NP

Completeness

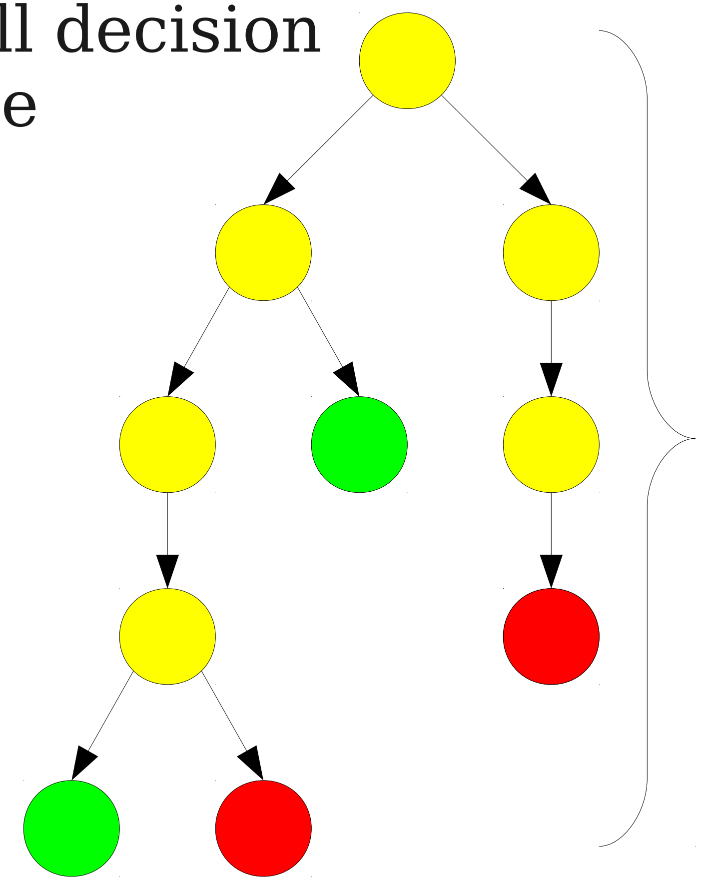
Announcements

- Friday Four Square!
  - Today at 4:15PM, outside Gates.
- Problem Set 8 due right now.
- Problem Set 9 out, due next Friday at 2:15PM.
  - Explore **P**, **NP**, and their connection.
- Did you lose a phone in my office?

Previously on CS103…

NTIME

- The time complexity of a nondeterministic Turing machine is the length of the longest execution path of that NTM on a string of length n.
- The class **NTIME**($f(n)$) consists of all decision problems that can be decided in time $O(f(n))$ by a single-tape NTM.



The Complexity Class **NP**

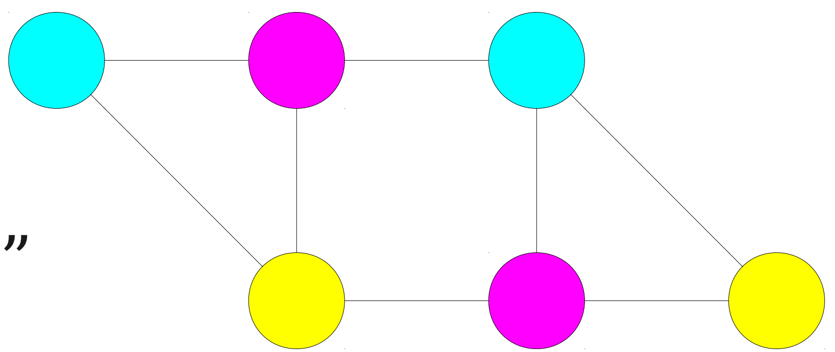
- The complexity class **NP** (**nondeterministic polynomial time**) contains all problems that can be solved in polynomial time by a single-tape NTM.
- Formally:

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

- Equivalently: A language is in **NP** iff there is a polynomial-time verifier for it.

A Problem in **NP**

- A **graph coloring** is a way of assigning colors to nodes in an undirected graph such that no two nodes joined by an edge have the same color.
  - Applications in compilers, cell phone towers, etc.
- Question: Can graph G be colored with at most k colors?
- M = "On input $\langle G, k, C \rangle$, where C is an alleged coloring:
  - **Deterministically** check whether C is a legal k-coloring of G.
  - If so, accept; if not, reject."

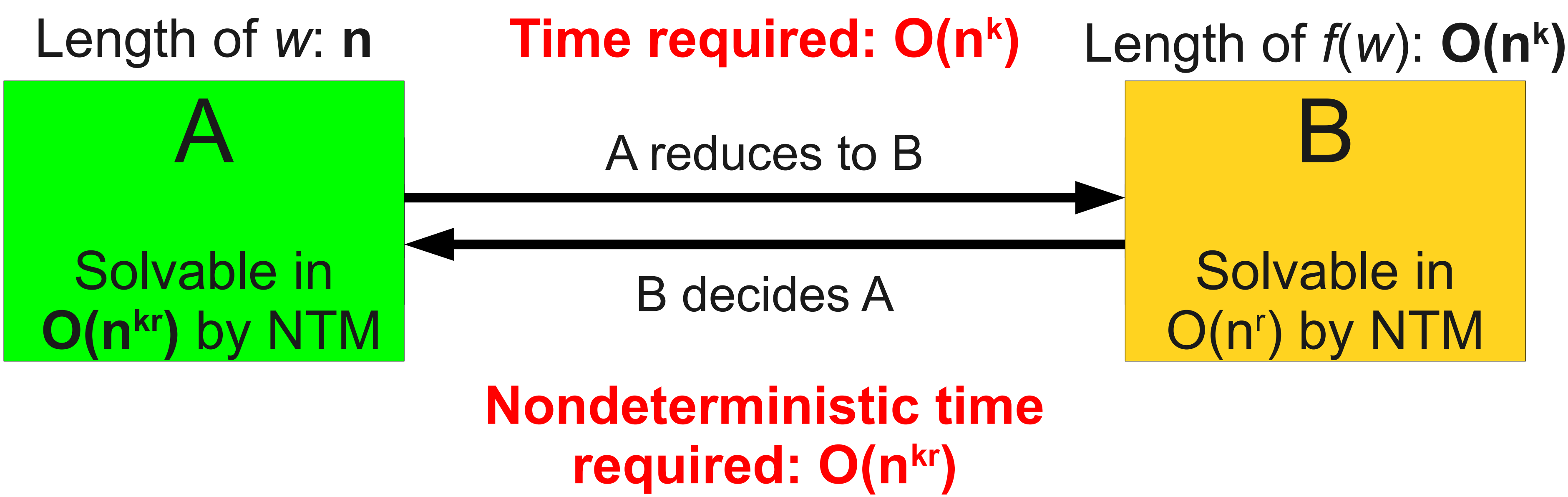


Proving Languages are in **NP**

- **Build a polynomial-time NTM for *L*.**
  - Build an NTM for the language L.
  - Prove that it runs in nondeterministic time $O(n^k)$.
- **Build a polynomial-time verifier for *L*.**
  - Build a TM that verifies a string, given a certificate.
  - Prove that it runs in deterministic time $O(n^k)$.
- **Reduce *L* to a language in NP.**
  - Show how a polynomial-time verifier or polynomial-time NTM for some language L' can be used to decide L.

