NP-Completeness

Part II

Please evaluate this course on Axess.

Your comments really do make a difference.

Announcements

- Problem Set 8 due tomorrow at 12:50PM **sharp** with one late day.
- Problem Set 9 out, due this Friday at 12:50PM.
  - Due at 11:30AM on Monday, March 18 with a late day; **no submissions can be accepted after this time**. (Sorry – this is university policy).
  - Since the final ends at this time, it's okay if you submit as soon as you're done with the exam.

Final Exam Logistics

- **Final exam** is next Monday, March 18 from 8:30AM – 11:30AM in Cemex Auditorium.
  - Covers material up through an including today's lecture.
  - Focus is on material from the second half of the course, though there will be at least one midterm-style question.
- If you need to take the exam at an alternate time, please email the staff list no later than **tonight at 11:59PM**.

Change to Open-Computer Policy

- **We are changing the open-computer policy for the final exam.**
- You may only use your computer to do the following:
  - Visit the course website and access materials available there (including the DFA/NFA/Regex editors).
  - Read over ***your own notes*** for which you are the original author and which do not contain proofs copied from online sources.
  - Read over an electronic copy of Sipser or Rosen (if you happen to have one.)
- You **must not** consult any other sources, even if you have them already downloaded on your computer.
- ***We will take any violations extremely seriously.***

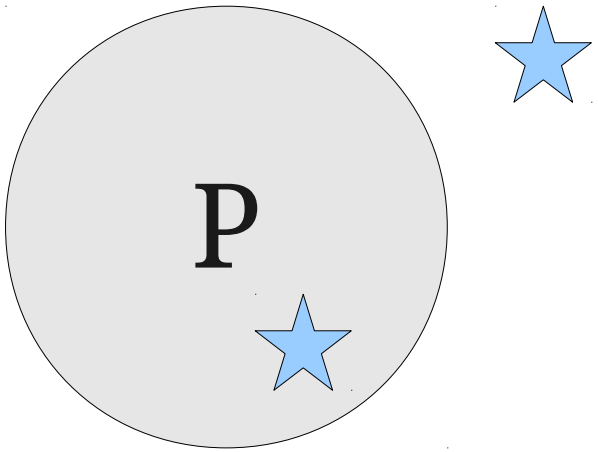
Practice Exams

- Two new practice exams released.
  - Not for extra credit – sorry!
  - Solutions released this Friday.

Recap from Last Time

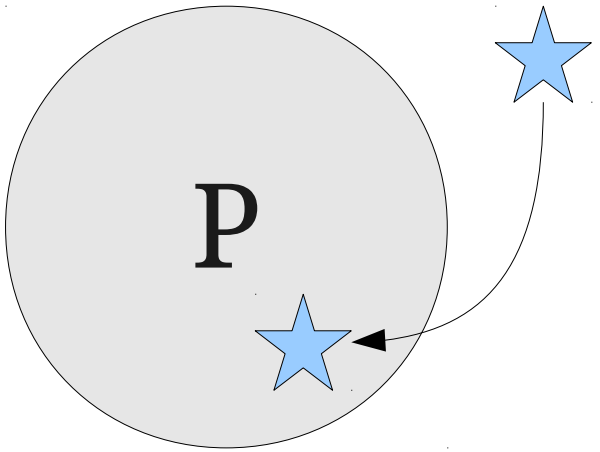
Polynomial-Time Reductions

- If $L_1 \leq_P L_2$ and $L_2 \in \mathbf{P}$, then $L_1 \in \mathbf{P}$.



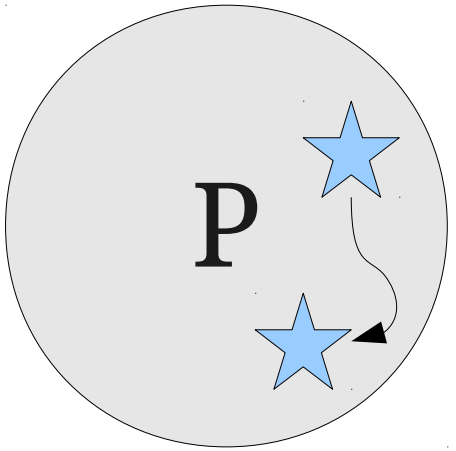
Polynomial-Time Reductions

- If $L_1 \leq_P L_2$ and $L_2 \in \mathbf{P}$, then $L_1 \in \mathbf{P}$.



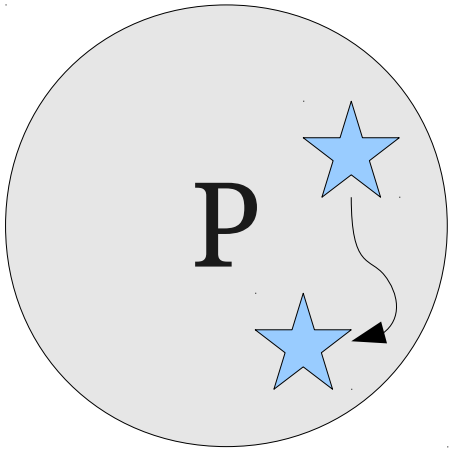
Polynomial-Time Reductions

- If $L_1 \leq_P L_2$ and $L_2 \in \mathbf{P}$, then $L_1 \in \mathbf{P}$.



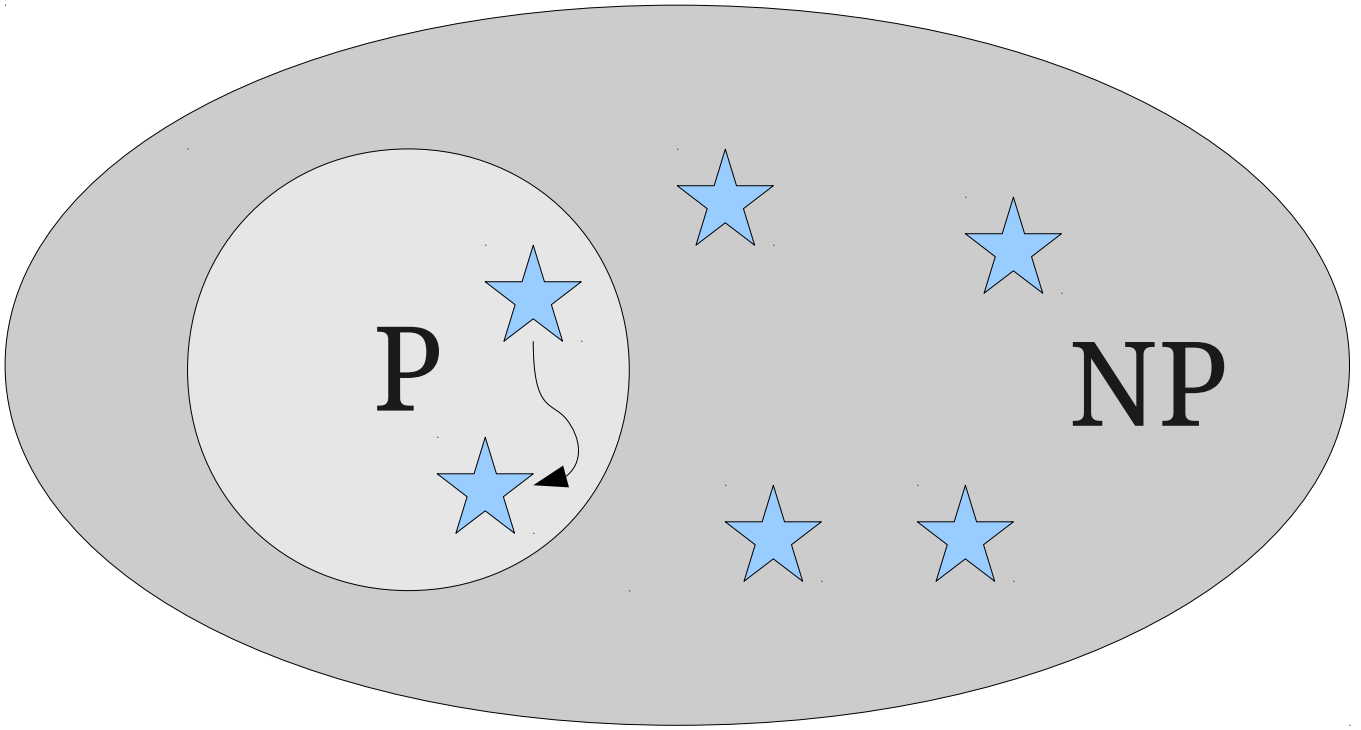
Polynomial-Time Reductions

- If $L_1 \leq_P L_2$ and $L_2 \in \mathbf{P}$, then $L_1 \in \mathbf{P}$.
- If $L_1 \leq_P L_2$ and $L_2 \in \mathbf{NP}$, then $L_1 \in \mathbf{NP}$.



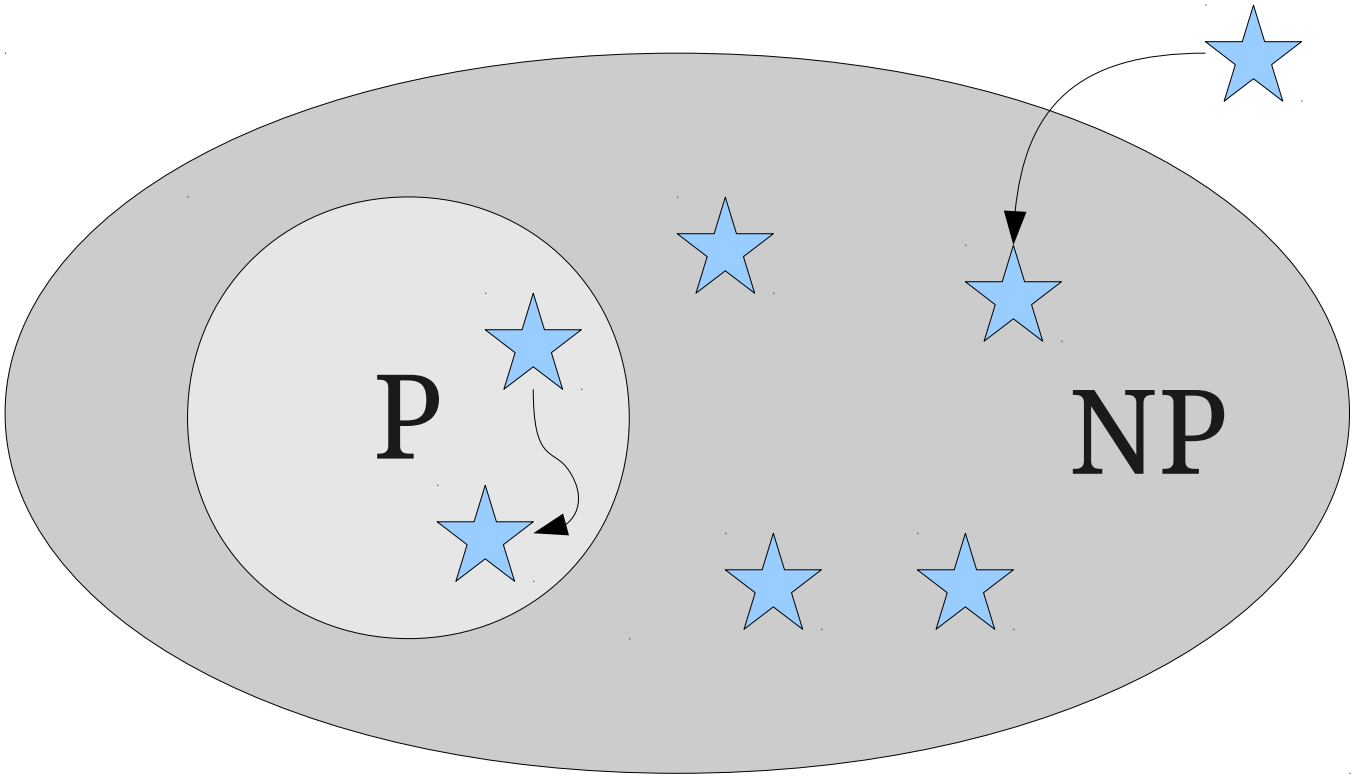
Polynomial-Time Reductions

- If $L_1 \leq_P L_2$ and $L_2 \in \mathbf{P}$, then $L_1 \in \mathbf{P}$.
- If $L_1 \leq_P L_2$ and $L_2 \in \mathbf{NP}$, then $L_1 \in \mathbf{NP}$.



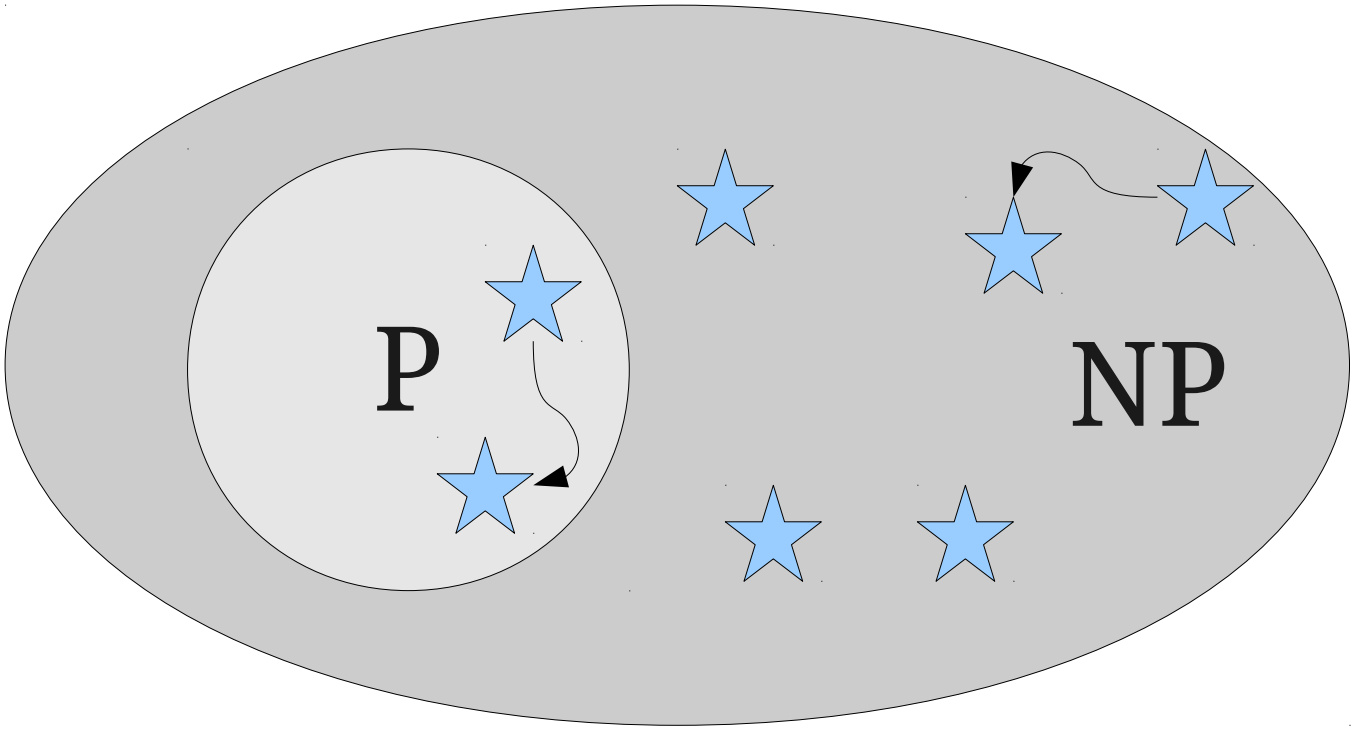
Polynomial-Time Reductions

- If $L_1 \leq_P L_2$ and $L_2 \in \mathbf{P}$, then $L_1 \in \mathbf{P}$.
- If $L_1 \leq_P L_2$ and $L_2 \in \mathbf{NP}$, then $L_1 \in \mathbf{NP}$.



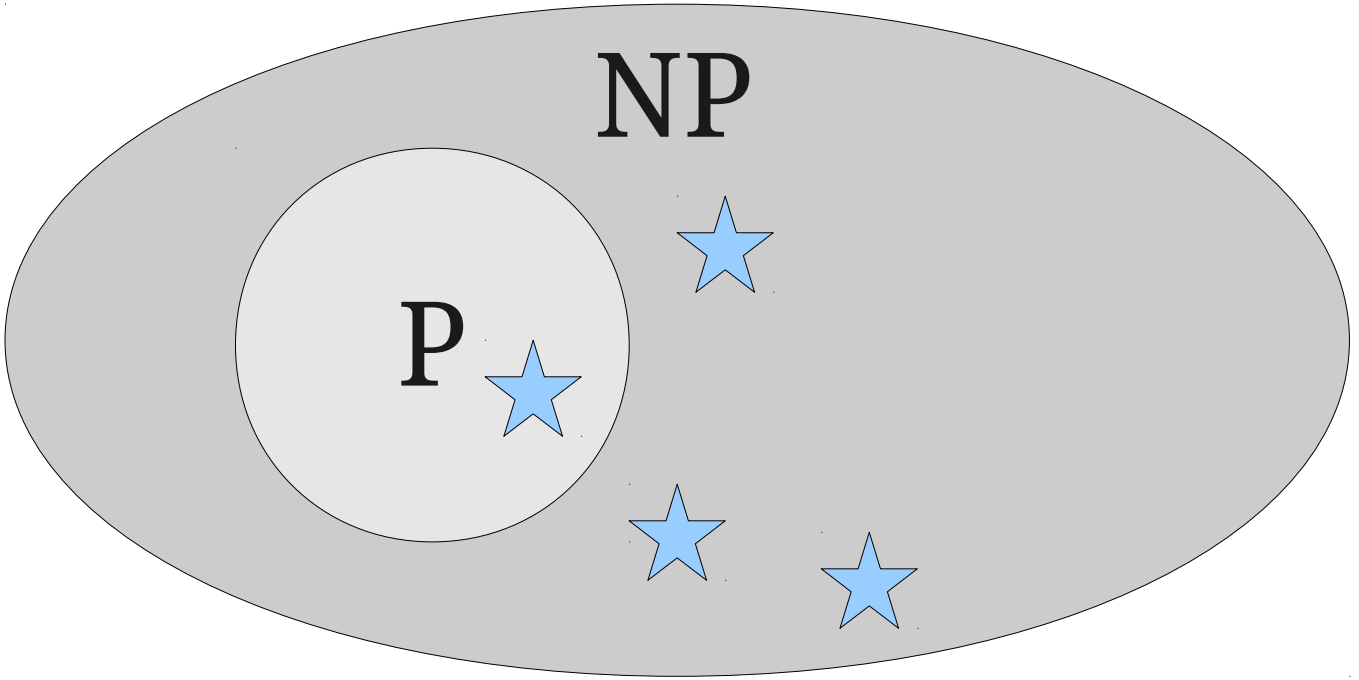
Polynomial-Time Reductions

- If $L_1 \leq_P L_2$ and $L_2 \in \mathbf{P}$, then $L_1 \in \mathbf{P}$.
- If $L_1 \leq_P L_2$ and $L_2 \in \mathbf{NP}$, then $L_1 \in \mathbf{NP}$.



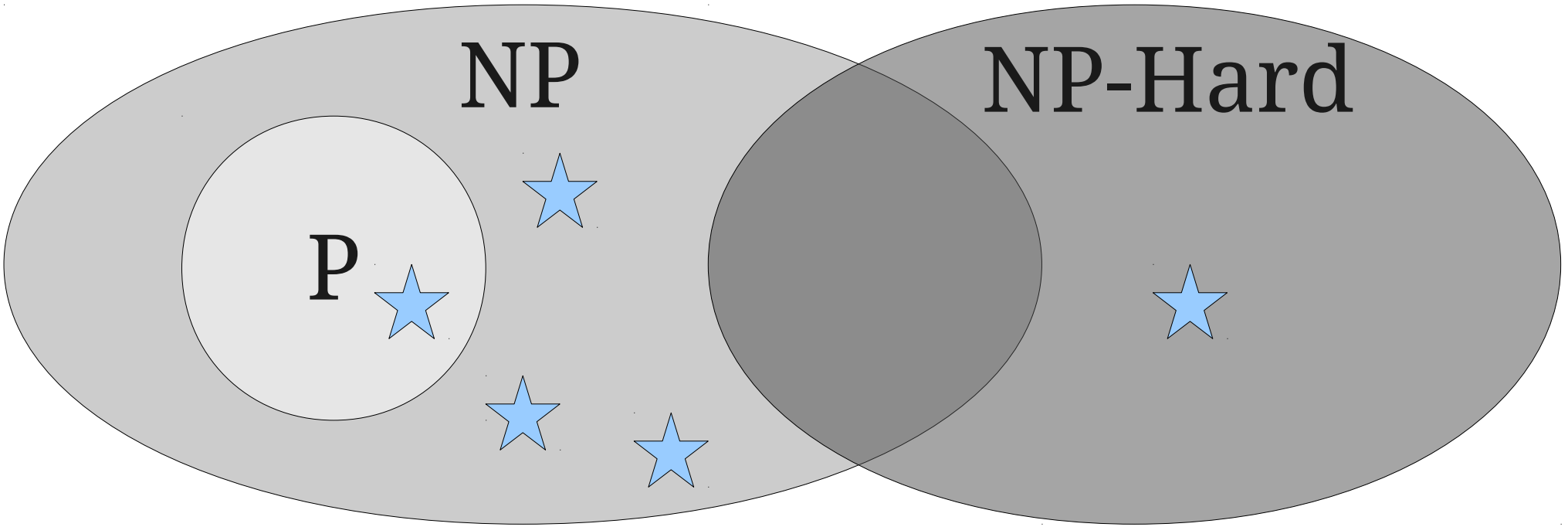
NP-Hardness

- A language L is called **NP-hard** iff for *every* $L' \in \mathbf{NP}$, we have $L' \leq_P L$.



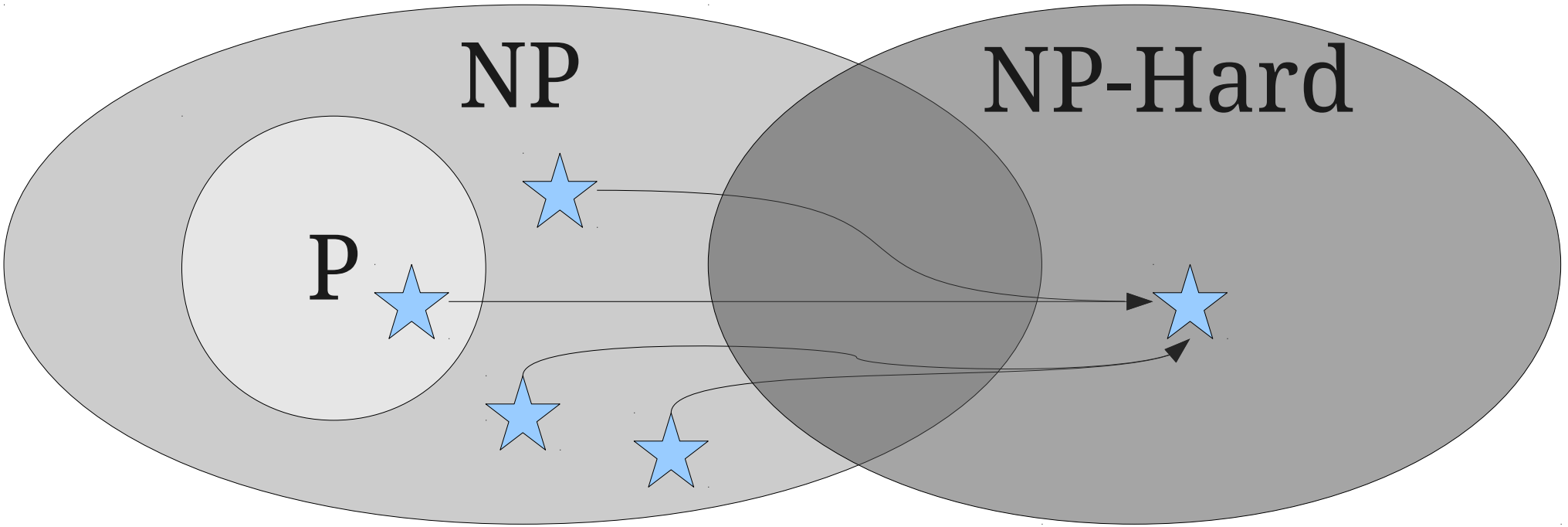
NP-Hardness

- A language L is called **NP-hard** iff for *every* $L' \in$ **NP**, we have $L' \leq_P L$.



NP-Hardness

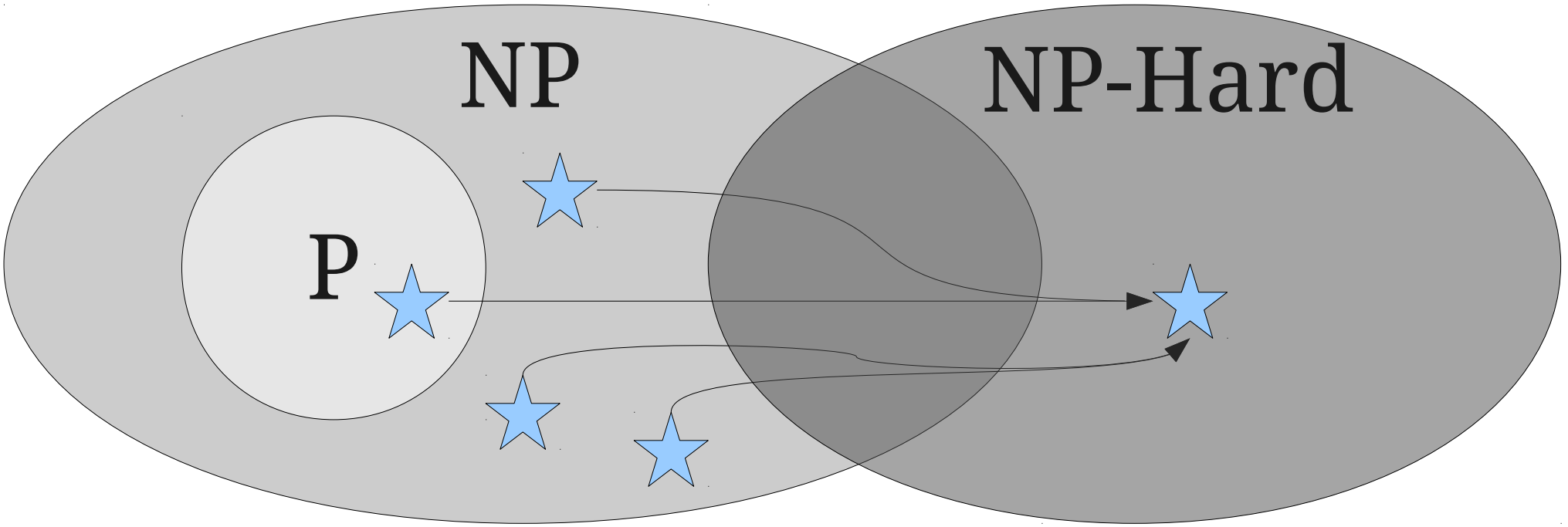
- A language L is called **NP-hard** iff for *every* $L' \in$ **NP**, we have $L' \leq_P L$.



NP-Hardness

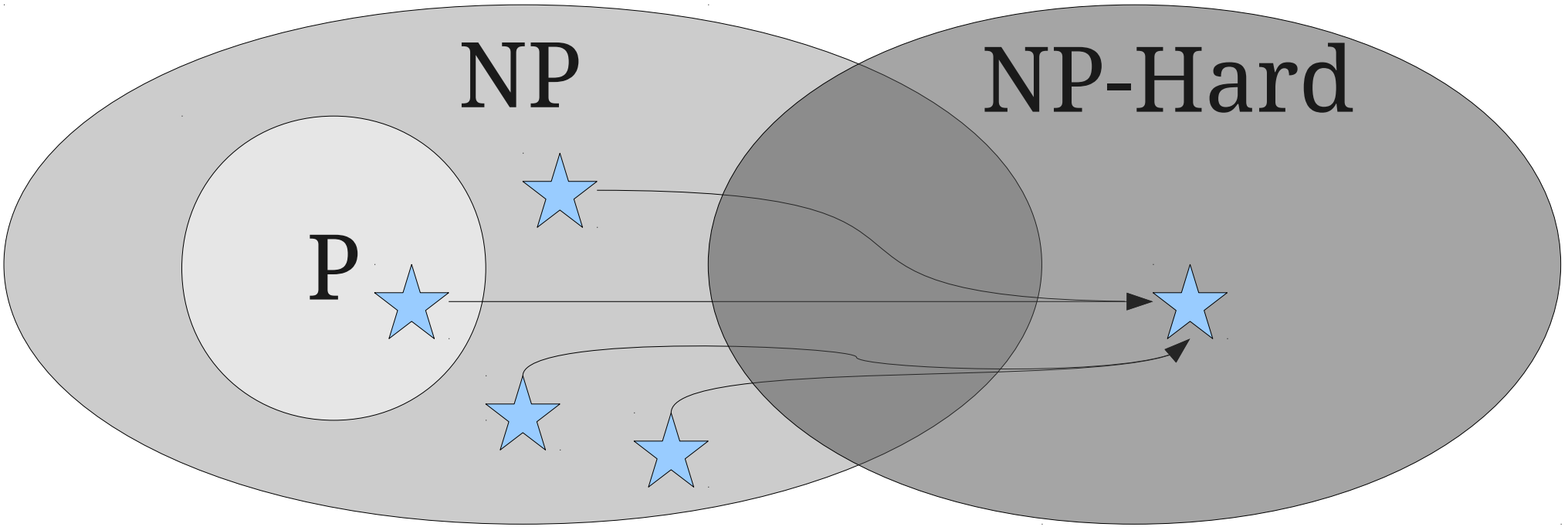
- A language L is called **NP-hard** iff for *every* $L' \in \mathbf{NP}$, we have $L' \leq_P L$.

Intuitively: L has to be at least as hard as every problem in **NP**, since an algorithm for L can be used to decide all problems in **NP**.



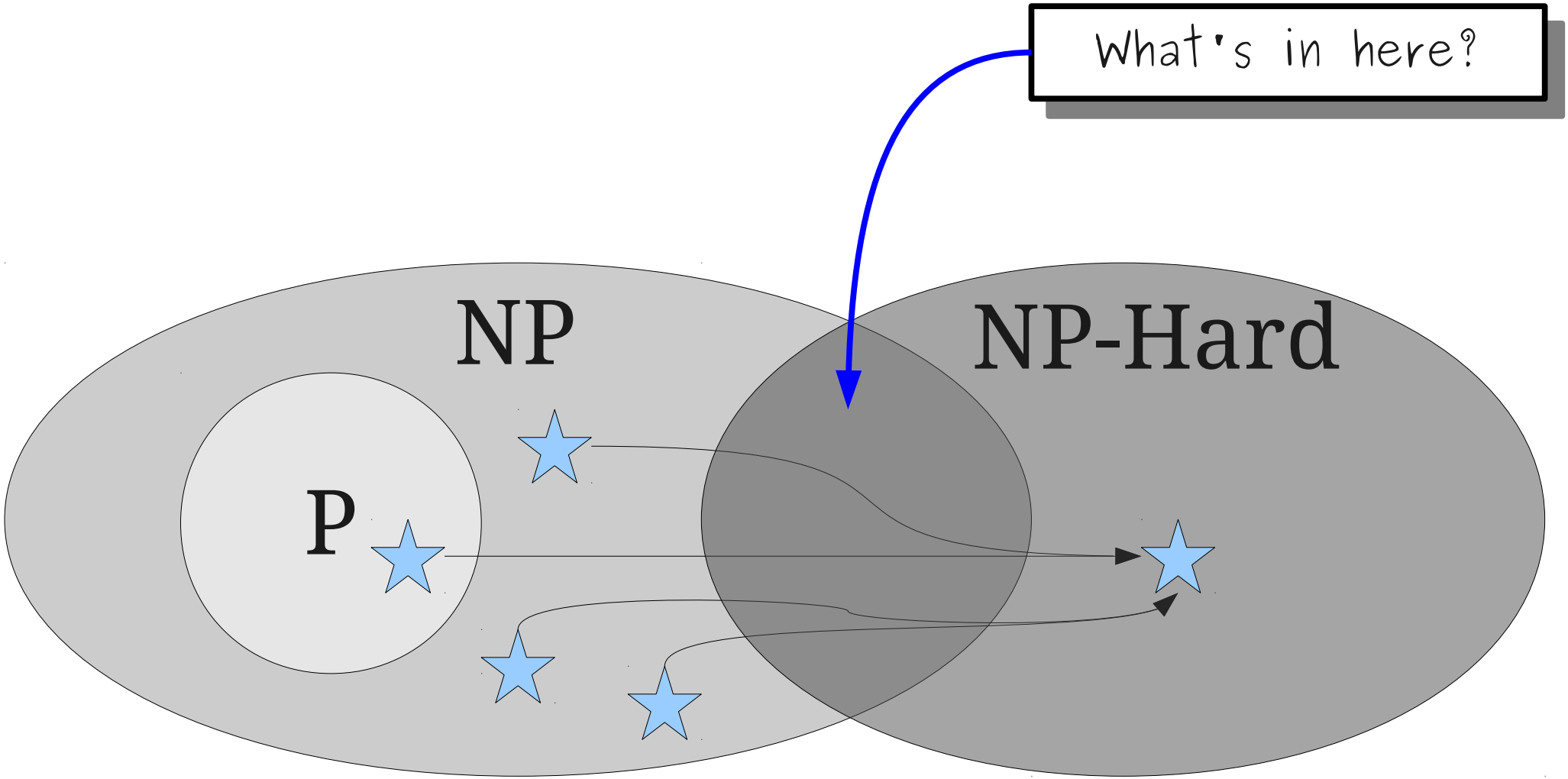
NP-Hardness

- A language L is called **NP-hard** iff for *every* $L' \in$ **NP**, we have $L' \leq_P L$.



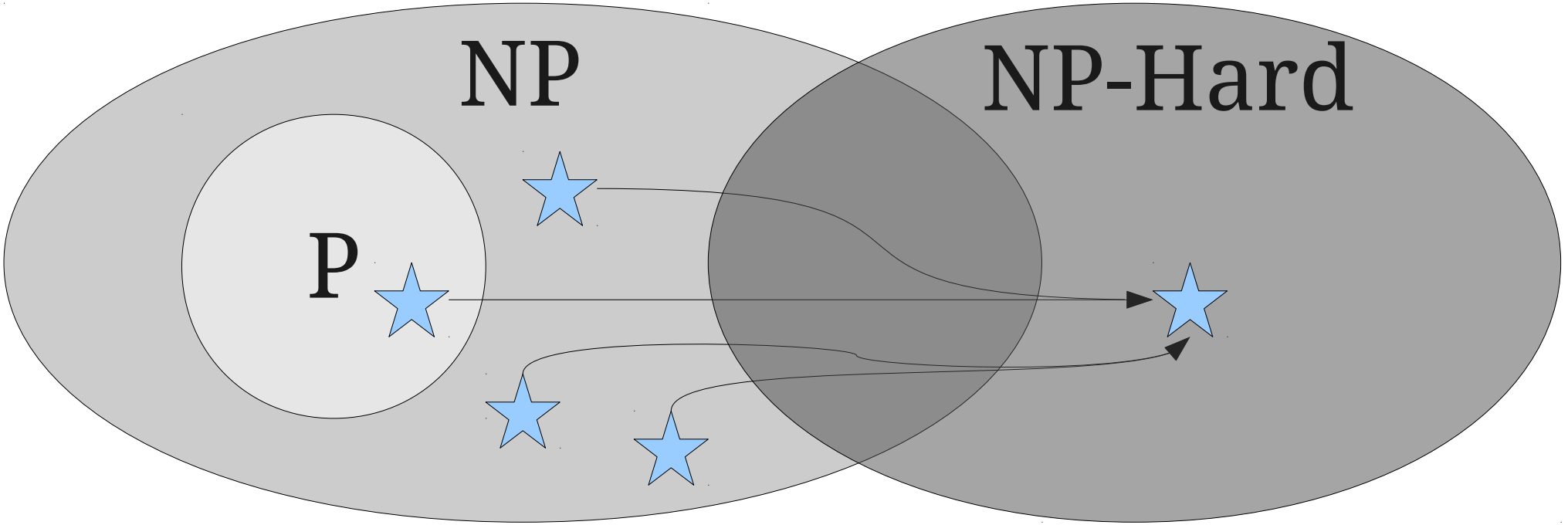
NP-Hardness

- A language L is called **NP-hard** iff for *every* $L' \in$ **NP**, we have $L' \leq_P L$.



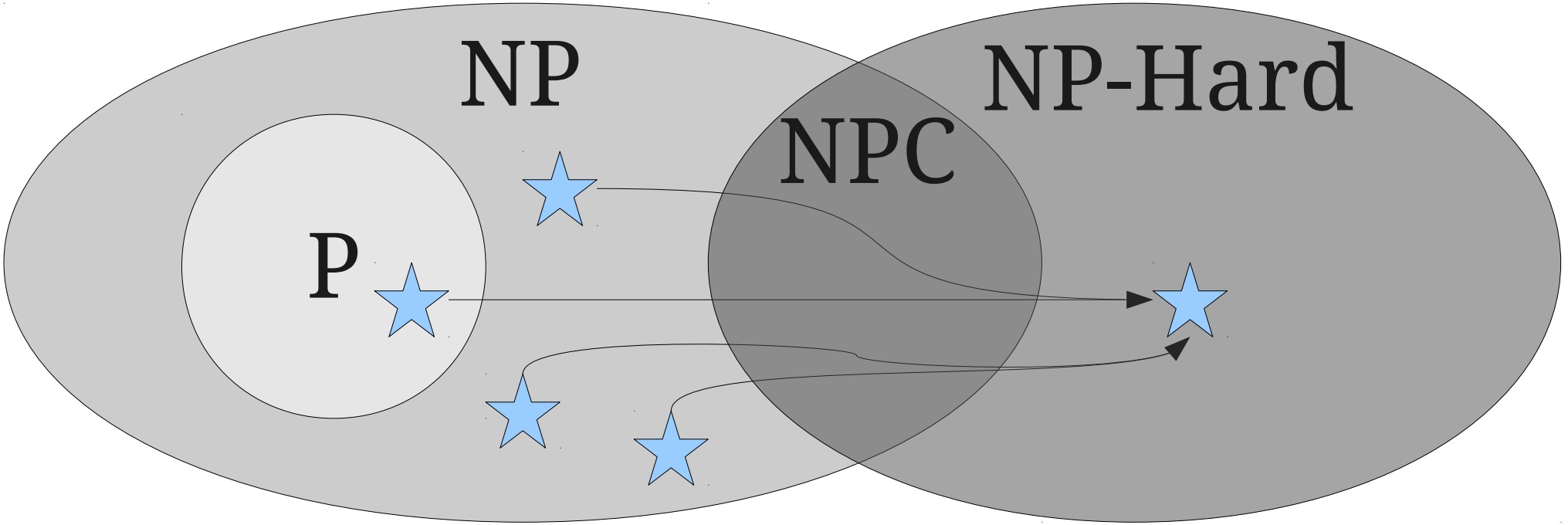
NP-Hardness

- A language L is called **NP-hard** iff for *every* $L' \in \mathbf{NP}$, we have $L' \leq_P L$.
- A language in L is called **NP-complete** iff L is **NP**-hard and $L \in \mathbf{NP}$.
- The class **NPC** is the set of **NP**-complete problems.



