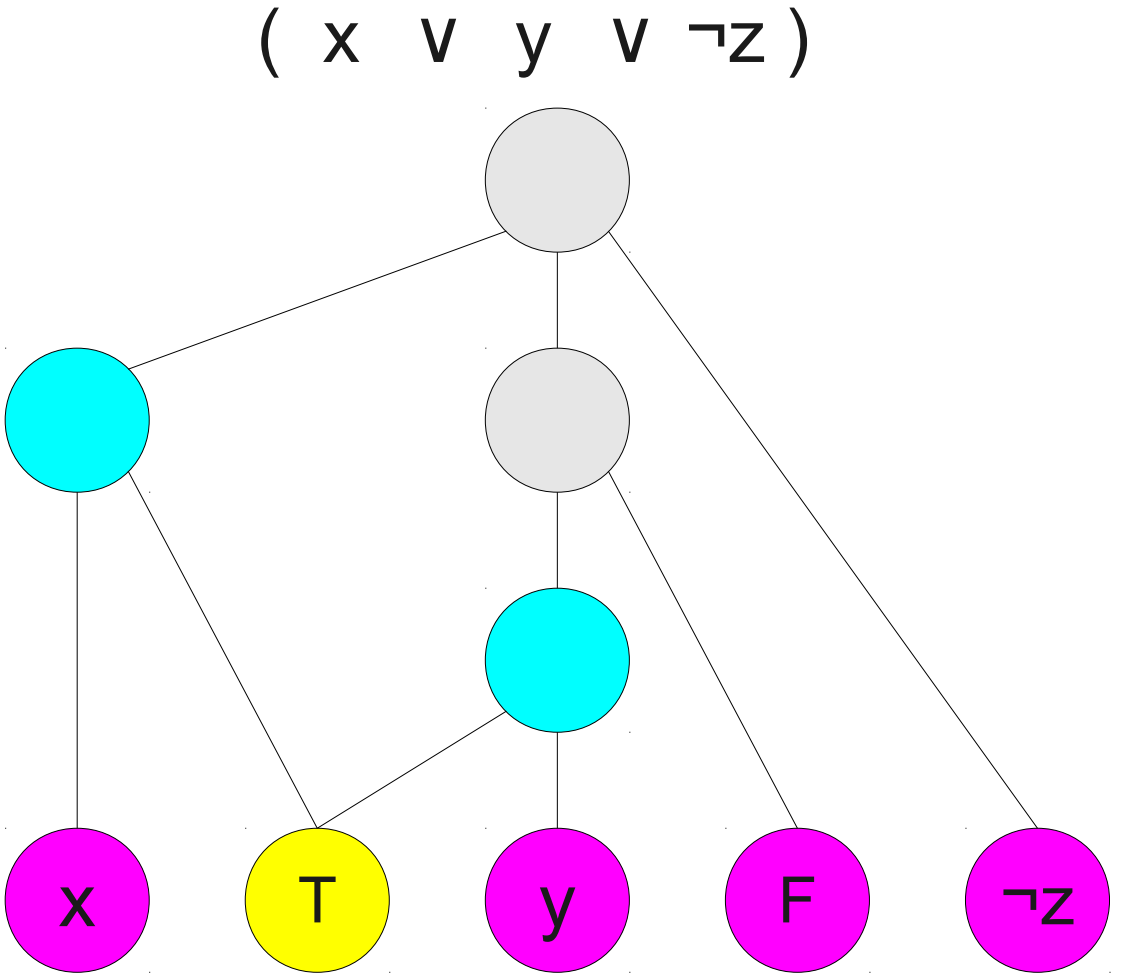
Gadget Three: Clause Satisfiability



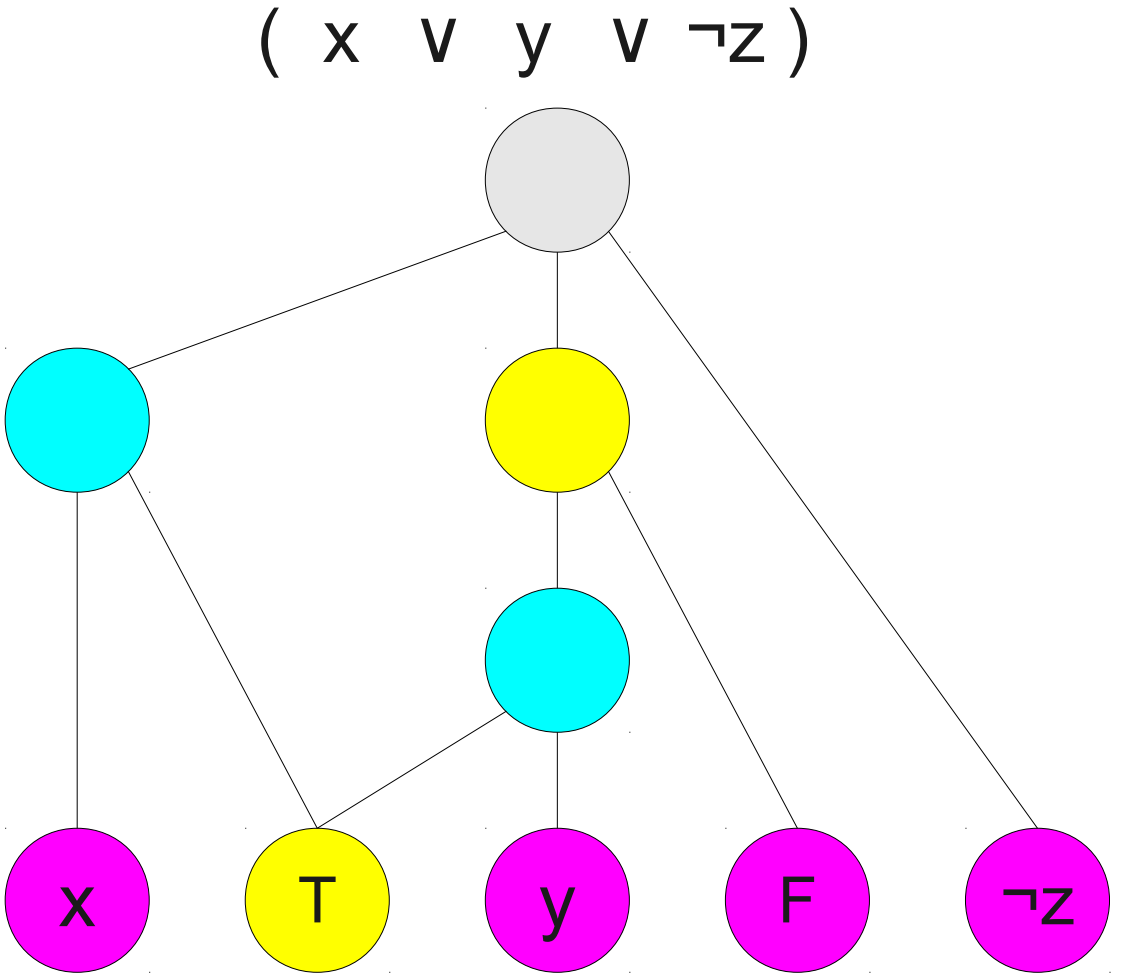
Gadget Three: Clause Satisfiability



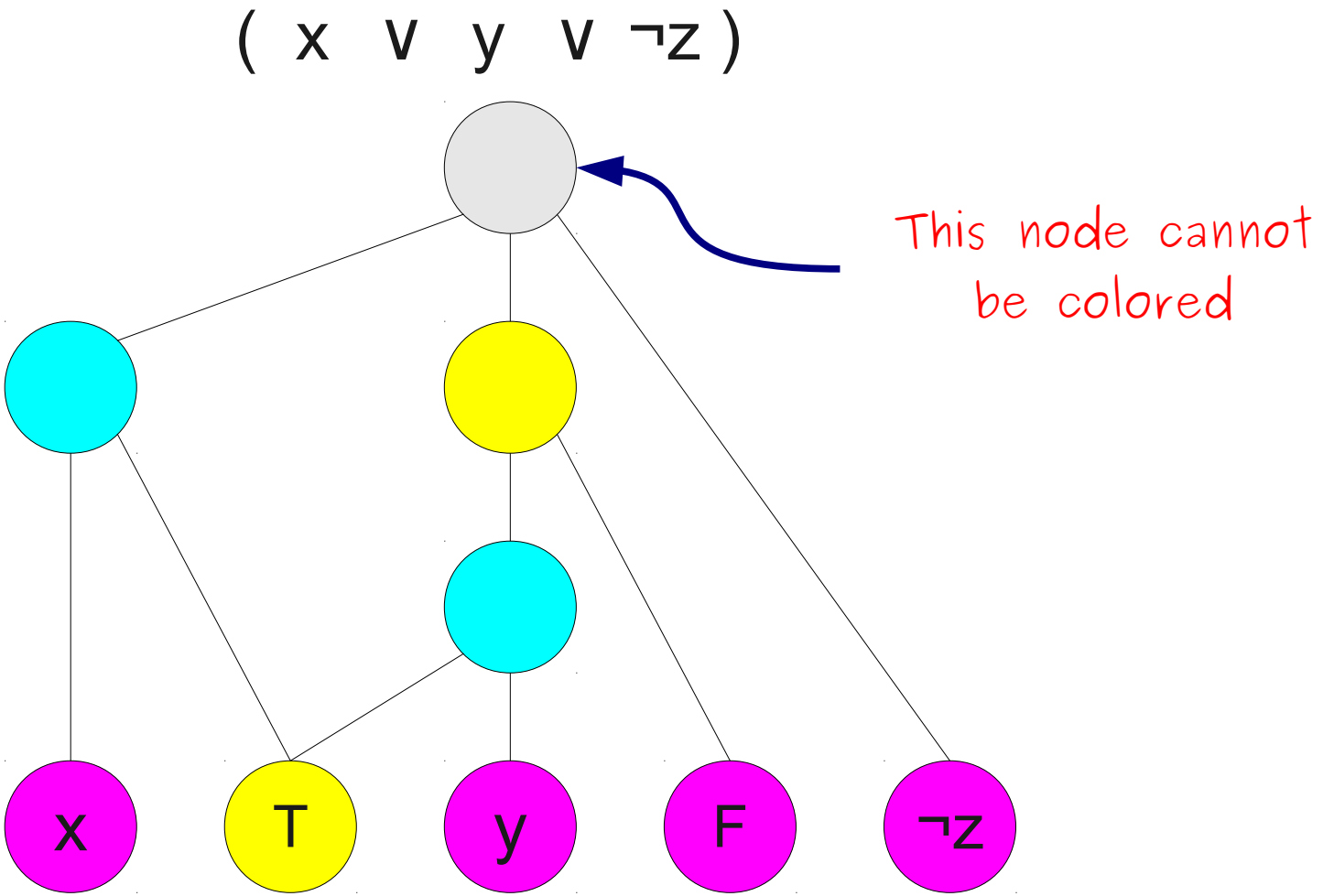
Gadget Three: Clause Satisfiability