Polynomial-Time Reductions

- Suppose that we know that $B \in \mathbf{NP}$.
- Suppose that $A \leq_P B$.
- Then $A \in \mathbf{NP}$.



The

Most Important Question

in

Theoretical Computer Science

What is the connection between **P** and **NP**?

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

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

$$\mathrm{TIME}(n^k) \subseteq \mathrm{NTIME}(n^k)$$

$$\mathbf{P} \subseteq \mathbf{NP}$$

Does $\mathbf{P} = \mathbf{NP}$?

$\mathbf{P} \stackrel{?}{=} \mathbf{NP}$

- The question of $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ is the most important question in 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.

Why This Matters

- The following problems are known to be efficiently verifiable, but have no known efficient solutions:
  - Determining whether an electrical grid can be built to link up some number of houses for some price (Steiner tree problem).
  - Determining whether a simple DNA strand exists that multiple gene sequences could be a part of (shortest common supersequence).
  - Determining the best way to assign hardware resources in a compiler (optimal register allocation).
  - Determining the best way to distribute tasks to multiple workers to minimize completion time (job scheduling).
  - **And many more**.
- If **P = NP**, **all** of these problems have efficient solutions.
- If **P ≠ NP**, **none** of these problems have efficient solutions.

Why This Matters

- If **P = NP**:
  - A huge number of seemingly difficult problems could be solved efficiently.
  - Our capacity to solve many problems will scale well with the size of the problems we want to solve.
- If **P ≠ NP**:
  - Enormous computational power would be required to solve many seemingly easy tasks.
  - Our capacity to solve problems will fail to keep up with our curiosity.

What We Know

- Resolving $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ has proven ***extremely difficult.***
- In the past 35 years:
  - Not a single correct proof either way has been found.
  - Many types of proofs have been shown to be insufficiently powerful to determine whether $\mathbf{P} = \mathbf{NP}$.
  - It is commonly believed that $\mathbf{P} \neq \mathbf{NP}$, but no one knows for sure.
- Interesting read: Interviews with leading thinkers about $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$:
  - http://web.ing.puc.cl/~jabaier/iic2212/poll-1.pdf

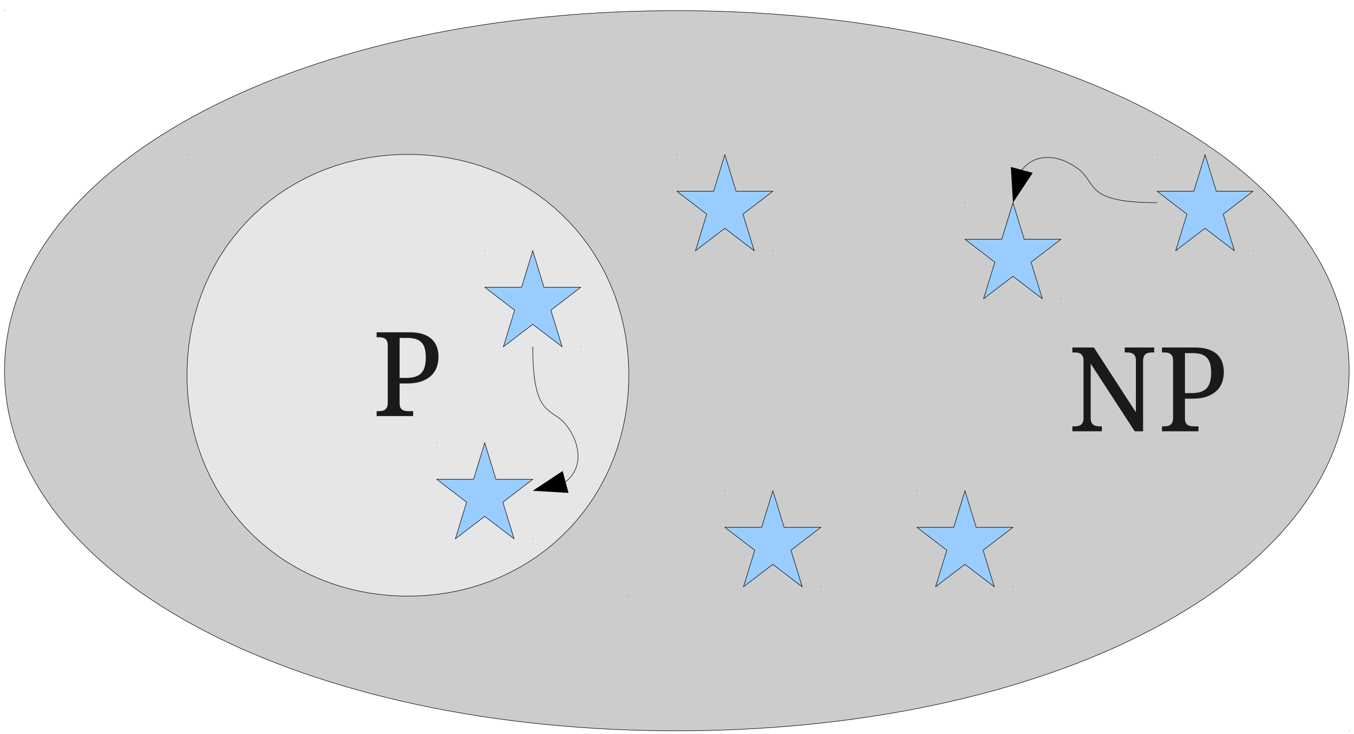
The Million-Dollar Question

The Clay Mathematics Institute has offered a **$1,000,000 prize** to anyone who proves or disproves **P = NP**.

