More Reductions

CSE 421 Winter 2023
Lecture 22

Today

Some more context on P, NP, etc.

When you suspect your problem is NP-hard, how do you prove it?

A Formal Definition

We need a formal definition of a reduction.

We will say "A reduces to B in polynomial time" (or "A is polynomial time reducible to B" or "A reduces to B" or "$A \leq_P B$" or "$A \leq B$") if:

There is an algorithm to solve problem A, which, if given access to a library function for solving problem B,

- Calls the library at most polynomially-many times
- Takes at most polynomial-time otherwise excluding the calls to the library.

A note on $\leq_P$

Let A, B be the decision versions of any problems we've solved this quarter (2-color, MST, is there a stable matching, is there a flow…)

$A \leq_P B$ and $B \leq_P A$.

So they're all "equal" in difficulty.

$\leq_P$ is a very "coarse" definition of difficulty.

Only enough to (maybe) separate NP-hard problems from problems in P.

It won't tell you the difference between a problem that can be solved in $\Theta(n^2)$ and one that can be solved in $\Theta(n)$ or $\Theta(n^{100})$.

Solve vs. Verify

To solve a problem, you get the instance, an algorithm decides whether the correct answer is "YES" or "NO"

"Is there a spanning tree of weight at most k?" can be solved with Kruskal's algorithm

To **verify,** you get an instance AND a claimed "certificate", an algorithm decides whether the certificate PROVES the instance is a YES instance.

"Is there a spanning tree of weight at most k?" is **verified** by getting (1) the graph (2) the proposed spanning tree and returning "is the spanning tree *you gave me* weight at most k**?"**

NP

Our second set of problems have the property that "I'll know it when I see it"
We're looking for **something**, and if someone shows it to me, we can recognize it quickly (it just might be hard to find)

NP (stands for "nondeterministic polynomial")

The set of all decision problems such that for every YES-instance (of size n), there is a certificate (of size $O(n^k)$) for that instance which can be verified in polynomial time.

A "verifier" takes in: an instance of the NP problem, and a "proof"
And returns "true" if it received a valid proof that the instance is a YES instance, and "false" if it did not receive a valid proof

NP problems have "verifiers" that run in polynomial time.
Do they have **solvers** that run in polynomial time? The definition doesn't say.

NP

Our second set of problems have the property that "I'll know it when I see it"
We're looking for **something,** and if someone shows it to me, we can recognize it quickly (it just might be hard to find)

NP (stands for "nondeterministic polynomial")

The set of all decision problems such that for every YES-instance (of size n), there is a certificate (of size $O(n^k)$) for that instance which can be verified in polynomial time.

If you have a "YES" instance, a little birdy can magically find you this certificate-thing, and you'll say "Oh yeah, that's totally a yes instance!"

What if it's a NO instance? No guarantee.

NP

NP

The set of all decision problems such that for every YES-instance (of size n), there is a certificate (of size $O(n^k)$) for that instance which can be verified in polynomial time.

Decision Problems such that:
If the answer is YES, you can prove the answer is yes by
- Being given a "proof" or a "certificate"
- Verifying that certificate in polynomial time.

What certificate would be convenient for short paths?
- The path itself. Easy to check the path is really in the graph and really short.

Light Spanning Tree:
Is there a spanning tree of graph G of weight at most k?

The spanning tree itself.
Verify by checking it really connects every vertex and its weight.

3-Coloring:
Can you color vertices of a graph red, blue, and green so every edge has differently colored endpoints?

The coloring.
Verify by checking each edge.

Large flow:
Is there a flow from s to t in G of value at least k?

The flow itself.
Verify the capacity constraints, conservation, and that flow value at least k.

NP

Our second set of problems have the property that "I'll know it when I see it"
We're looking for **something,** and if someone shows it to me, we can recognize it quickly (it just might be hard to find)

NP (stands for "nondeterministic polynomial")

The set of all decision problems such that for every YES-instance (of size n), there is a certificate (of size $O(n^k)$) for that instance which can be verified in polynomial time.