NP-Hardness

- A language L is called **NP-hard** iff for *every* $L' \in$ **NP**, we have $L' \leq_P L$.
- A language in L is called **NP-complete** iff L is **NP**-hard and $L \in$ **NP**.
- The class **NPC** is the set of **NP**-complete problems.



The Tantalizing Truth

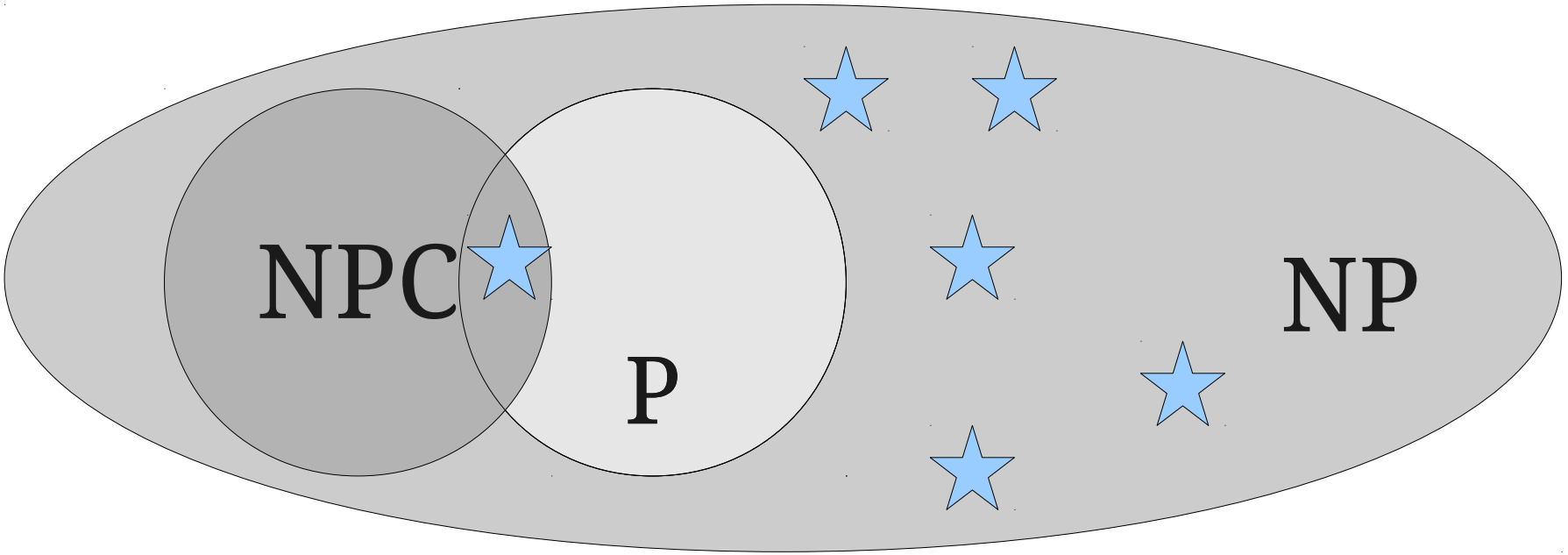
Theorem: If ***any*** **NP**-complete language is in **P**, then **P** = **NP**.

Proof: If $L \in$ **NPC** and $L \in$ **P**, we know for any $L' \in$ **NP** that $L' \leq_P L$, because L is **NP**-complete. Since $L' \leq_P L$ and $L \in$ **P**, this means that $L' \in$ **P** as well. Since our choice of L' was arbitrary, any language $L' \in$ **NP** satisfies $L' \in$ **P**, so **NP** $\subseteq$ **P**. Since **P** $\subseteq$ **NP**, this means **P** = **NP**. ■

The Tantalizing Truth

Theorem: If ***any*** **NP**-complete language is in **P**, then **P** = **NP**.

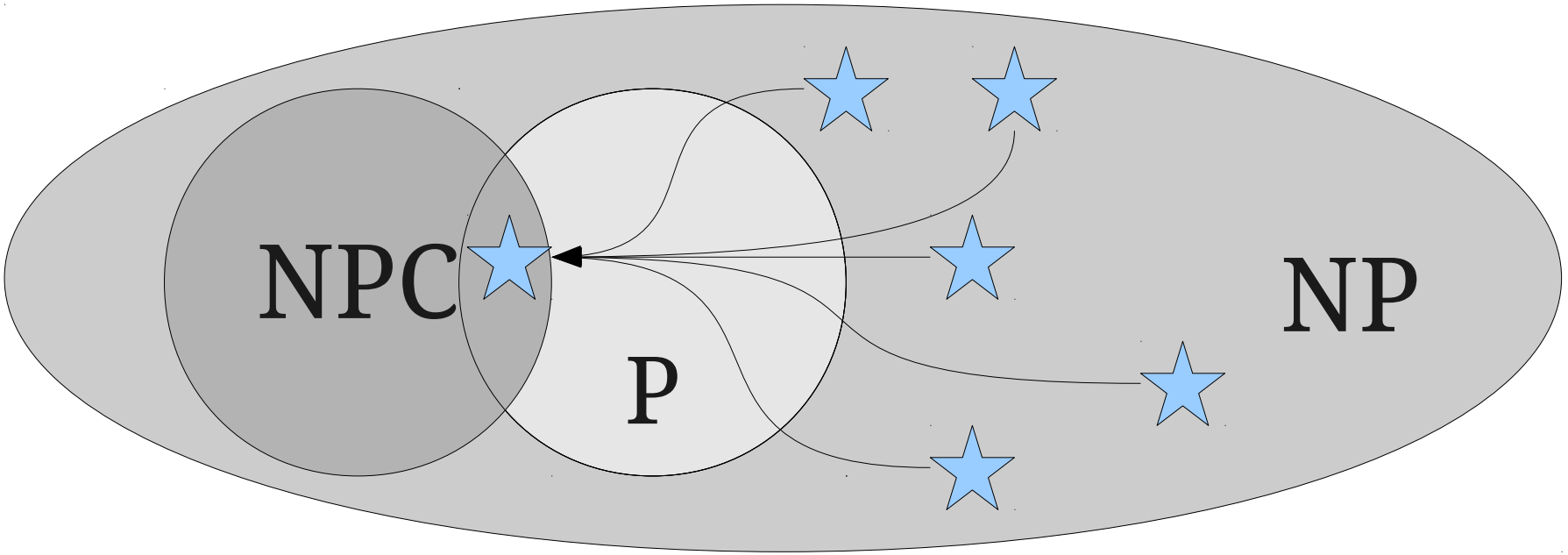
Proof: If $L \in$ **NPC** and $L \in$ **P**, we know for any $L' \in$ **NP** that $L' \leq_P L$, because L is **NP**-complete. Since $L' \leq_P L$ and $L \in$ **P**, this means that $L' \in$ **P** as well. Since our choice of L' was arbitrary, any language $L' \in$ **NP** satisfies $L' \in$ **P**, so **NP** $\subseteq$ **P**. Since **P** $\subseteq$ **NP**, this means **P** = **NP**. ■



The Tantalizing Truth

Theorem: If ***any*** **NP**-complete language is in **P**, then **P** = **NP**.

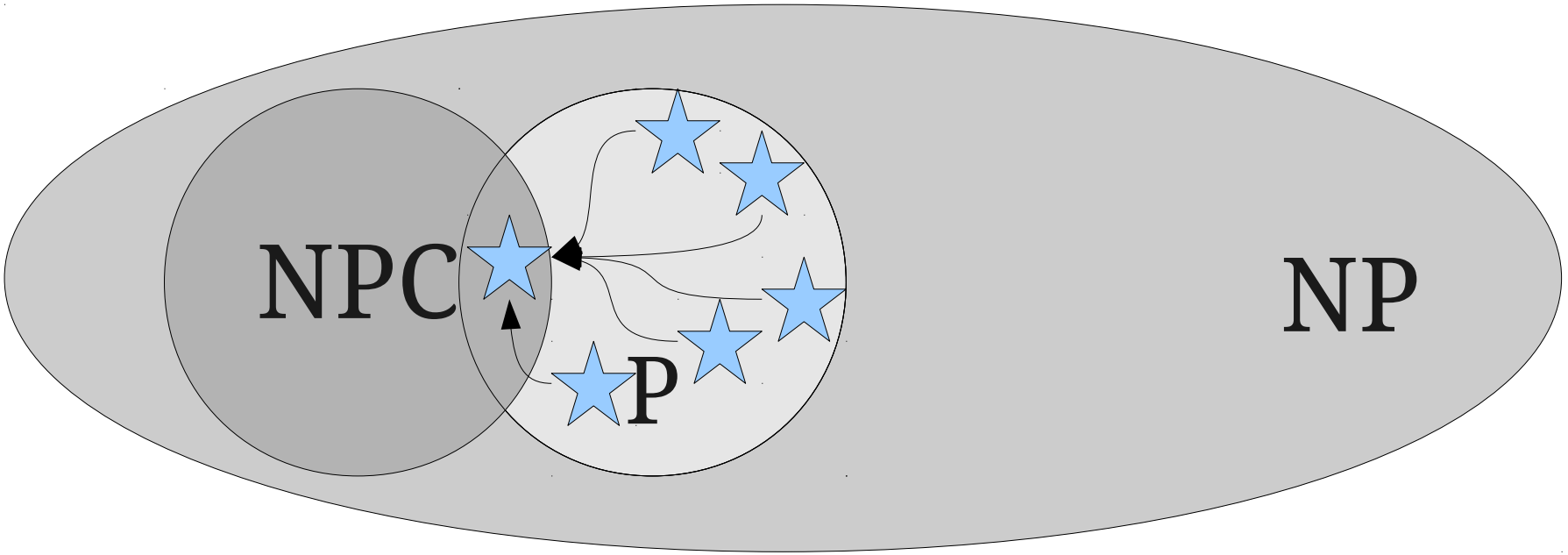
Proof: If $L \in$ **NPC** and $L \in$ **P**, we know for any $L' \in$ **NP** that $L' \leq_P L$, because L is **NP**-complete. Since $L' \leq_P L$ and $L \in$ **P**, this means that $L' \in$ **P** as well. Since our choice of L' was arbitrary, any language $L' \in$ **NP** satisfies $L' \in$ **P**, so **NP** $\subseteq$ **P**. Since **P** $\subseteq$ **NP**, this means **P** = **NP**. ■



The Tantalizing Truth

Theorem: If ***any*** **NP**-complete language is in **P**, then **P** = **NP**.

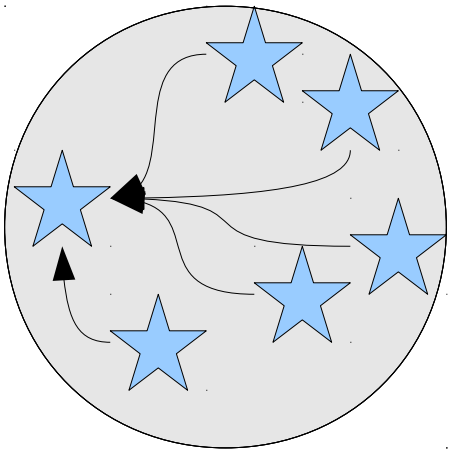
Proof: If $L \in$ **NPC** and $L \in$ **P**, we know for any $L' \in$ **NP** that $L' \leq_P L$, because L is **NP**-complete. Since $L' \leq_P L$ and $L \in$ **P**, this means that $L' \in$ **P** as well. Since our choice of L' was arbitrary, any language $L' \in$ **NP** satisfies $L' \in$ **P**, so **NP** $\subseteq$ **P**. Since **P** $\subseteq$ **NP**, this means **P** = **NP**. ■



The Tantalizing Truth

Theorem: If ***any*** **NP**-complete language is in **P**, then **P** = **NP**.

Proof: If $L \in$ **NPC** and $L \in$ **P**, we know for any $L' \in$ **NP** that $L' \leq_P L$, because L is **NP**-complete. Since $L' \leq_P L$ and $L \in$ **P**, this means that $L' \in$ **P** as well. Since our choice of L' was arbitrary, any language $L' \in$ **NP** satisfies $L' \in$ **P**, so **NP** $\subseteq$ **P**. Since **P** $\subseteq$ **NP**, this means **P** = **NP**. ■

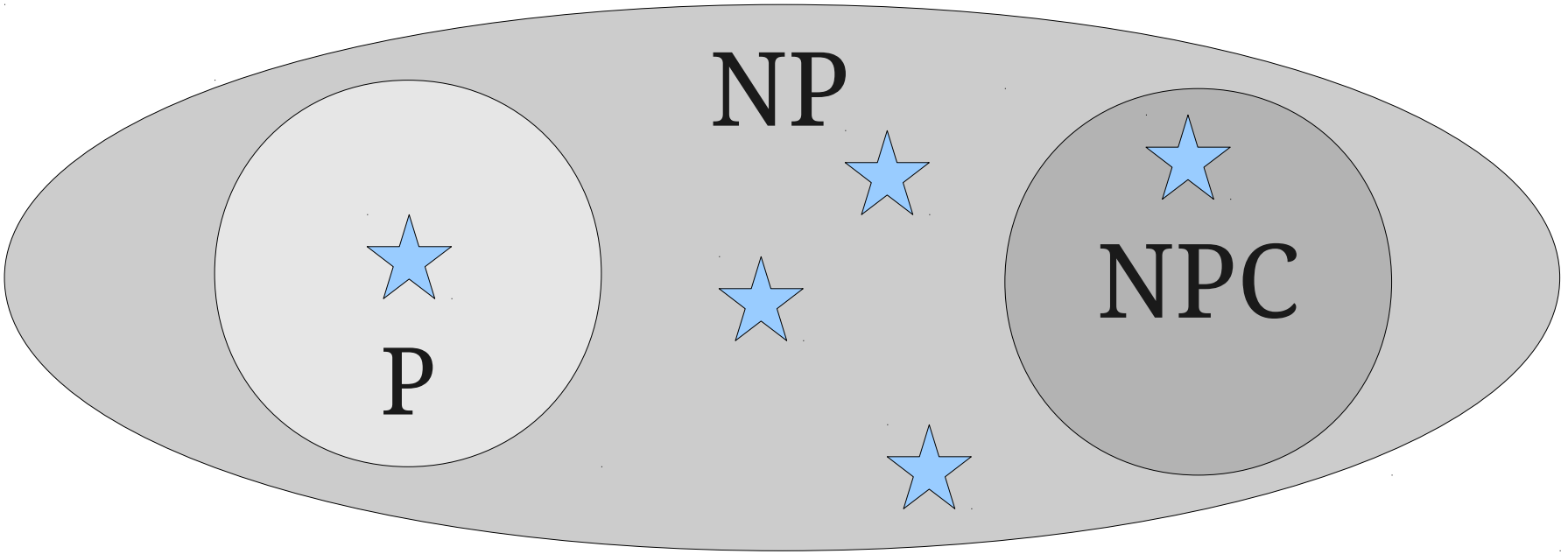


P = NP

The Tantalizing Truth

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

Proof: If $L \in$ **NPC**, then $L \in$ **NP**. Thus if $L \notin$ **P**, then $L \in$ **NP** – **P**.
This means that **NP** – **P** ≠ Ø, so **P ≠ NP**. ■



A Feel for **NP**-Completeness

- If a problem is **NP**-complete, then under the assumption that **P** ≠ **NP**, there cannot be an efficient algorithm for it.
- In a sense, **NP**-complete problems are the hardest problems in **NP**.
- All known **NP**-complete problems are enormously hard to solve:
  - All known algorithms for **NP**-complete problems run in worst-case exponential time.
  - Most algorithms for **NP**-complete problems are infeasible for reasonably-sized inputs.

What Problems are **NP**-Complete?

- **NP**-complete problems give a promising approach for resolving $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$:
  - 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 φ is called **satisfiable** if there is some assignment to its variables that makes it evaluate to true.
- An assignment of true and false to the variables of φ that makes it evaluate to true is called a **satisfying assignment**.

SAT

- The **boolean satisfiability problem** (**SAT**) is the following:

  Given a propositional logic formula φ, is φ satisfiable?

- Formally:

  SAT = { ⟨φ⟩ | φ is a satisfiable PL formula }

Theorem (Cook-Levin): SAT is **NP**-complete.

A Simpler **NP**-Complete Problem

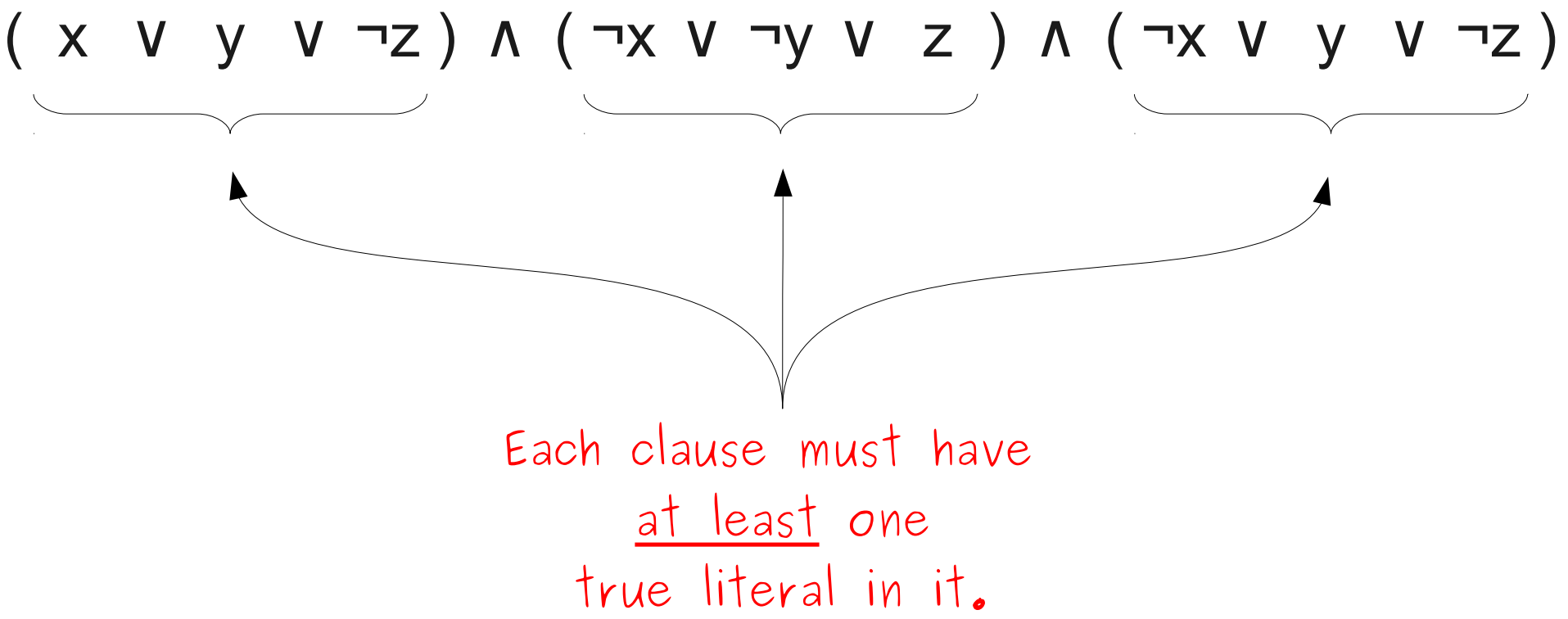
Literals and Clauses

- A **literal** in propositional logic is a variable or its negation:
  - x
  - $\neg y$
  - But not $x \wedge y$.
- A **clause** is a many-way OR (*disjunction*) of literals.
  - $\neg x \vee y \vee \neg z$
  - x
  - But not $x \vee \neg(y \vee z)$

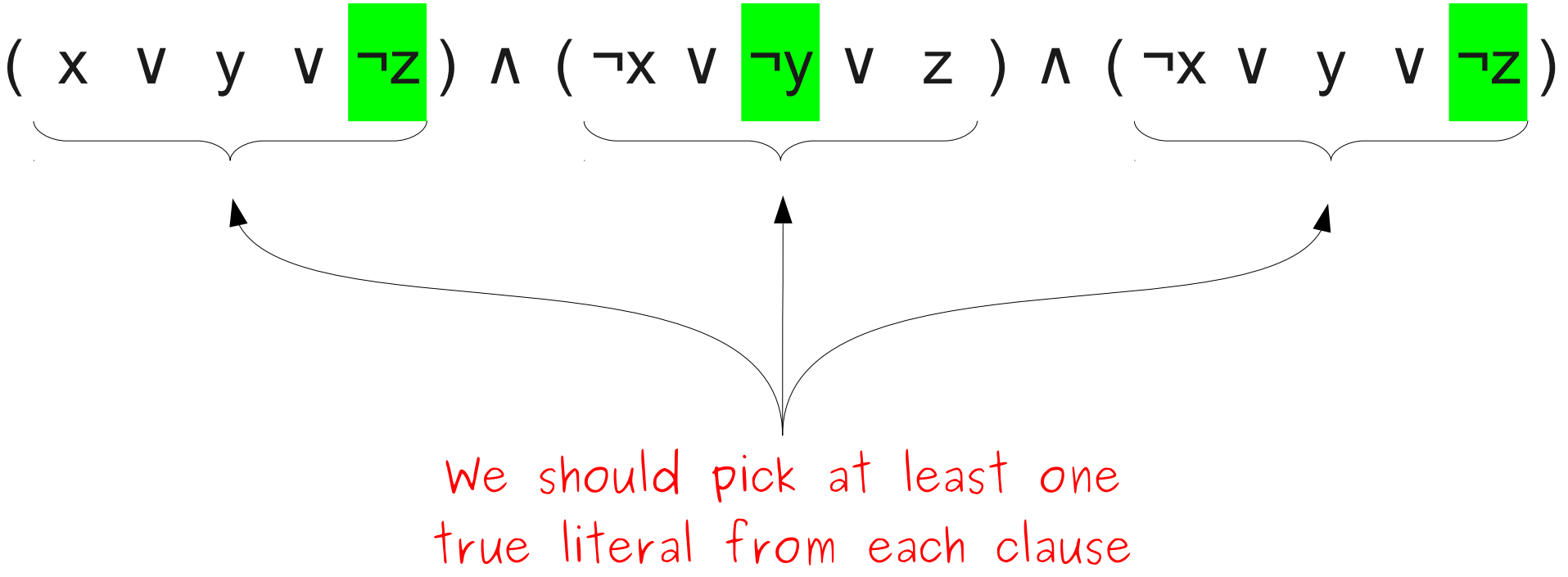
Conjunctive Normal Form

- A propositional logic formula φ is in **conjunctive normal form** (**CNF**) if it is the many-way AND (*conjunction*) of clauses.
  - $(x \vee y \vee z) \wedge (\neg x \vee \neg y) \wedge (x \vee y \vee z \vee \neg w)$
  - $x \vee z$
  - But not $(x \vee (y \wedge z)) \vee (x \vee y)$
- Only legal operators are $\neg$, $\vee$, $\wedge$.
- No nesting allowed.

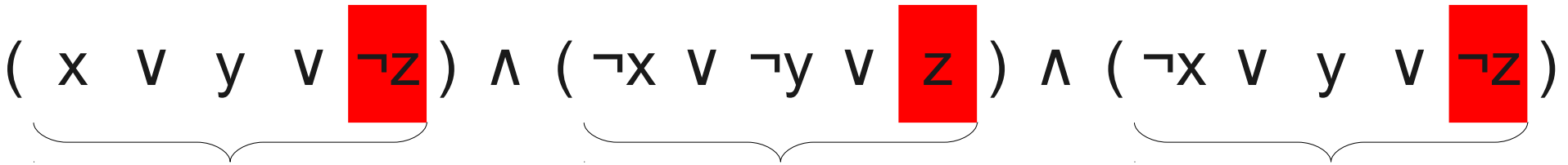
The Structure of CNF



The Structure of CNF



The Structure of CNF



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

3-CNF

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

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

Theorem: 3SAT is **NP**-Complete

Using the Cook-Levin Theorem

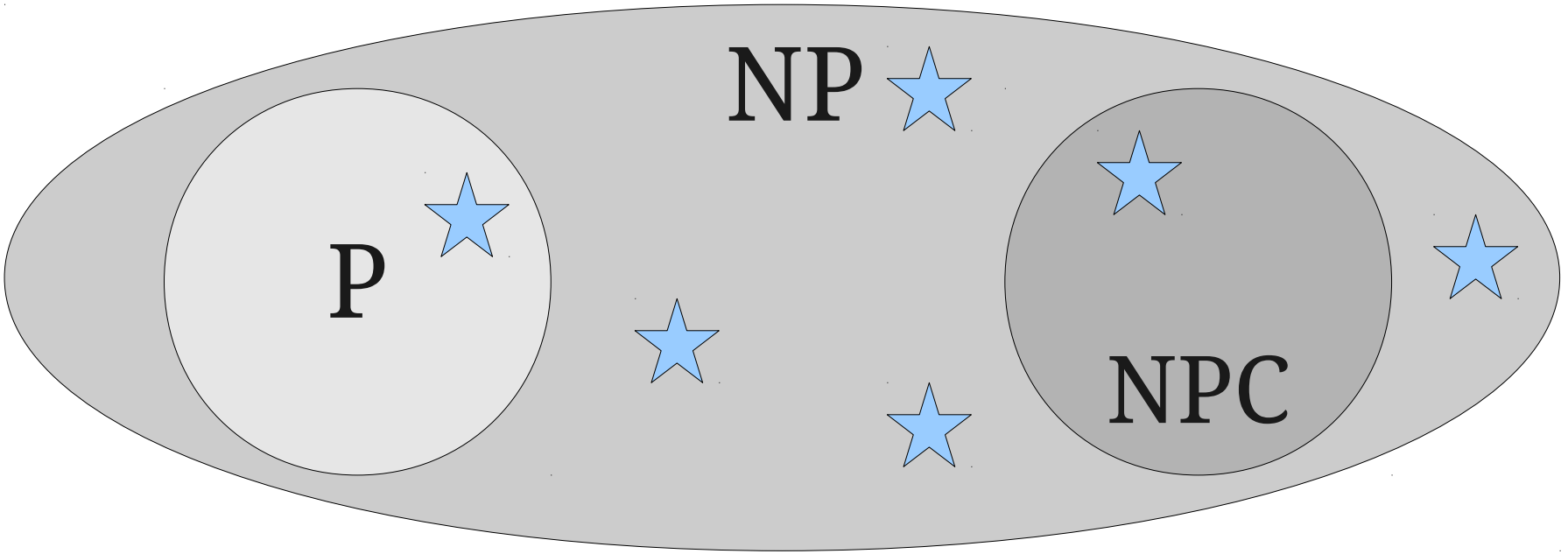
- When discussing decidability, we used the fact that $A_{TM} \notin \mathbf{R}$ as a starting point for finding other undecidable languages.
  - **Idea:** Reduce A_{TM} to some other language.
- When discussing **NP**-completeness, we will use the fact that 3SAT $\in$ **NPC** as a starting point for finding other **NPC** languages.
  - **Idea**: Reduce 3SAT to some other language.

NP-Completeness

- **Theorem**: If $L \in \mathbf{NPC}$, $L \leq_P L'$, and $L' \in \mathbf{NP}$, then $L' \in \mathbf{NPC}$.
- **Proof**: Consider any language $X \in \mathbf{NP}$. Since $L \in \mathbf{NPC}$, we know that $X \leq_P L$. Since $L \leq_P L'$, we have $X \leq_P L'$. Since our choice of X was arbitrary, this means L' is **NP**-hard. Since L' is **NP**-hard and $L' \in \mathbf{NP}$, we have $L' \in \mathbf{NPC}$. ■

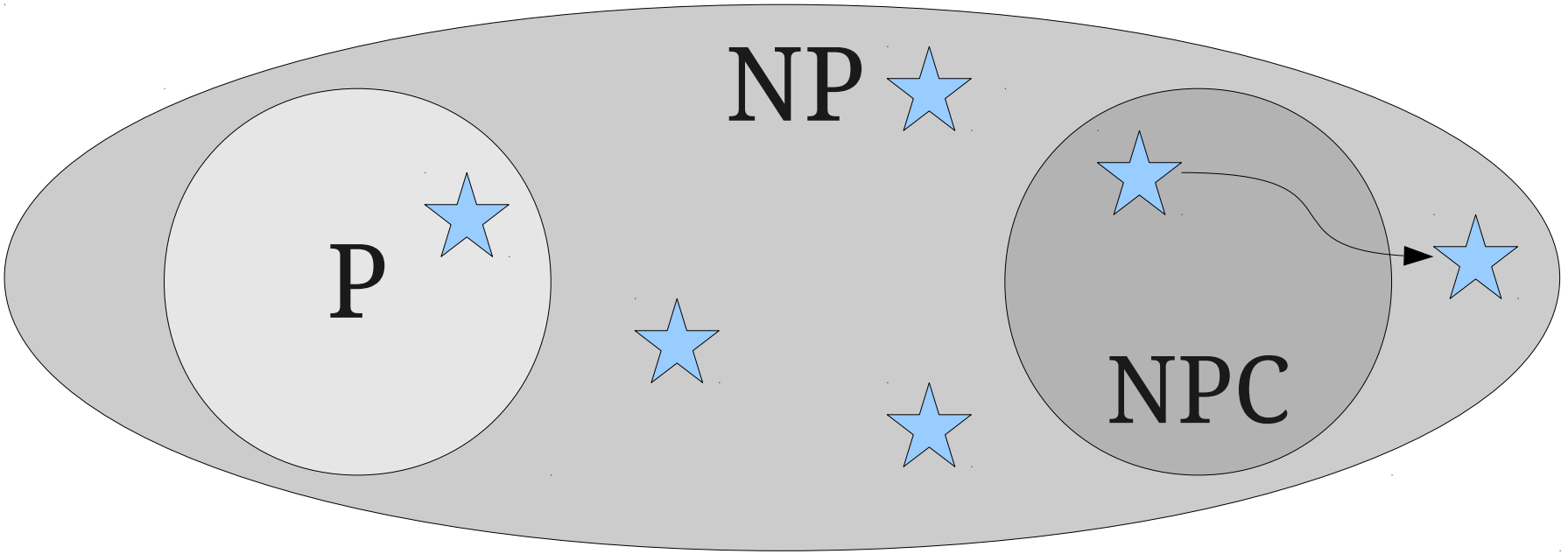
NP-Completeness

- **Theorem**: If $L \in \mathbf{NPC}$, $L \leq_P L'$, and $L' \in \mathbf{NP}$, then $L' \in \mathbf{NPC}$.
- **Proof**: Consider any language $X \in \mathbf{NP}$. Since $L \in \mathbf{NPC}$, we know that $X \leq_P L$. Since $L \leq_P L'$, we have $X \leq_P L'$. Since our choice of X was arbitrary, this means L' is **NP**-hard. Since L' is **NP**-hard and $L' \in \mathbf{NP}$, we have $L' \in \mathbf{NPC}$. ■



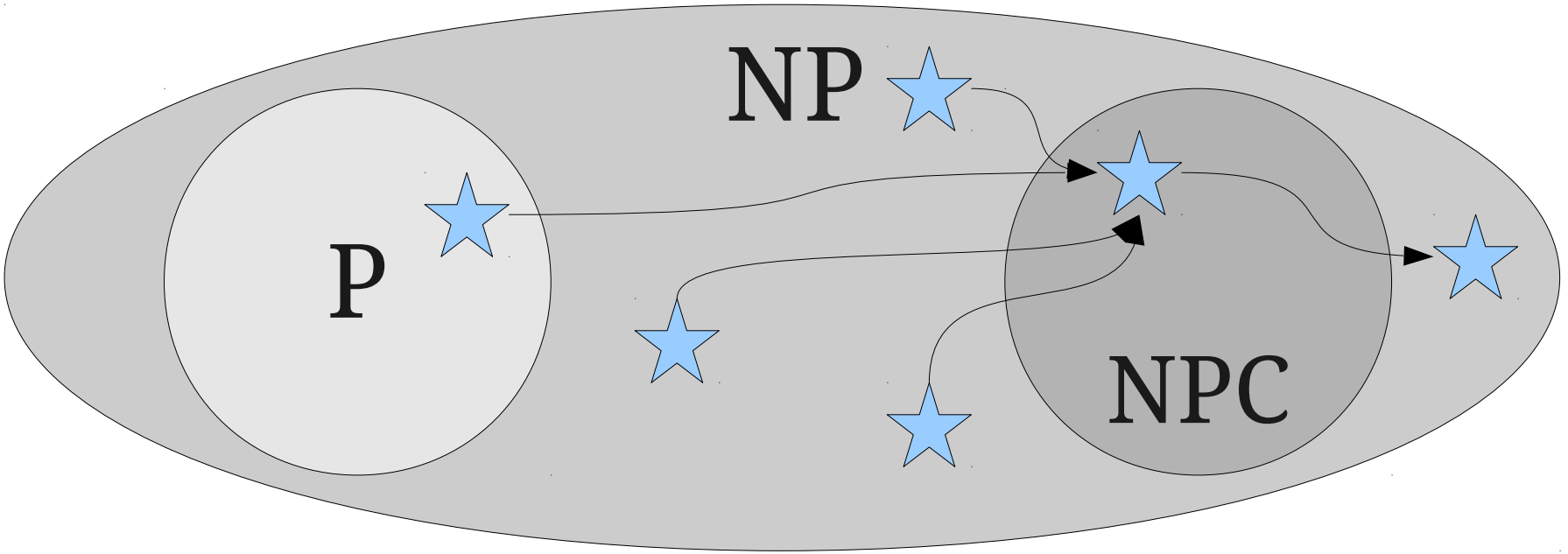
NP-Completeness

- **Theorem**: If $L \in$ **NPC**, $L \leq_P L'$, and $L' \in$ **NP**, then $L' \in$ **NPC**.
- **Proof**: Consider any language $X \in$ **NP**. Since $L \in$ **NPC**, we know that $X \leq_P L$. Since $L \leq_P L'$, we have $X \leq_P L'$. Since our choice of X was arbitrary, this means L' is **NP**-hard. Since L' is **NP**-hard and $L' \in$ **NP**, we have $L' \in$ **NPC**. ■



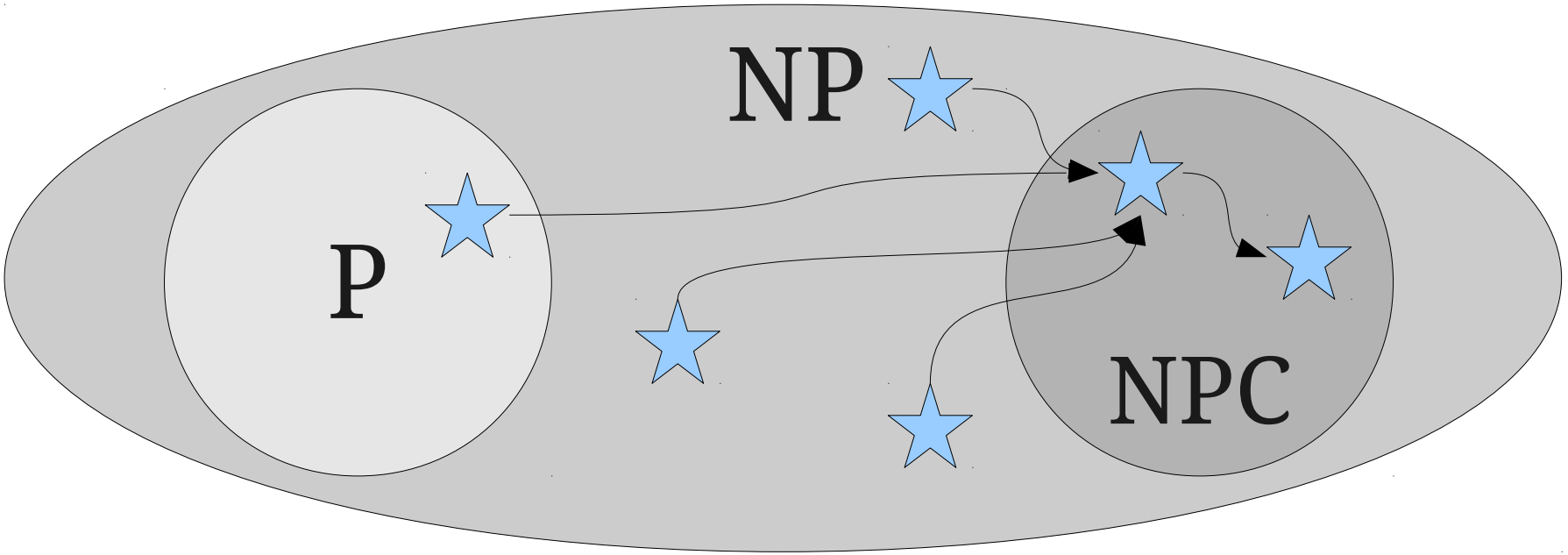
NP-Completeness

- **Theorem**: If $L \in$ **NPC**, $L \leq_P L'$, and $L' \in$ **NP**, then $L' \in$ **NPC**.
- **Proof**: Consider any language $X \in$ **NP**. Since $L \in$ **NPC**, we know that $X \leq_P L$. Since $L \leq_P L'$, we have $X \leq_P L'$. Since our choice of X was arbitrary, this means L' is **NP**-hard. Since L' is **NP**-hard and $L' \in$ **NP**, we have $L' \in$ **NPC**. ■



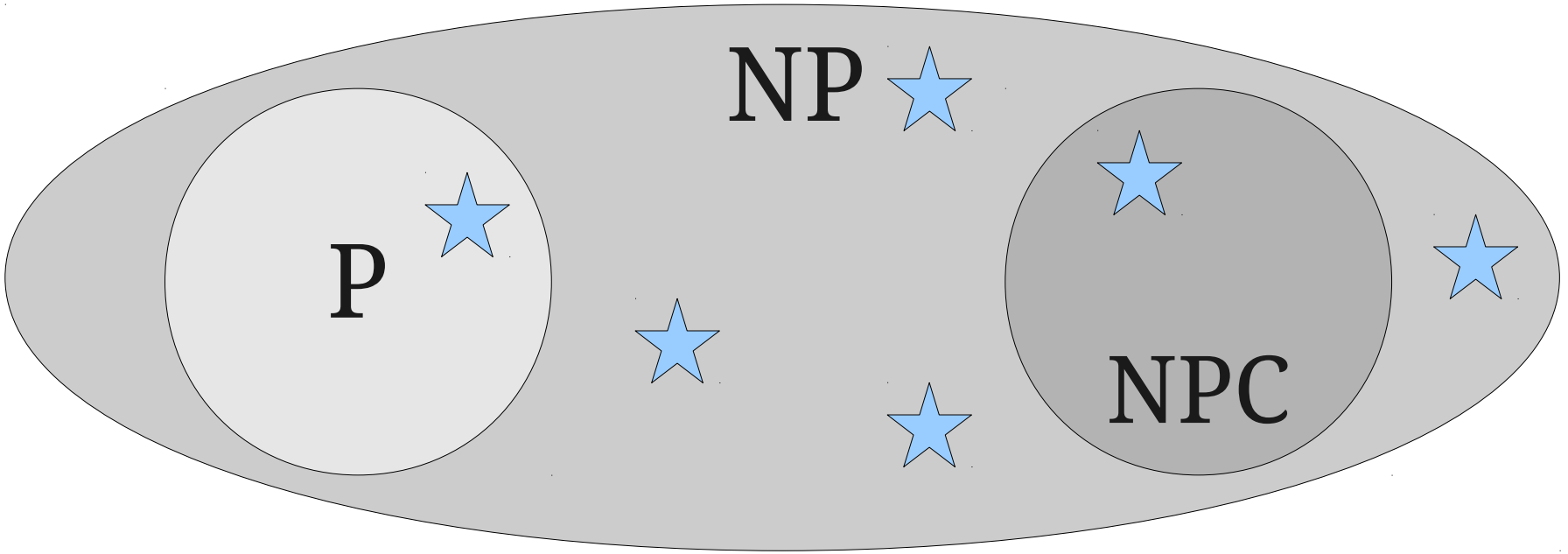
NP-Completeness

- **Theorem**: If $L \in$ **NPC**, $L \leq_P L'$, and $L' \in$ **NP**, then $L' \in$ **NPC**.
- **Proof**: Consider any language $X \in$ **NP**. Since $L \in$ **NPC**, we know that $X \leq_P L$. Since $L \leq_P L'$, we have $X \leq_P L'$. Since our choice of X was arbitrary, this means L' is **NP**-hard. Since L' is **NP**-hard and $L' \in$ **NP**, we have $L' \in$ **NPC**. ■



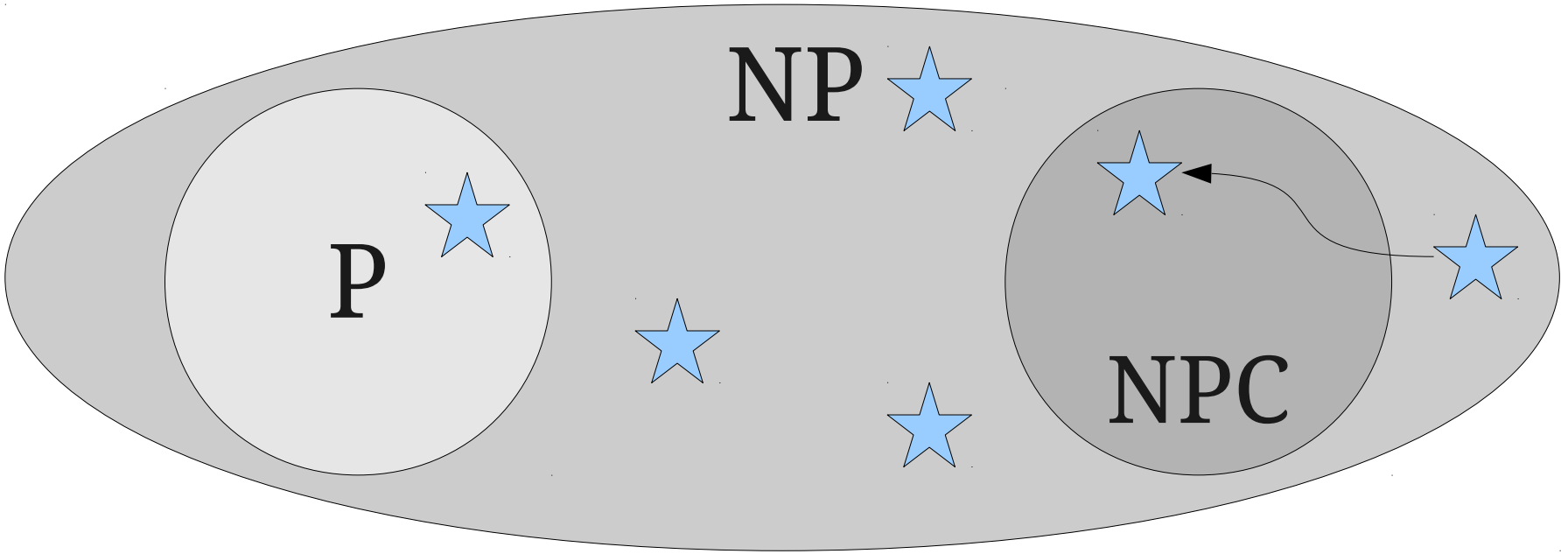
Be Careful!