NP-Completeness

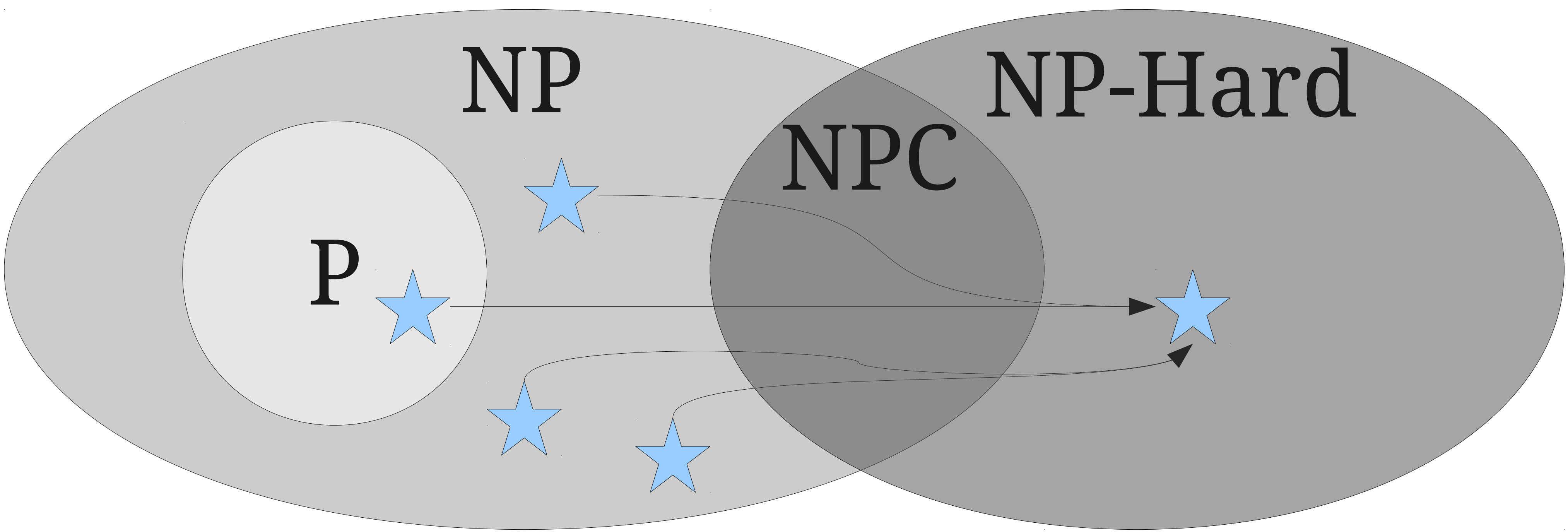
Polynomial-Time Reductions

- 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

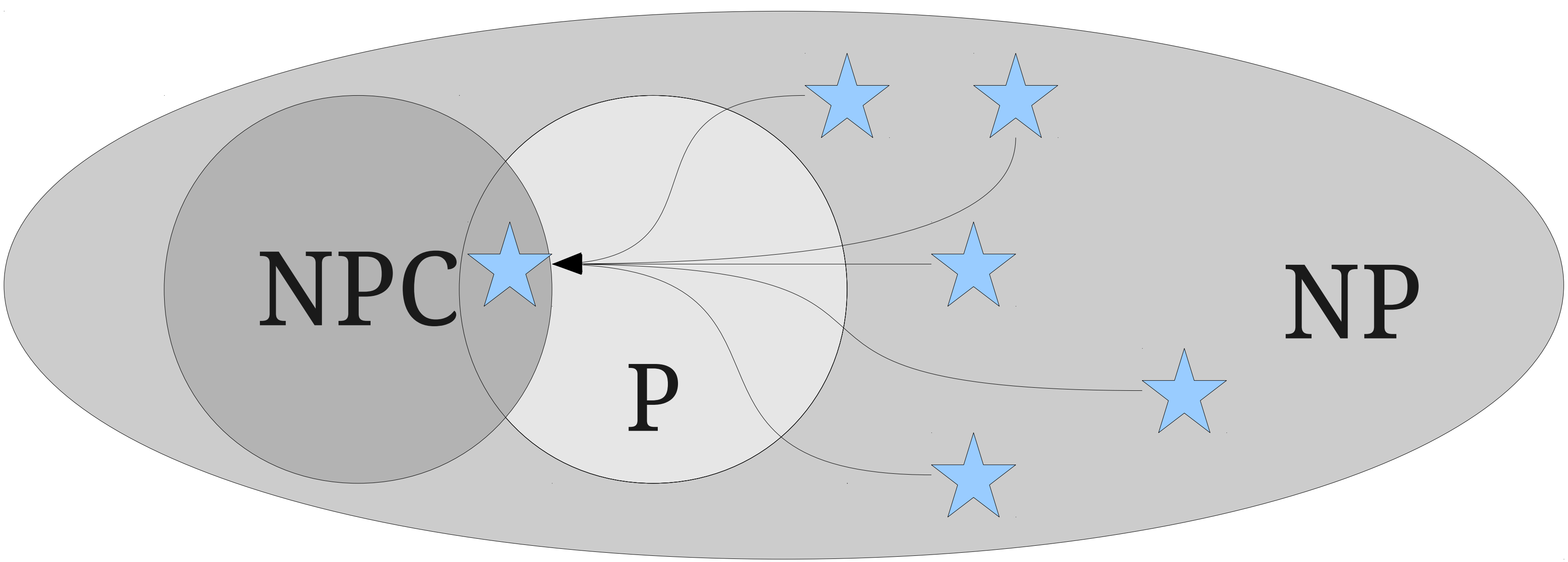
- A language L is called **NP-hard** iff for *every* $L' \in$ **NP**, we have $L' \leq_{\mathrm{P}} L$.
- A language in L is called **NP-complete** iff L is **NP**-hard and $L \in$ **NP**.
- The class **NPC** is the set of **NP**-complete problems.



The Tantalizing Truth

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

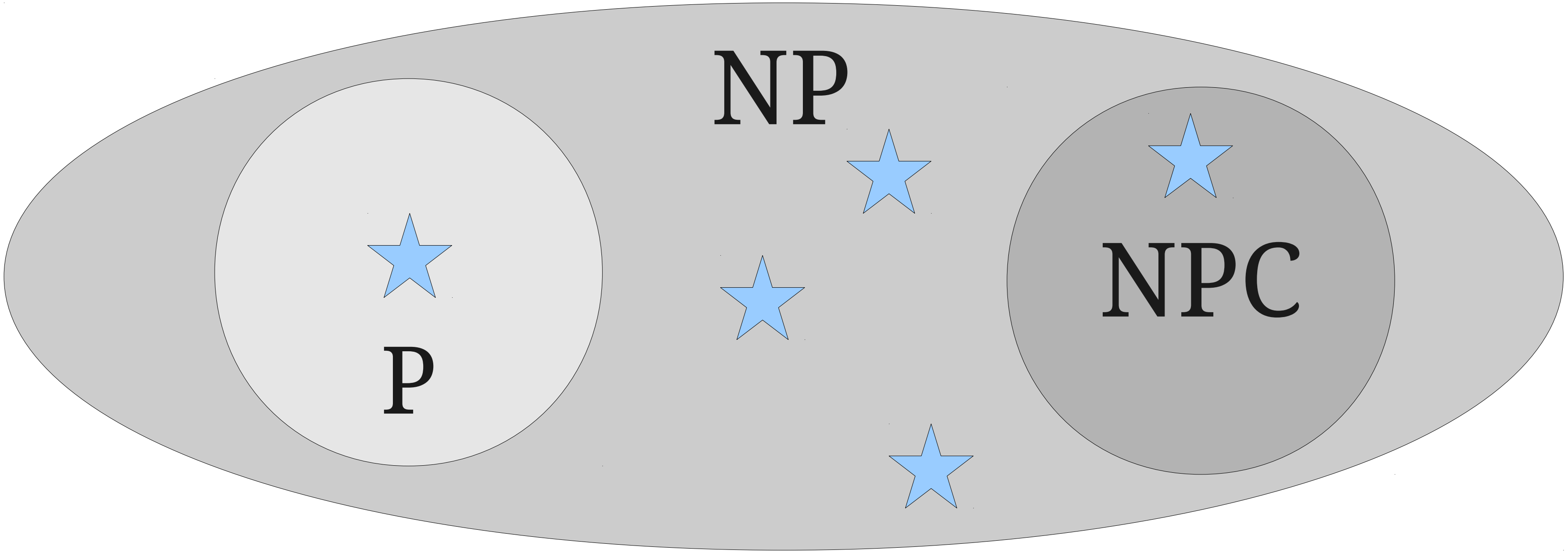
Proof: If $L \in$ **NPC** and $L \in$ **P**, we know for any $L' \in$ **NP** that $L' \leq_P L$, because L is **NP**-complete. Since $L' \leq_P L$ and $L \in$ **P**, this means that $L' \in$ **P** as well. Since our choice of L' was arbitrary, any language $L' \in$ **NP** satisfies $L' \in$ **P**, so **NP** ⊆ **P**. Since **P** ⊆ **NP**, this means **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 (commonly-held) assumption that $\mathbf{P} \neq \mathbf{NP}$, 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 $\mathbf{P} = \mathbf{NP}$.
  - If any **NPC** problem is not in **P**, then $\mathbf{P} \neq \mathbf{NP}$.
- 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**.
- Similar terms:
  - φ is **tautological** if it is always true.
  - φ is **satisfiable** if it *can* be made true.
  - φ is **unsatisfiable** if it is always false.