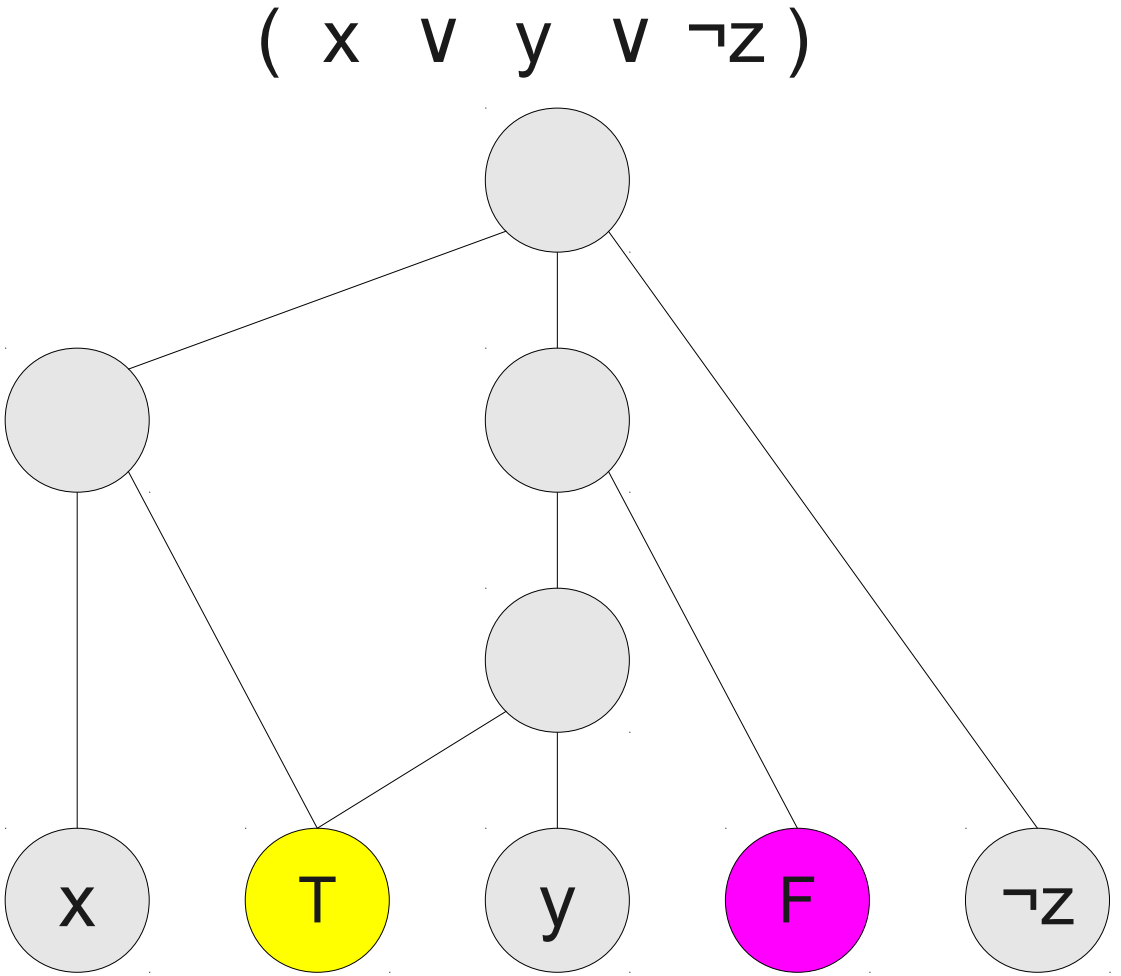
Gadget Three: Clause Satisfiability



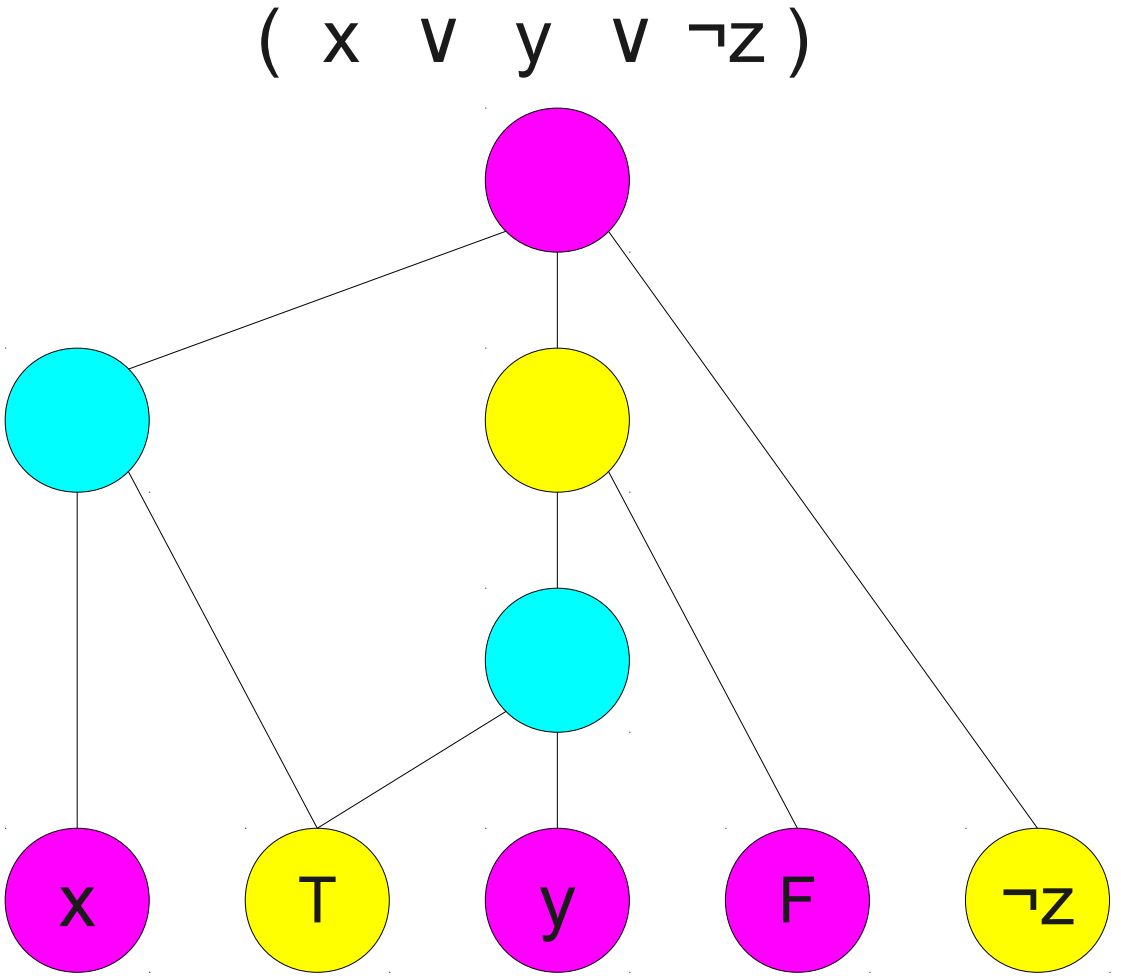
Gadget Three: Clause Satisfiability



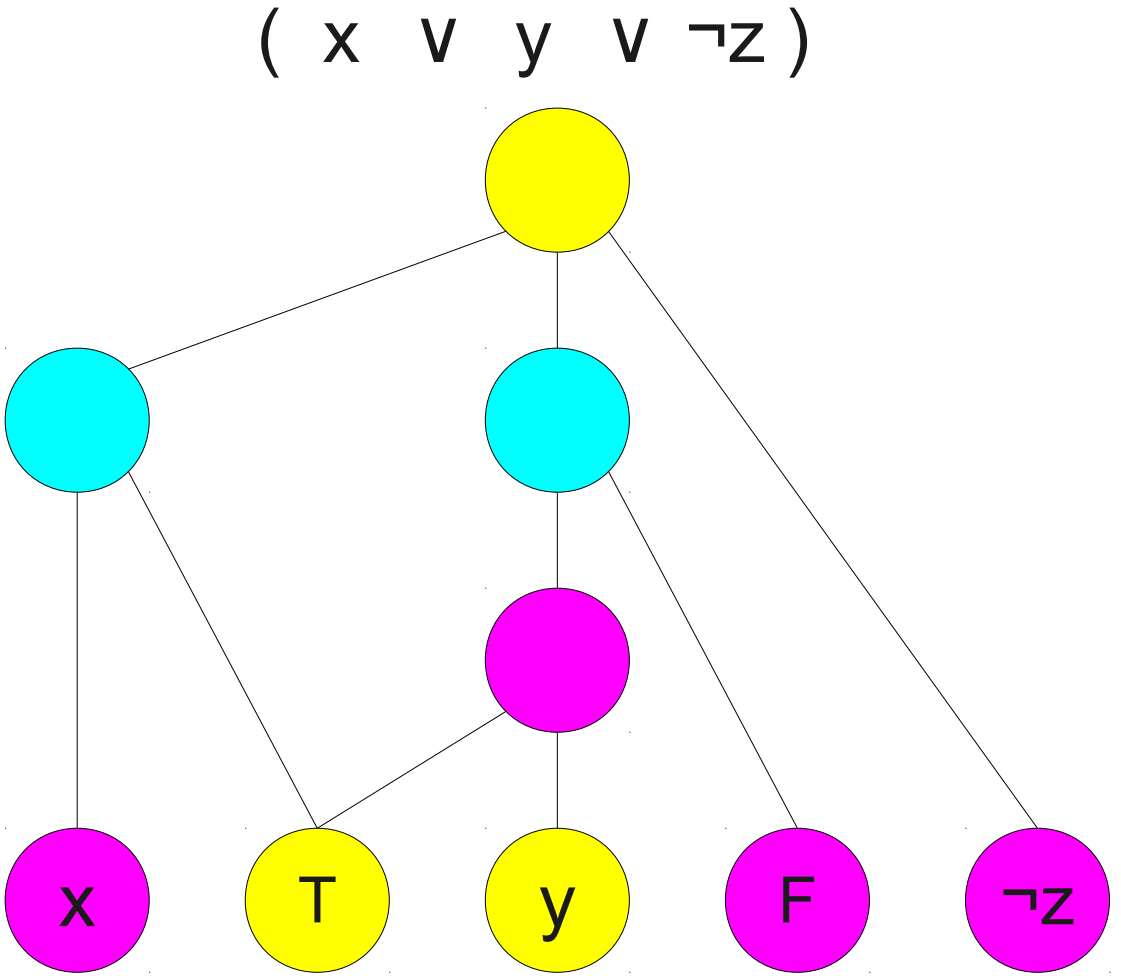
Gadget Three: Clause Satisfiability