- To prove that some language L is **NP**-complete, show that $L \in$ **NP**, then reduce some known **NP**-complete problem to L.
- **Do not** reduce L to a known **NP**-complete problem.
  - We already knew you could do this; *every* **NP** problems is reducible to any **NP**-complete problem!



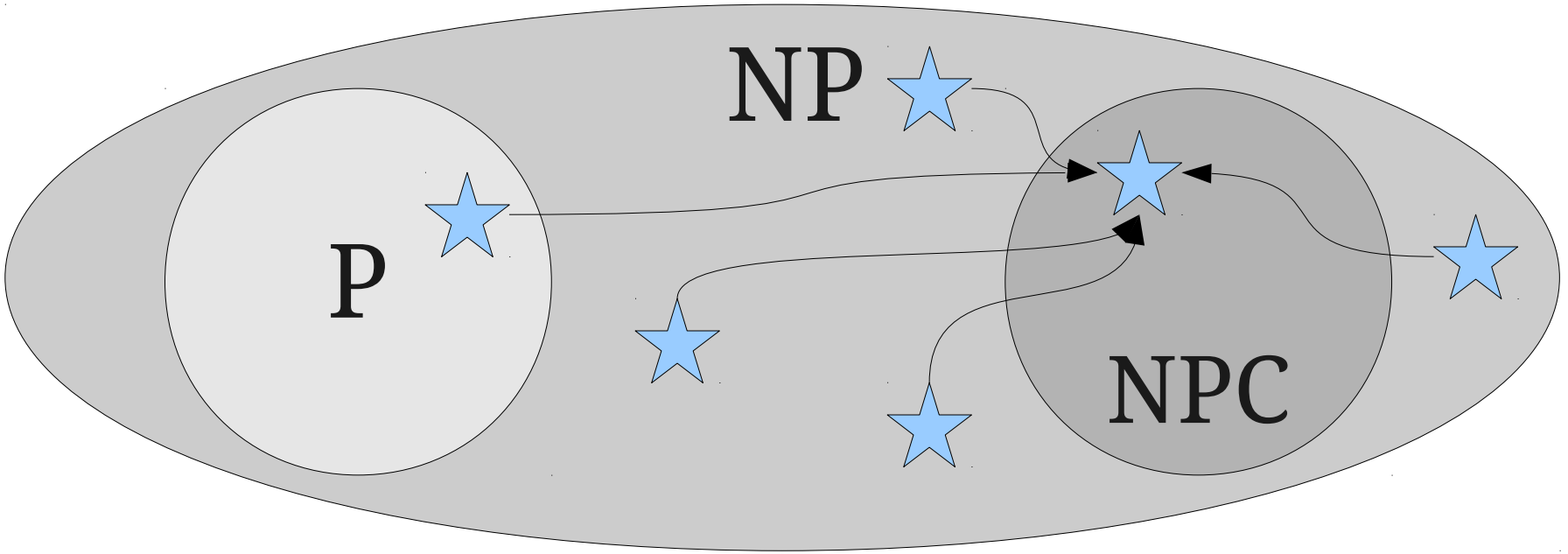
Be Careful!

- To prove that some language L is **NP**-complete, show that $L \in$ **NP**, then reduce some known **NP**-complete problem to L.
- **Do not** reduce L to a known **NP**-complete problem.
  - We already knew you could do this; *every* **NP** problems is reducible to any **NP**-complete problem!



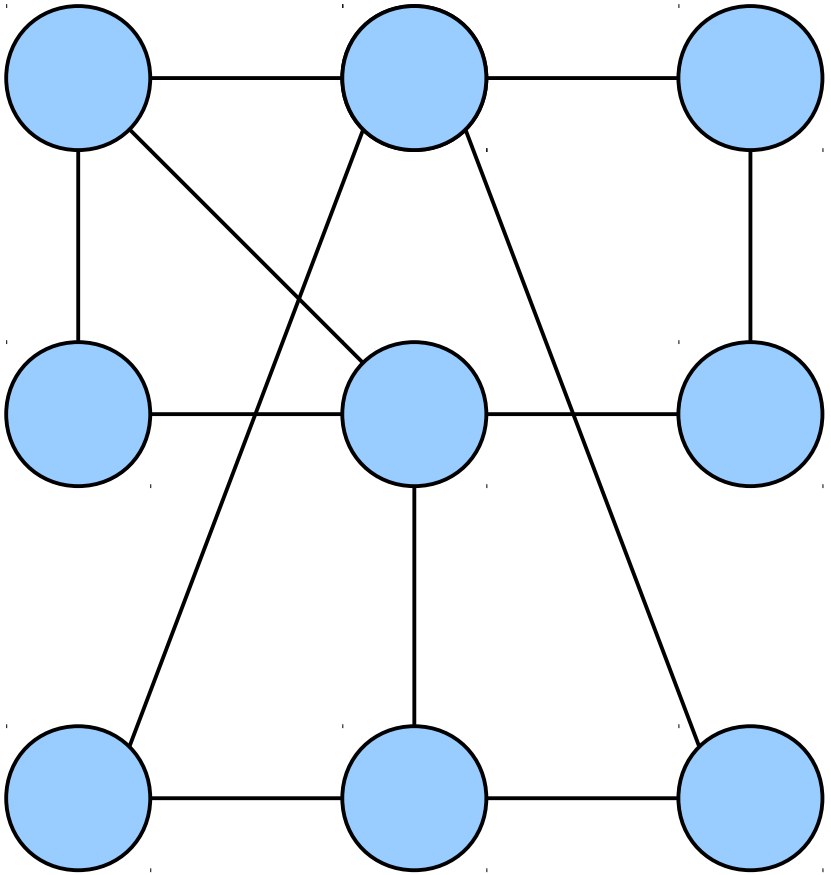
Be Careful!

- To prove that some language L is **NP**-complete, show that $L \in$ **NP**, then reduce some known **NP**-complete problem to L.
- **Do not** reduce L to a known **NP**-complete problem.
  - We already knew you could do this; *every* **NP** problems is reducible to any **NP**-complete problem!

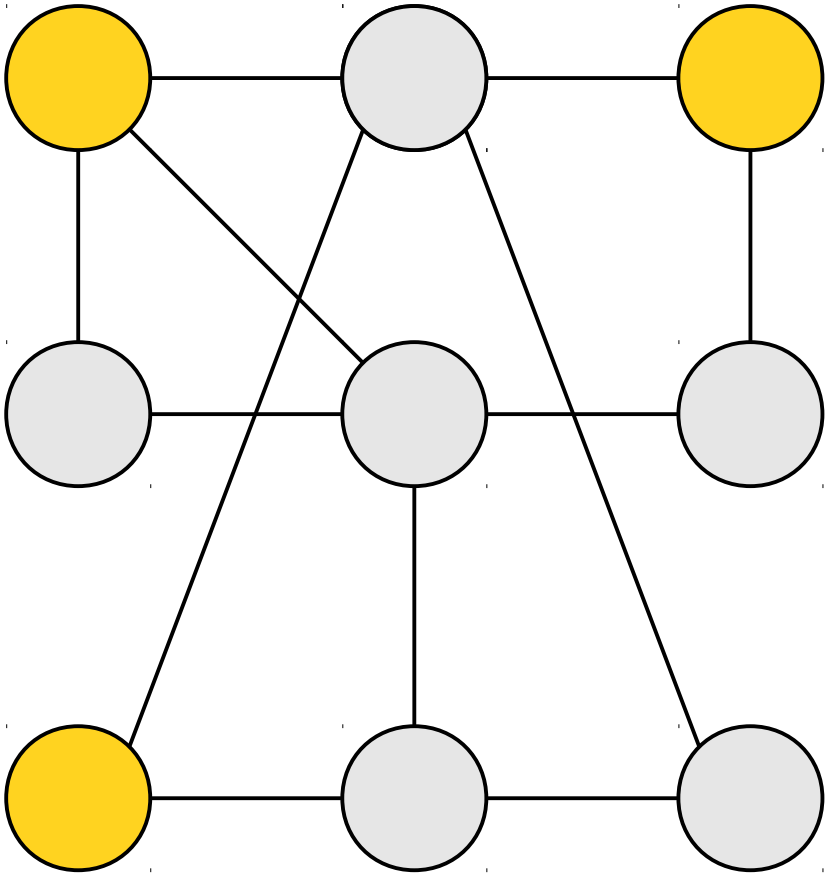


So what other problems are **NP**-complete?

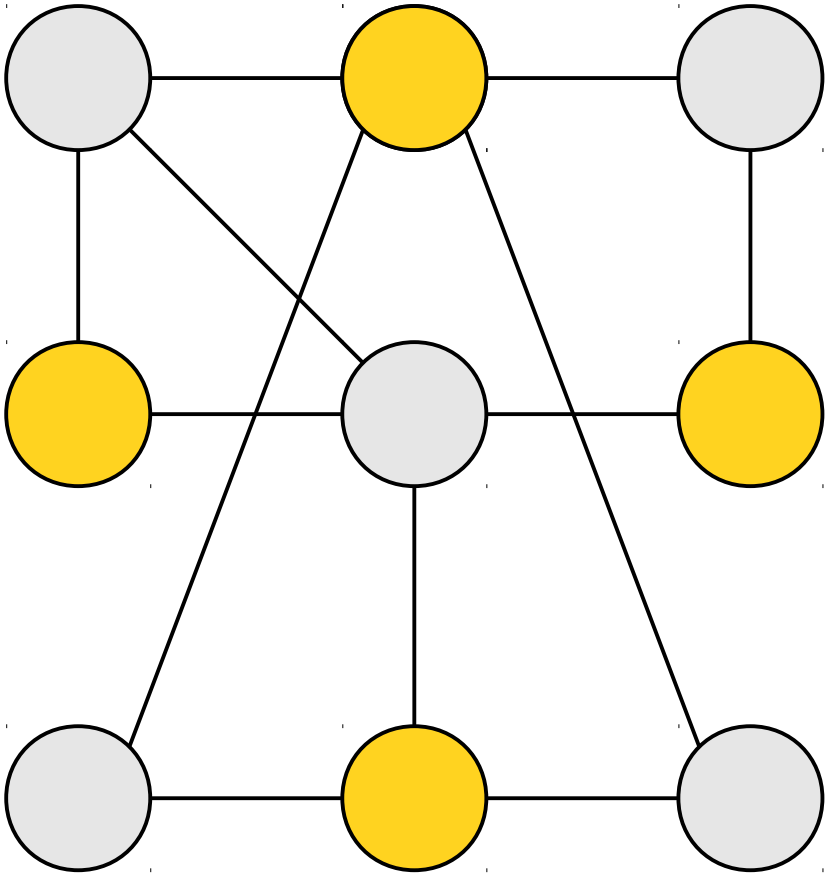
An **independent set** in an undirected graph
is a set of vertices that have no edges between them



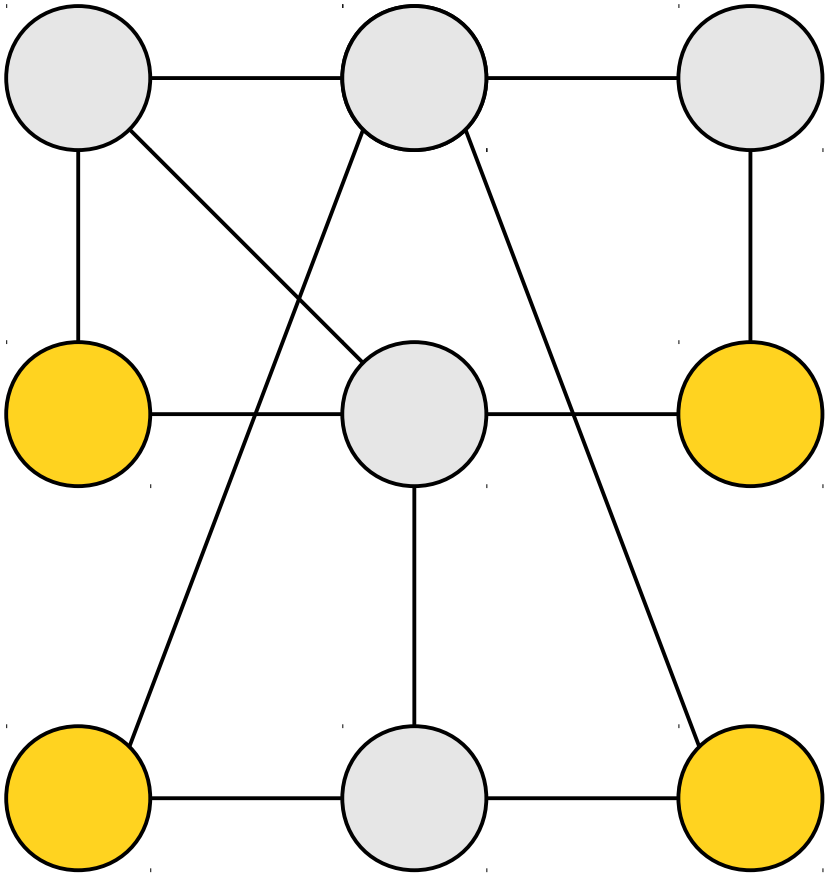
An **independent set** in an undirected graph
is a set of vertices that have no edges between them



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 = { $\langle G, n \rangle$ | G is an undirected graph with an independent set of size at least n }

INDSET ∈ **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 $\langle\langle G, n\rangle, S\rangle$:
    - If $|S| < n$, reject.
    - For each edge in G, if both endpoints are in S, reject.
    - Otherwise, accept."

INDSET ∈ **NPC**

- The *INDSET* problem is **NP**-complete.
- To prove this, we will find a polynomial-time reduction from 3SAT to *INDSET*.
- Goal: Given a 3CNF formula φ, construct a graph G and number n such that φ is satisfiable iff G has an independent set of size n.
- How can we accomplish this?

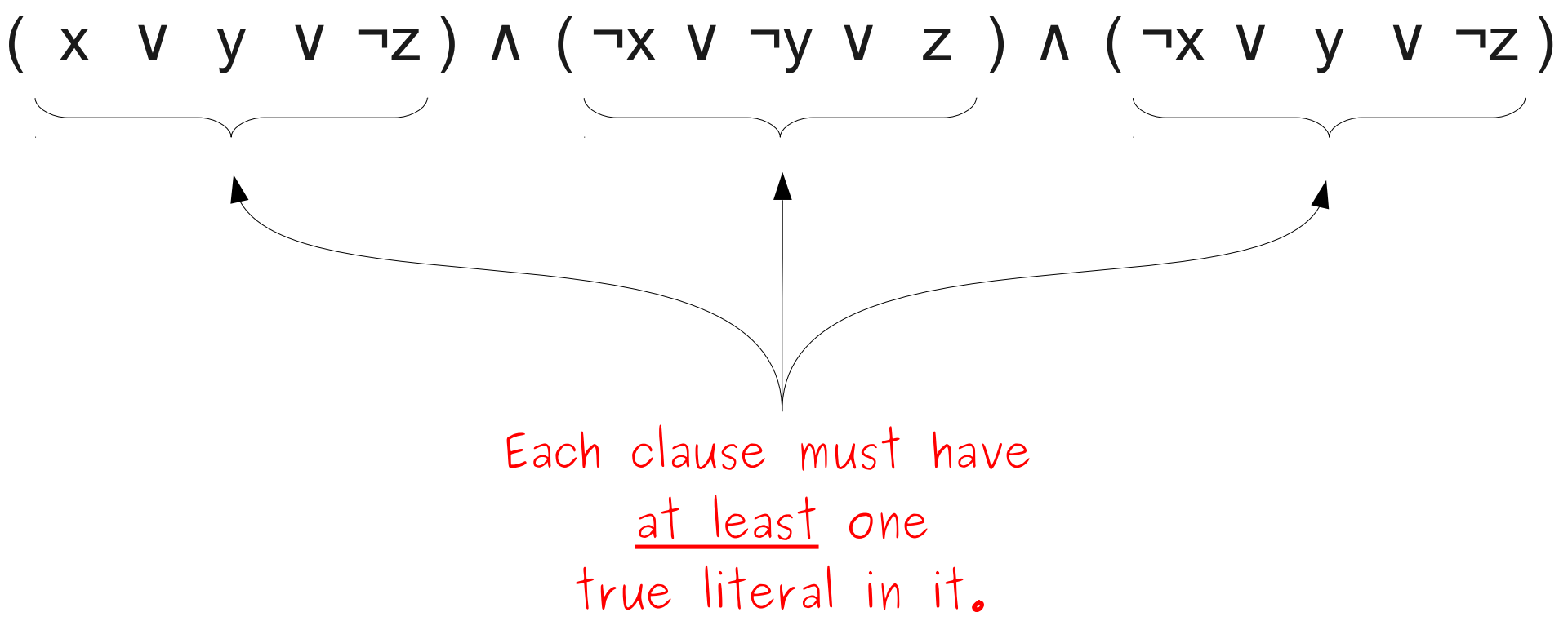
The Structure of 3CNF

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

The Structure of 3CNF

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

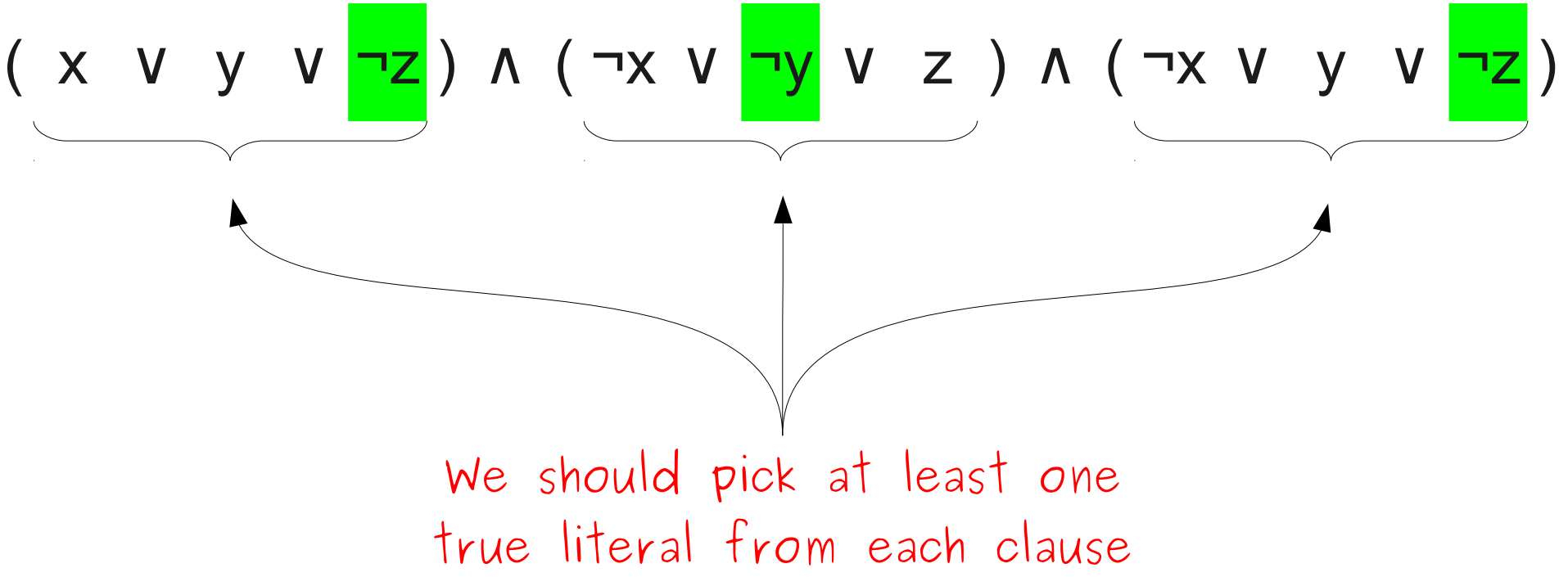
The Structure of 3CNF



The Structure of 3CNF

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

The Structure of 3CNF



The Structure of 3CNF

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

The Structure of 3CNF

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

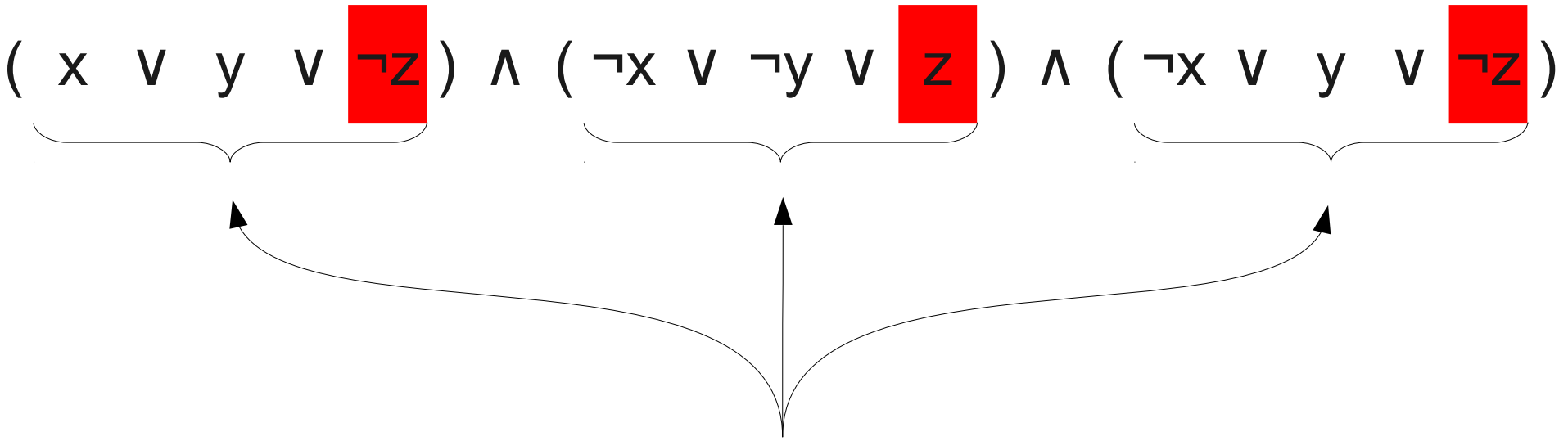
The Structure of 3CNF

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

The Structure of 3CNF

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

The Structure of 3CNF



From 3SAT to INDSET

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

From 3SAT to INDSET

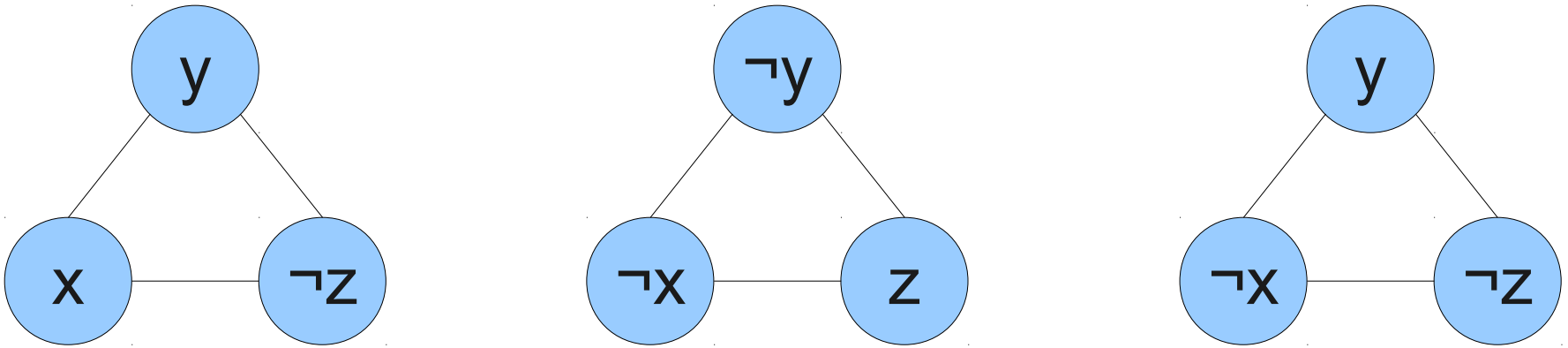
$$(x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee z) \wedge (\neg x \vee y \vee \neg z)$$

From 3SAT to INDSET

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

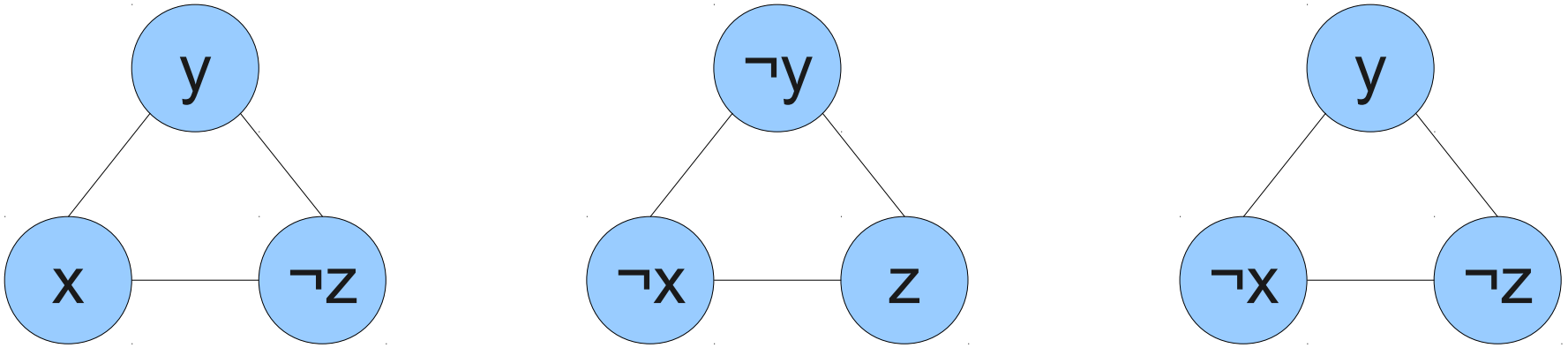
From 3SAT to INDSET

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



From 3SAT to INDSET

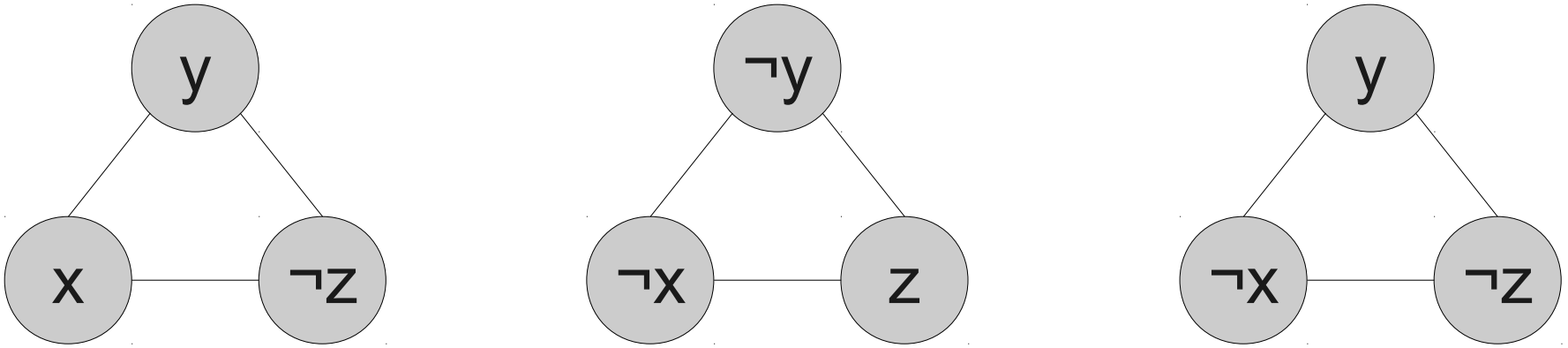
$$(x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee z) \wedge (\neg x \vee y \vee \neg z)$$



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

From 3SAT to INDSET

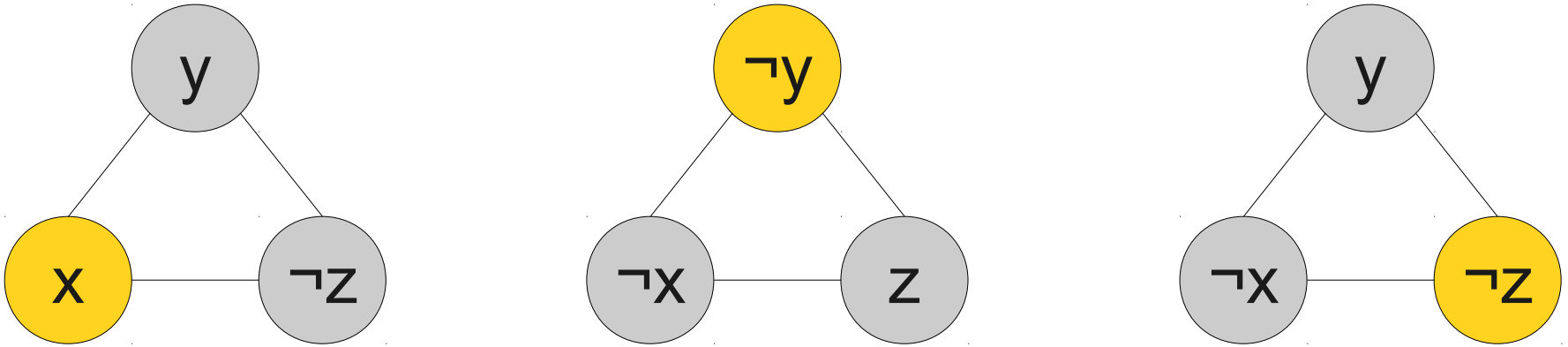
$$(x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee z) \wedge (\neg x \vee y \vee \neg z)$$



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

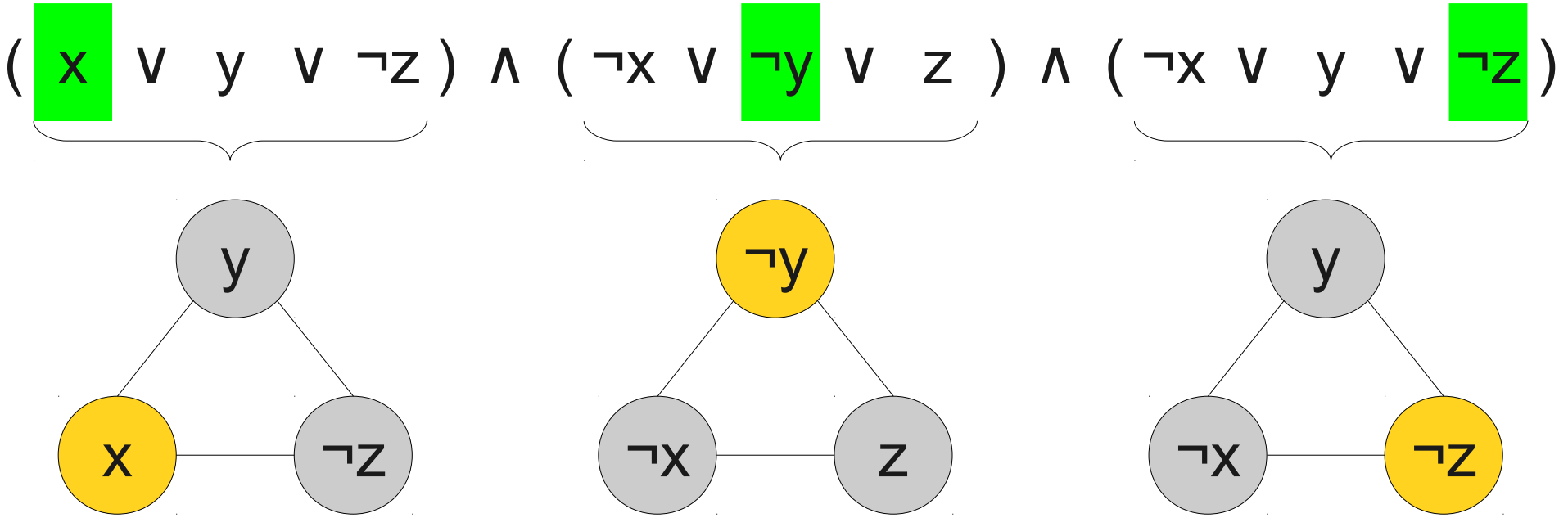
From 3SAT to INDSET

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



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

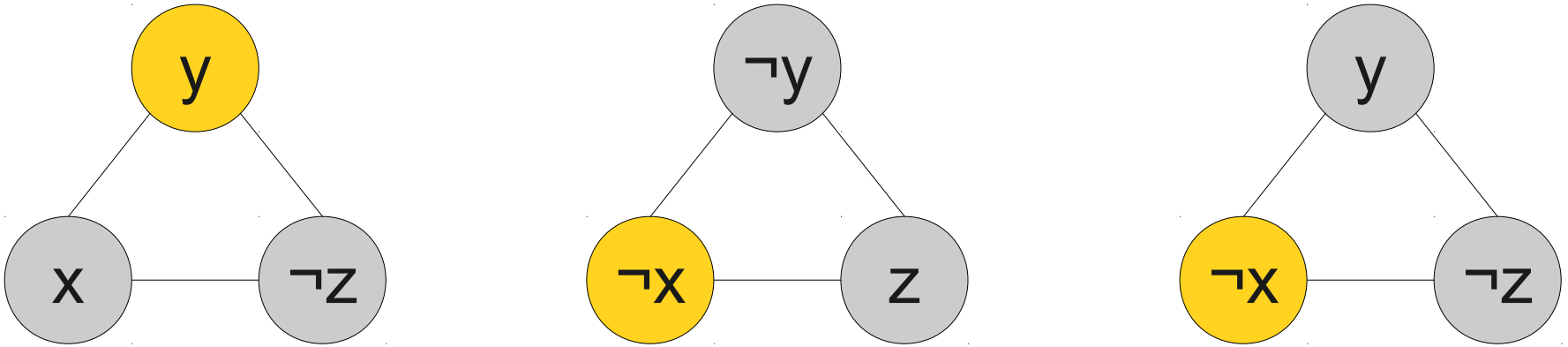
From 3SAT to INDSET



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

From 3SAT to INDSET

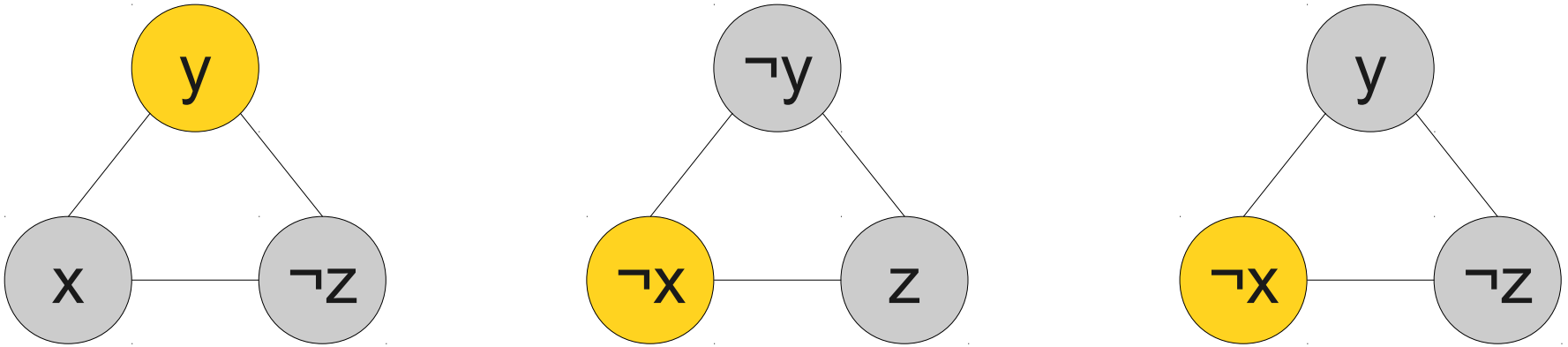
$$(x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee z) \wedge (\neg x \vee y \vee \neg z)$$