SAT

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

Given a propositional logic formula φ, is φ satisfiable?

- Formally:

$$\textbf{SAT} = \{ \langle \varphi \rangle \mid \varphi \textbf{ is a satisfiable PL formula} \}$$

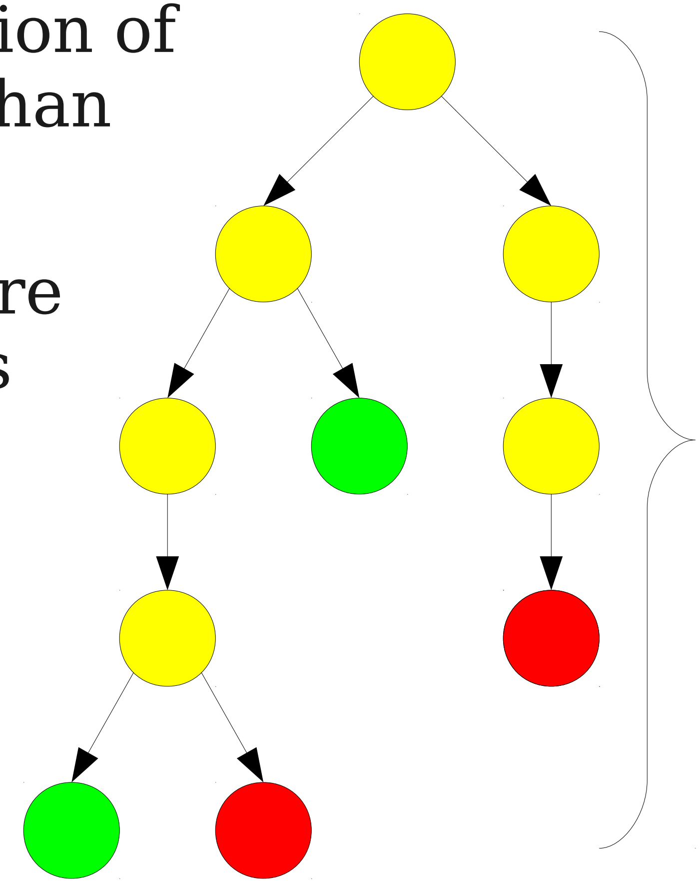
Theorem (Cook-Levin): SAT is **NP**-complete.

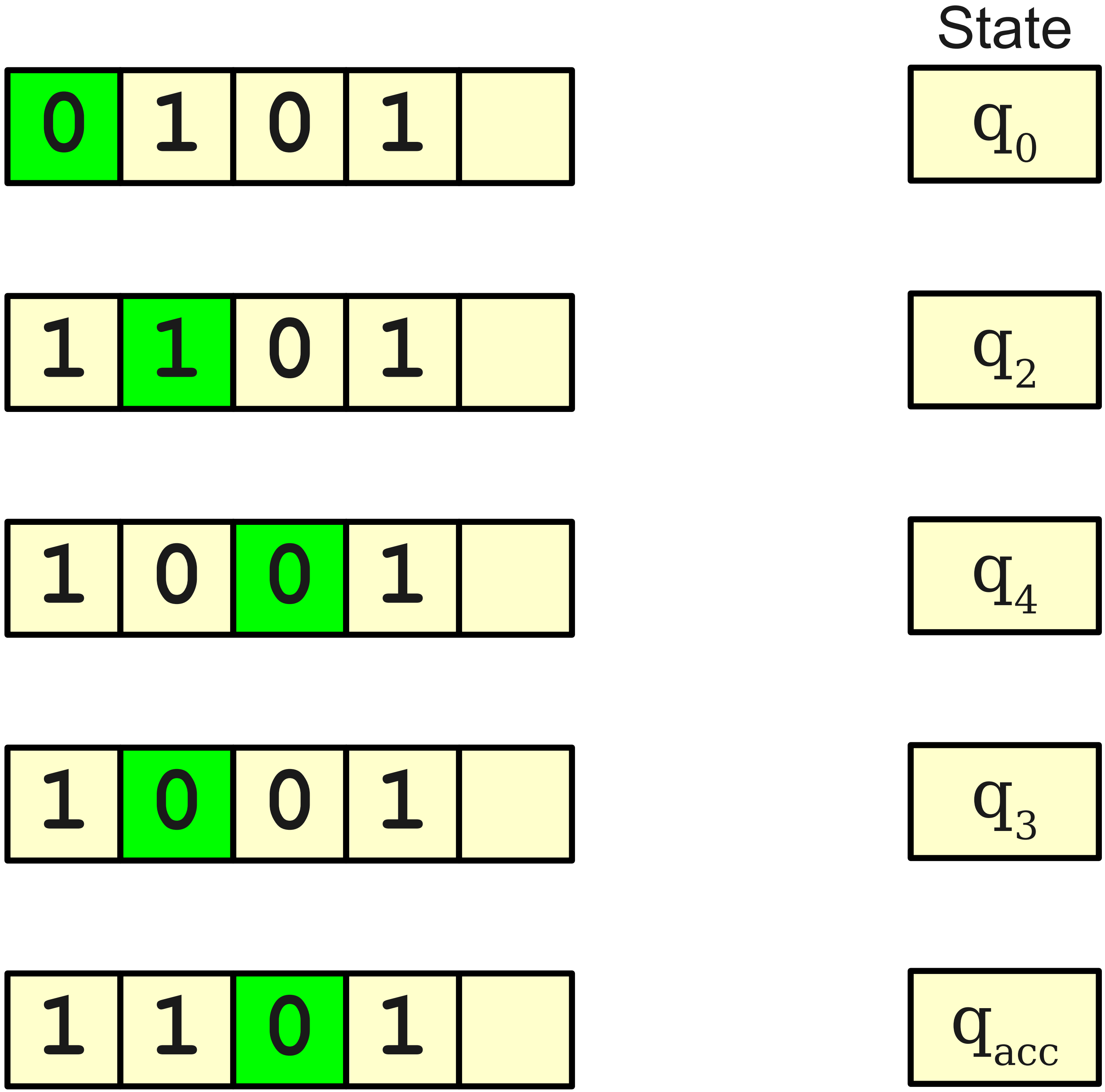
Sketch of the Proof

- We need to show that every single language in **NP** has a polynomial-time reduction to SAT.
- To do so, we will use the fact that every language in **NP** has a polynomial-time NTM.
- We can build a SAT formula that encodes the rules for how that NTM operates.
- If there is some set of choices where the NTM accepts, our formula will be satisfiable.
- If there are no choices we can make where the NTM accepts, our formula will be unsatisfiable.