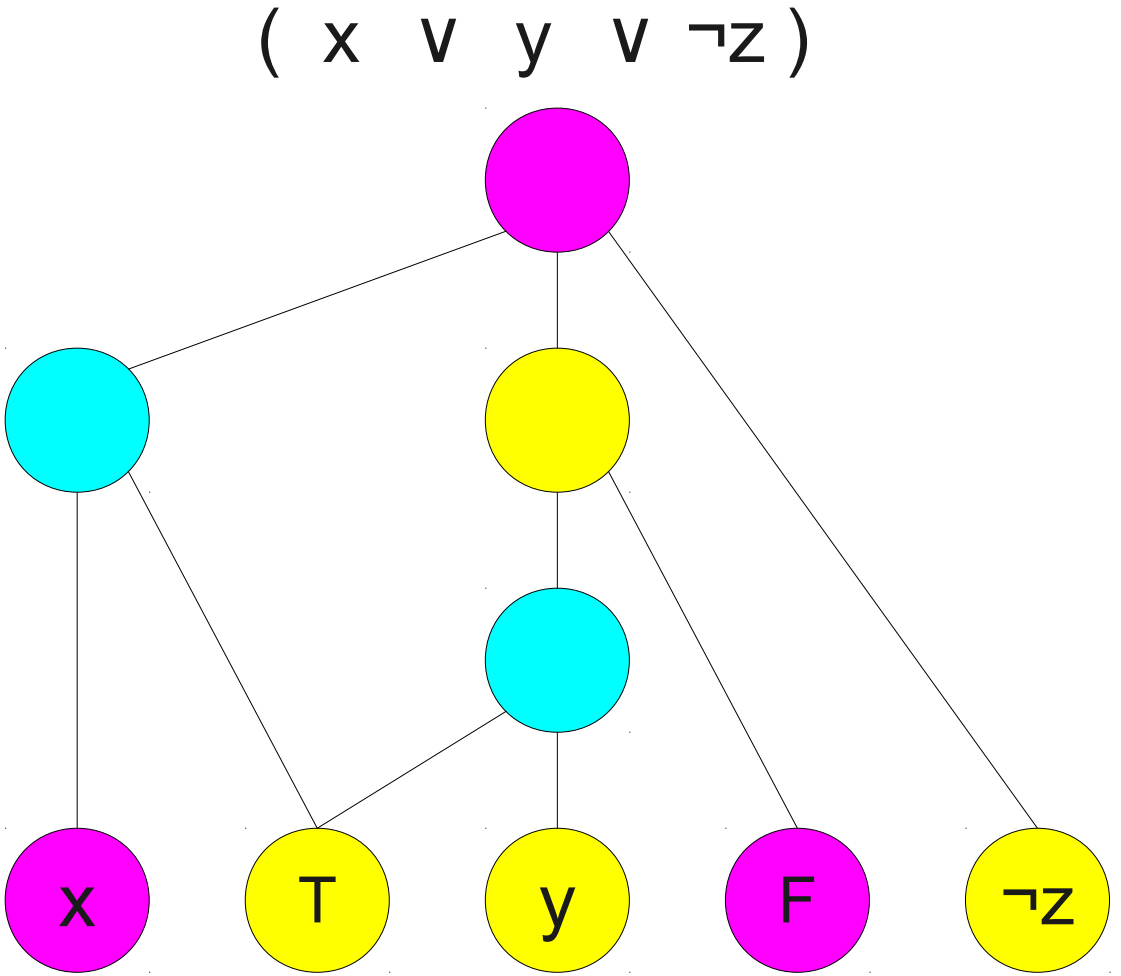
Gadget Three: Clause Satisfiability



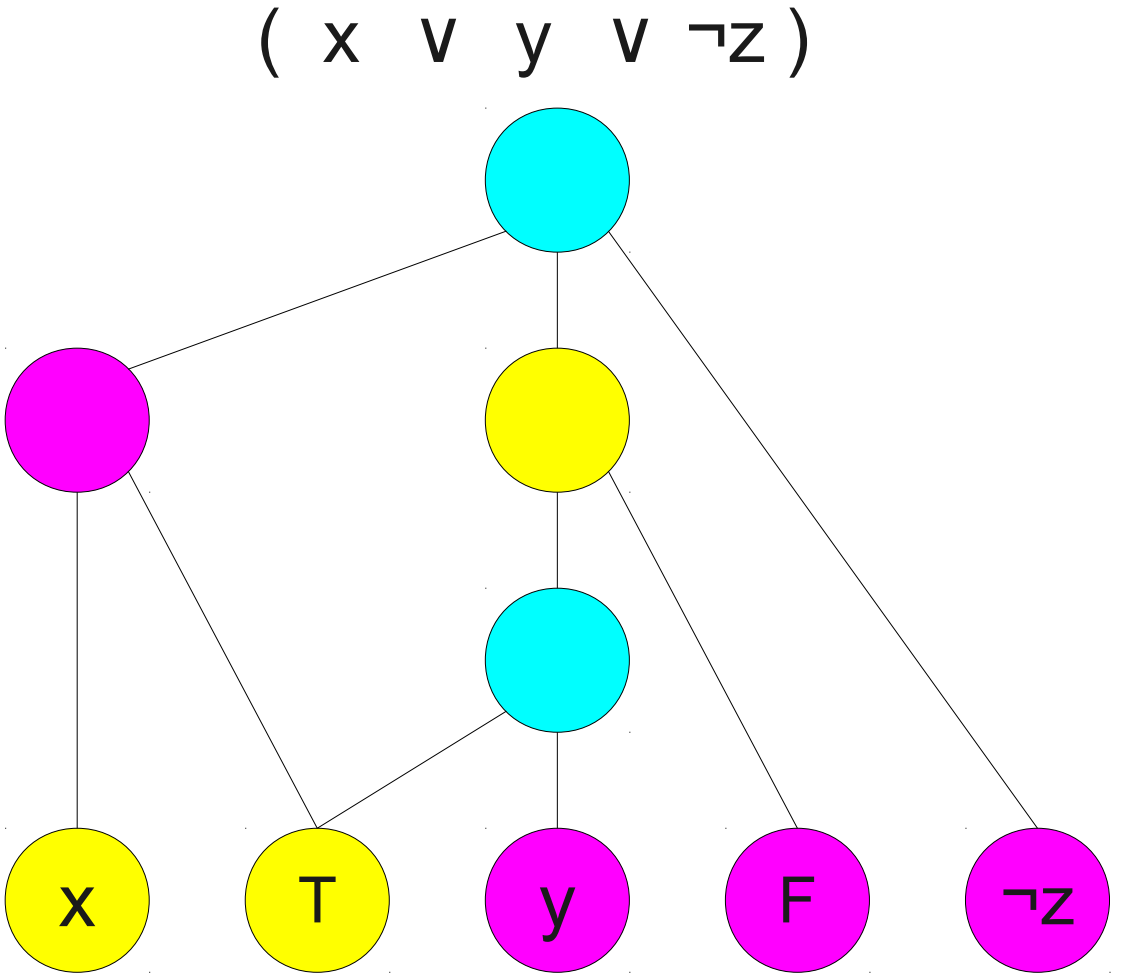
Gadget Three: Clause Satisfiability



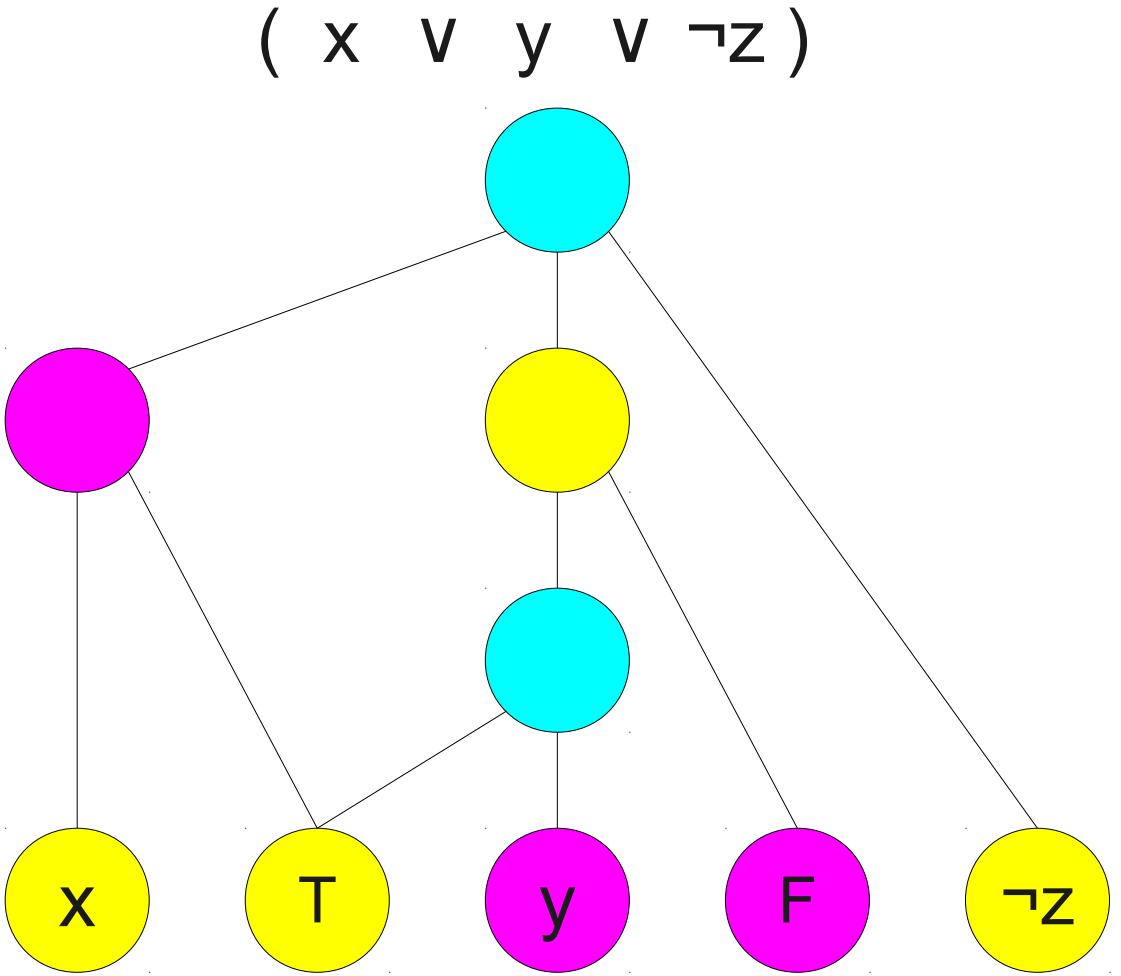
Gadget Three: Clause Satisfiability