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

From 3SAT to INDSET

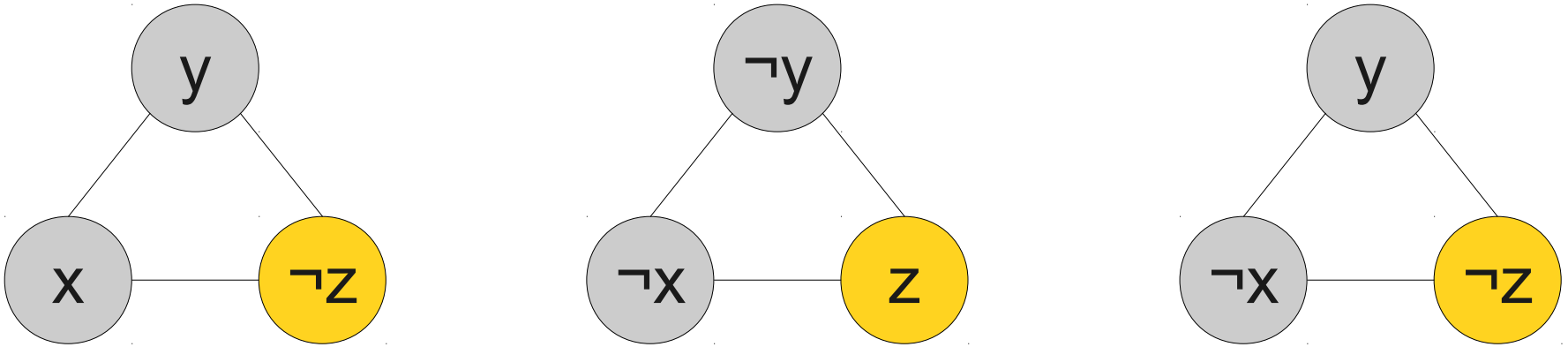
$$(\ x \vee y \vee \neg z\ ) \wedge (\ \neg x \vee \neg y \vee z\ ) \wedge (\ \neg x \vee y \vee \neg z\ )$$



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

From 3SAT to INDSET

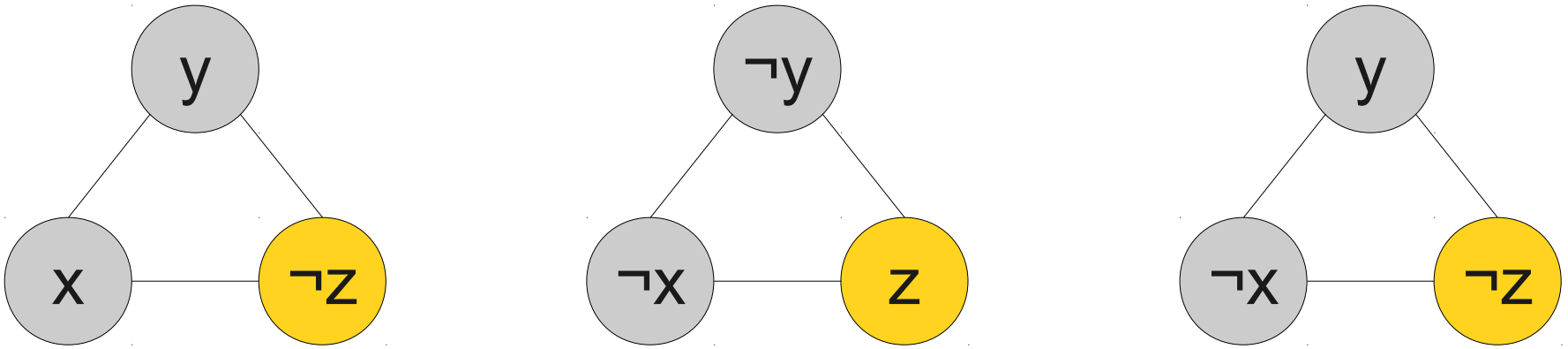
$$(x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee z) \wedge (\neg x \vee y \vee \neg z)$$



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

From 3SAT to INDSET

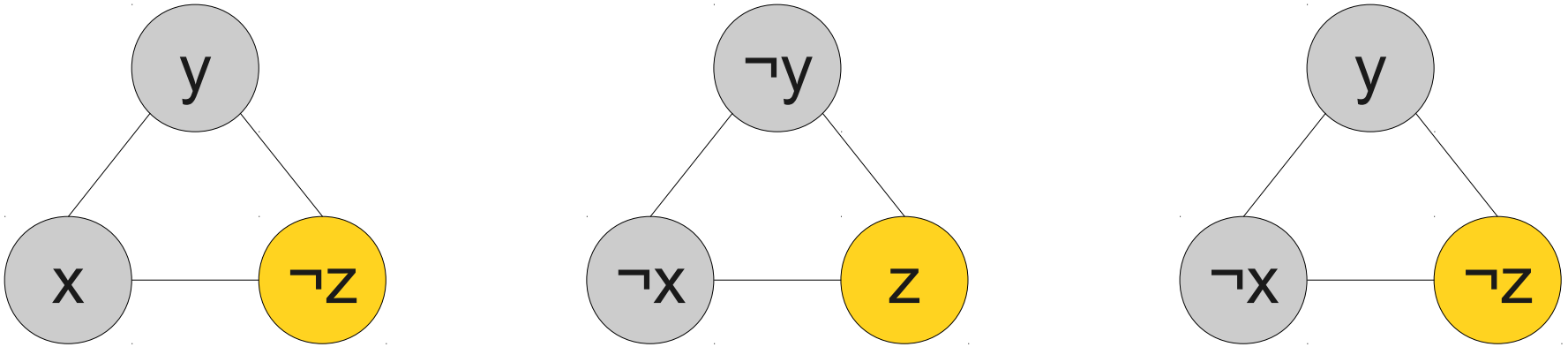
$$(x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee z) \wedge (\neg x \vee y \vee \neg z)$$



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

From 3SAT to INDSET

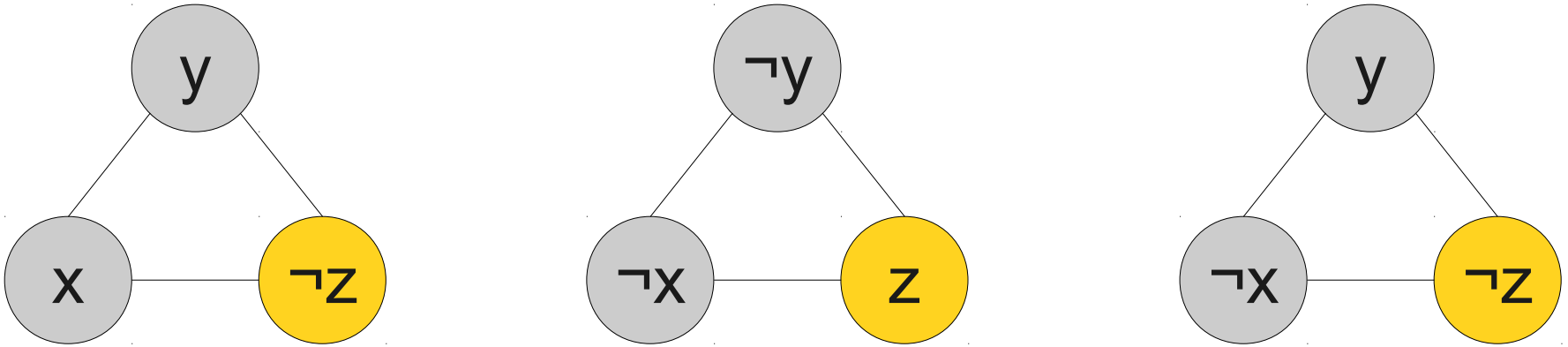
$$(x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee z) \wedge (\neg x \vee y \vee \neg z)$$



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

From 3SAT to INDSET

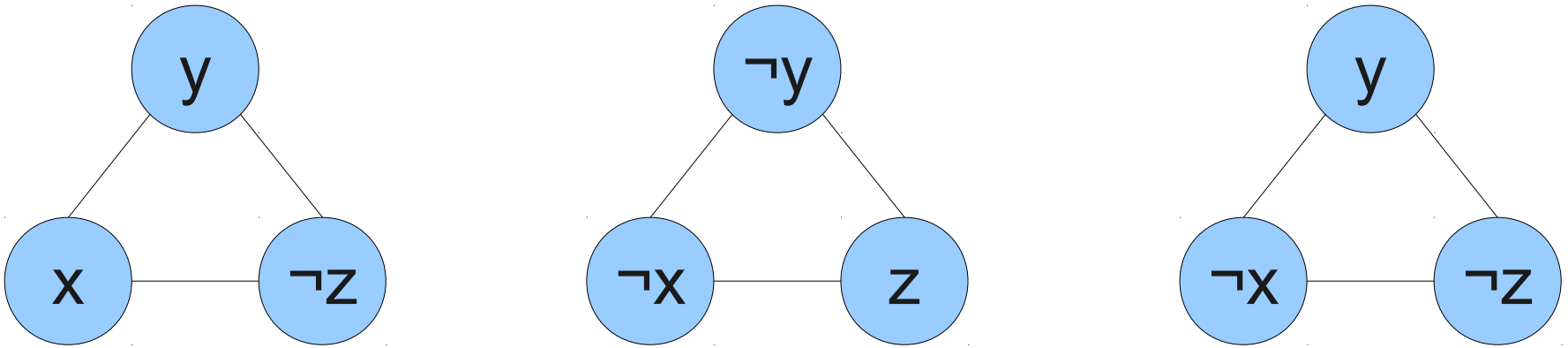
$$(x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee z) \wedge (\neg x \vee y \vee \neg z)$$



We need a way to ensure we never pick a literal and its negation.

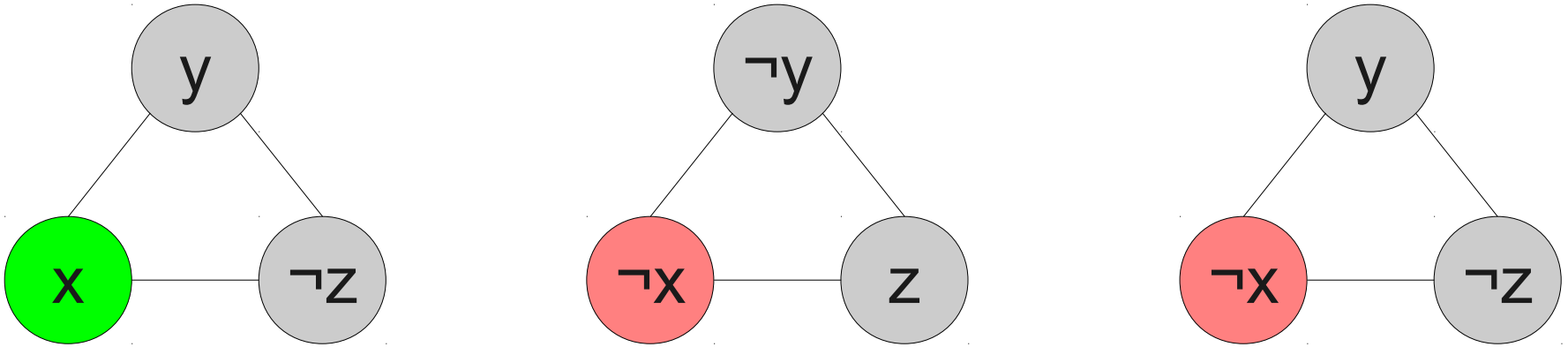
From 3SAT to INDSET

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

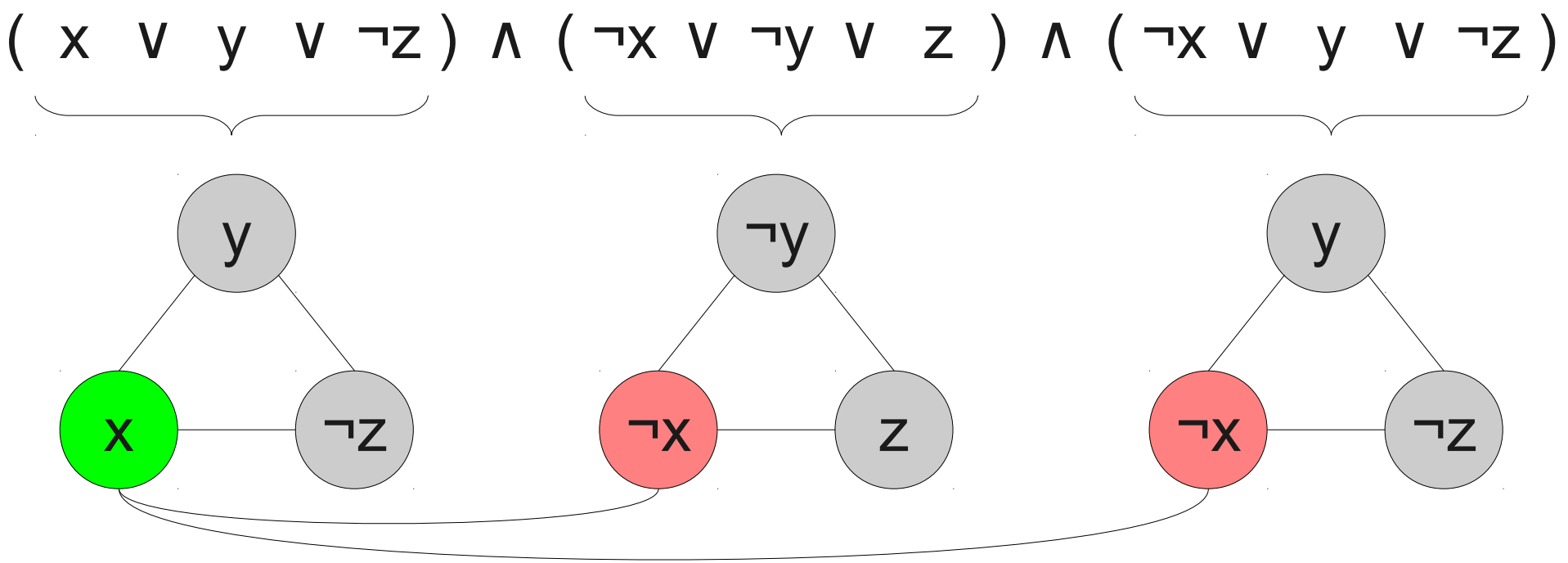


From 3SAT to INDSET

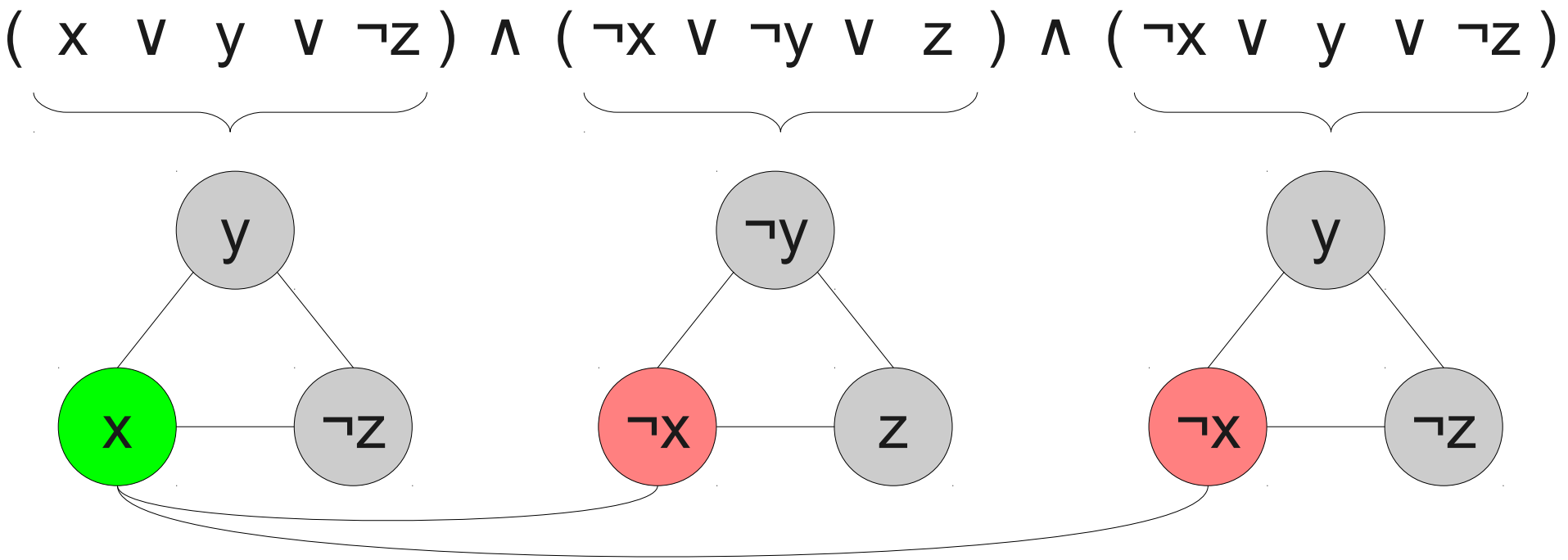
$$(\ x \ \vee \ y \ \vee \neg z\ ) \wedge (\ \neg x \vee \neg y \vee \ z\ ) \wedge (\ \neg x \vee \ y \ \vee \neg z\ )$$



From 3SAT to INDSET

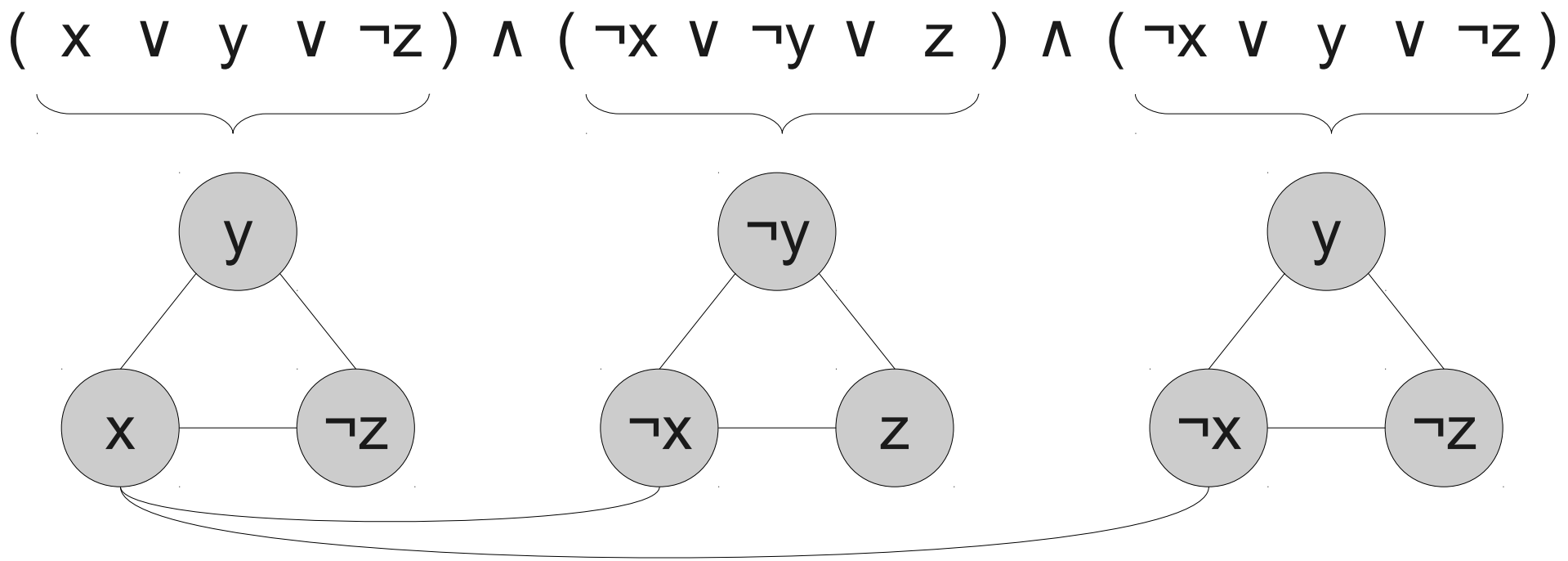


From 3SAT to INDSET

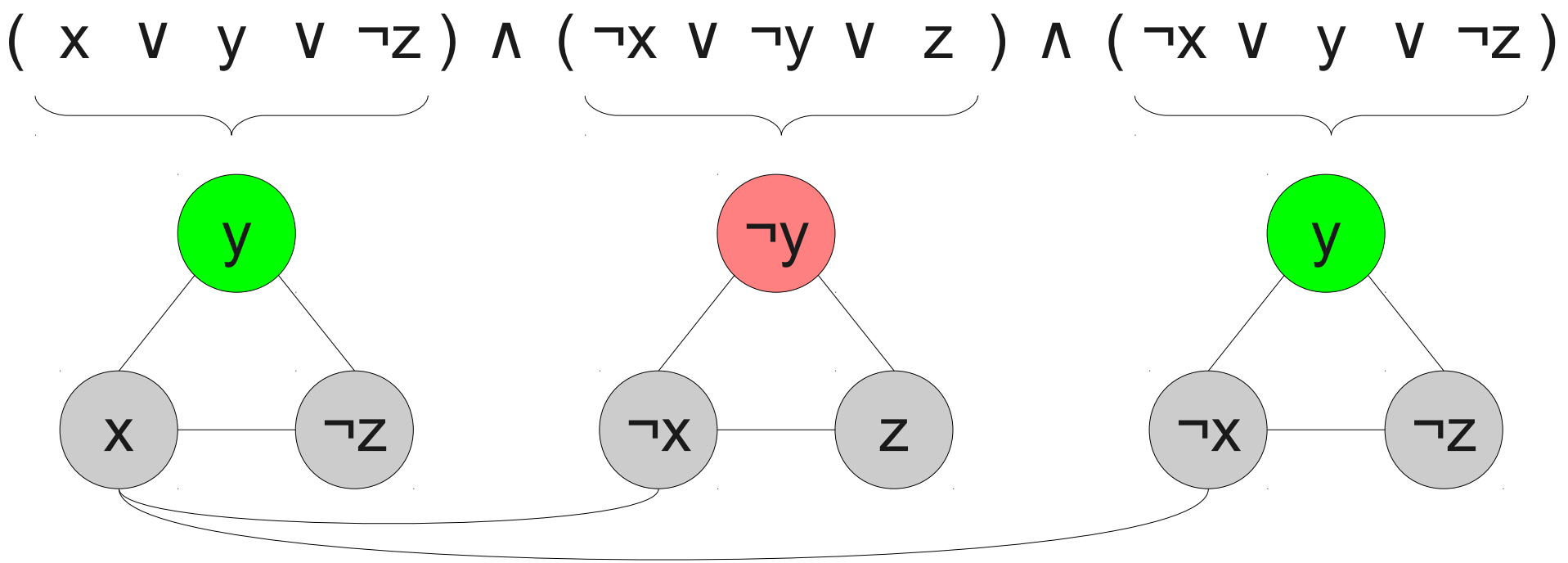


No independent set in this graph can choose two nodes labeled **x** and **¬x**.