Polynomial-Time NTMs

- Recall: The runtime of an NTM on a string w is the height of its computation tree on w.
- If an NTM runs in polynomial time, there is some polynomial $p(n)$ such that no execution of the NTM on a string w takes more than $p(|w|)$ time on any branch.
- This means the NTM never uses more than $p(|w|)$ tape on any branch of its computation on w.




State
0 1 0 1
q_0
1 1 0 1
q_2
1 0 0 1
q_4
1 0 0 1
q_3
1 1 0 1
q_{acc}

Proving Cook-Levin

- Build a PL formula that encodes the following idea:
  - Machine M begins with w written on its tape, followed by blanks.
  - Each step of the computation legally follows from the previous step.
  - The machine ends in an accepting state.
- This formula is satisfiable iff there is some series of choices M can make such that M accepts w.
- This formula has size polynomial in $|w|$.
- **See Sipser for Details.**

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 ¬, ∨, ∧.
- No nesting allowed.

The Structure of CNF

$$\underbrace{(x \lor y \lor \neg z)} \land \underbrace{(\neg x \lor \neg y \lor z)} \land \underbrace{(\neg x \lor y \lor \neg z)}$$

Each clause must have
at least one
true literal in it.

The Structure of CNF

$$(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)$$

We should pick at least one
true literal from each clause

The Structure of CNF

$$(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)$$

… 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:
  - $(x \lor y \lor z) \land (\neg x \lor \neg y \lor z)$
  - $(x \lor x \lor x) \land (y \lor \neg y \lor \neg x) \land (x \lor y \lor \neg y)$
  - But not $(x \lor y \lor z \lor w) \land (x \lor y)$
- The language **3SAT** is defined as follows:

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

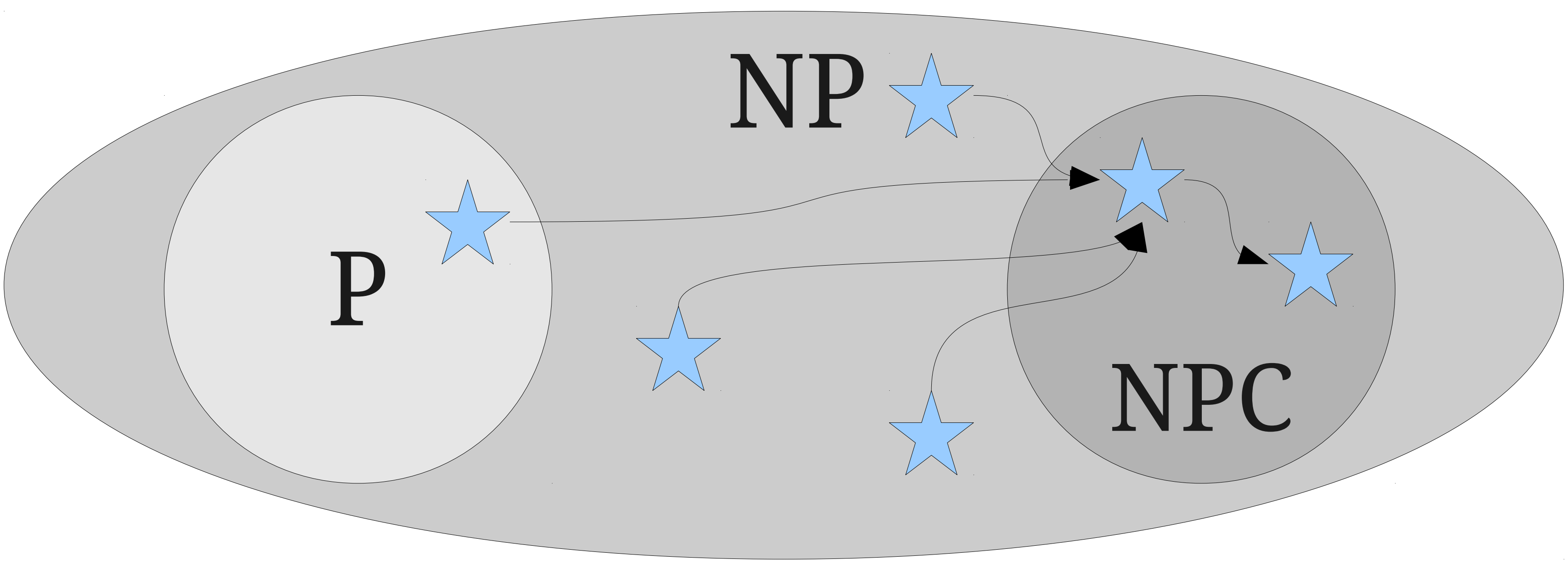
Theorem (Cook-Levin): 3SAT is **NP**-Complete

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 ∈ **NPC** as a starting point for finding other **NPC** languages.
  - **Idea**: Reduce 3SAT to some other language.

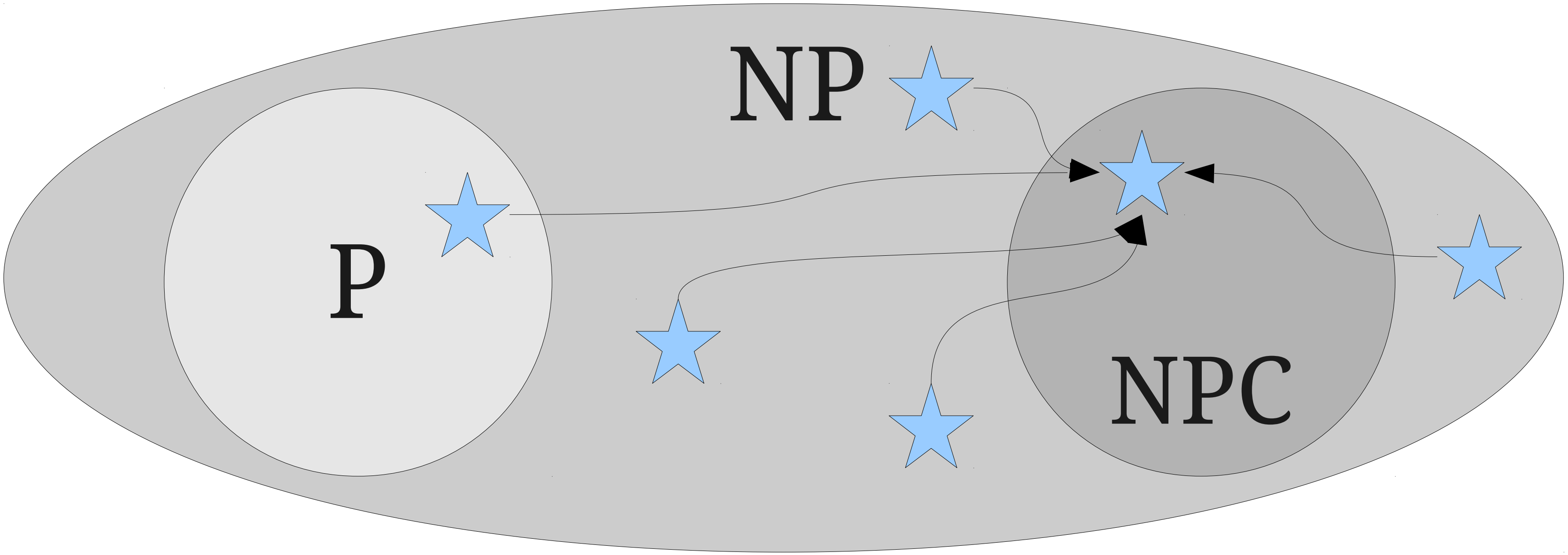
NP-Completeness

- **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 $L' \in \mathbf{NPC}$. ■