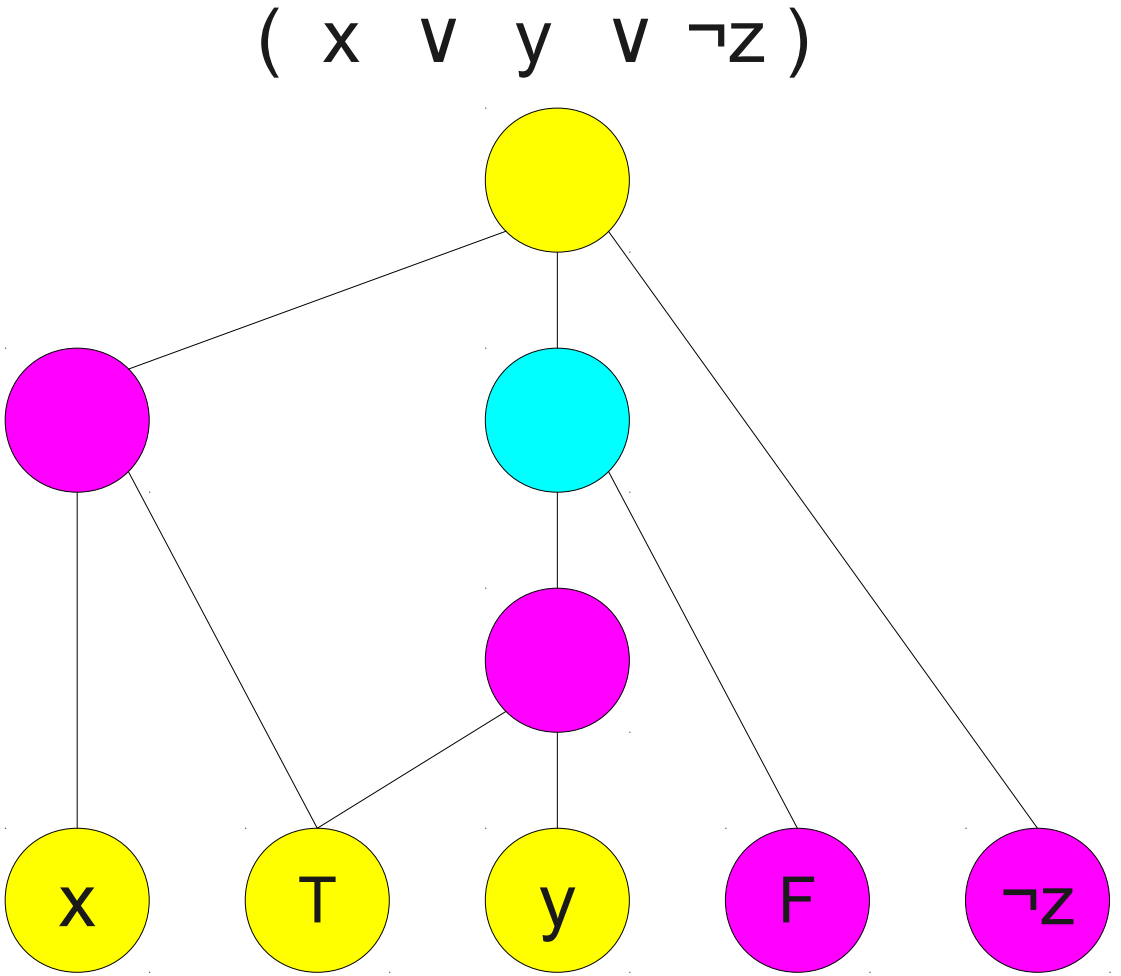
Gadget Three: Clause Satisfiability



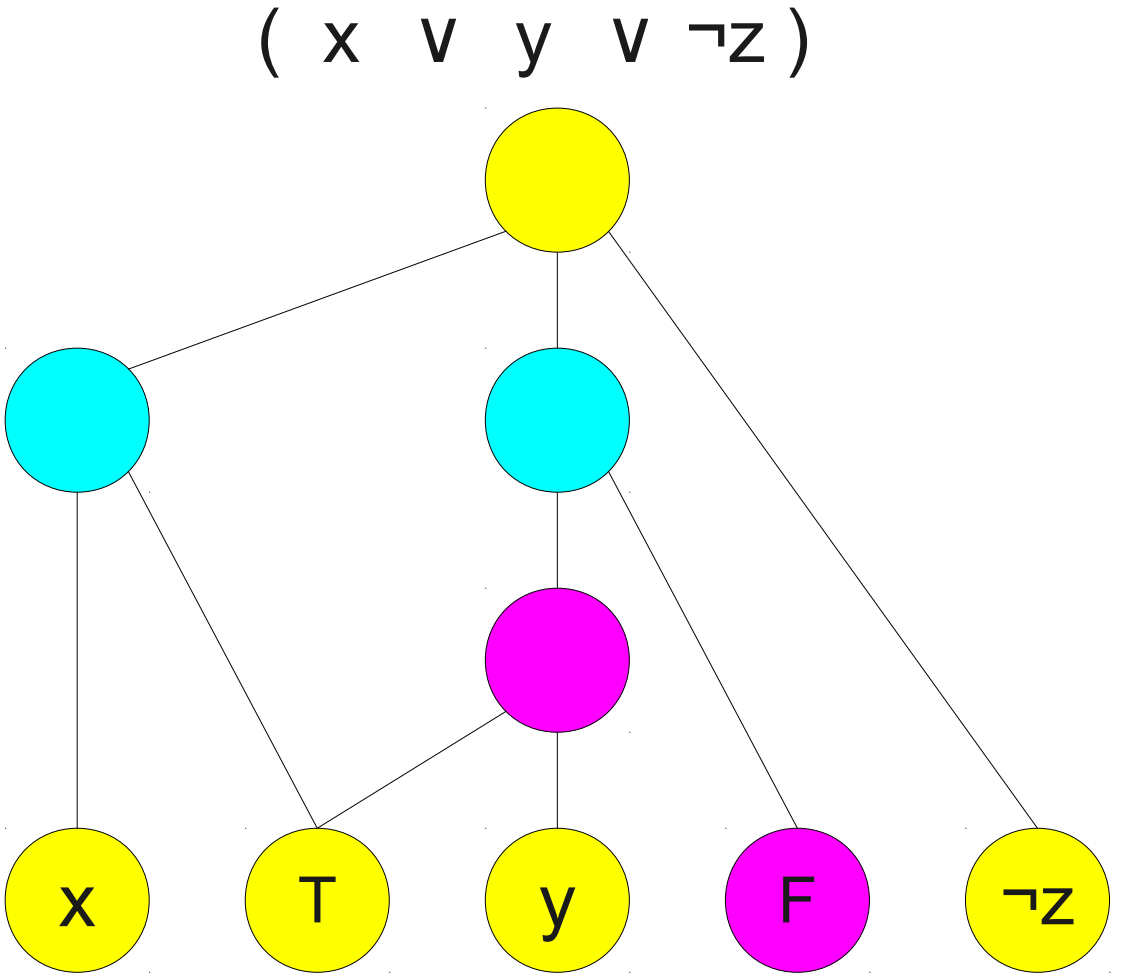
Gadget Three: Clause Satisfiability



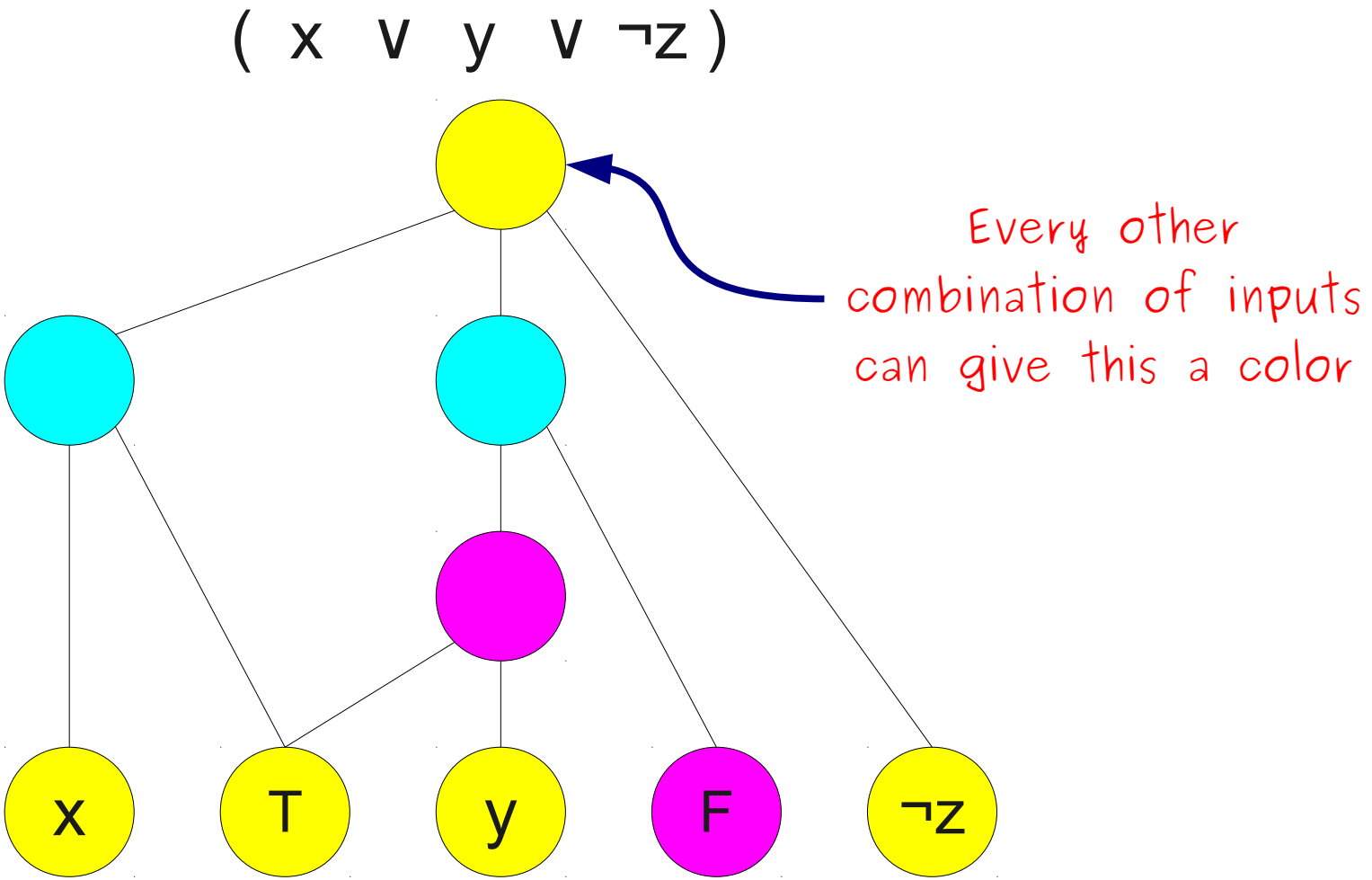
Gadget Three: Clause Satisfiability