It's a common misconception that NP stands for "not polynomial"

Never, ever, ever, ever say "NP" stands for "not polynomial"
Please
Every time someone says that, a theoretical computer scientist sheds a single tear
(That theoretical computer scientist is me)

Solve vs. Verify

Verification is equivalent to the “magic” of nondeterminism

Remember NFAs? That could “magically” decide which state to jump to?

It’s that kind of nondeterminism. But instead of just jumping state-to-state, you can also “magically” write bits to memory too.

More on this soon.

P vs. NP

P vs. NP

Are P and NP the same complexity class?
That is, can every problem that can be verified **in polynomial time also be** solved **in polynomial time.**

If you'll know it when you see it, can you also search to find it efficiently?

No one knows the answer to this question.

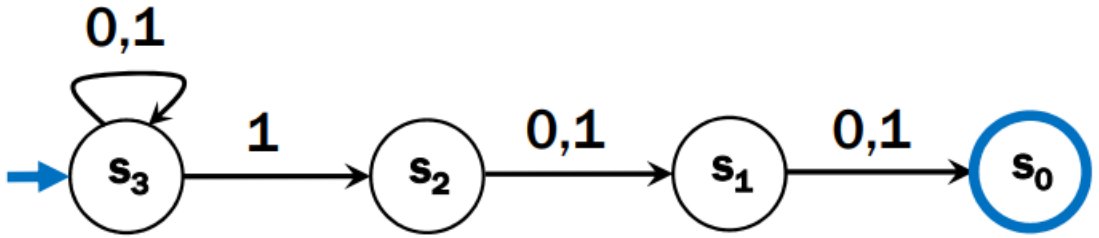
In fact, it's the biggest unsolved question in Computer Science.

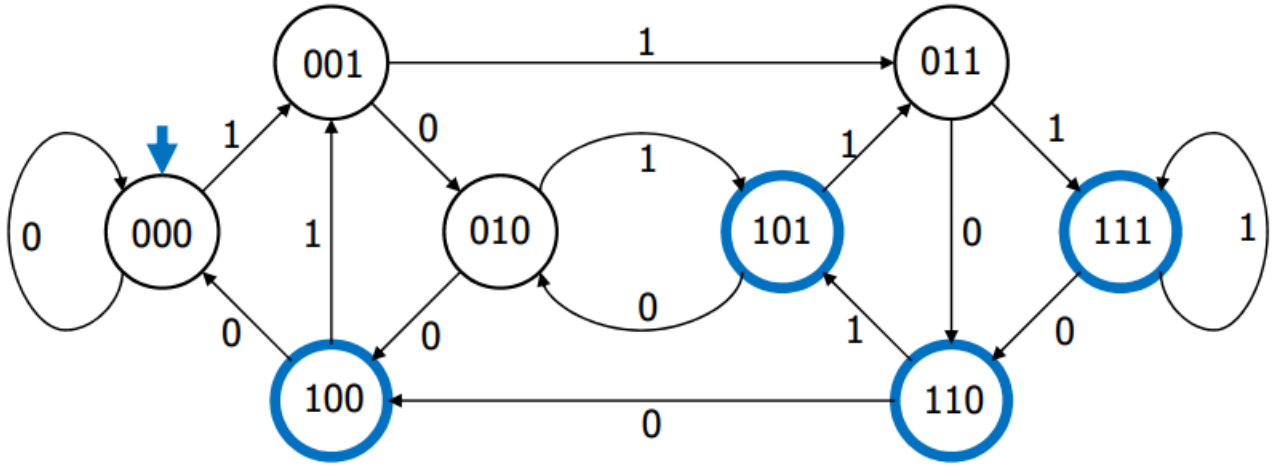
Why is it called NP?

You've seen nondeterministic computation before.
Way back in 311.
What do verifiers have to do with…anything?

NFAs would "magically" decide among a set of valid transitions.
Always choosing one that would lead to an accept state (if such a transition exists).

An NFA and a DFA for the language
"binary strings with a 1 in the 3rd position from the end."





From Kevin & Paul's 311 Lecture 23.

Nondeterminism

The really formal definitions (take 431) use Turing machines.