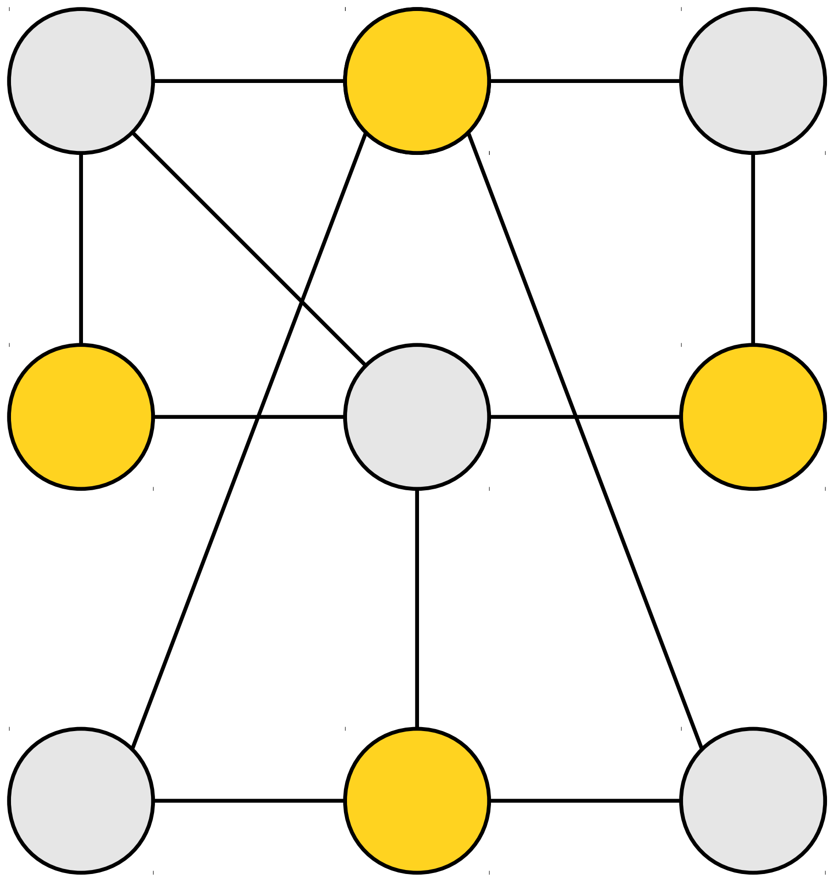


Be Careful!

- 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

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:

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

$\mathit{INDSET} \in \mathbf{NP}$

- 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 $\langle G, n, S \rangle$:
    - 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

$$(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)$$

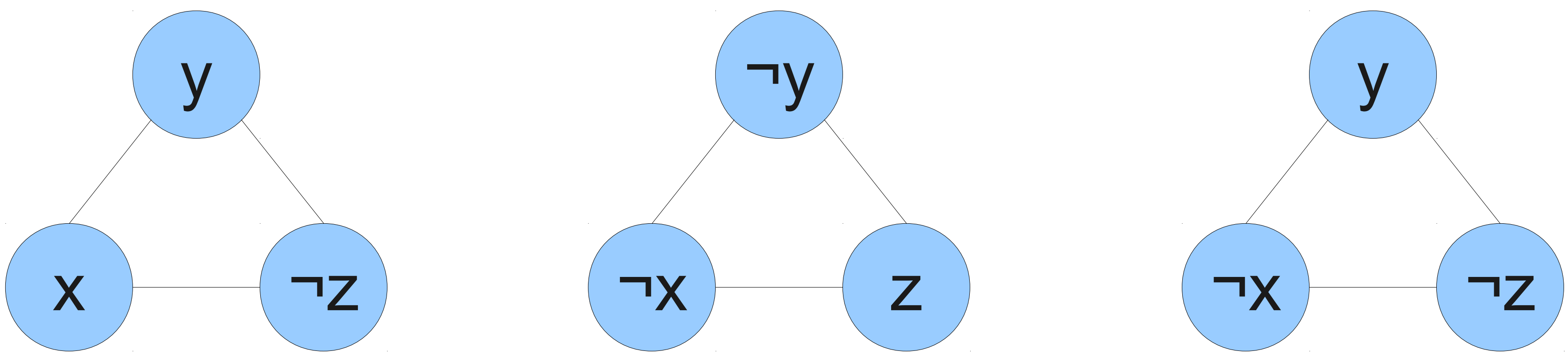
Each clause must have
<u>at least</u> one
true literal in it.

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

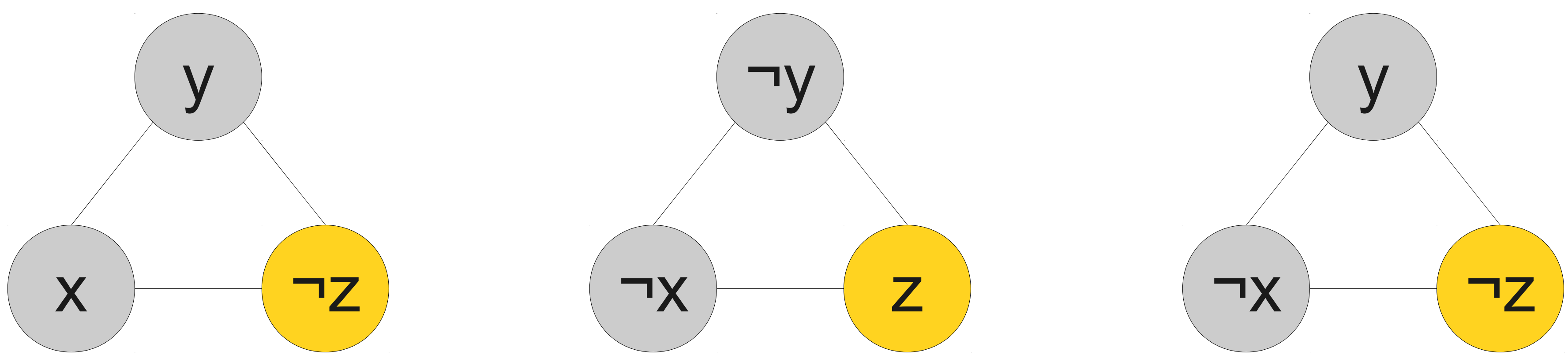
$$(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)$$