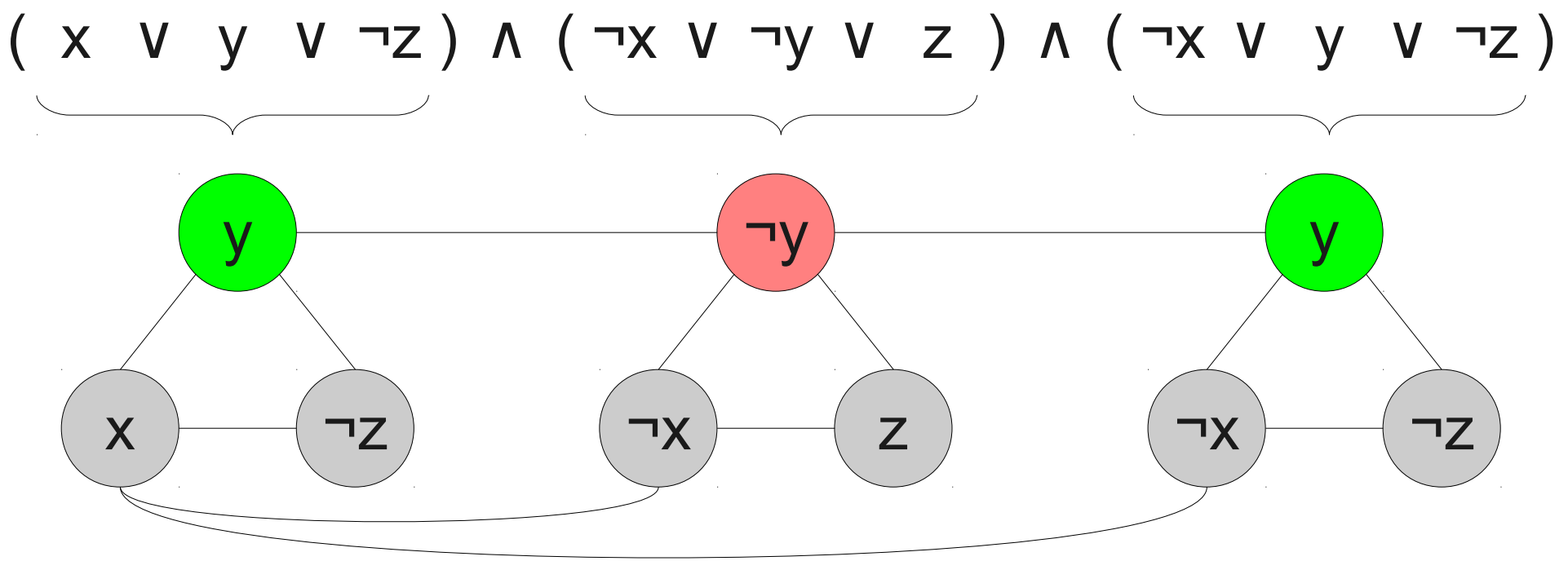
From 3SAT to INDSET



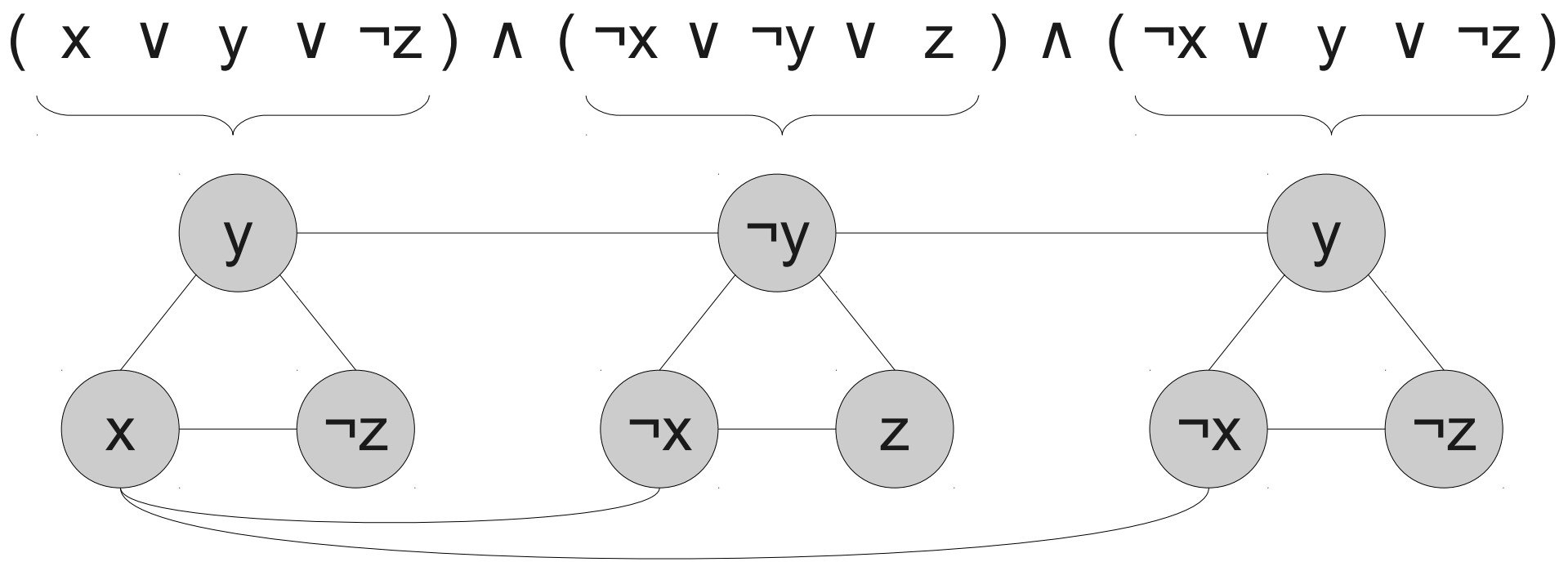
From 3SAT to INDSET



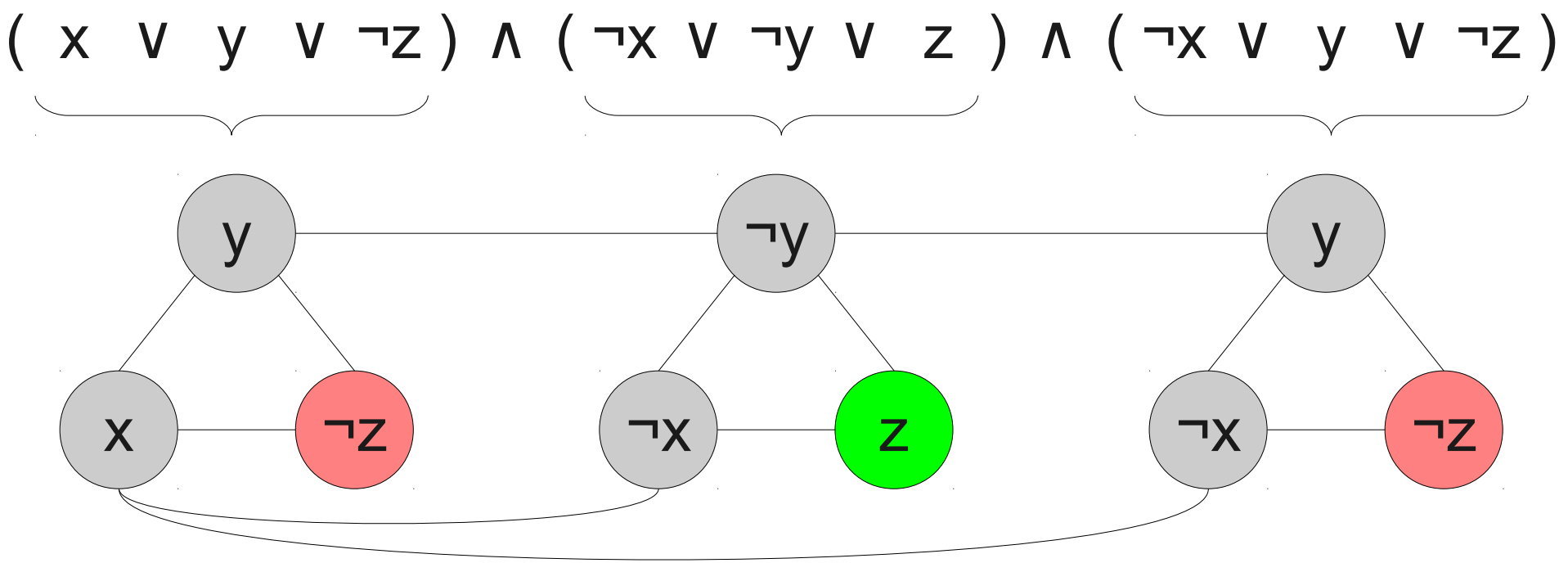
From 3SAT to INDSET



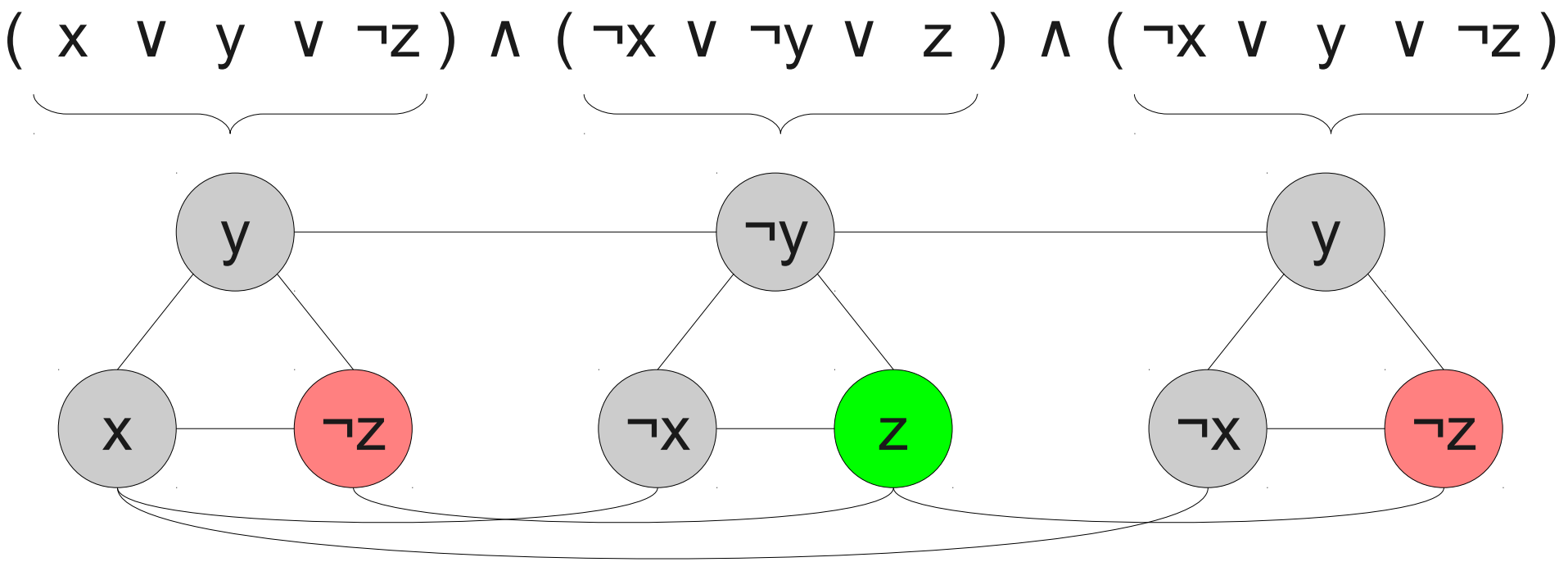
From 3SAT to INDSET



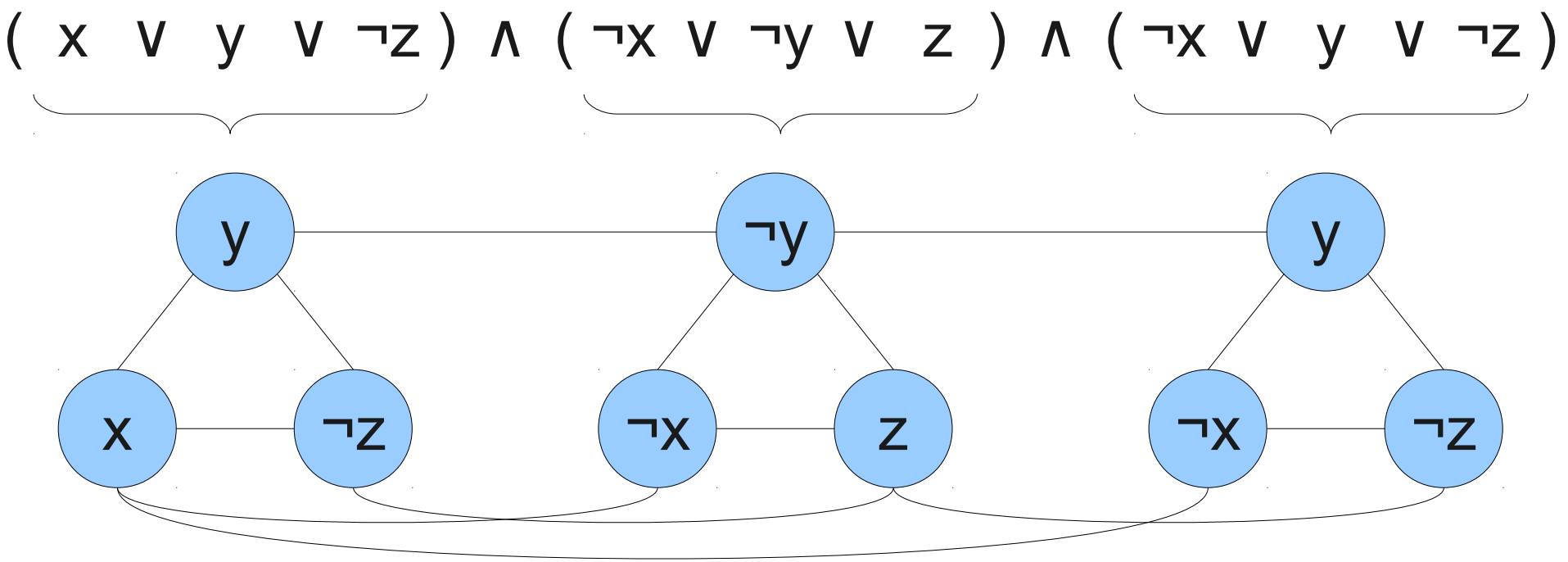
From 3SAT to INDSET



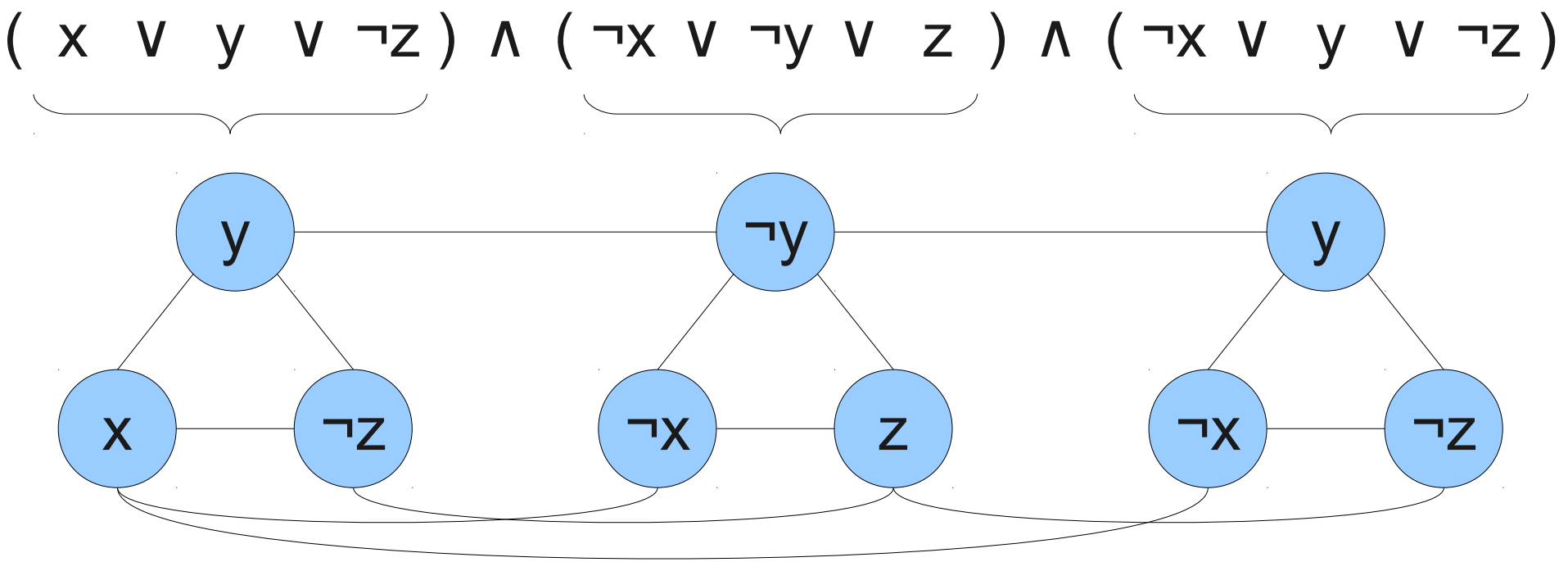
From 3SAT to INDSET



From 3SAT to INDSET

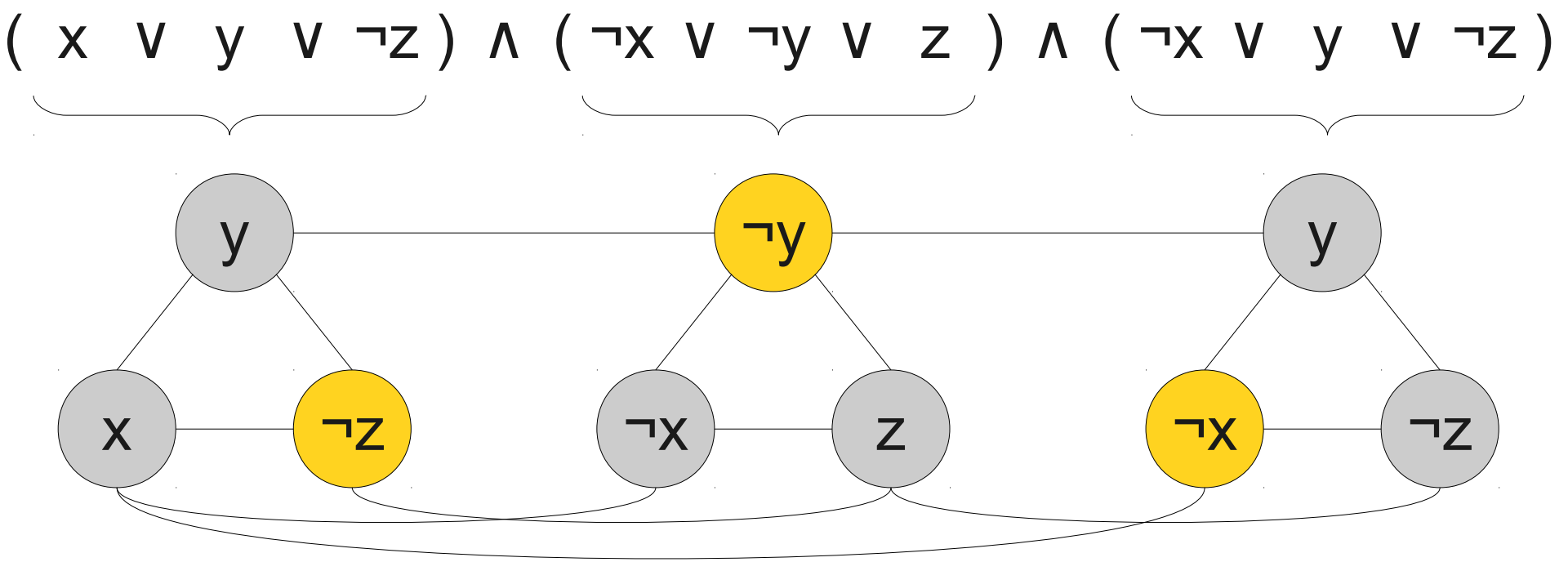


From 3SAT to INDSET



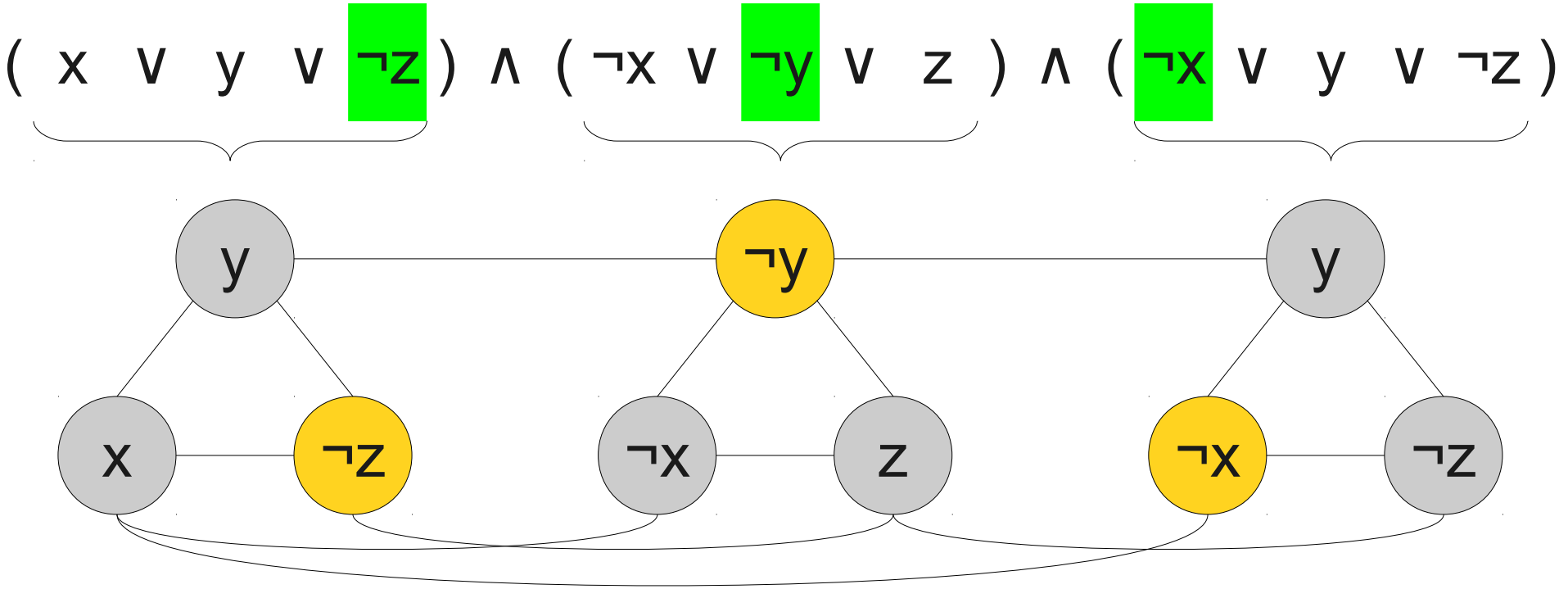
If this graph has an independent set of size three, the original formula is satisfiable.

From 3SAT to INDSET



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

From 3SAT to INDSET

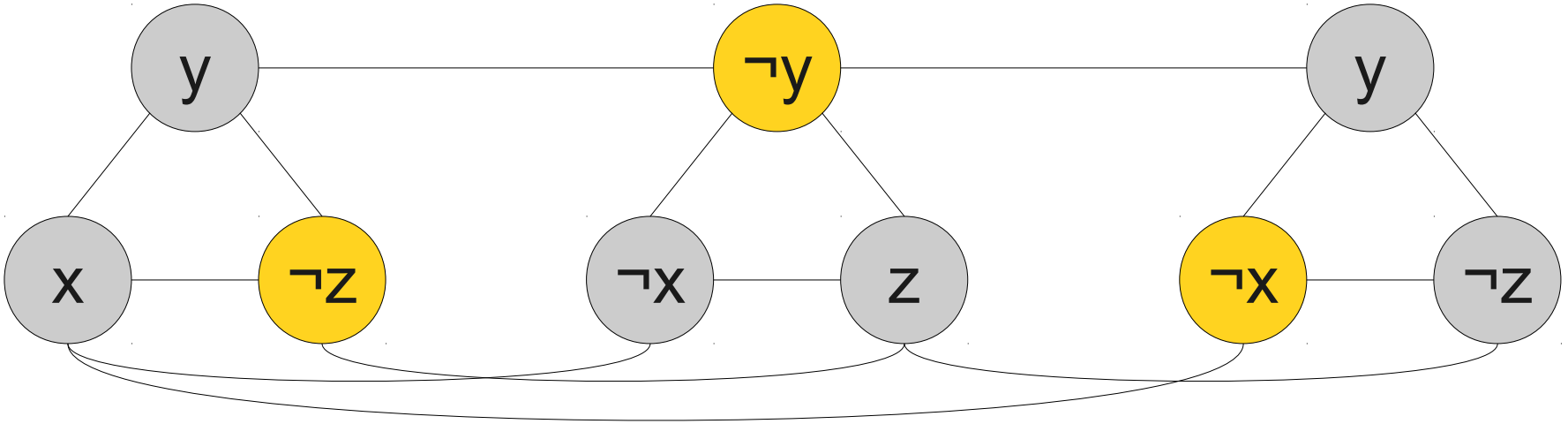


If this graph has an independent set of size three, the original formula is satisfiable.

From 3SAT to INDSET

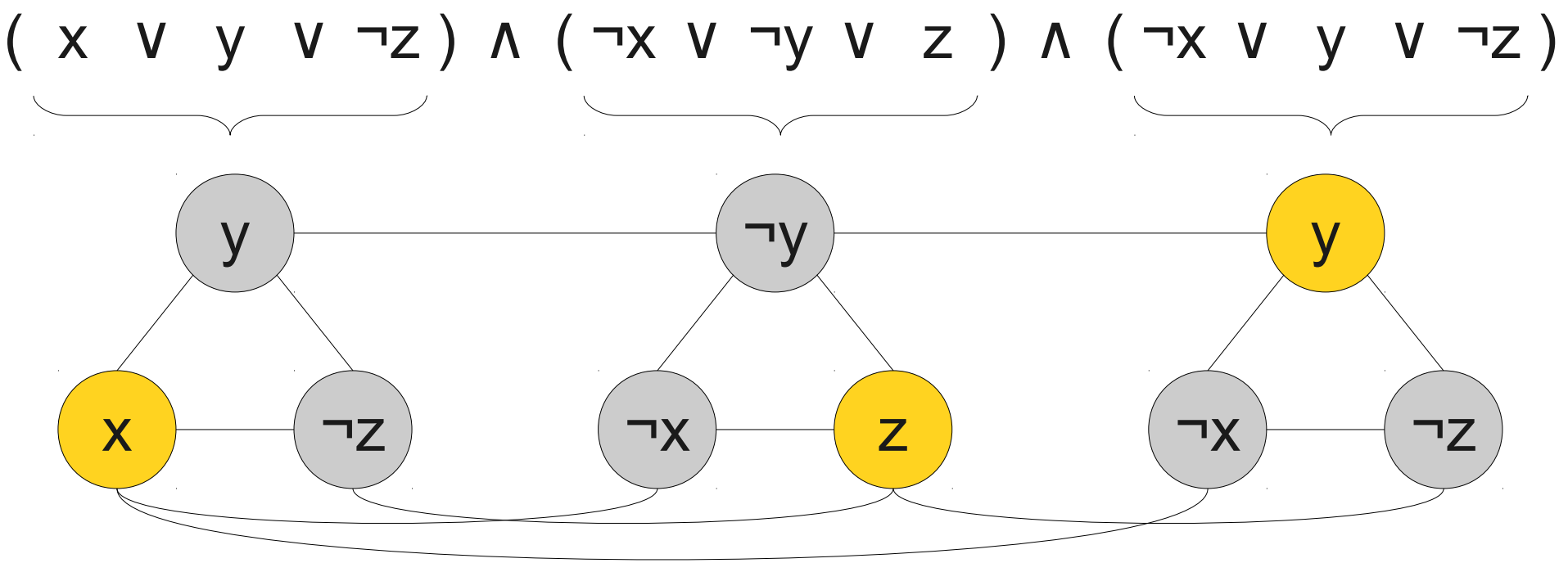
x = false, y = false, z = false.

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



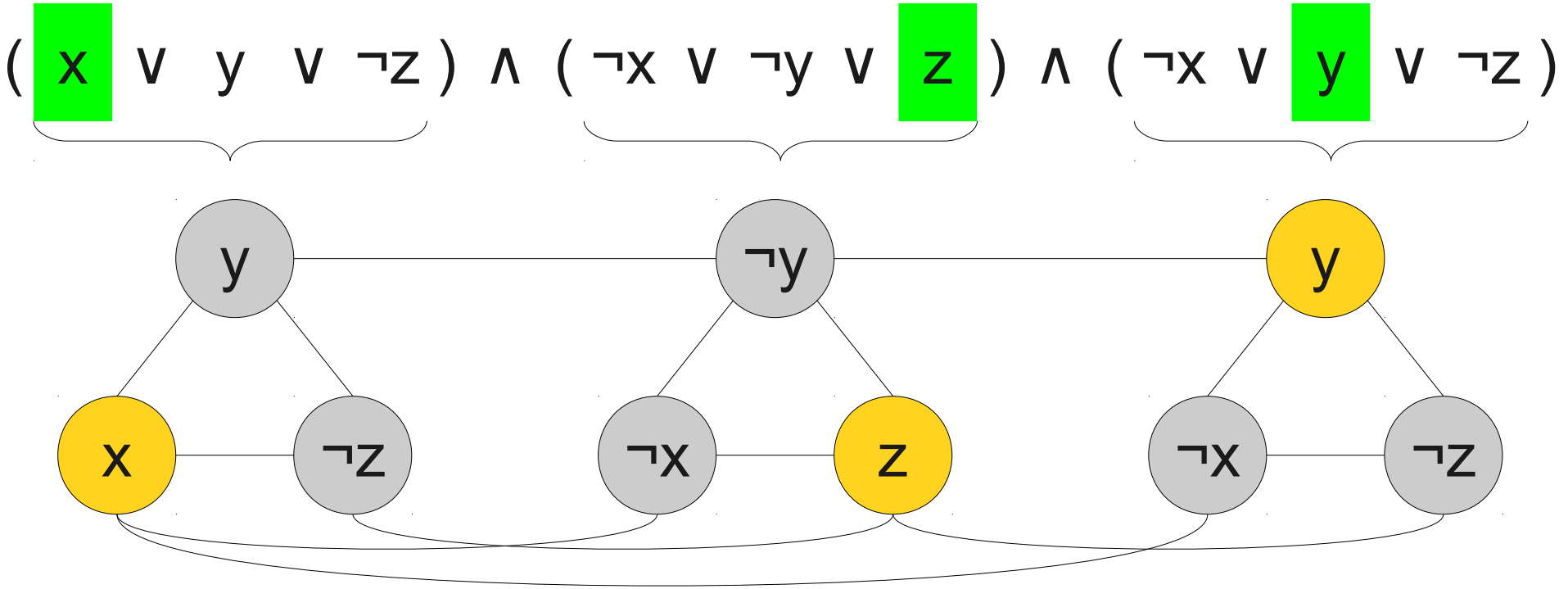
If this graph has an independent set of size three, the original formula is satisfiable.

From 3SAT to INDSET



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

From 3SAT to INDSET

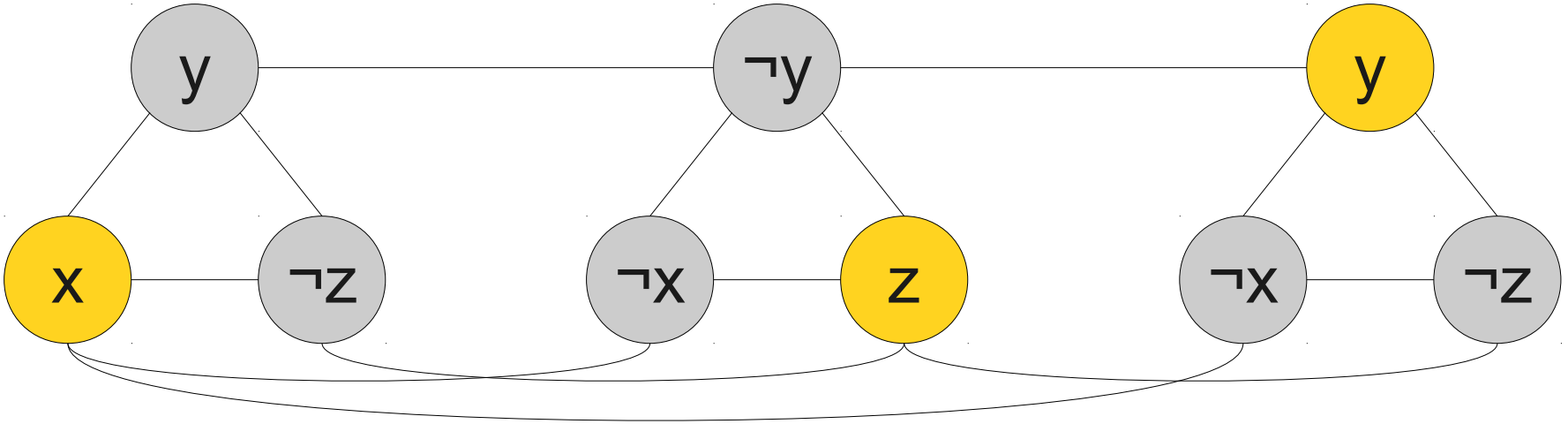


If this graph has an independent set of size three, the original formula is satisfiable.

From 3SAT to INDSET

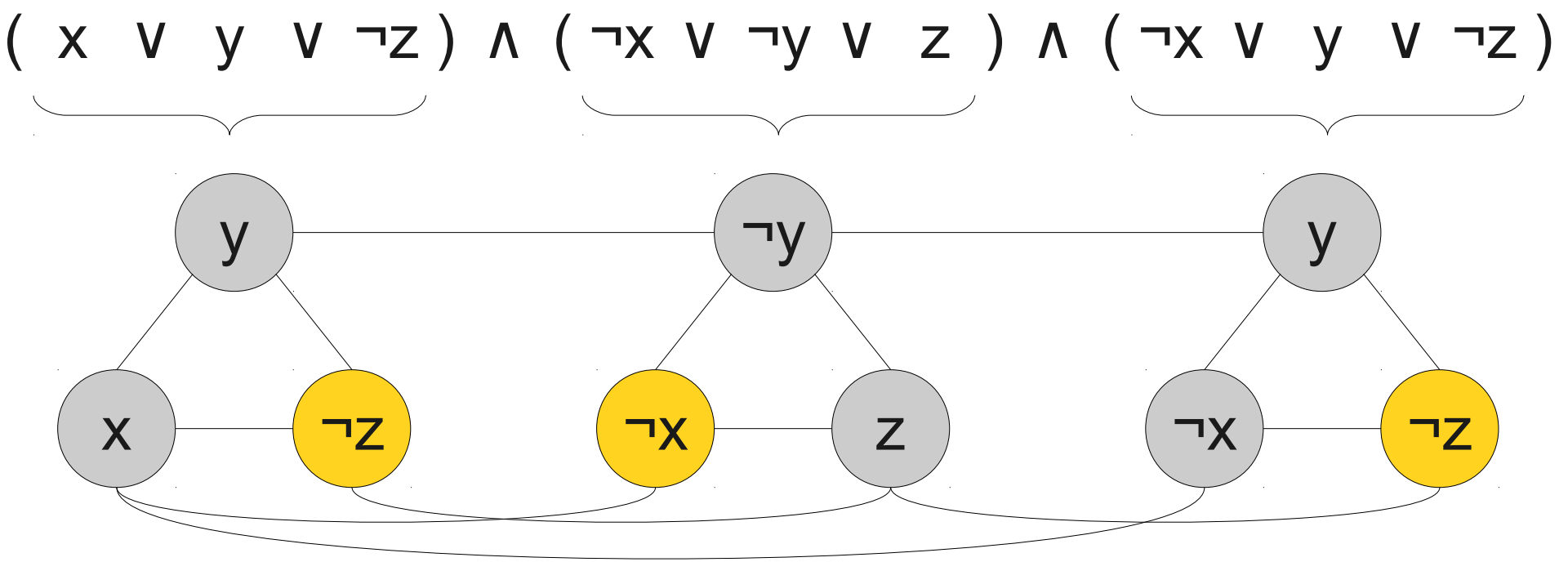
x = true, y = true, z = true.

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



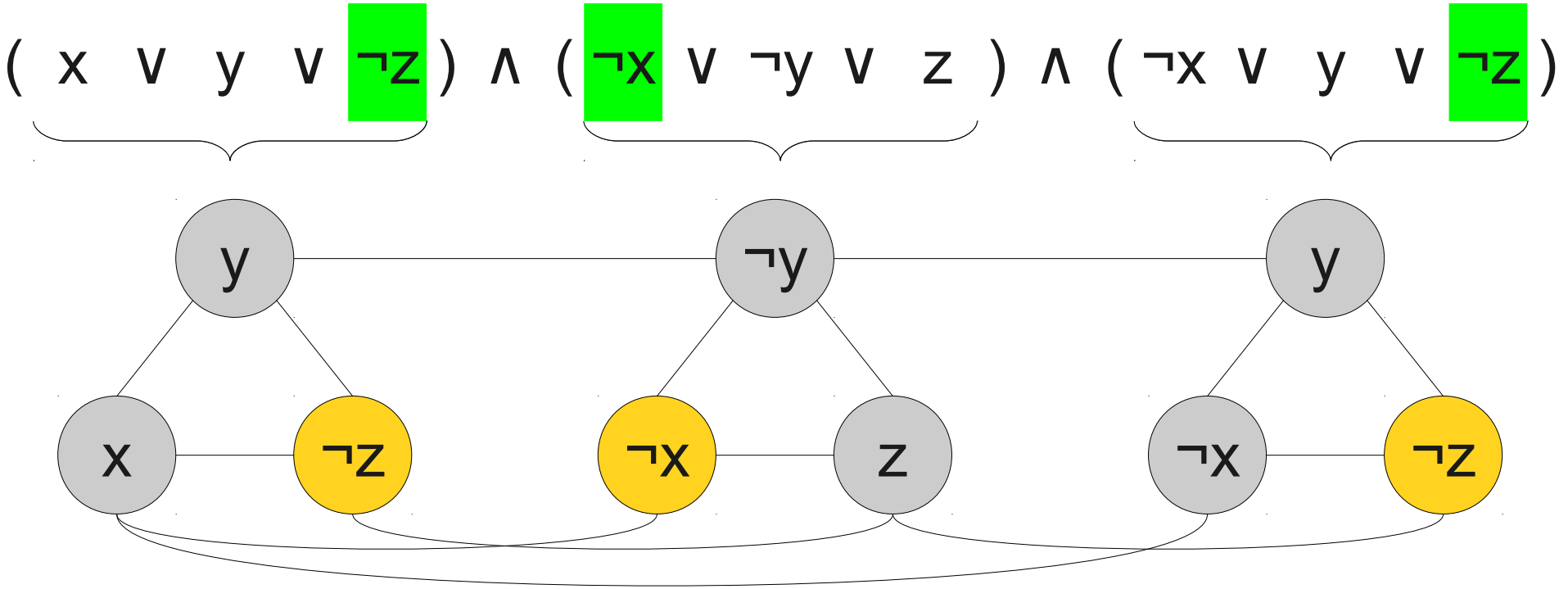
If this graph has an independent set of size three, the original formula is satisfiable.

From 3SAT to INDSET



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

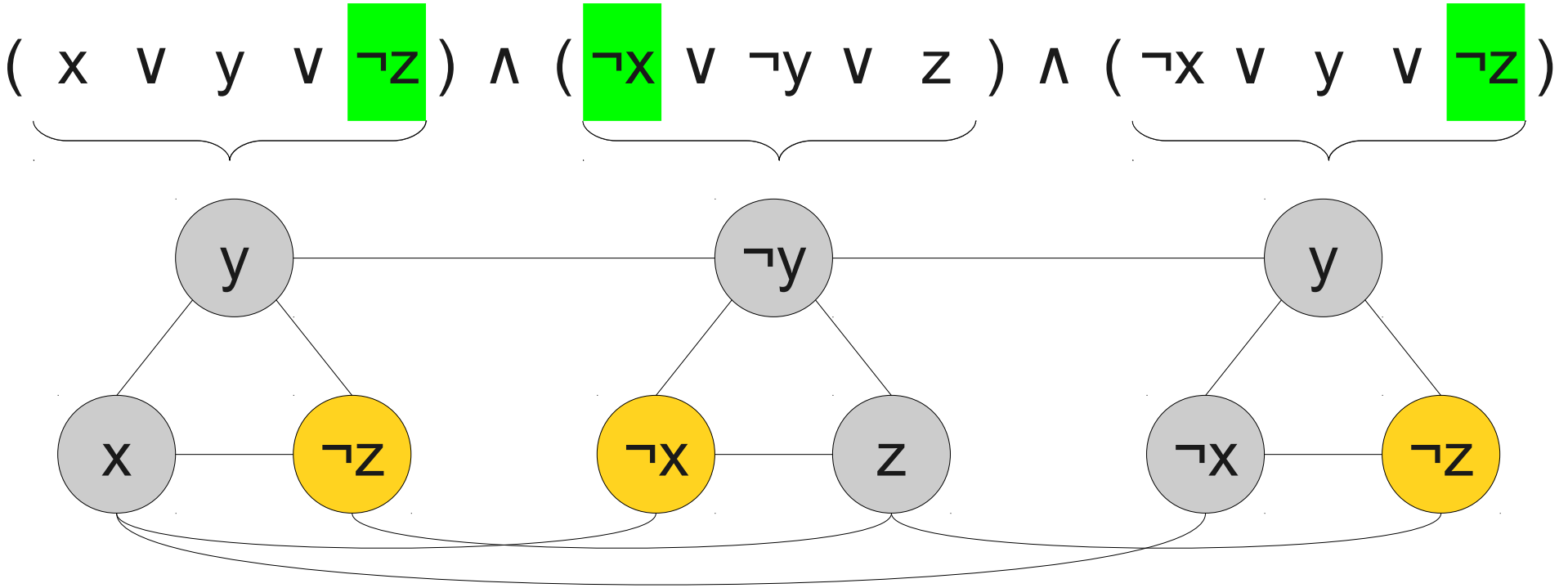
From 3SAT to INDSET



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

From 3SAT to INDSET

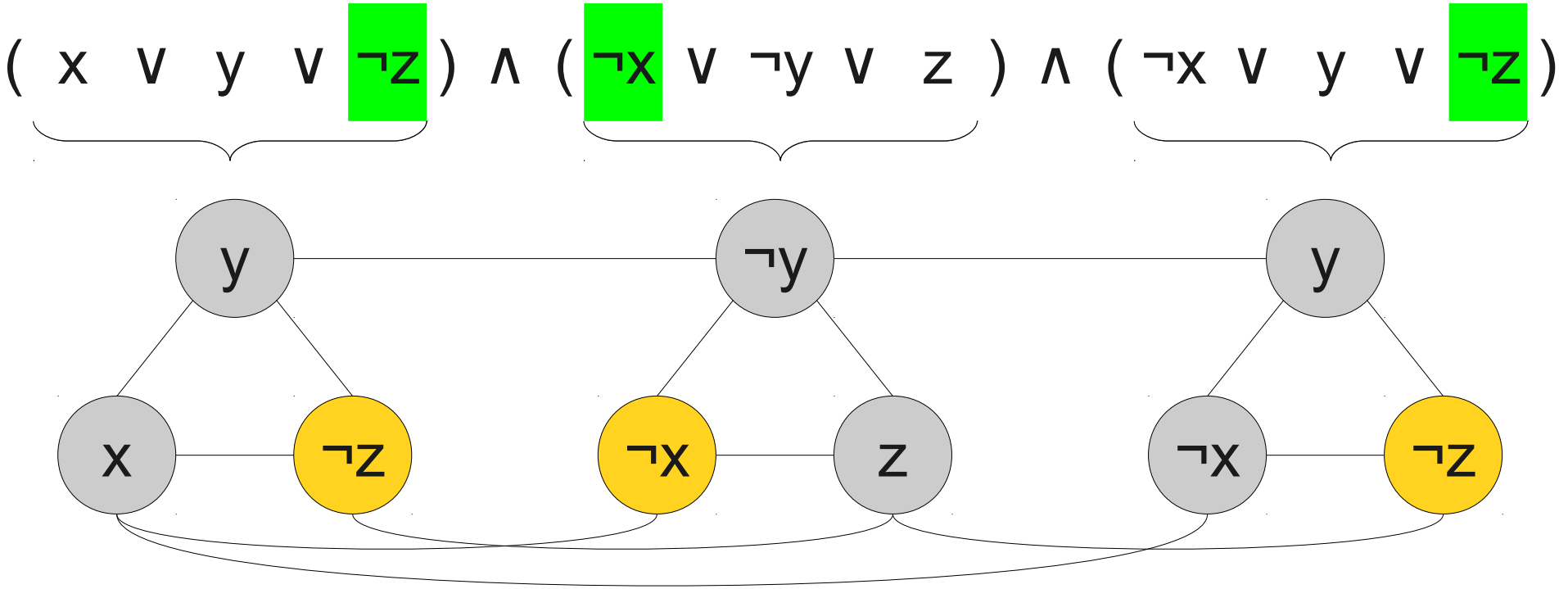
x = false, y = ??, z = false.



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

From 3SAT to INDSET

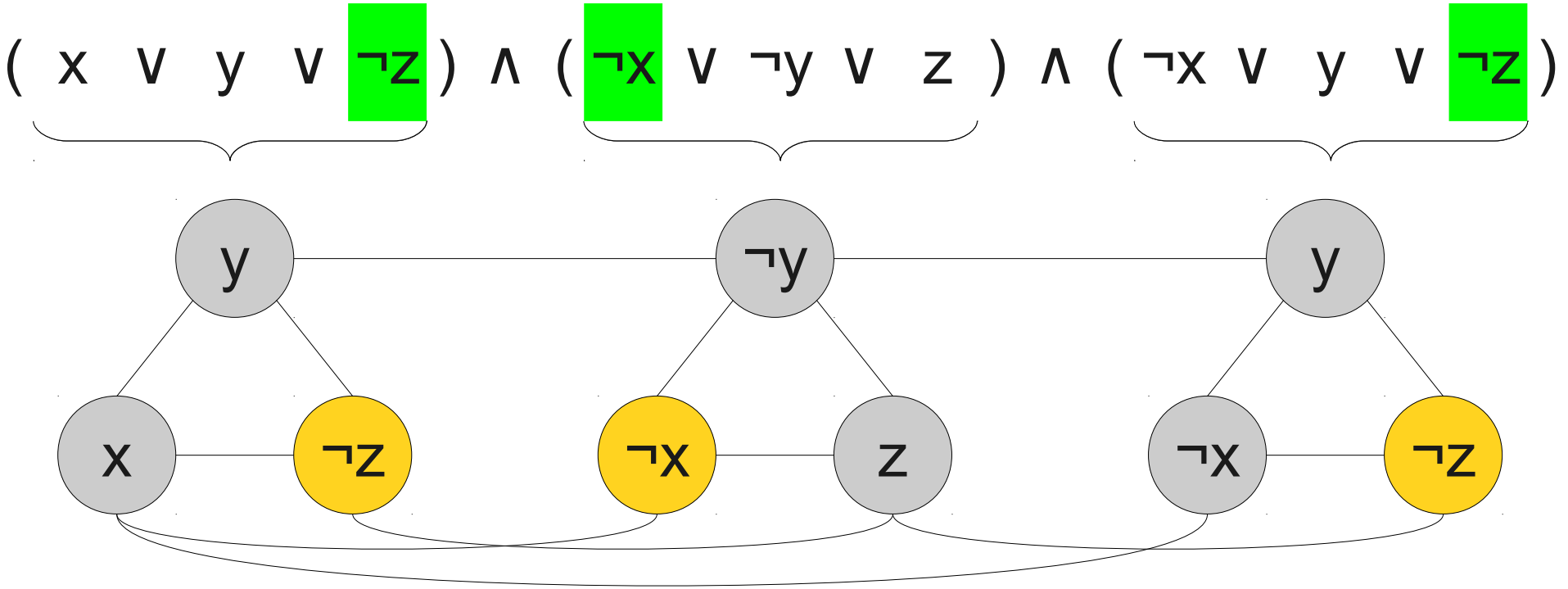
x = false, y = true, z = false.



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

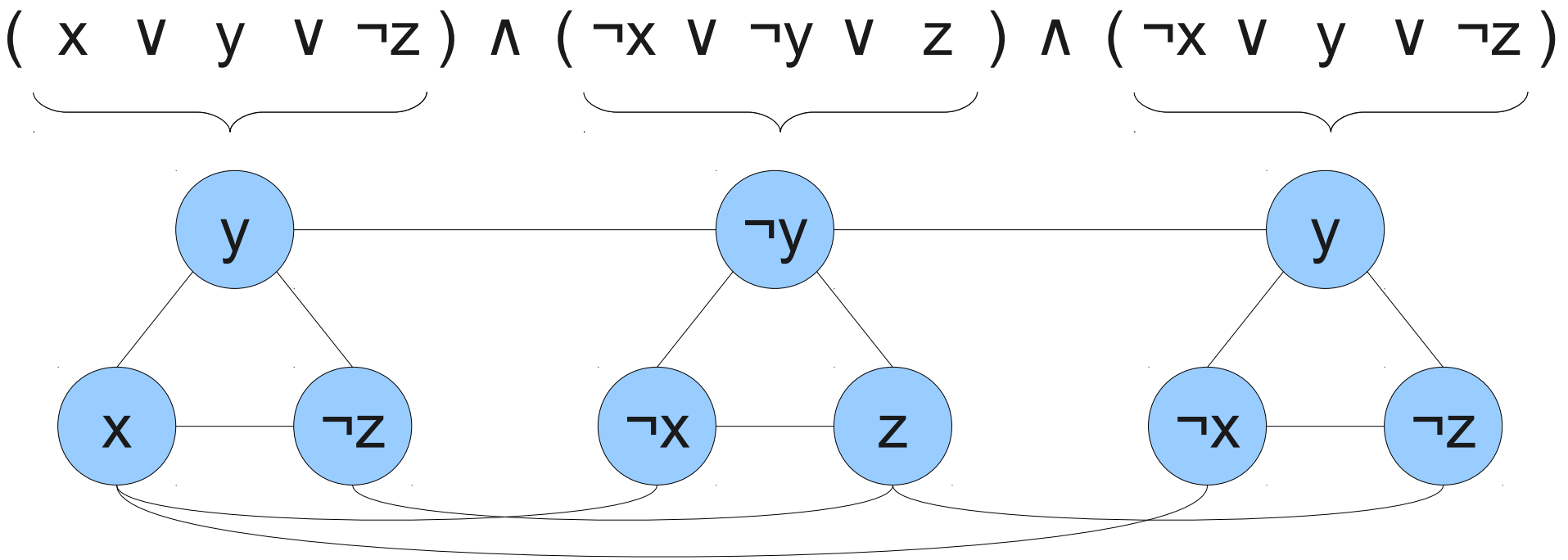
From 3SAT to INDSET

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



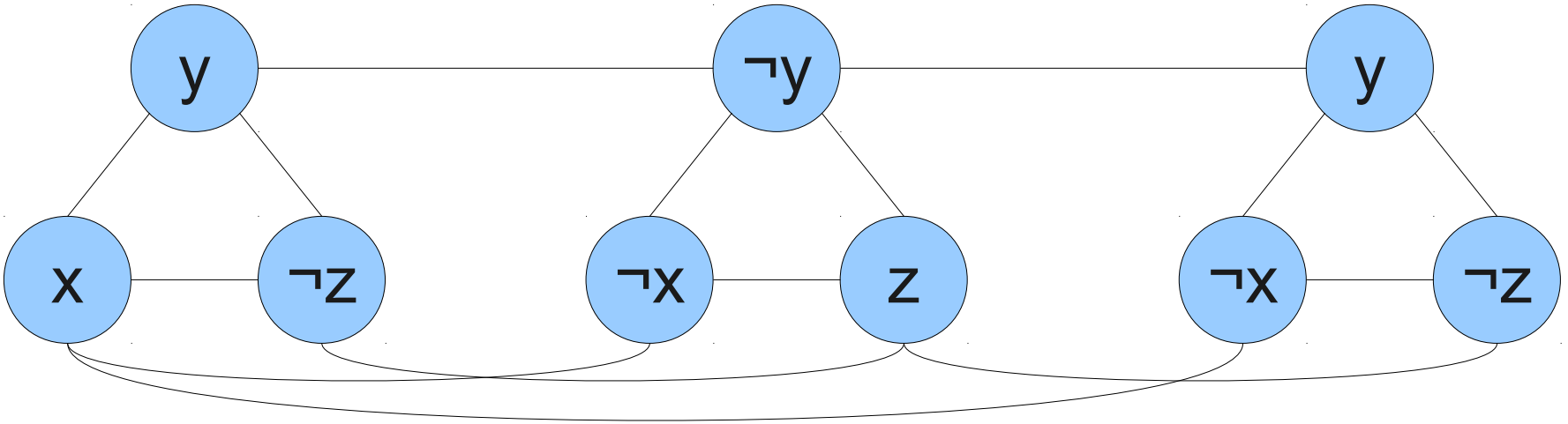
If this graph has an independent set of size three, the original formula is satisfiable.