What would a nondeterministic **computer** look like?

It can run all the usual commands,

But it can also magically (i.e. nondeterministically) decide to set any bit of memory to 0 or 1.

Always choosing 0 or 1 to cause the computer to output YES,

(if such a choice exists).

If we had a nondeterministic computer…

Can you think of a polynomial time algorithm on a nondeterministic computer to:

Solve 2-COLOR?

Solve 3-COLOR?

If we had a nondeterministic computer…

Can you think of a polynomial time algorithm on a nondeterministic computer to:

Solve 2-COLOR?

Just run our regular deterministic polynomial time algorithm

Or nondeterministically guess colors, output if they work.

Solve 3-COLOR?

Nondeterministically guess colors, output if they work.

The pattern? Nondeterministically guess the certificate! Then **verify** that it works. A nondeterministic computer can *solve* a problem in NP if and only if a regular computer can verify a given certificate.

Analogue of NFA/DFA equivalence

You showed in 311 that the set of languages decided by NFAs and DFAs were the same.

I.e. NFAs didn't let you solve more problems than DFAs.

But it did sometimes make the process a lot easier.

There are languages such that the best DFA is exponentially larger than the best NFA. (like the one from a few slides ago).

P vs. NP is an analogous question. Does non-determinism let us use exponentially fewer resources to solve some problems?

Hard Problems

Let's say we want to figure out if every problem in NP can actually be solved efficiently.

We might want to start with a really hard problem in NP.

What is the hardest problem in NP?

What does it mean to be a hard problem?

Reductions are a good definition:

If A reduces to B then "$A \leq B$" (in terms of difficulty)

- Once you have an algorithm for B, you have one for A automatically from the reduction!

NP-hardness

NP-hard

The problem B is NP-hard if
for all problems A in NP, A reduces to B.

An NP-hard problem is "hard enough" to design algorithms for that if you write an efficient algorithm for it, you've (by accident) designed an algorithm that works for every problem in NP.

What does it look like? Let A be in NP, and let B be the NP-hard problem you solved, on an input to A, "run the reduction" and plug in your actual algorithm for B!

NP-Completeness

NP-Complete

The problem B is NP-complete if B is in NP
and B is NP-hard

An NP-complete problem is a "hardest" problem in NP.

If you have an algorithm to solve an NP-complete problem, you have an algorithm for **every** problem in NP.

An NP-complete problem is a **universal language** for encoding "I'll know it when I see it" problems.

Why is being NP-hard/-complete interesting?