Gadget Three: Clause Satisfiability



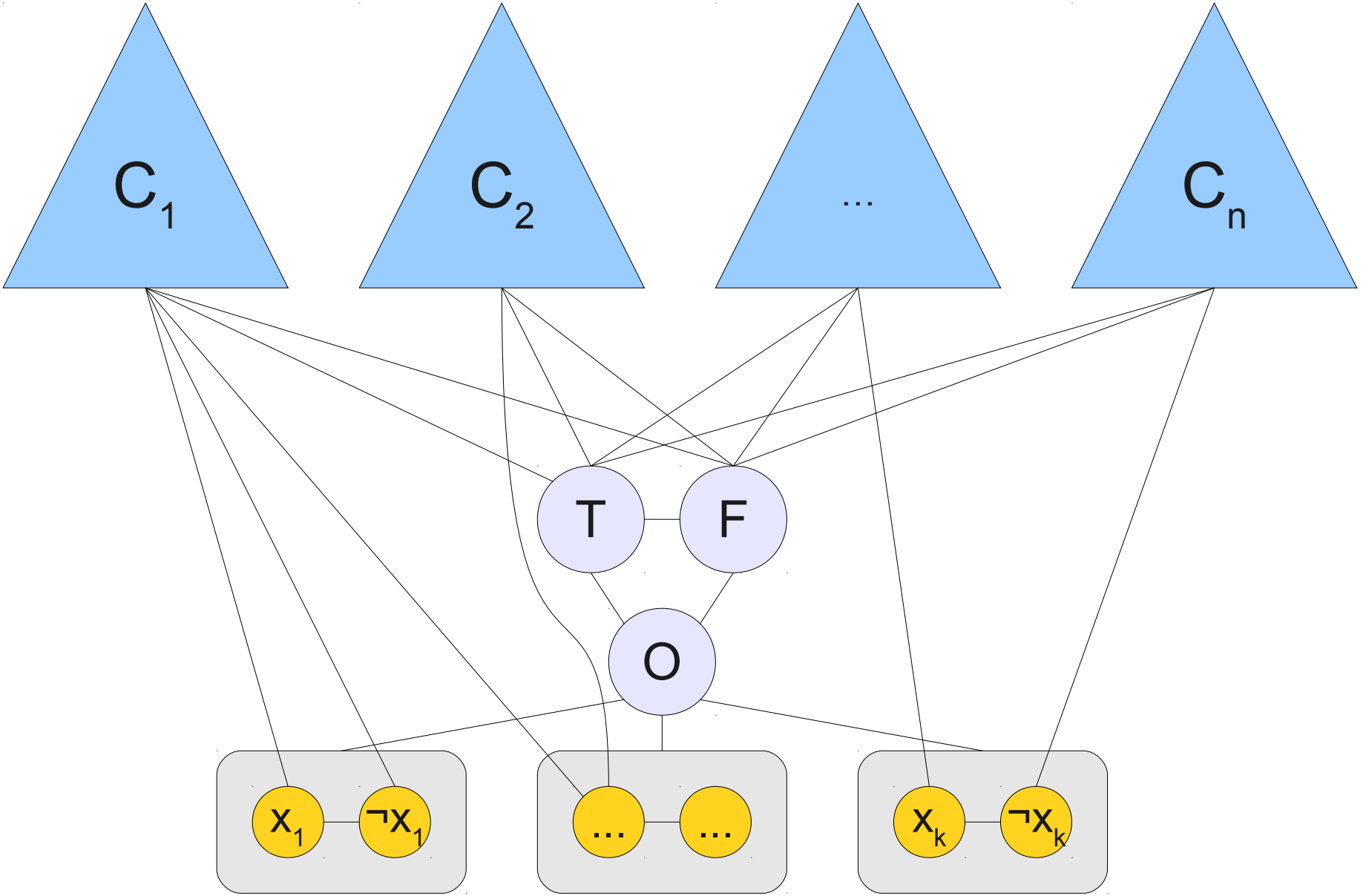
Gadget Three: Clause Satisfiability



Putting It All Together

- Construct the first gadget so we have a consistent definition of true and false.
- For each variable v:
  - Construct nodes v and ¬v.
  - Add an edge between v and ¬v.
  - Add an edge between v and O and between ¬v and O.
- For each clause C:
  - Construct the earlier gadget from C by adding in the extra nodes and edges.

Putting It All Together



Analyzing the Reduction

- How large is the resulting graph?
- We have O(1) nodes to give meaning to “true” and “false.”
- Each variable gives O(1) nodes for its true and false values.
- Each clause gives O(1) nodes for its colorability gadget.
- Collectively, if there are n clauses, there are O(n) variables.
- Total size of the graph is O(n).

Next Time

- **More NP-Complete Problems**
- **The Proof of the Cook-Levin Theorem**