From 3SAT to INDSET



From 3SAT to INDSET

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

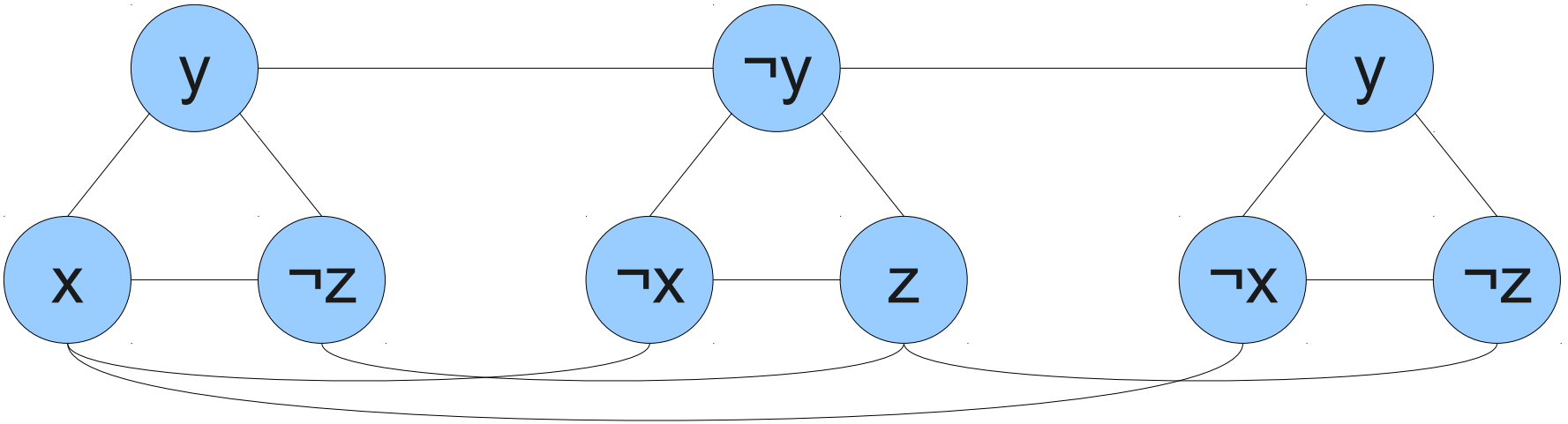


If the original formula is satisfiable,
this graph has an independent set of size three.

From 3SAT to INDSET

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

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

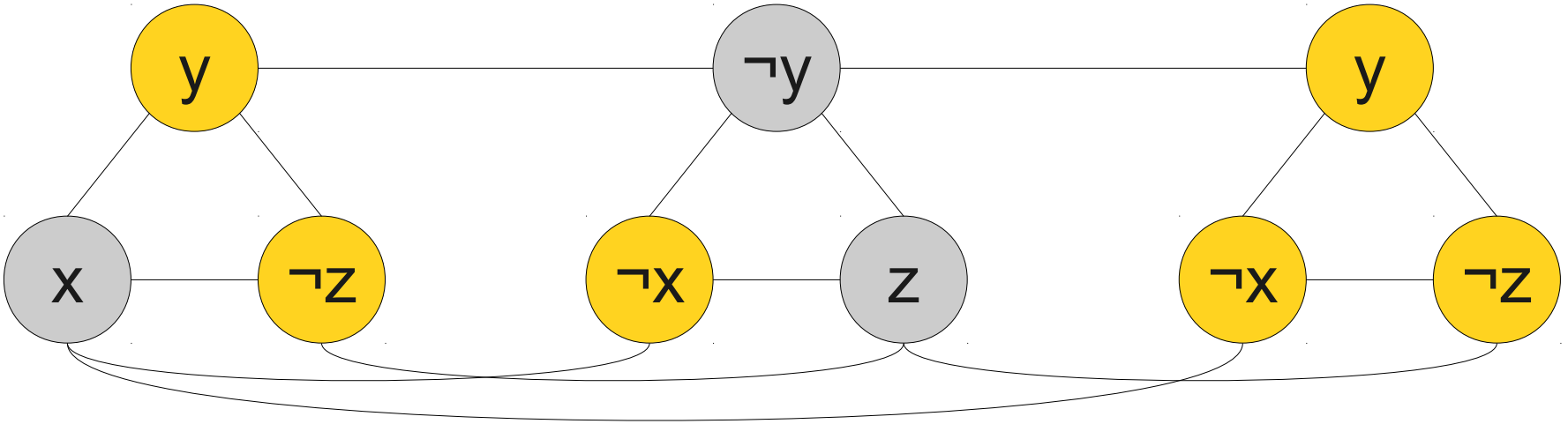


If the original formula is satisfiable,
this graph has an independent set of size three.

From 3SAT to INDSET

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

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

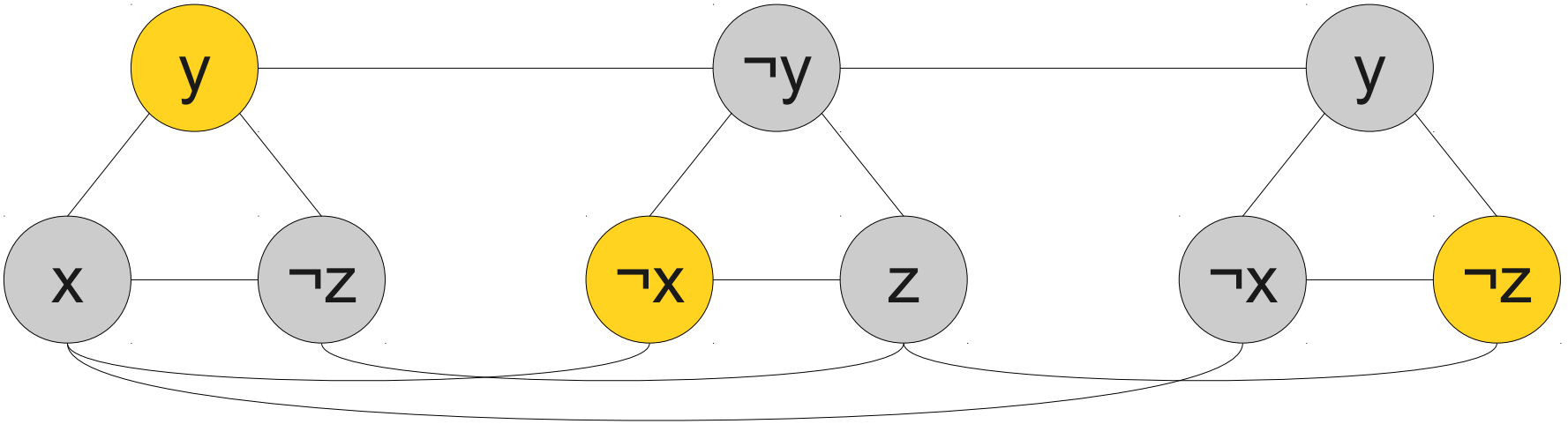


If the original formula is satisfiable,
this graph has an independent set of size three.

From 3SAT to INDSET

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

(x ∨ 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

- 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.

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, as required. ■

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 $\neg v$ 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 $\neg v$, 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 $\neg x$, then C contains $\neg x$, and since we set x to false, C is satisfied. Thus all clauses in φ are satisfied, so φ is satisfied by this assignment. ■

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 joined 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.

Proof: We know that INDSET ∈ **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, φ ∈ 3SAT iff ⟨G, n⟩ ∈ INDSET, and this reduction can be computed in polynomial time. Thus 3SAT is polynomial-time reducible to INDSET, so INDSET is **NP**-complete. ■

Structuring **NP**-Completeness Reductions

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.

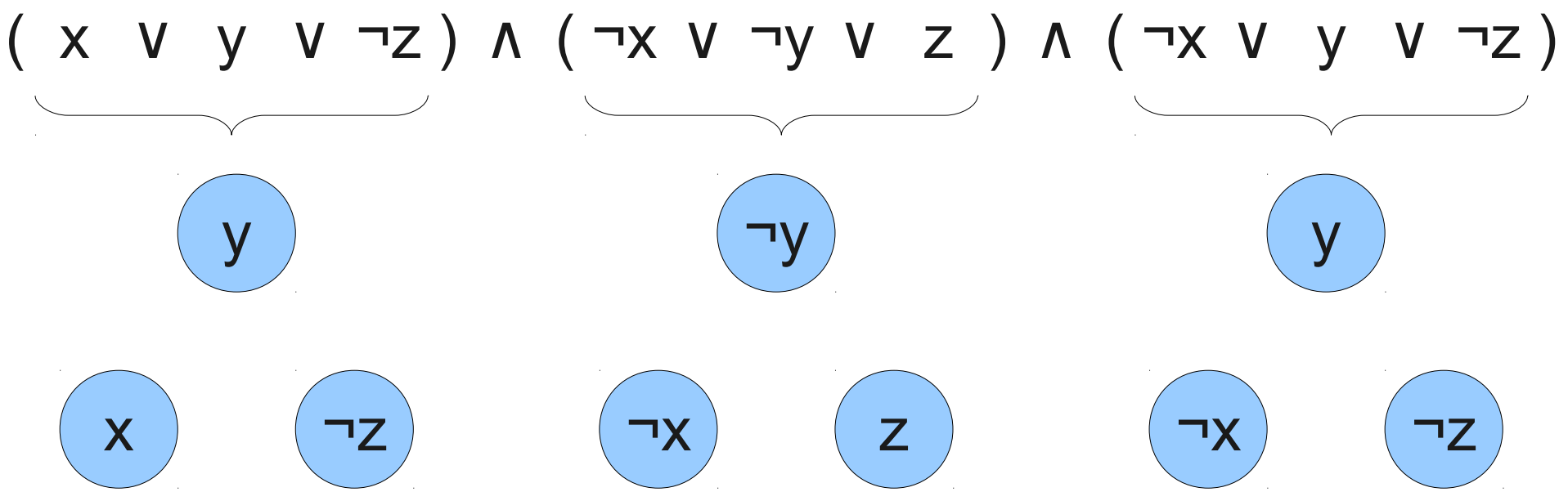
Gadgets in INDSET

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

Gadgets in INDSET

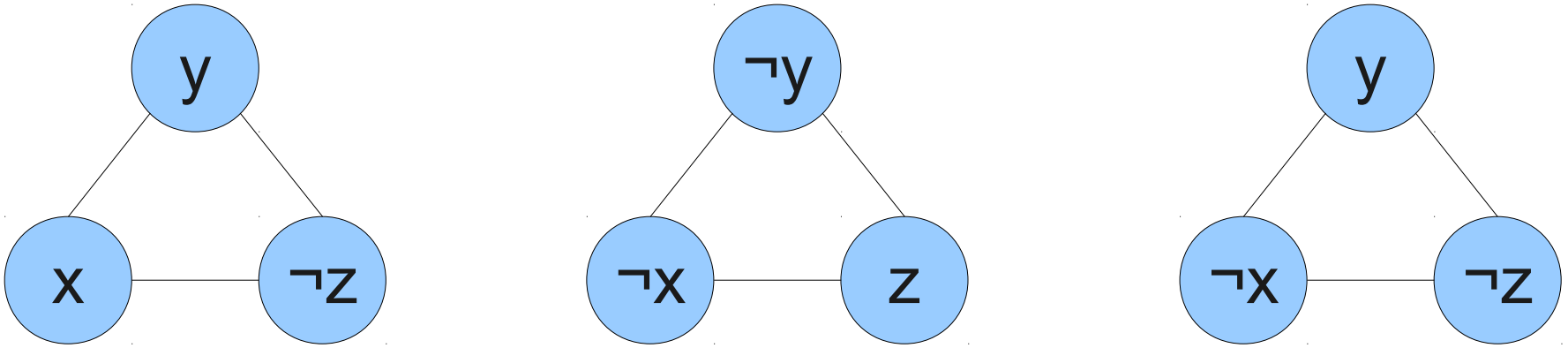
$$(\underbrace{x \vee y \vee \neg z}) \wedge (\underbrace{\neg x \vee \neg y \vee z}) \wedge (\underbrace{\neg x \vee y \vee \neg z})$$

Gadgets in INDSET



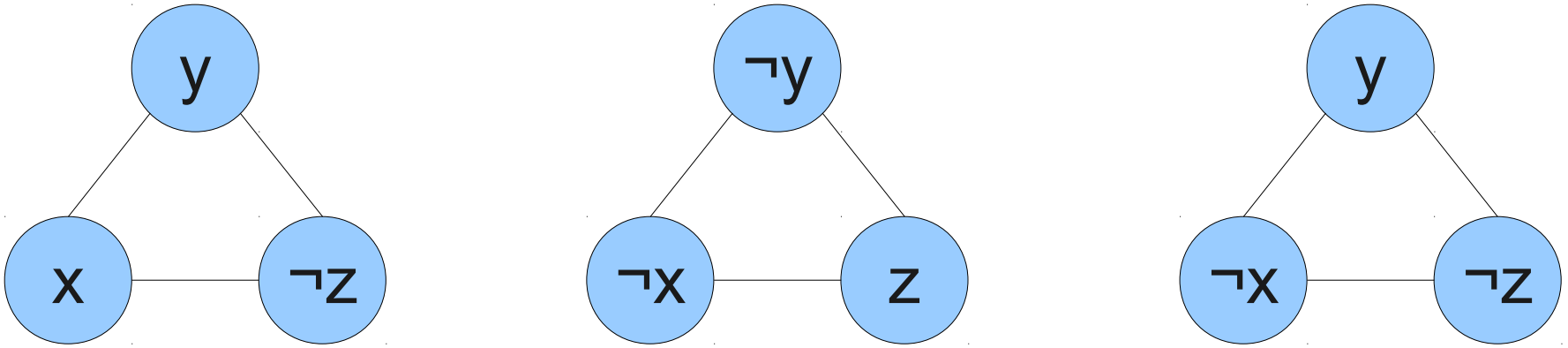
Gadgets in INDSET

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



Gadgets in INDSET

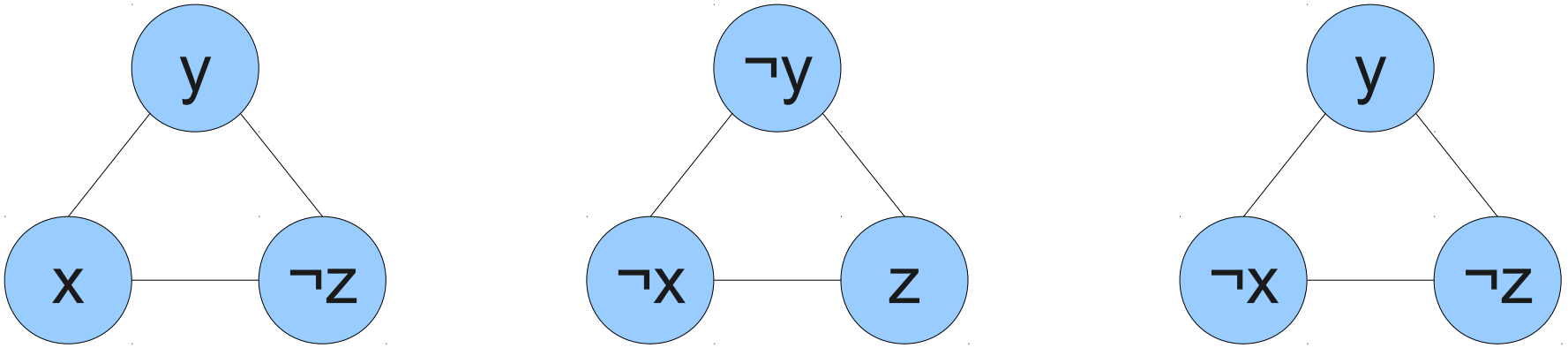
$$(x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee z) \wedge (\neg x \vee y \vee \neg z)$$



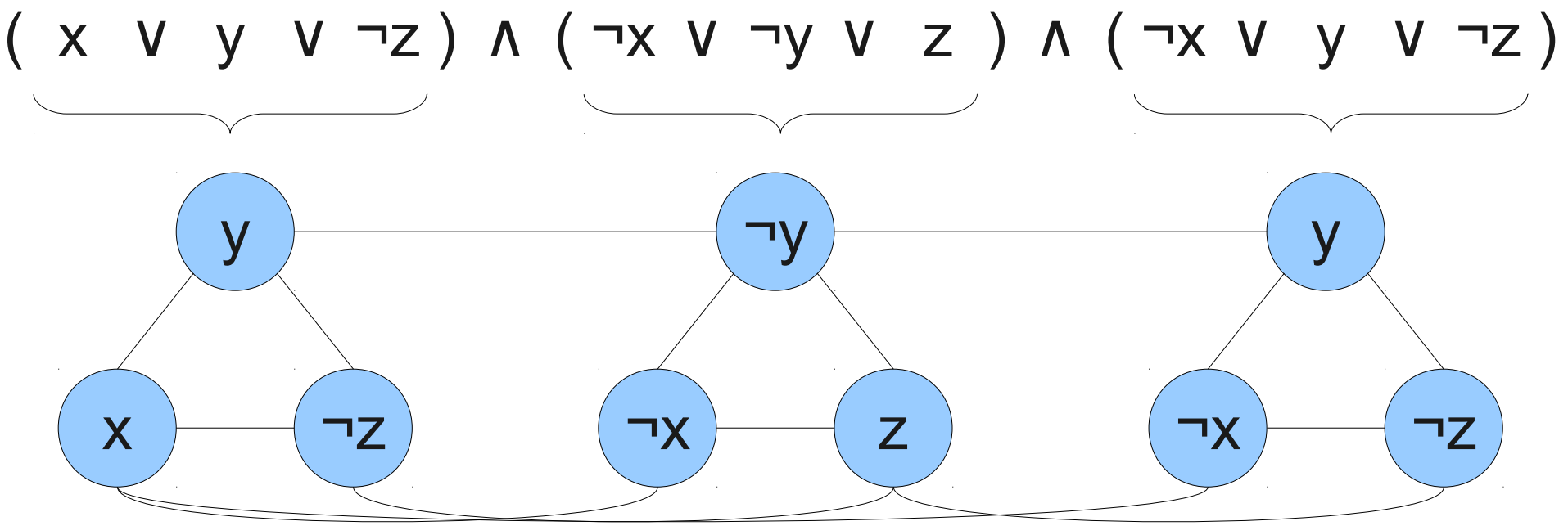
Each of these gadgets is designed to solve one part of the problem: ensuring each clause is satisfied.

Gadgets in INDSET

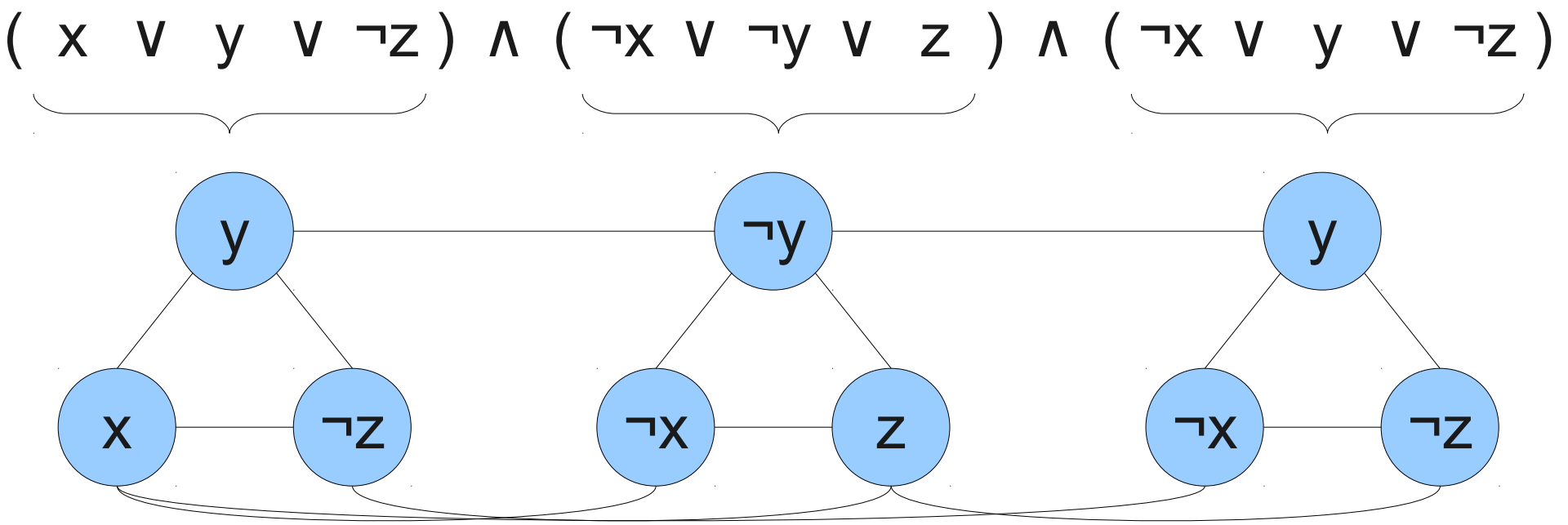
$$(\ x \ \vee \ y \ \vee \neg z\ ) \wedge (\ \neg x \vee \neg y \vee \ z\ ) \wedge (\ \neg x \vee \ y \ \vee \neg z\ )$$



Gadgets in INDSET

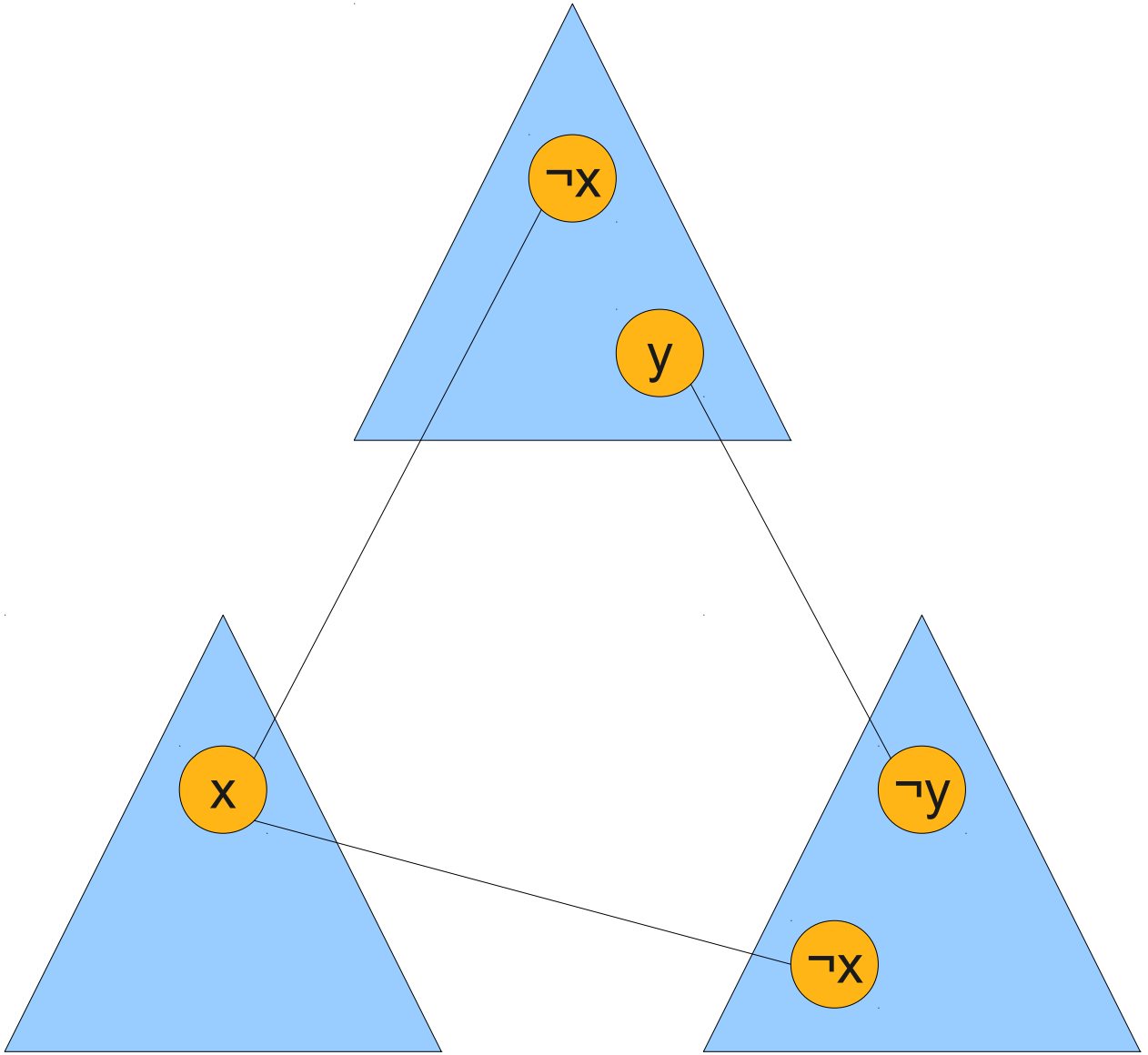


Gadgets in INDSET



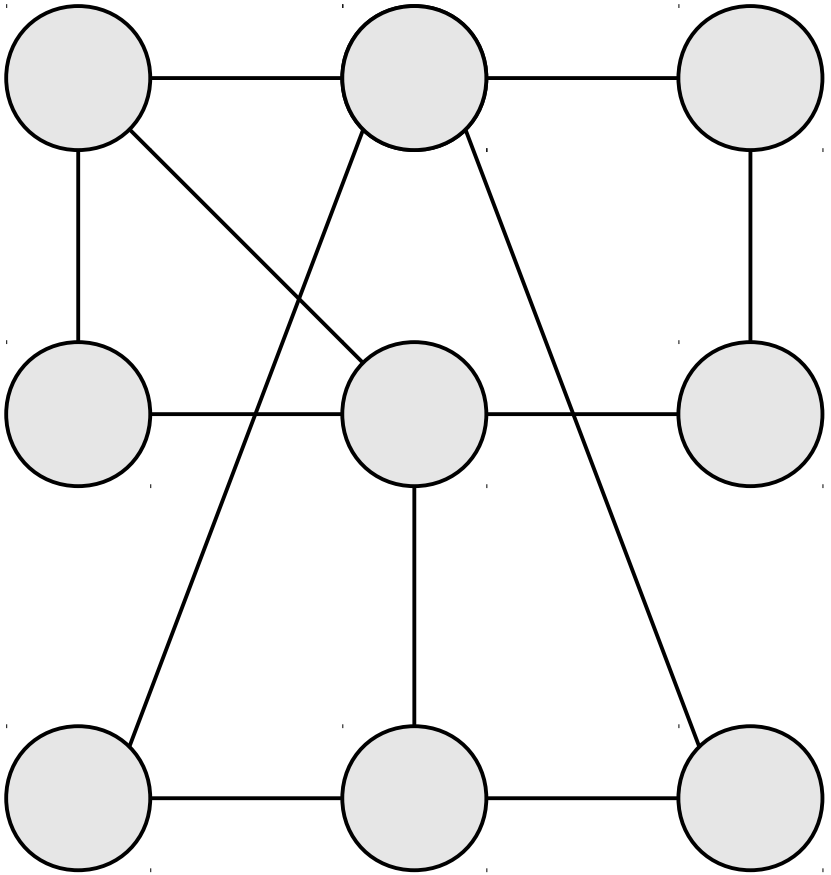
These connections ensure that the solutions to each gadget are linked to one another.

Gadgets in INDSET

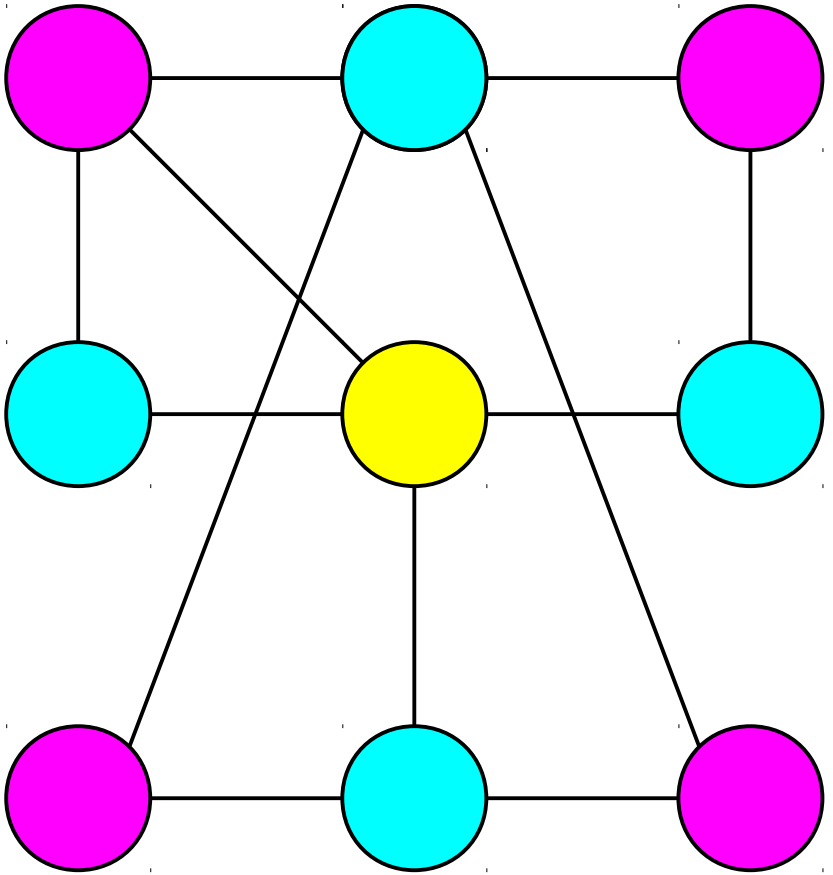


A More Complex Reduction

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 ∈ **NP**

- We can prove that 3COLOR ∈ **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.”

A Note on Terminology

- Although 3COLOR and 3SAT both have “3” in their names, the two are very different problems.
  - 3SAT means “there are three literals in every clause.” However, each literal can take on only one of two different values.
  - 3COLOR means “every node can take on one of three different colors.”
- **Key difference**:
  - In 3SAT variables have two choices of value.
  - In 3COLOR nodes have three choices of value.

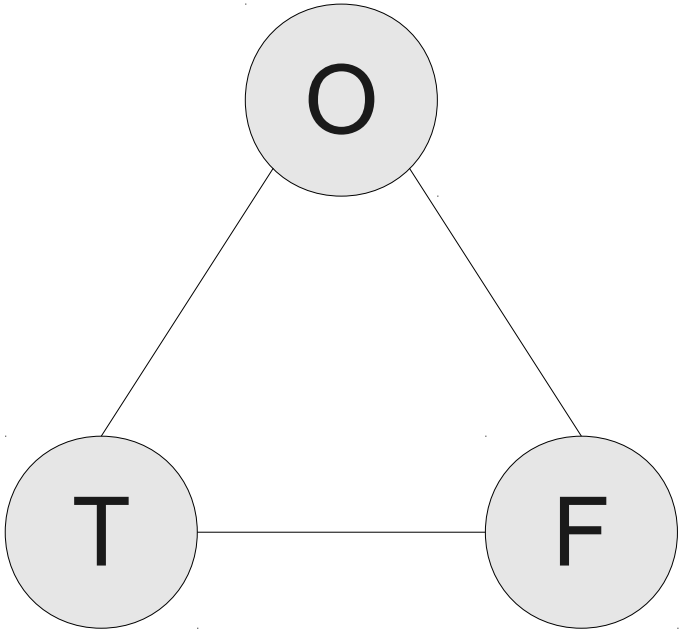
Why Not Two Colors?

- It would seem that 2COLOR (whether a graph has a 2-coloring) would be a better fit.
  - Every variable has one of two values.
  - Every node has one of two values.
- 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!

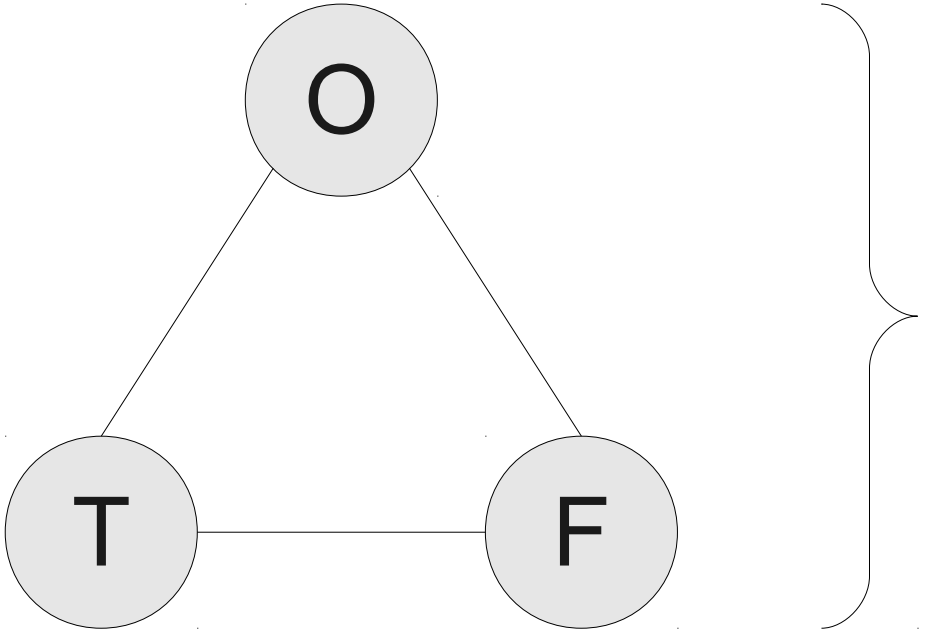
From 3SAT to 3COLOR

- 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.

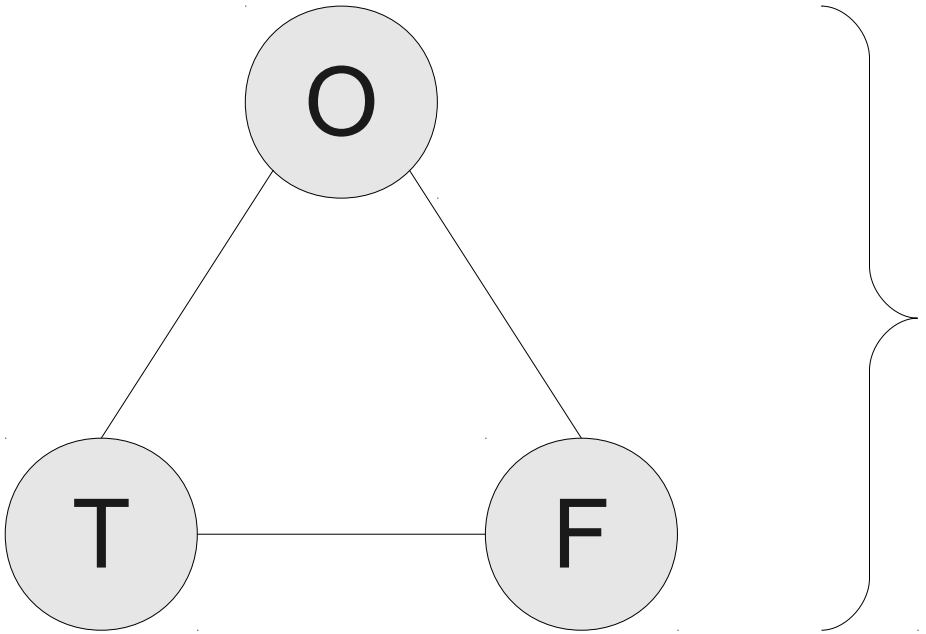
Gadget One: Assigning Meanings



Gadget One: Assigning Meanings

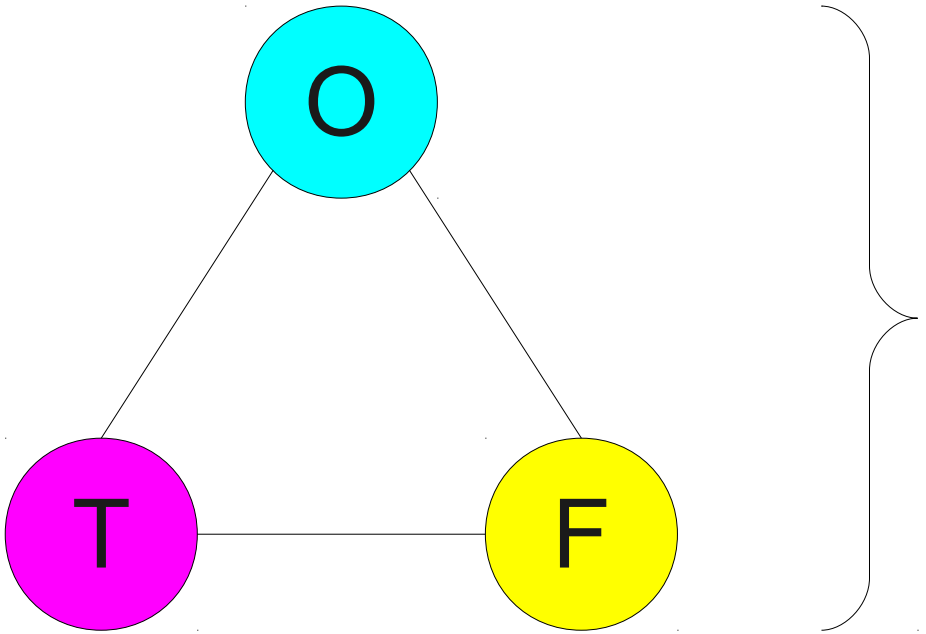


Gadget One: Assigning Meanings



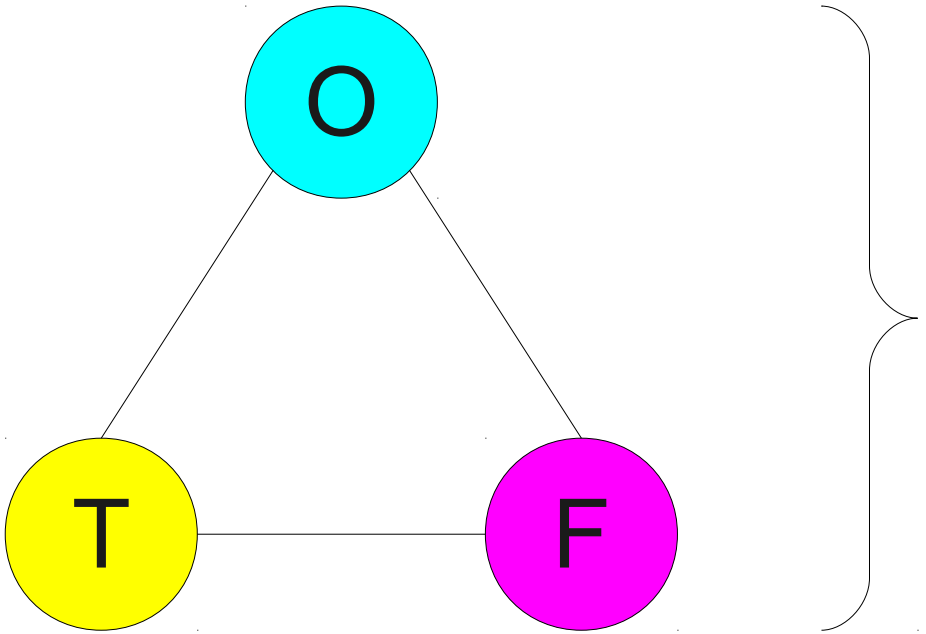
The color assigned to T will be interpreted as "true."
The color assigned to F will be interpreted as "false."
We do not associate any special meaning with O.