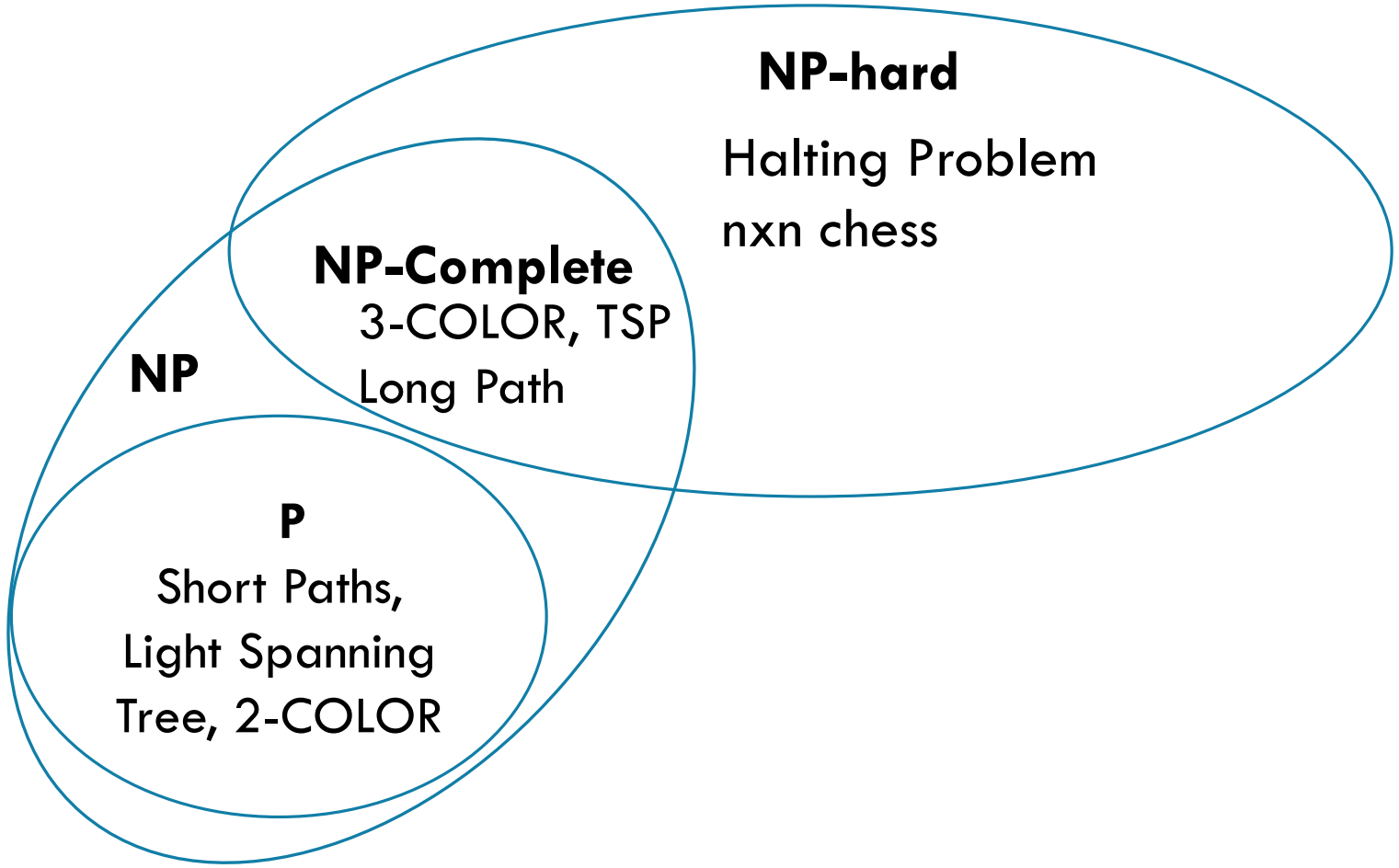
Let B be an NP-hard problem. Suppose you found a polynomial time algorithm for B. Why is that interesting?

You now have for free a polynomial time algorithm for **every** problem in NP. (if A is in NP, then $A \leq B$. So plug in your algorithm for B!)