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

From 3SAT to INDSET

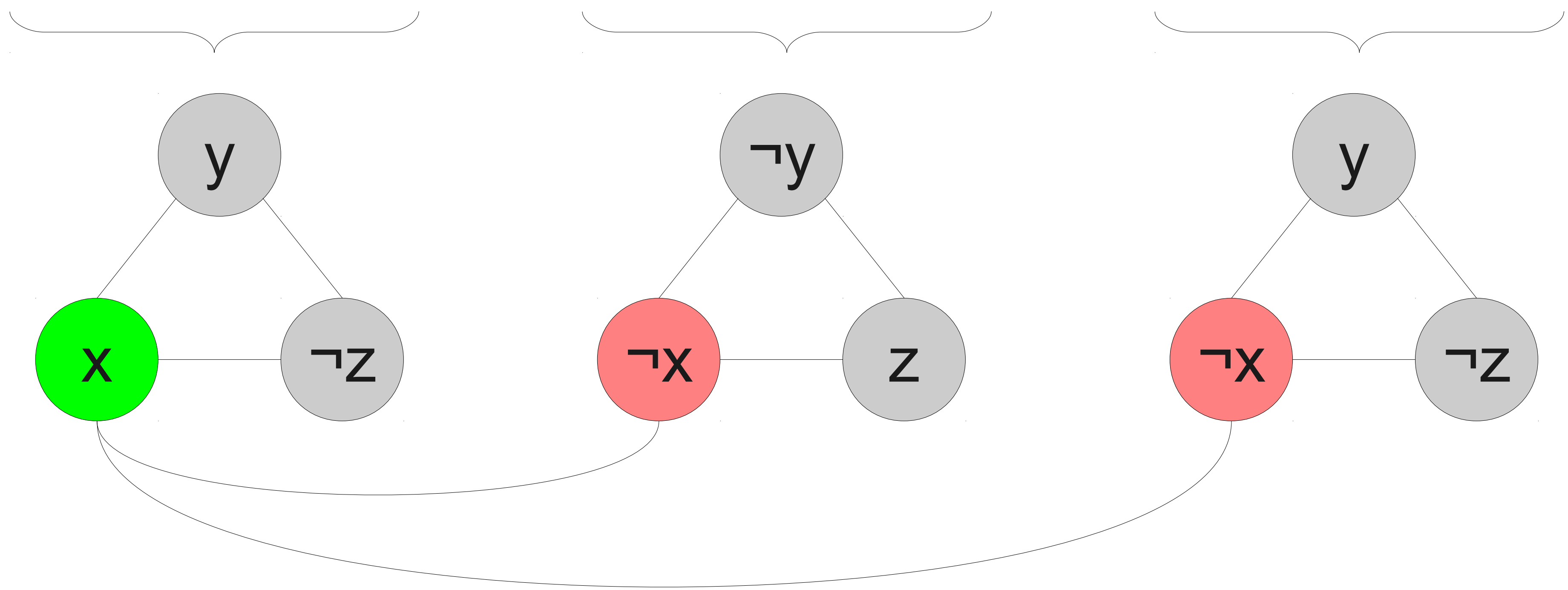
$$(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)$$



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

From 3SAT to INDSET

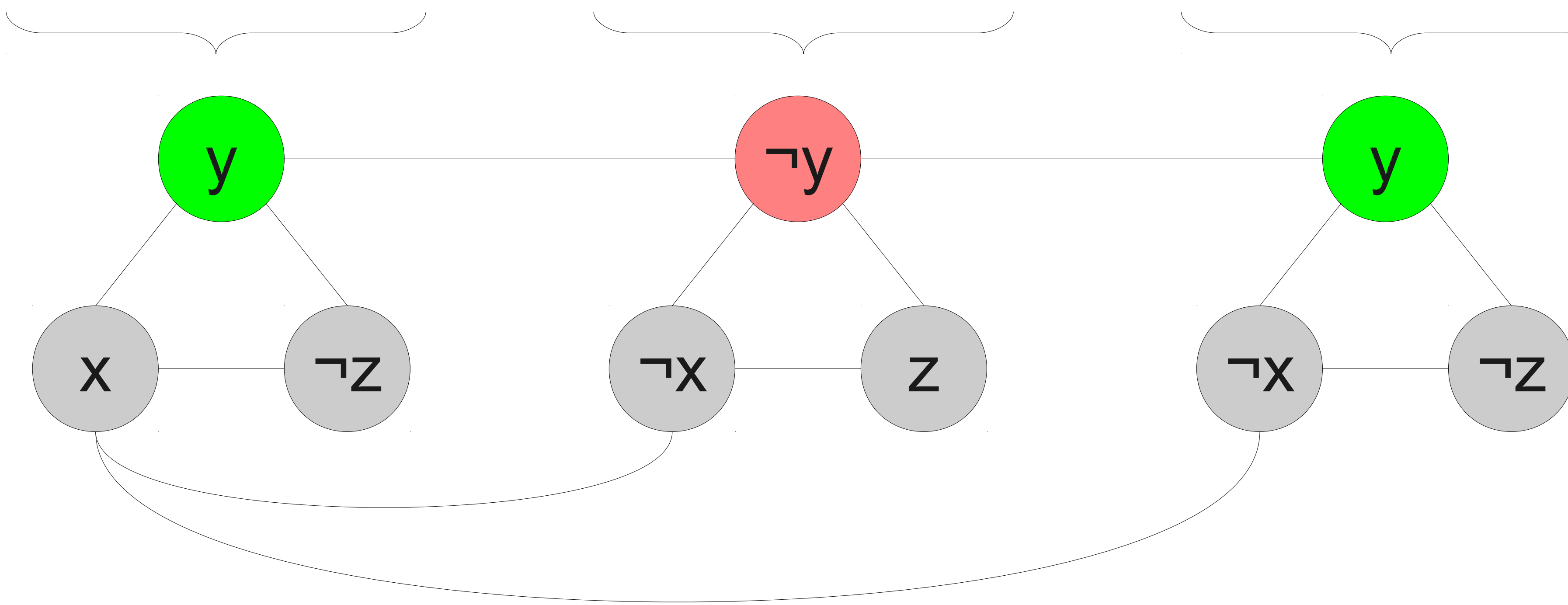
$$(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)$$



No independent set in this graph can choose two nodes labeled x and $\neg x$.