Gadget One: Assigning Meanings



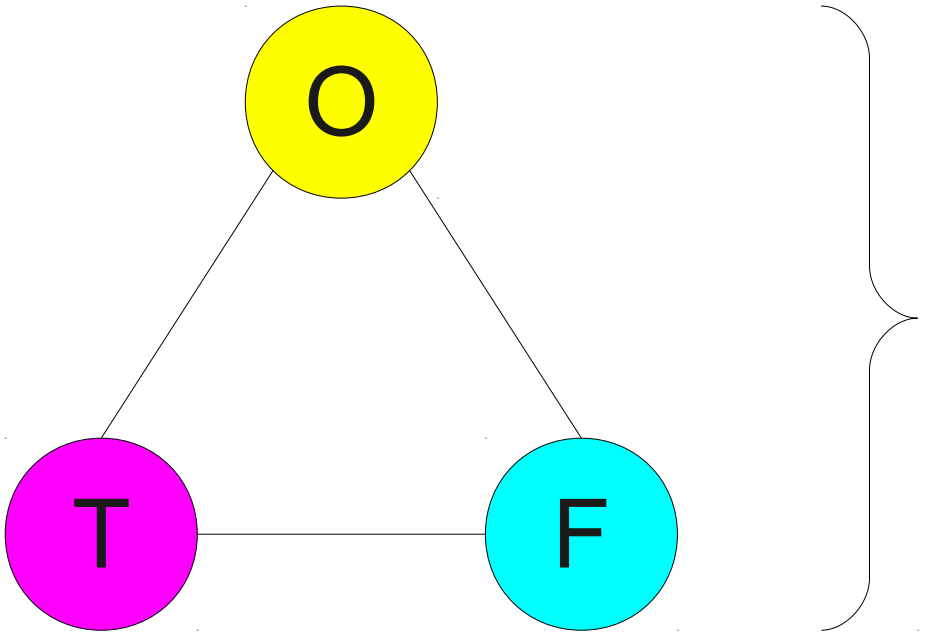
The color assigned to T will be interpreted as "true."
The color assigned to F will be interpreted as "false."
We do not associate any special meaning with O.

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 0.

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 0.

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

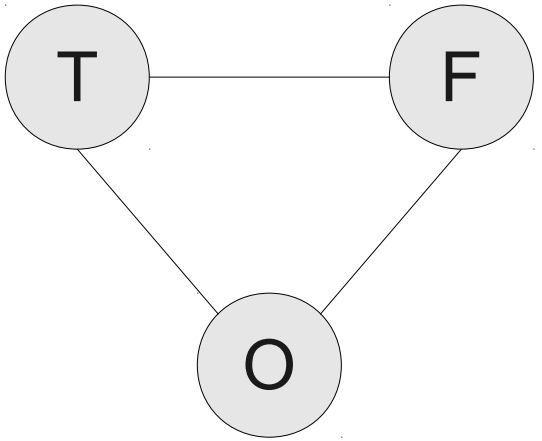
(x ∨ y ∨ ¬z) ∧ (¬x ∨ ¬y ∨ z) ∧ (¬x ∨ y ∨ ¬z)

Gadget Two: Forcing a Choice

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

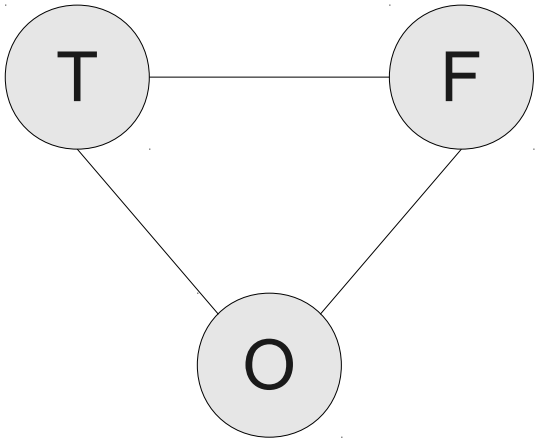
Gadget Two: Forcing a Choice

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



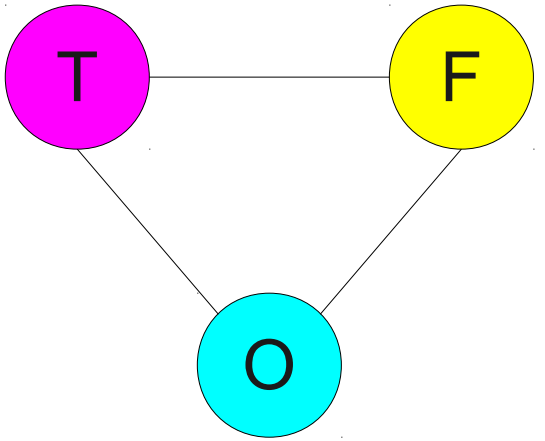
Gadget Two: Forcing a Choice

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



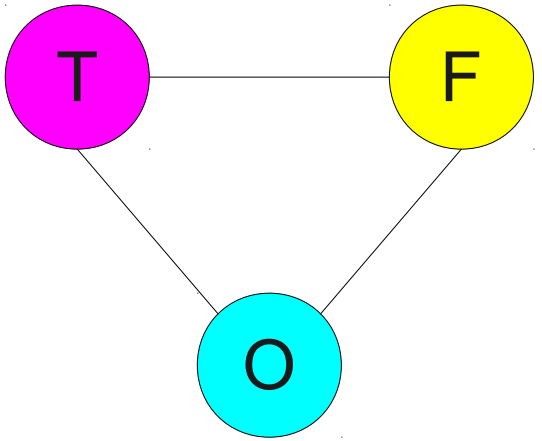
Gadget Two: Forcing a Choice

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



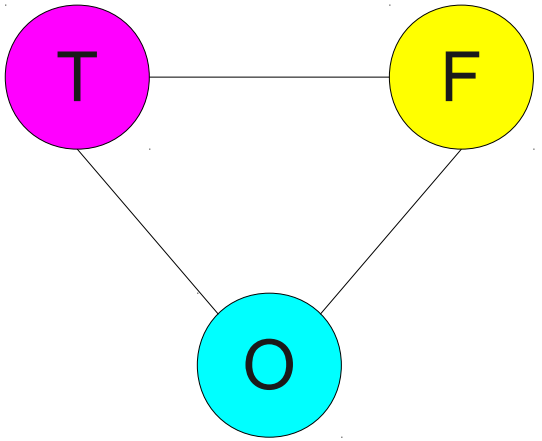
Gadget Two: Forcing a Choice

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



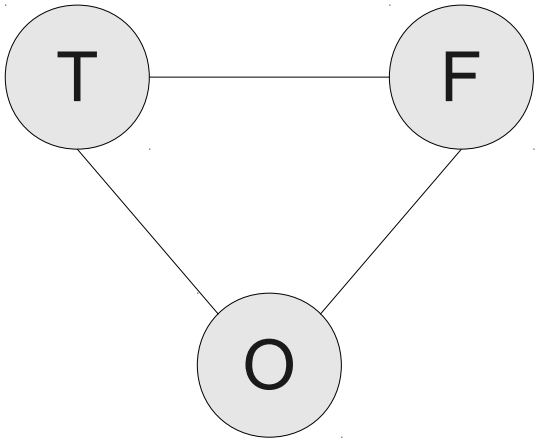
Gadget Two: Forcing a Choice

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



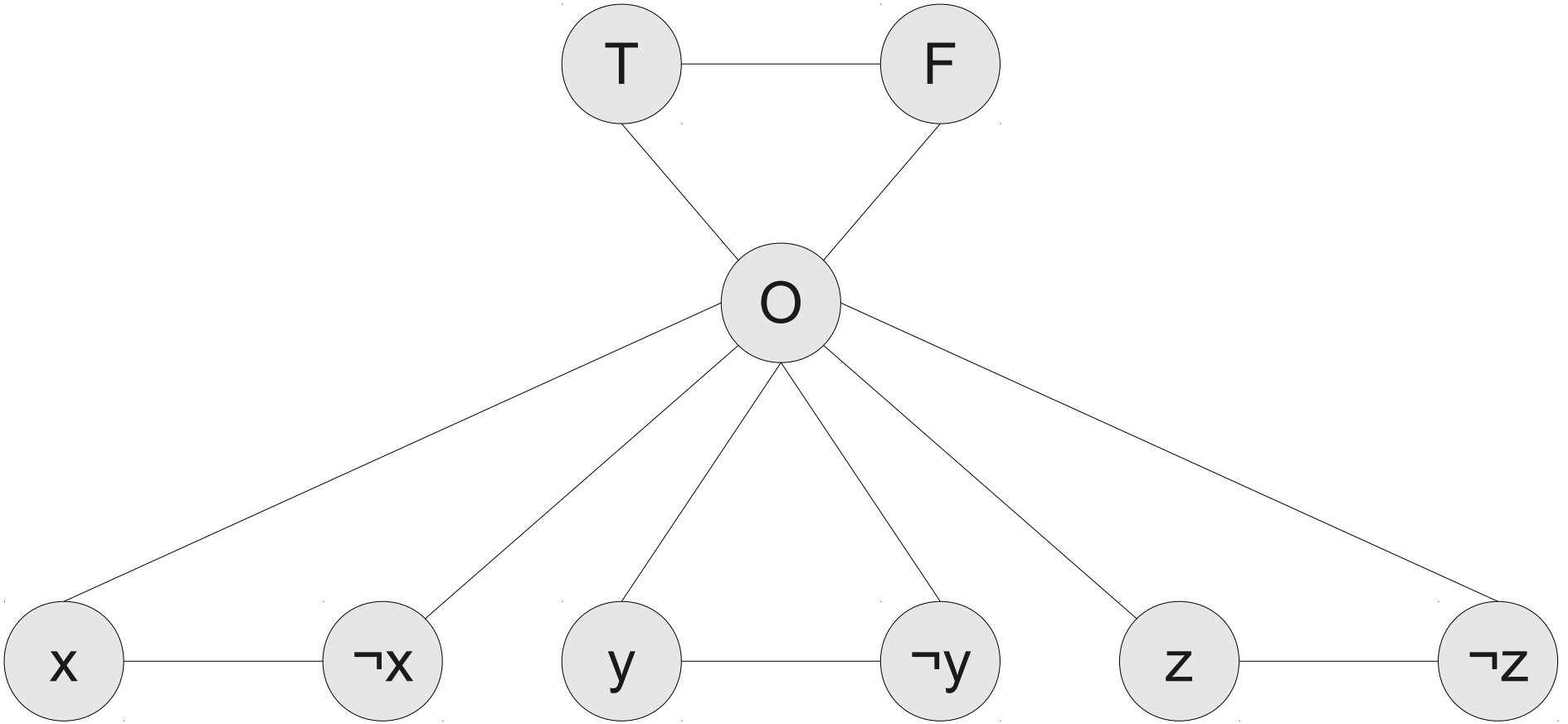
Gadget Two: Forcing a Choice

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



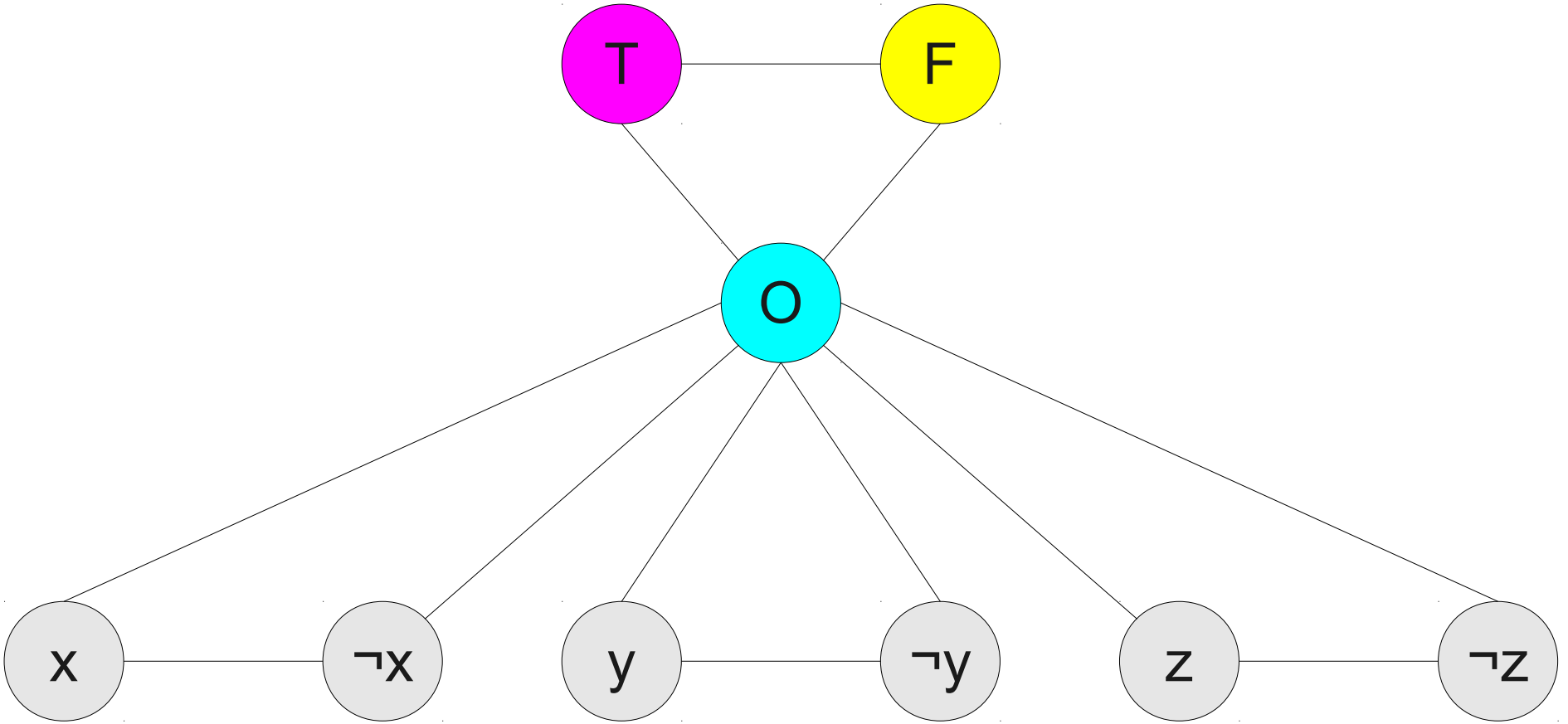
Gadget Two: Forcing a Choice

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



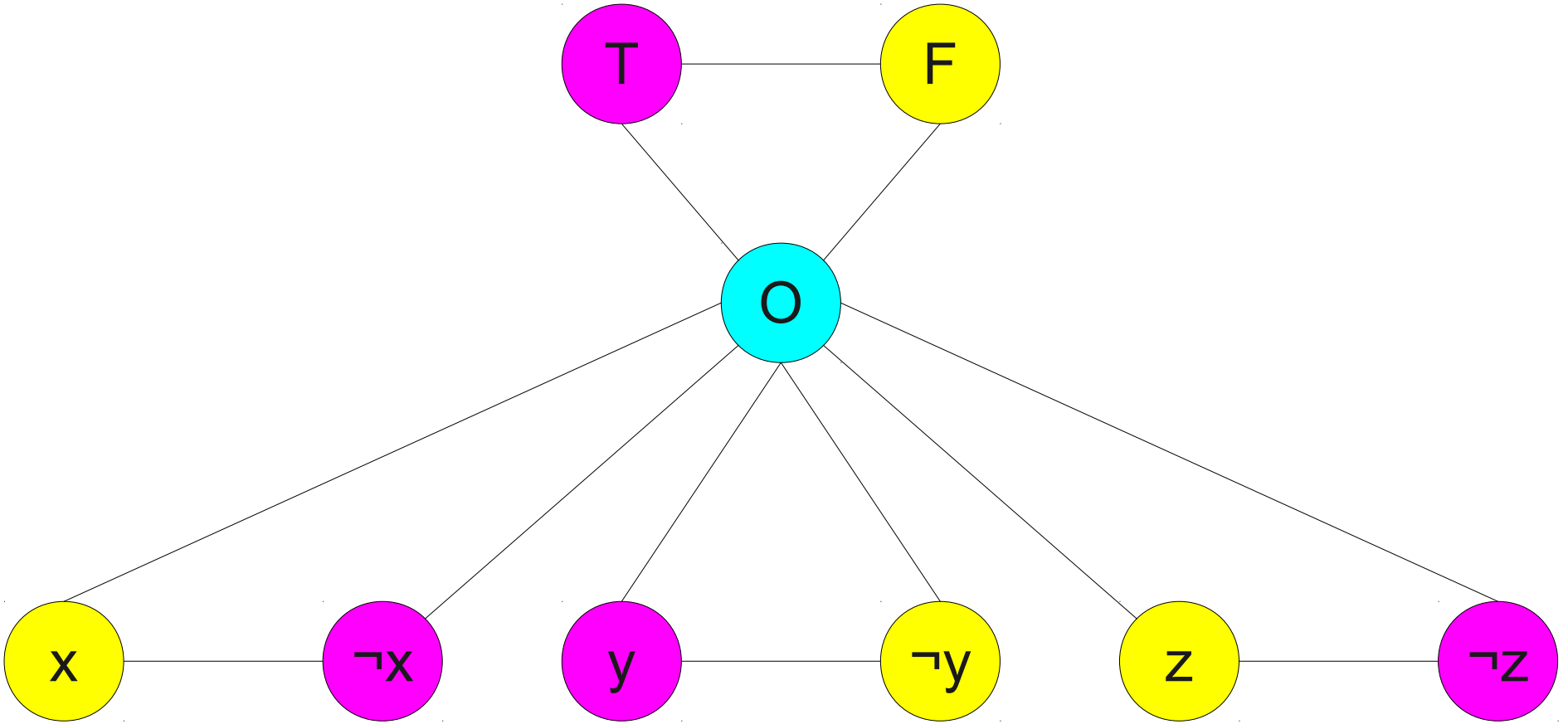
Gadget Two: Forcing a Choice

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



Gadget Two: Forcing a Choice

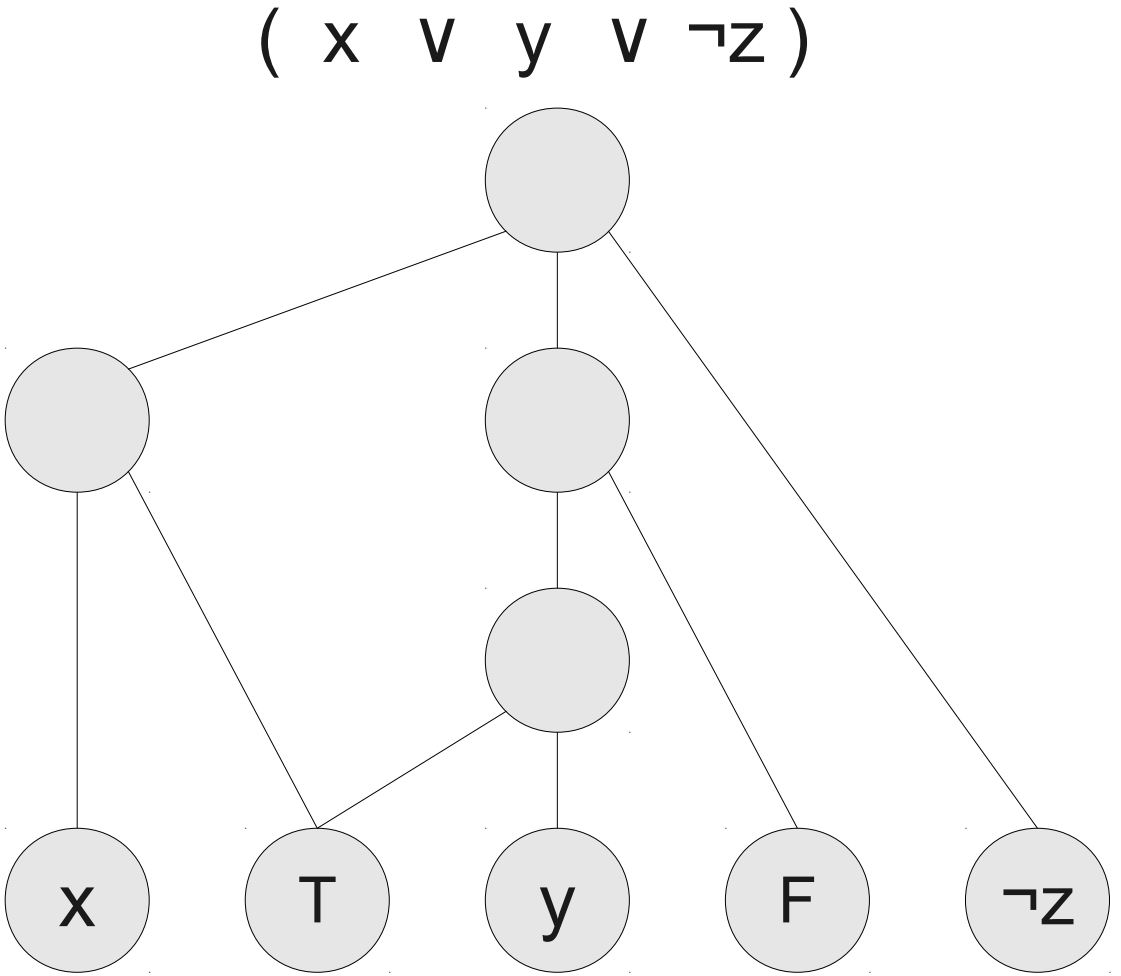
(x ∨ y ∨ ¬z) ∧ (¬x ∨ ¬y ∨ z) ∧ (¬x ∨ y ∨ ¬z)