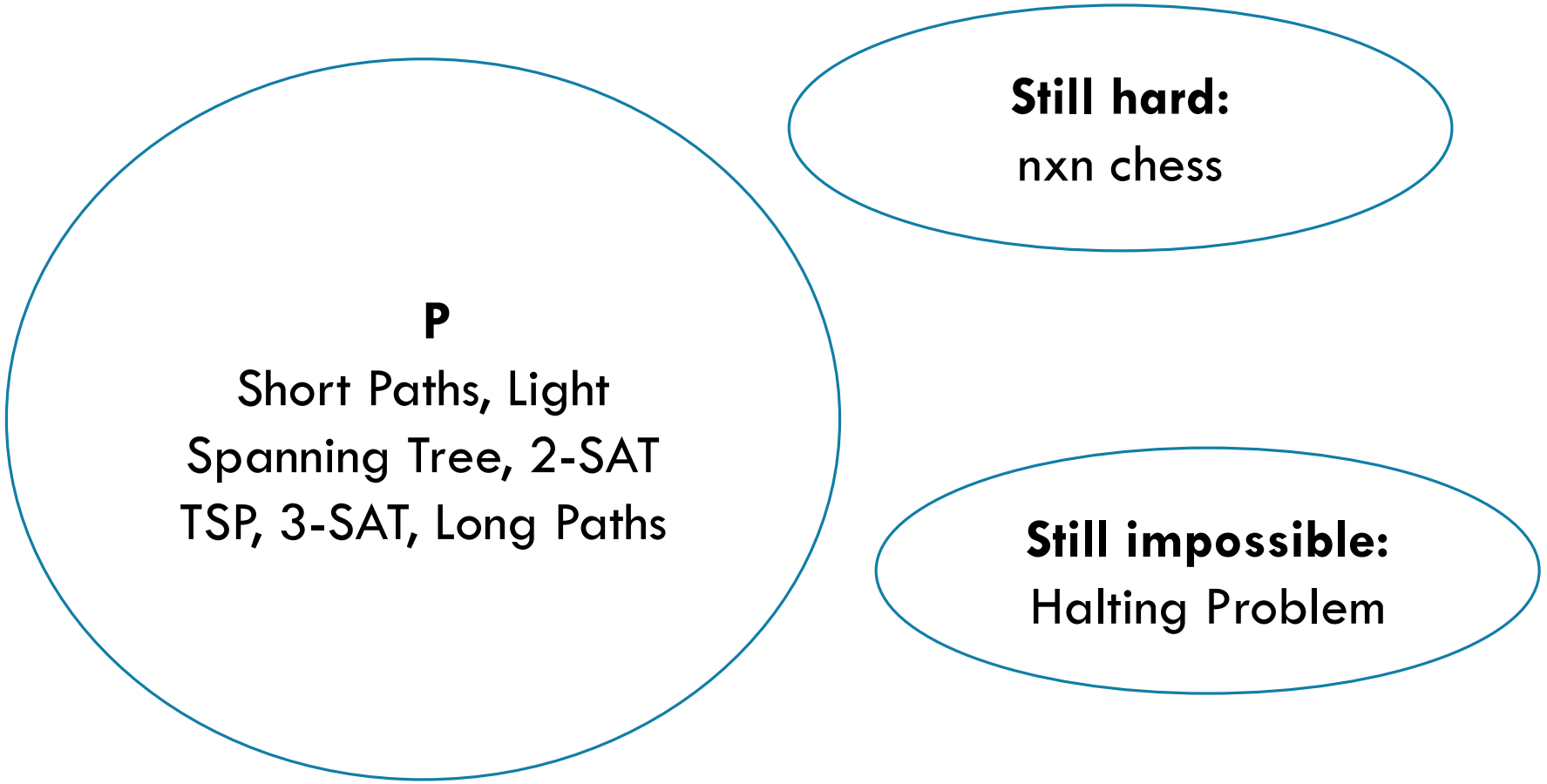
So $P = NP$. (if you find a polynomial time algorithm for an NP-hard problem).

On the other hand, if any problem in NP is not in P (any doesn't have a polynomial time algorithm), then no NP-complete problem is in P.

What The World Looks Like (We Think)



What The World Looks Like (If P=NP)



NP-Completeness

An NP-complete problem does exist!

Cook-Levin Theorem (1971)

3-SAT is NP-complete

Theorem 1: If a set S of strings is accepted by some nondeterministic Turing machine within polynomial time, then S is P-reducible to {DNF tautologies}.

This sentence (and the proof of it) won Cook the Turing Award.

What's 3-SAT?

Input: A list of Boolean variables $x_1, \ldots, x_n$

An expression in Conjunctive Normal Form, where each clause has exactly 3 literals.

"AND" of "ORs"
Λ outside parens
V inside parens

Something like:

$$(z_i \vee z_j \vee z_k) \wedge (z_i \vee z_\ell \vee z_a) \wedge \cdots \wedge (z_a \vee z_b \vee z_c)$$

One of the subexpressions inside parens

Where z is a "literal" a variable or the negation of a variable ($x_i, \neg x_j$, etc.).

Output: true if there is a setting of the variables where the expression evaluates to true, false otherwise.

Why is it called 3-SAT? 3 because you have 3 literals per clause
SAT is short for "satisfiability" can you satisfy all of the constraints?

3-SAT examples

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

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

3-SAT examples

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

Satisfiable (for example: $x = T; y = T, z = F$)

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

Not satisfiable

Really? All of them?

The idea that there is an NP-complete problem might be surprising.

Every problem in NP reduces to it? All of them? Like no exceptions?

Yes! Really all of them!