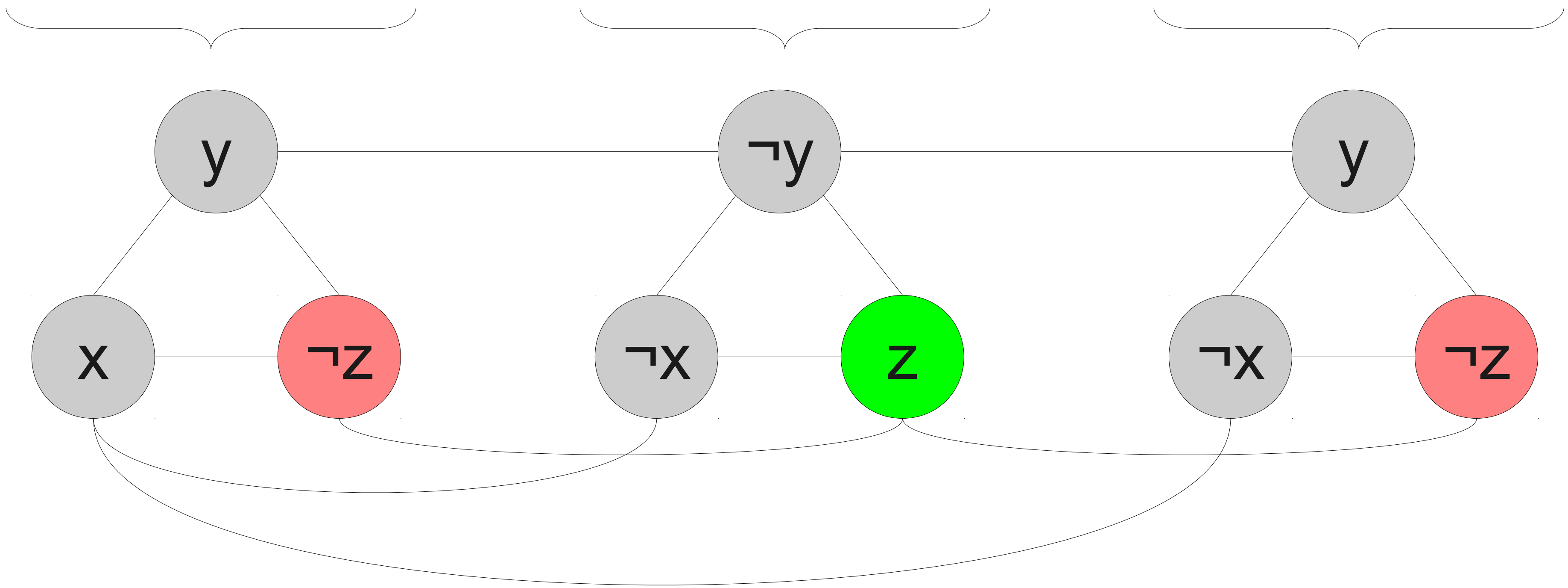
From 3SAT to INDSET

$$(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)$$



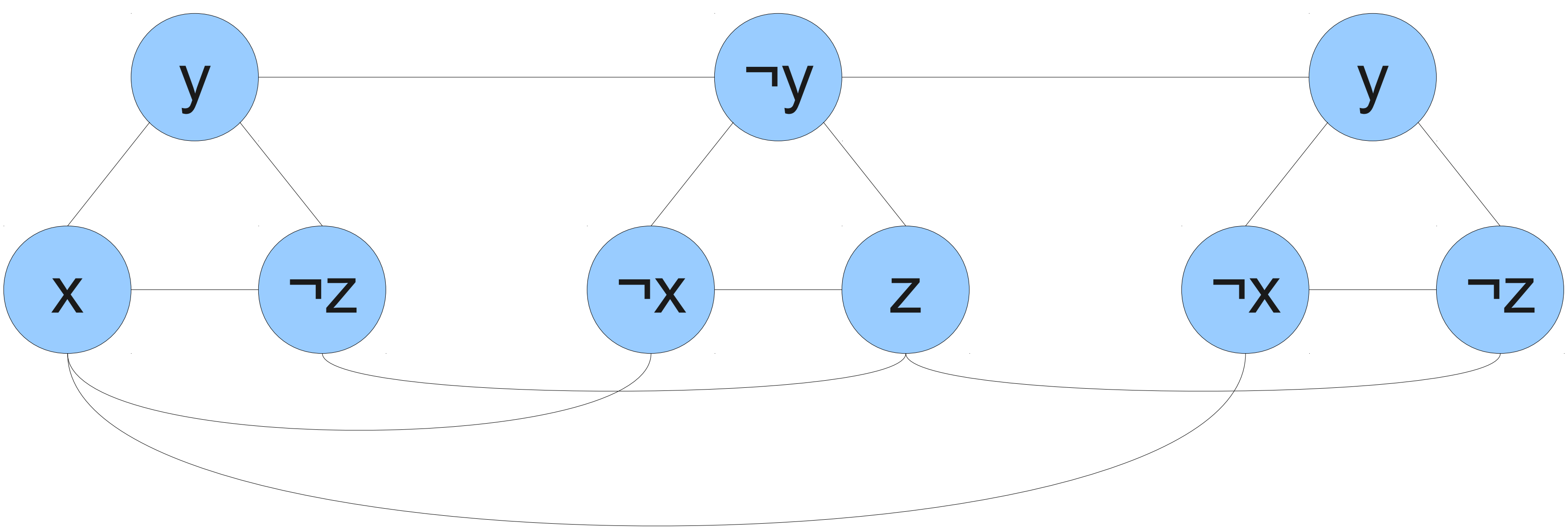
From 3SAT to INDSET

$(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)$



From 3SAT to INDSET

$$(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)$$

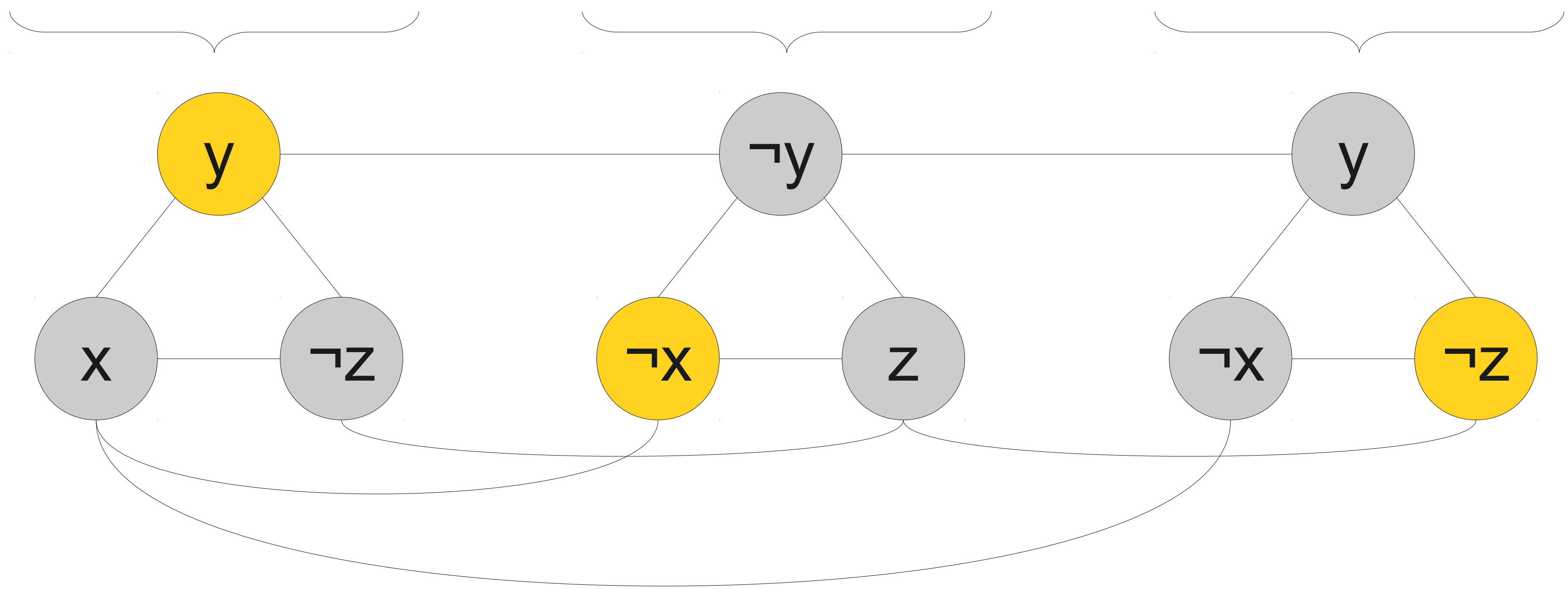


If this graph has an independent set of size three, the original formula is satisfiable.

From 3SAT to INDSET

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

$$(x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)$$



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.

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, $\varphi \in$ 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. ■

Next Time

- **More NP-Completeness**
  - A sampler of other **NP**-complete problems.
  - Problems from disaster relief, route planning, etc.