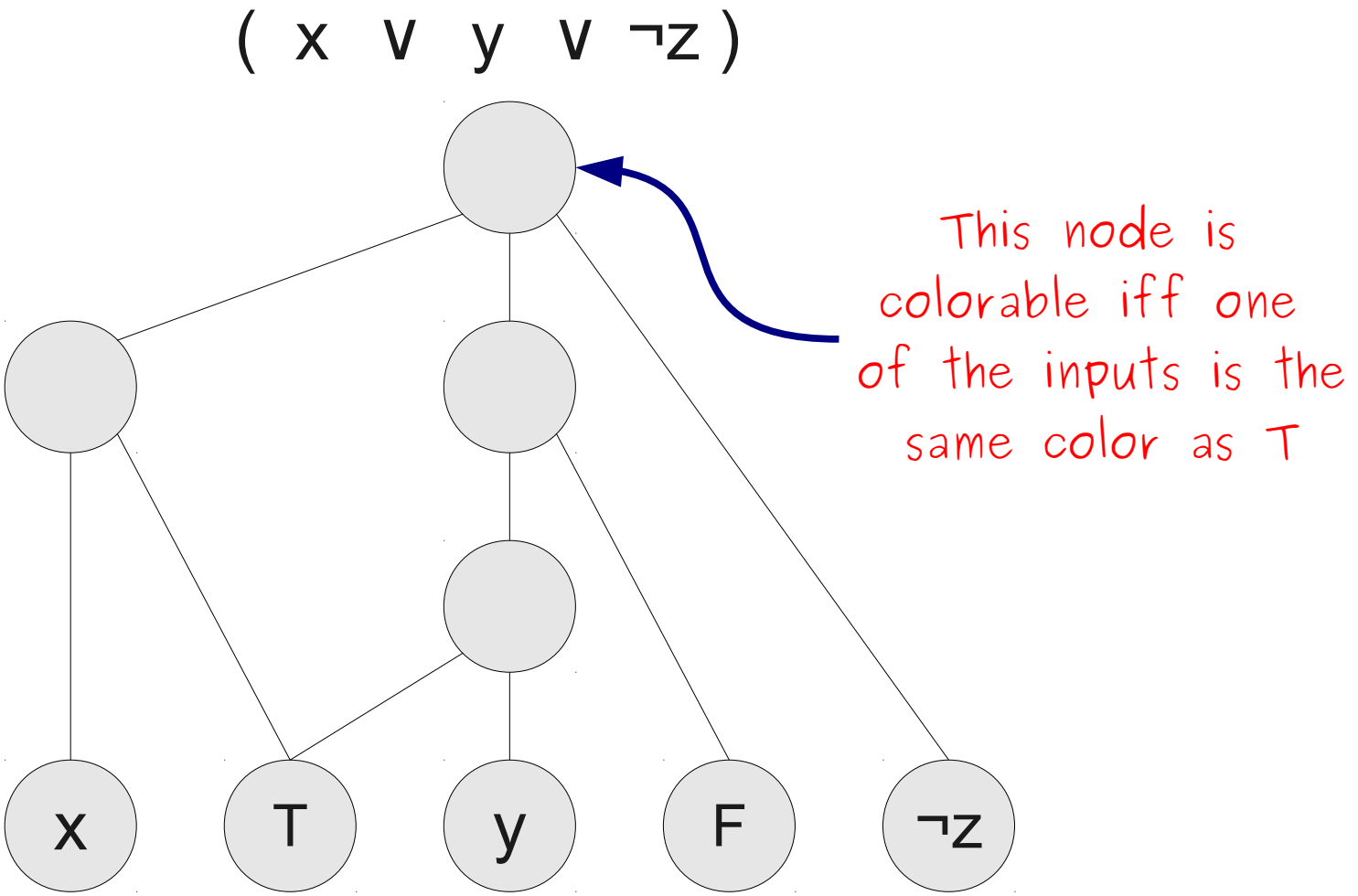
Gadget Three: Clause Satisfiability

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

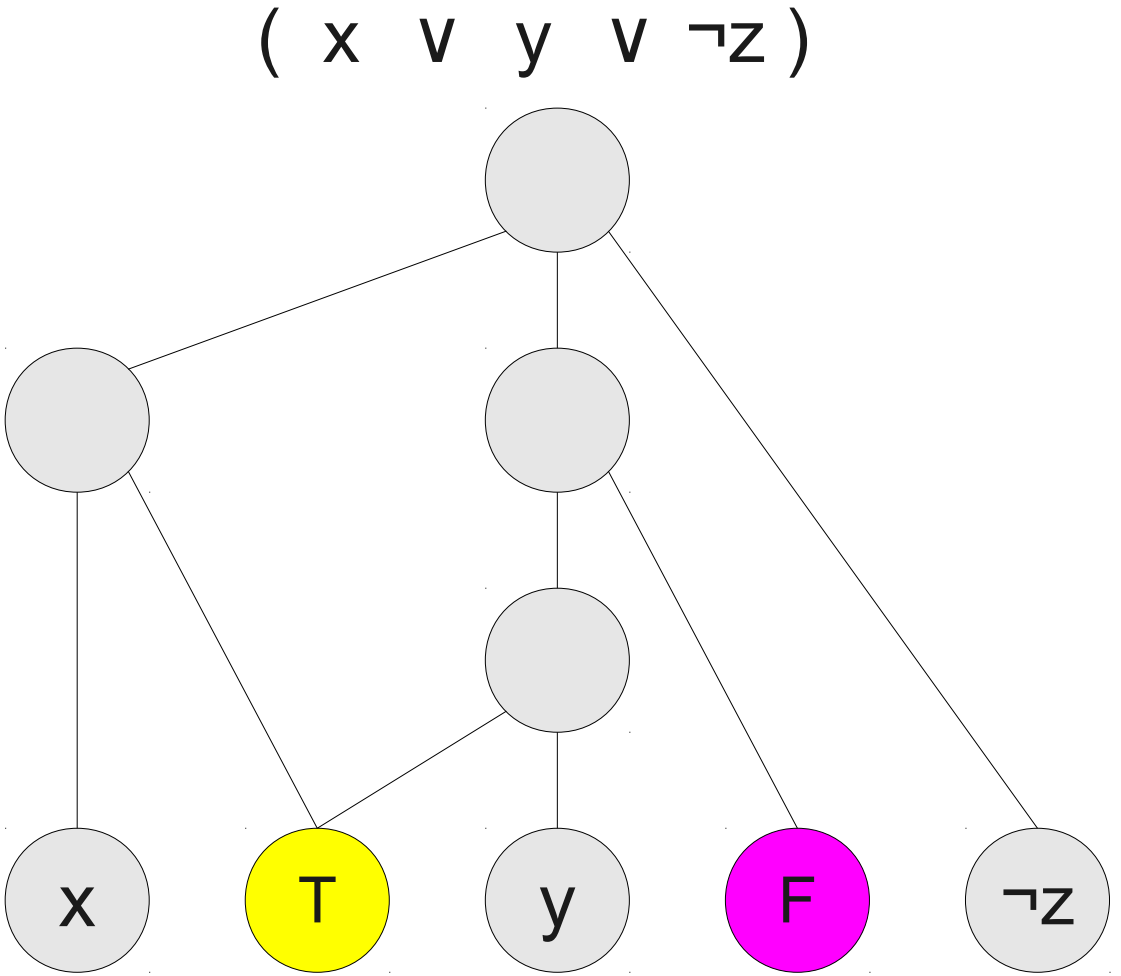
Gadget Three: Clause Satisfiability



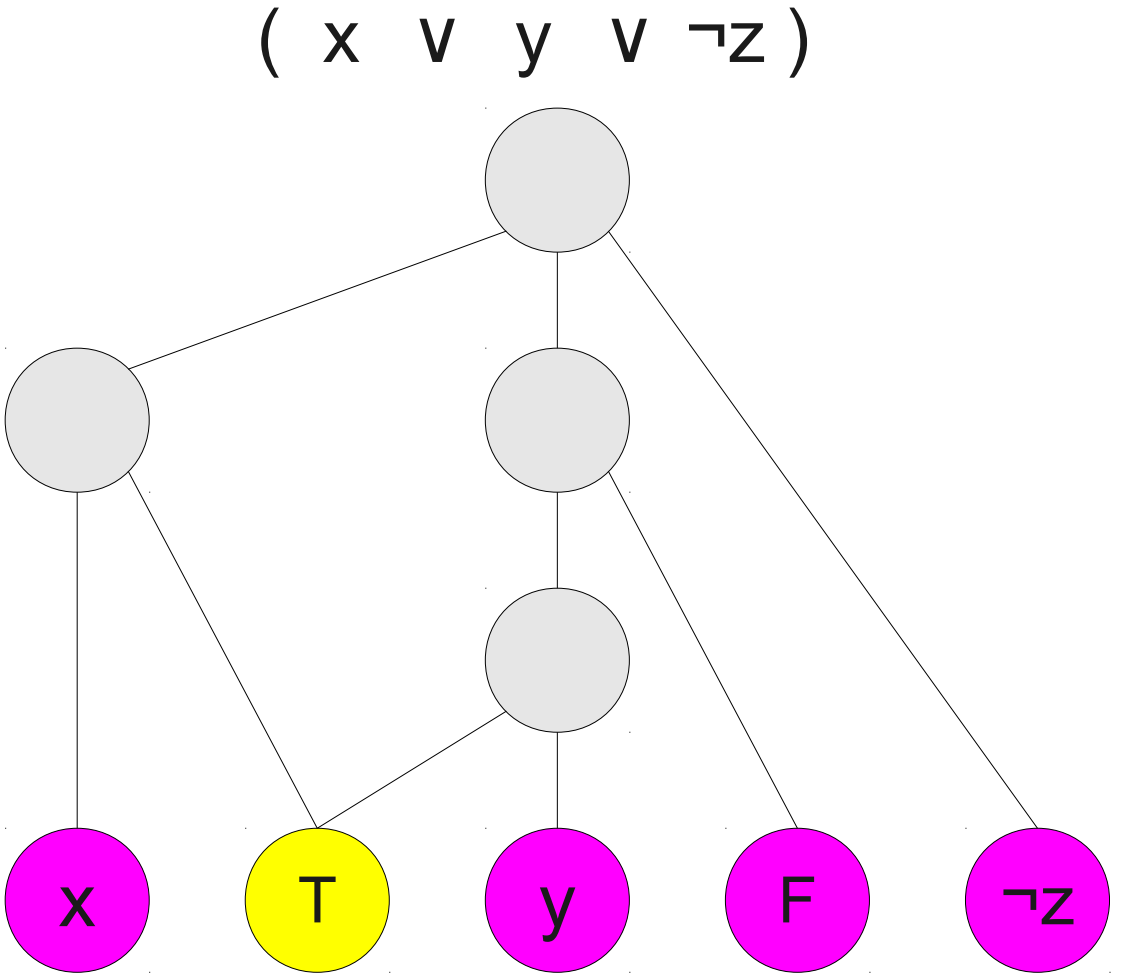
Gadget Three: Clause Satisfiability



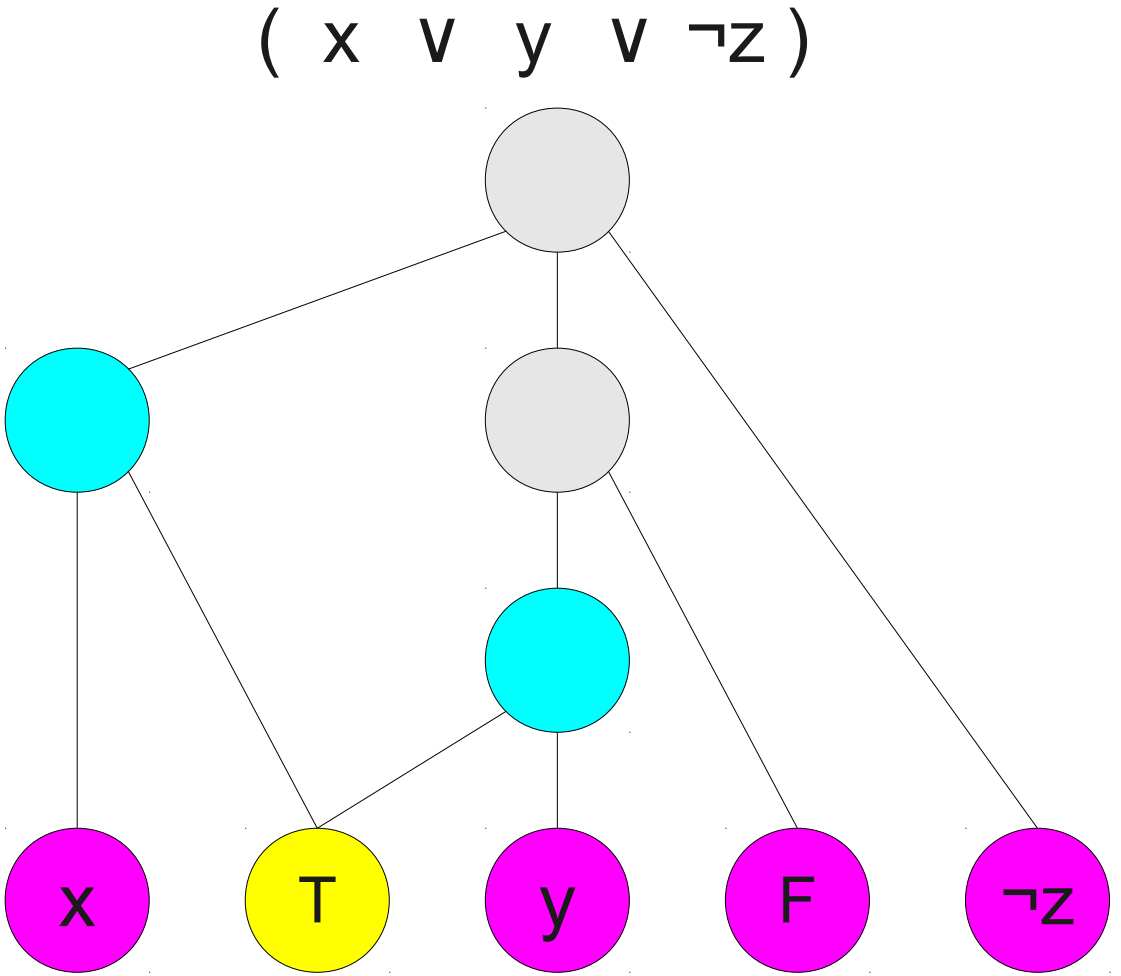
Gadget Three: Clause Satisfiability



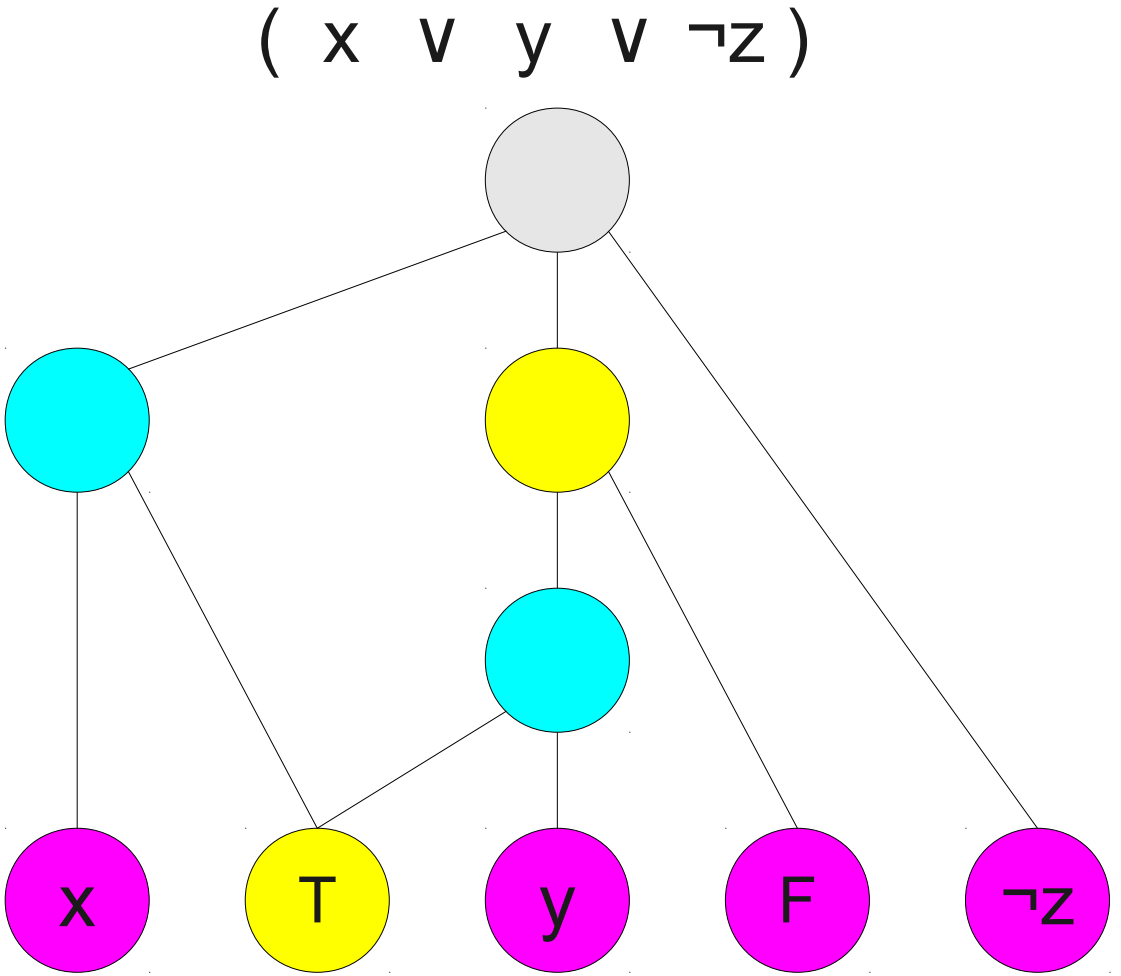
Gadget Three: Clause Satisfiability



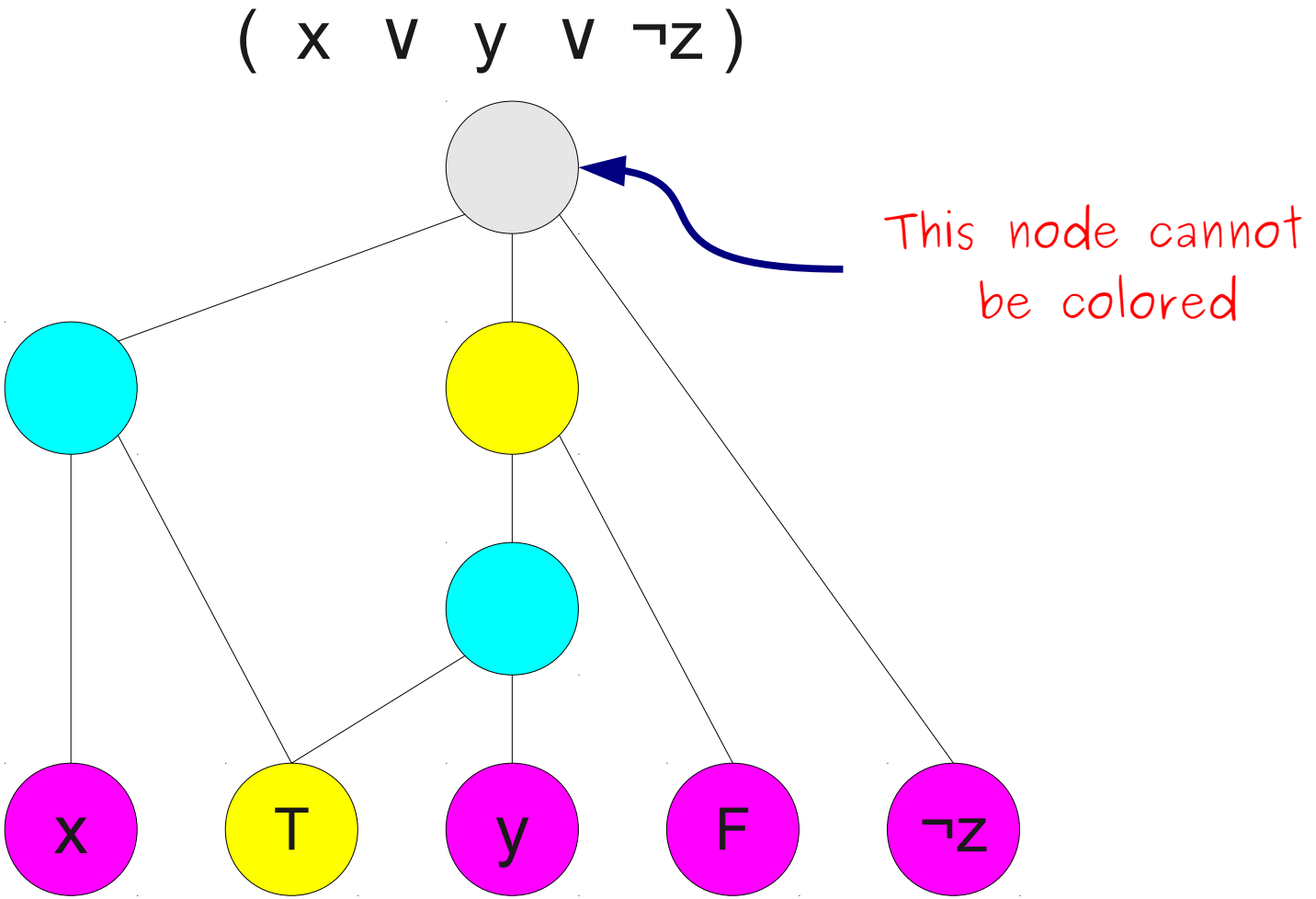
Gadget Three: Clause Satisfiability



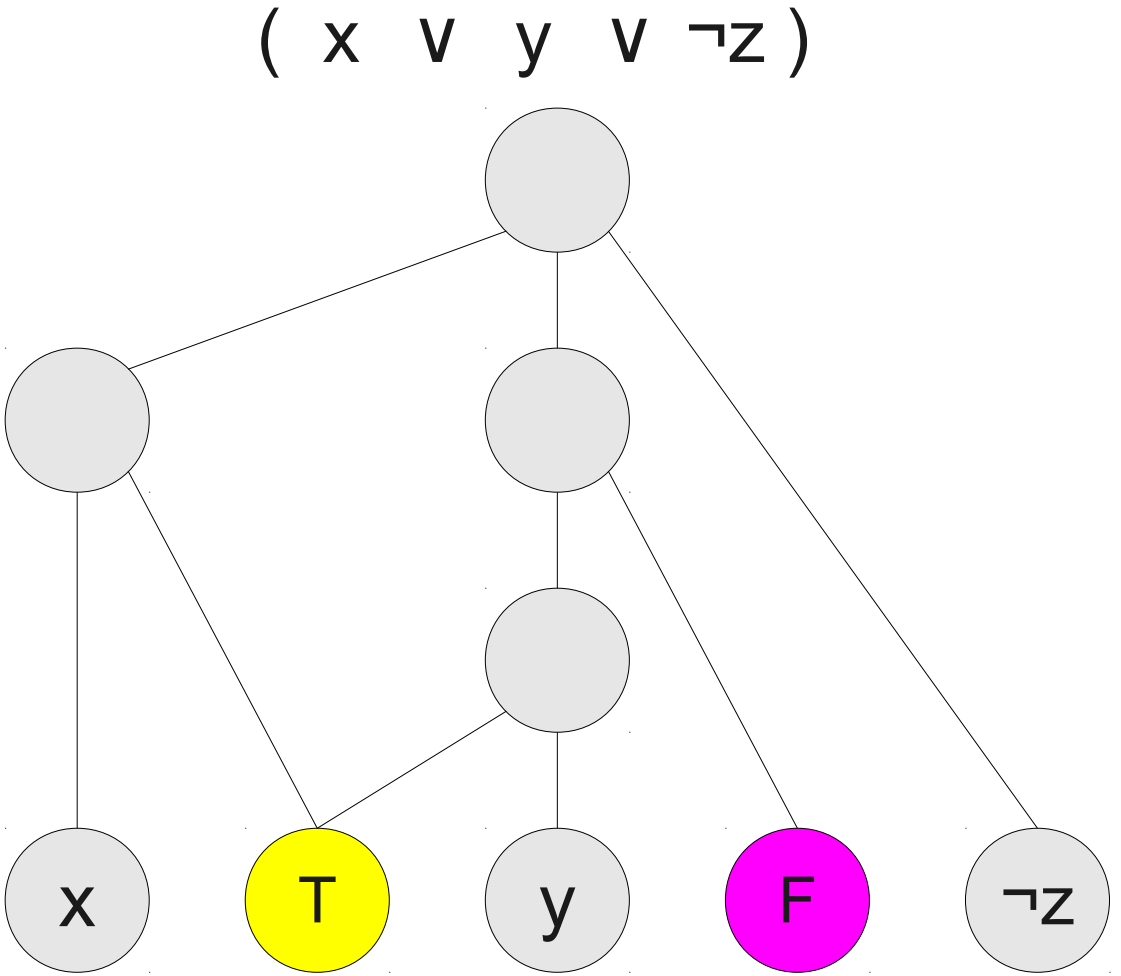
Gadget Three: Clause Satisfiability



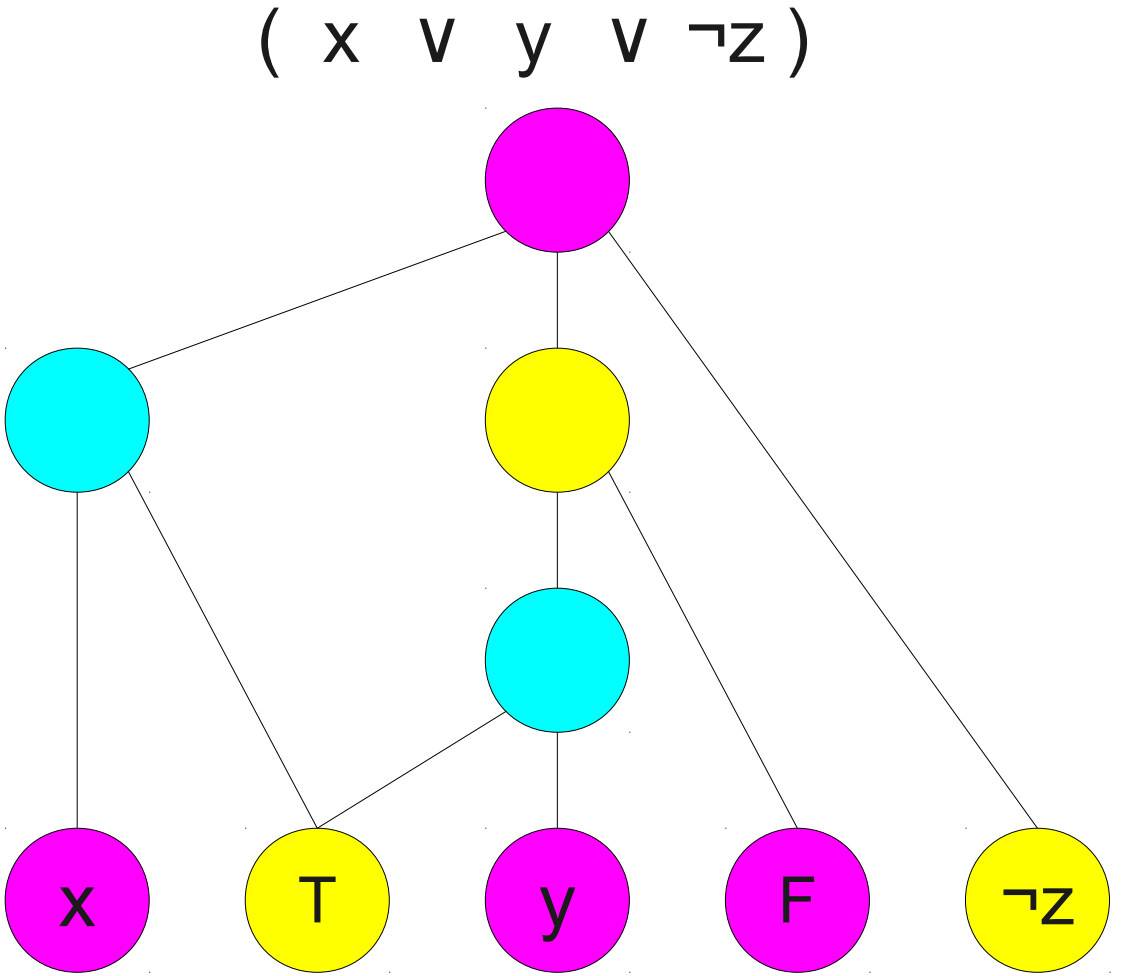
Gadget Three: Clause Satisfiability



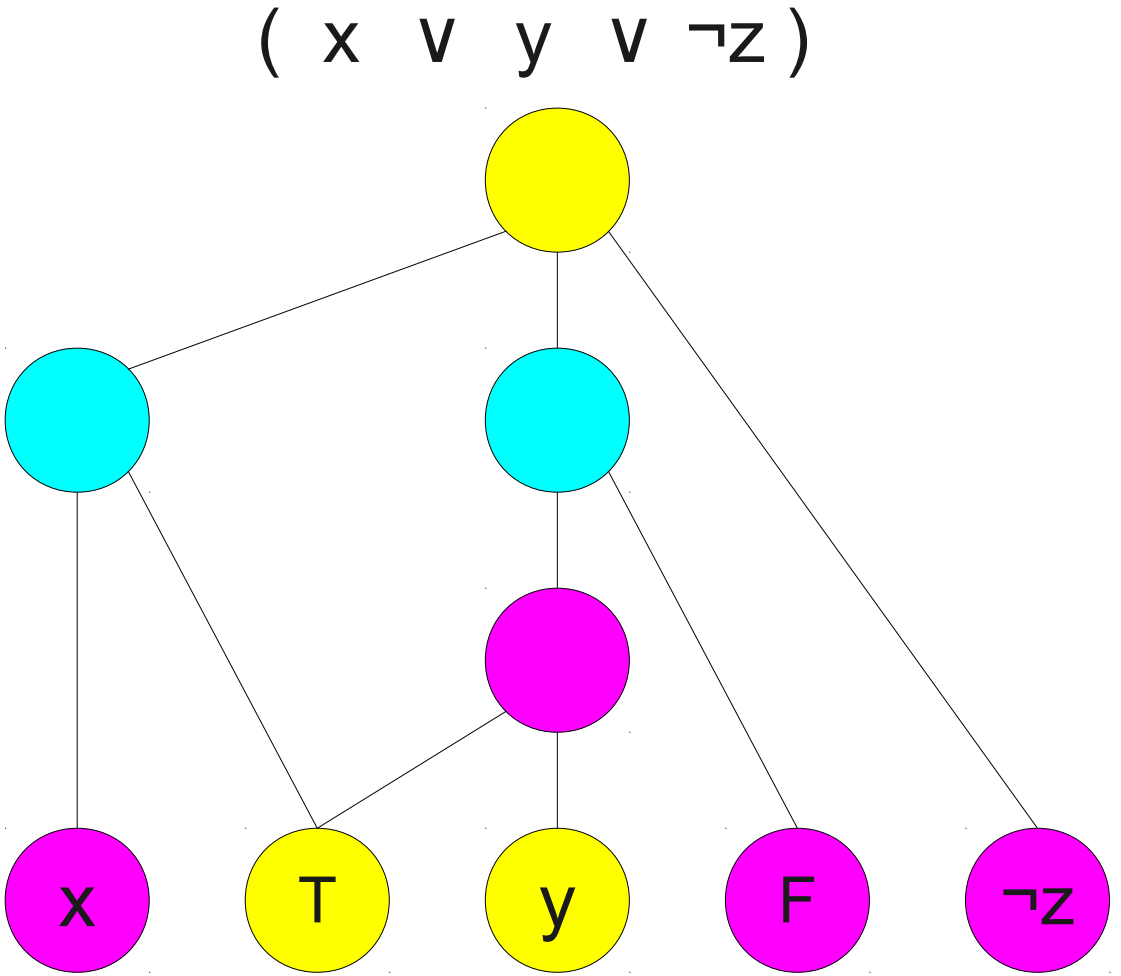
Gadget Three: Clause Satisfiability



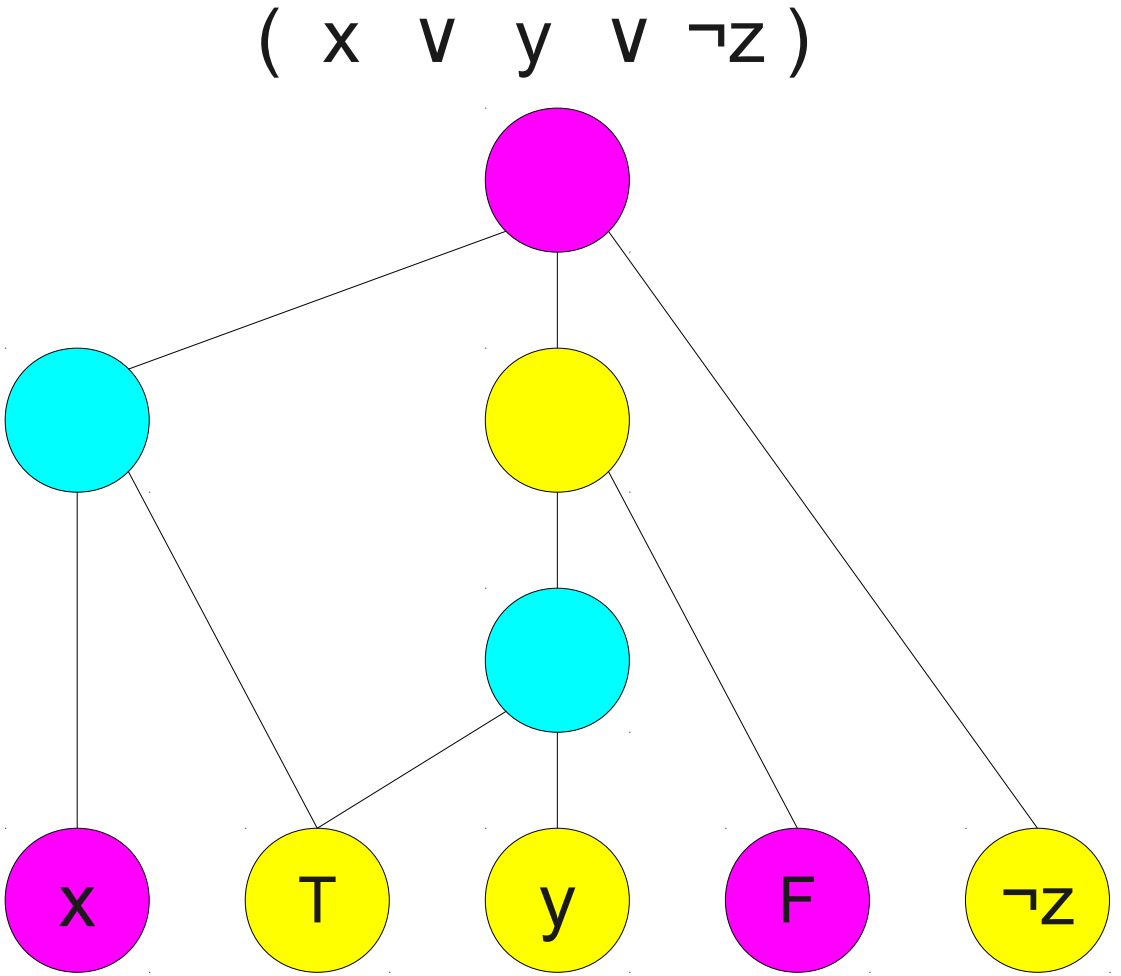
Gadget Three: Clause Satisfiability



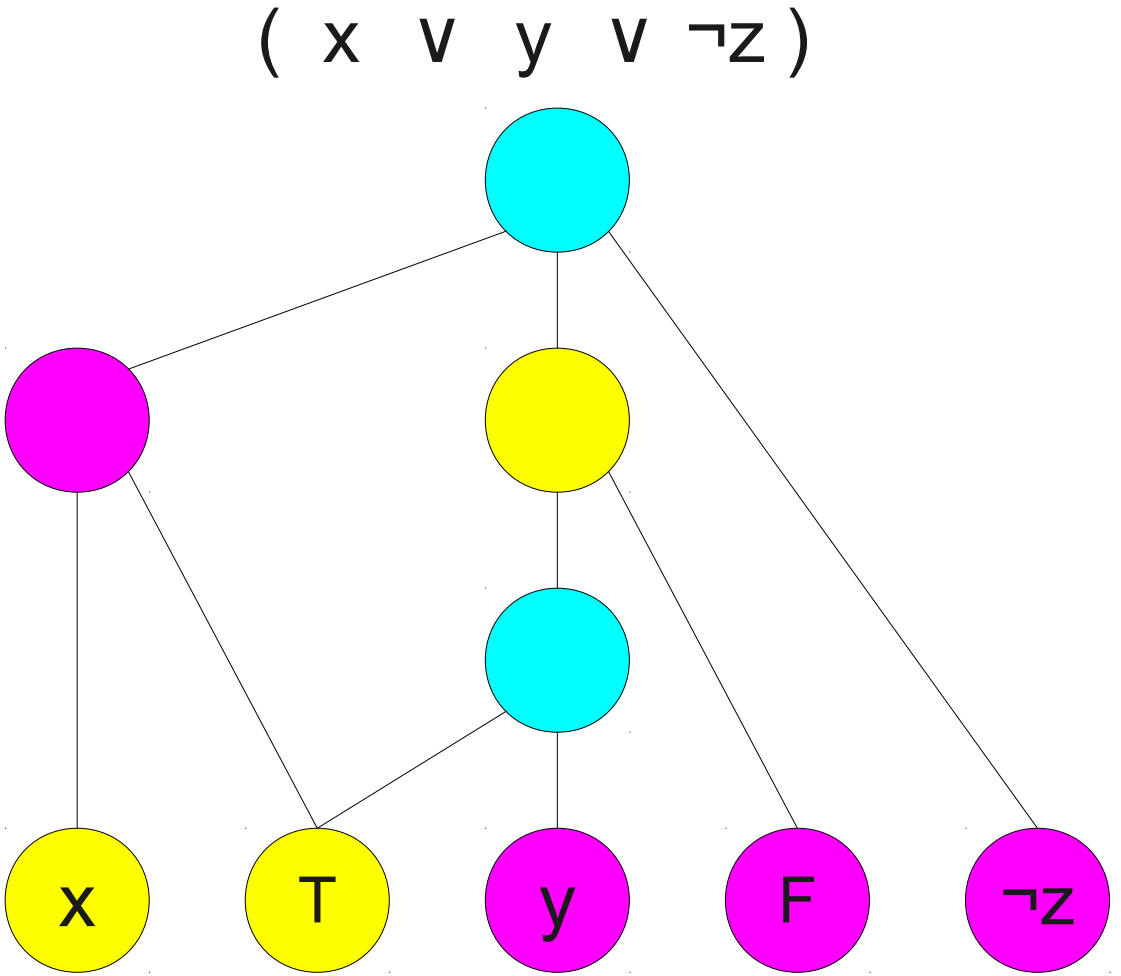
Gadget Three: Clause Satisfiability



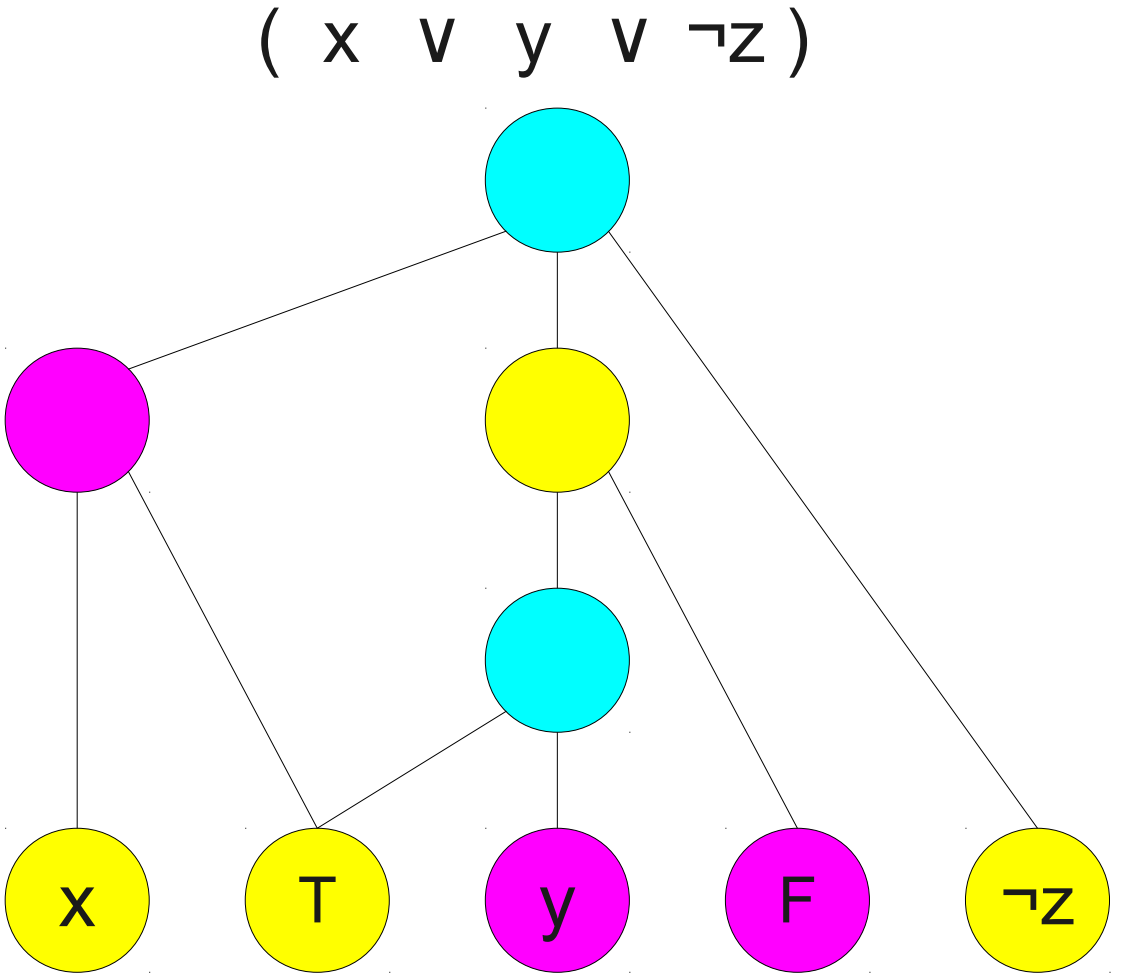
Gadget Three: Clause Satisfiability



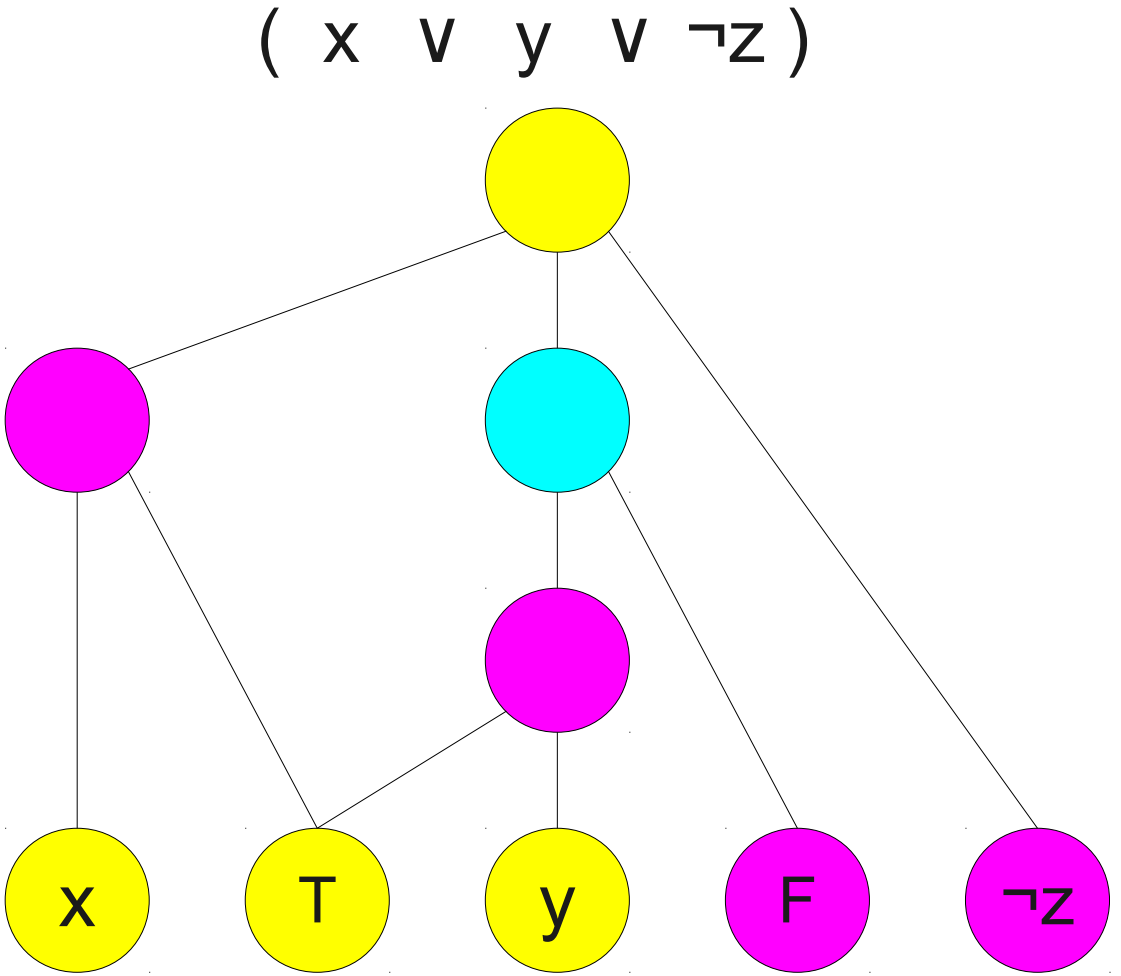
Gadget Three: Clause Satisfiability



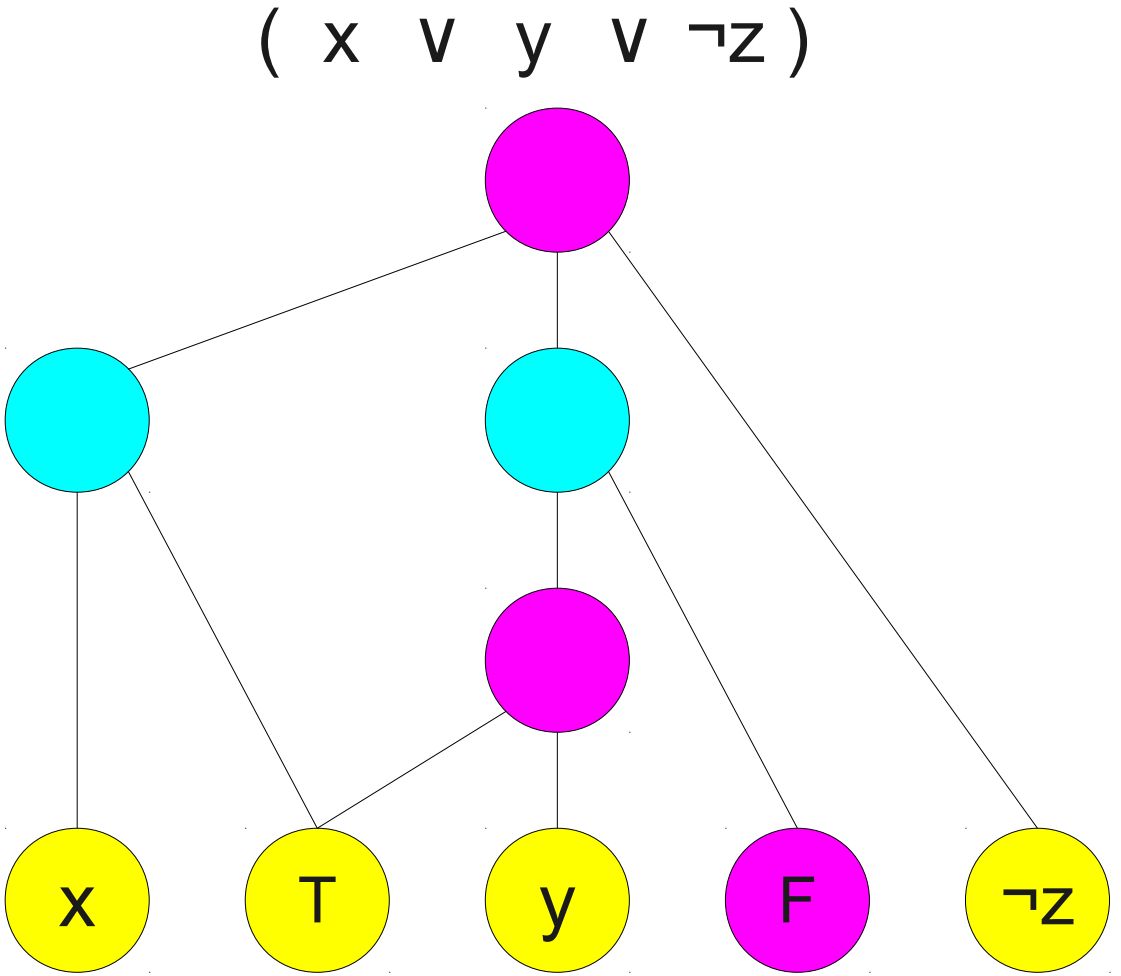
Gadget Three: Clause Satisfiability



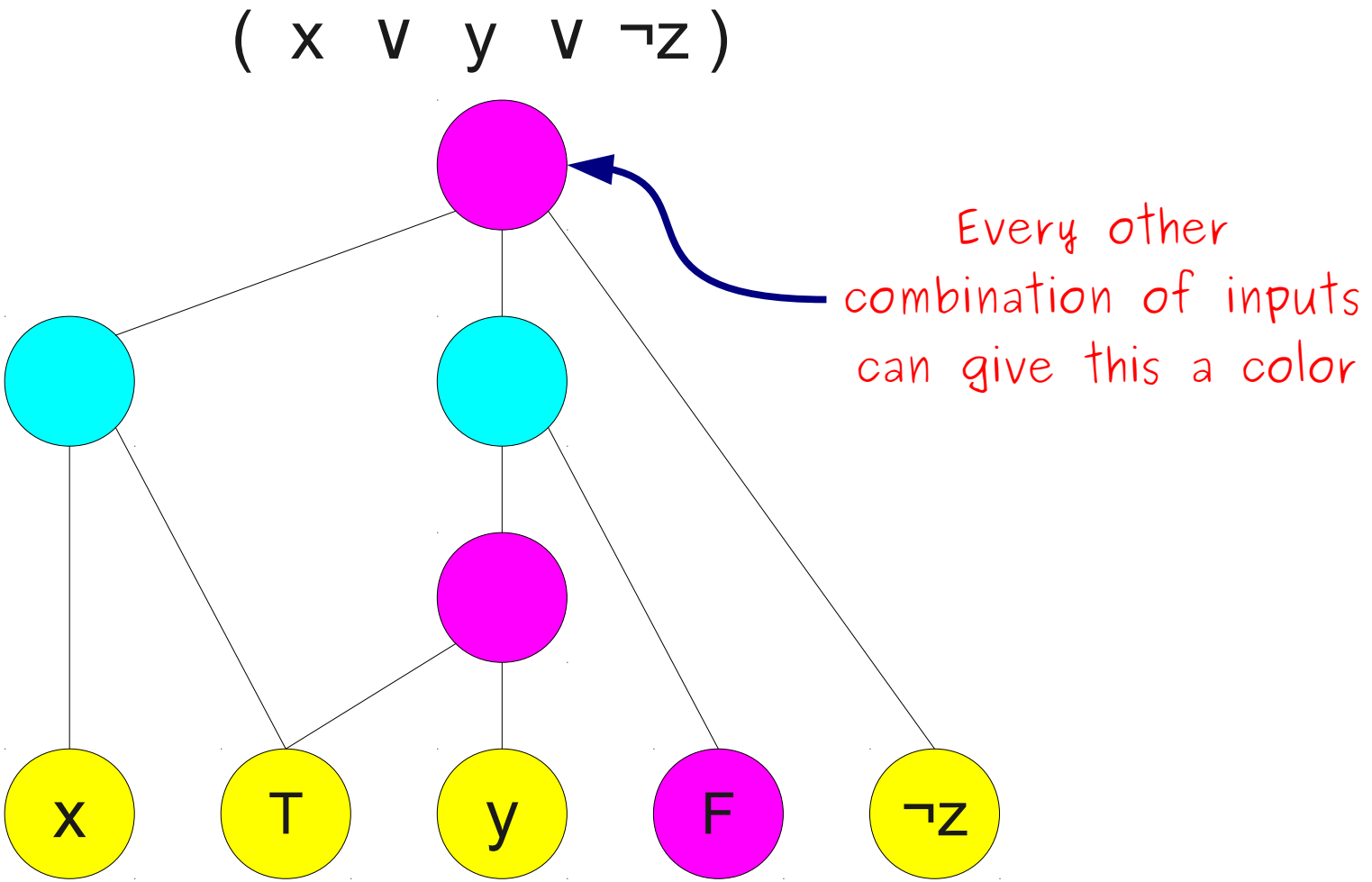
Gadget Three: Clause Satisfiability



Gadget Three: Clause Satisfiability



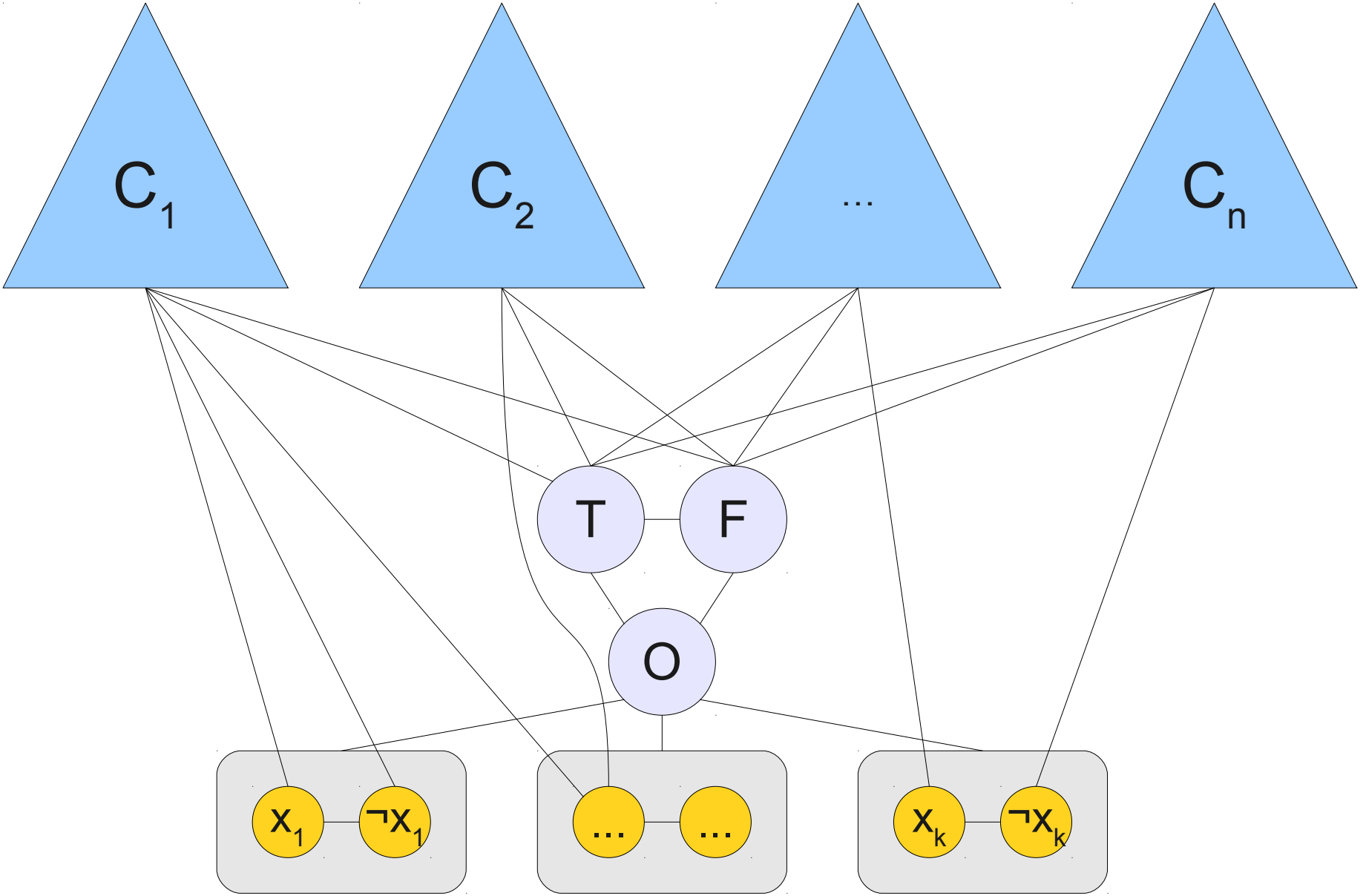
Gadget Three: Clause Satisfiability



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 $\neg v$.
  - Add an edge between v and $\neg v$.
  - Add an edge between v and O and between $\neg v$ 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

- **The Big Picture**
  - How do all of our results relate to one another?
- **Where to Go from Here**
  - What's next in CS theory?