Really? All of them?

The idea that there is an NP-complete problem might be surprising.

Every problem in NP reduces to it? All of them? Like no exceptions?

The proof is very fun, but also very a-full-lecture-long (take 431).

Hand-wavy intuition:

1. Let $A \in NP$, then it has a verifier that is "checking for something" in the certificate.

2. "checking for something" can be broken down into a bunch of tiny steps (because code can be broken down to tiny pieces).

3. each of those tiny pieces can be built up to a giant SAT expression (where the variables are the certificate).

Ok, so what?

If anyone ever asks me to solve 3-SAT exactly in polynomial time, I'll say no…

How often does that happen? How does this help?

Suppose you're interested in the problem B. You've tried to design a polynomial time algorithm; haven't succeeded yet…you start to wonder if it's possible at all…or maybe B is also NP-hard…

Well if you can show 3-SAT $\leq_P B$ then B is NP-hard too!

$A \leq_P$3-SAT and 3-SAT $\leq_P B \Rightarrow A \leq_P B$ ($\leq_P$ is transitive)

Which Direction?

To show B is NP-hard

How do you remember which direction?

The core idea of an NP-completeness reduction is a proof by contradiction:

Suppose, for the sake of contradiction, there were a polynomial time algorithm for B. But then if there were I could use that to design a polynomial time algorithm for problem A.

But we really, really, really don't think there's a polynomial time algorithm for problem A. So we should really, really, really think there isn't one for B either!

Let's Show a problem is NP-hard.

Once we have one NP-complete problem, the process gets a lot easier.

3-SAT is NP-complete. Prove that 3-COLOR is NP-hard.

Input: An undirected graph G.

Output: True if G can be 3-colored (each vertex red, blue, green and no edge has same-colored endpoints); false otherwise

To show 3-color is NP-complete

What do we need to show?

To show our new problem is NP-complete

A reduction from a known NP-hard problem to our new problem

That our new problem is in NP itself

(To show our new problems is NP-hard, just do the first step).

We need to show:

To show 3-color is NP-complete

What do we need to show?

To show our new problem is NP-complete

A reduction from a known NP-hard problem to our new problem

That our new problem is in NP itself

(To show our new problems is NP-hard, just do the first step).

We need to show:

3-color is in NP; 3-SAT $\leq_P$ 3-COLOR

To show 3-color is NP-complete

We need to show:

3-color is in NP; 3-SAT $\leq_P$ 3-COLOR

3-color is in NP (the certificate is the assignment of vertices to colors; a linear-time BFS can verify if the coloring is correct).

Get the direction of the reduction right! Double-check it!

The other reduction does exist! (because 3-SAT is NP-complete). You won't notice until it's too late. Check at the beginning!

This is a big claim!

3-SAT and 3-coloring aren't all that similar

They both have the number 3, I guess…

In 3-SAT we assign variables to true and false.

In 3-COLOR we assign vertices to red, blue, or green.

How could we do that?

Idea

Need to turn a 3-SAT instance into a 3-COLOR instance

(The reduction has access to a library for 3-COLOR)

And need to use 3-COLOR library to answer for the 3-SAT instance.

We'll want an assignment of variables to correspond to a coloring

So have a vertex for each variable so that you can color the graph iff you can make the expression true; colors should correspond to values (True, False, and…dummy?)

We'll tweak this later, but get this intuition first.

Idea

We’re going to need little subgraphs that make this happen.

We call them “gadgets.”

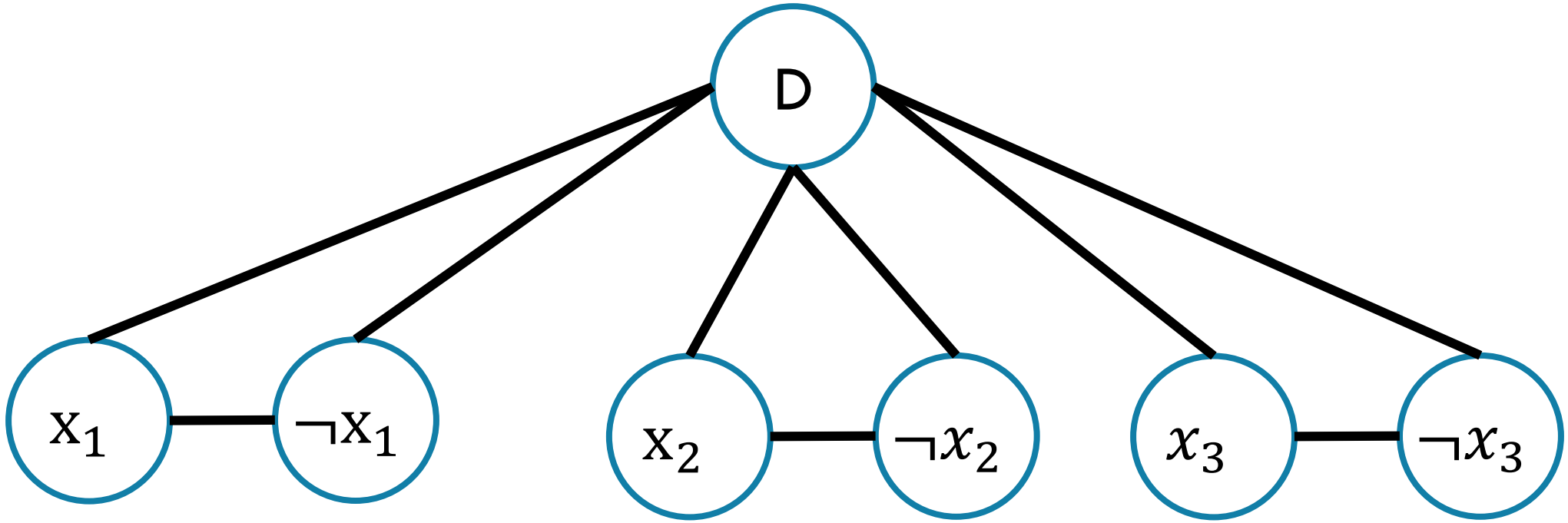
Gadget 1

Make the variables true and false.

Vertex for each **literal** (every vertex and its negation) attach x to $\neg x$

Need them to be different colors

And attach both to a shared vertex (the "dummy" color)



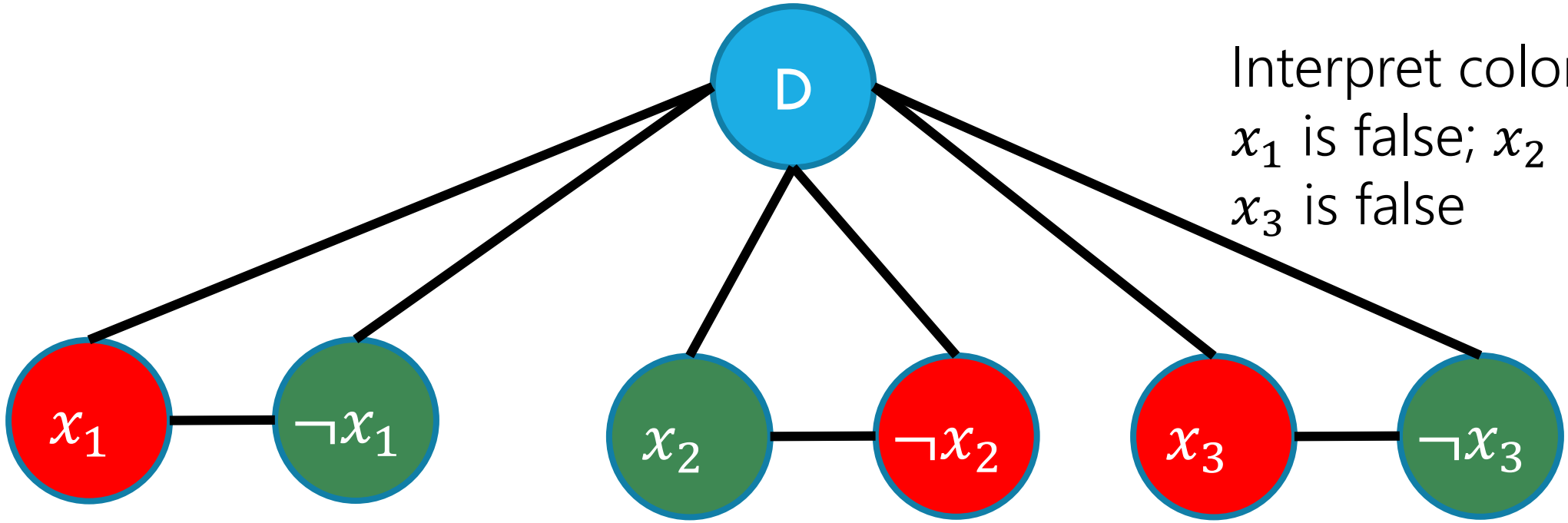
Gadget 1

Make the variables true and false.

Vertex for each **literal** (every vertex and its negation) attach x to $\neg x$

Need them to be different colors

And attach both to a shared vertex (the "dummy" color)



Interpret coloring as:
x_1 is false; x_2 is true;
x_3 is false

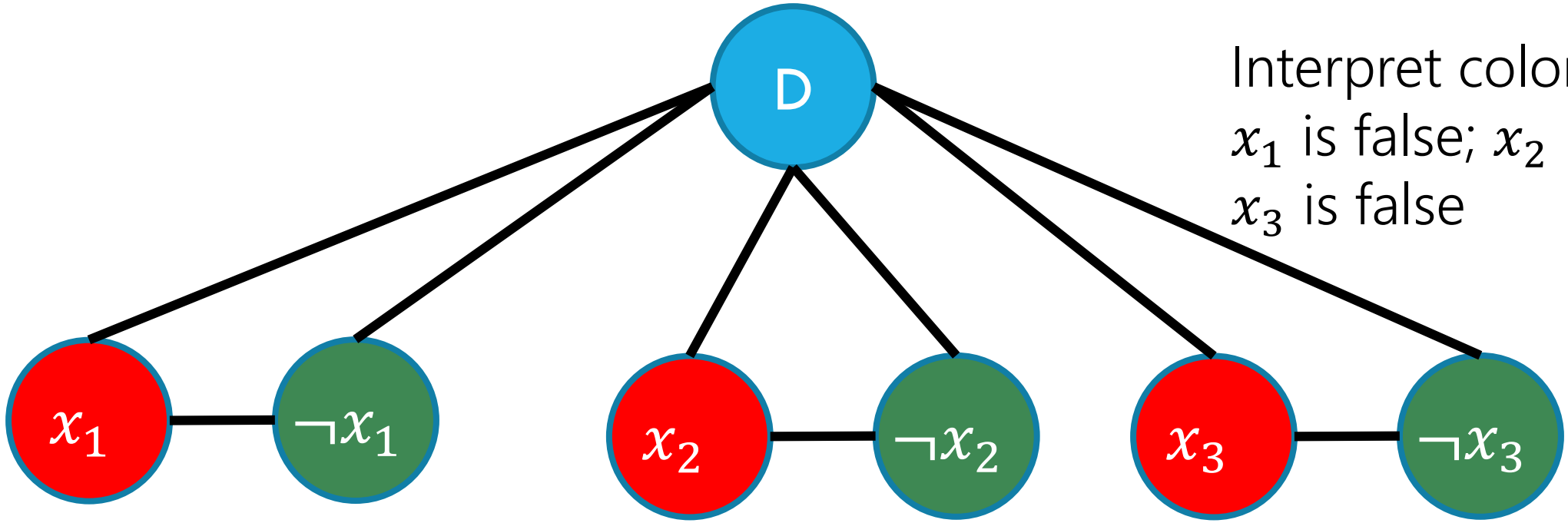
Gadget 1

Make the variables true and false.

Vertex for each **literal** (every vertex and its negation) attach x to $\neg x$

Need them to be different colors

And attach both to a shared vertex (the "dummy" color)



Interpret coloring as:
x_1 is false; x_2 is false;
x_3 is false

Are We Done?

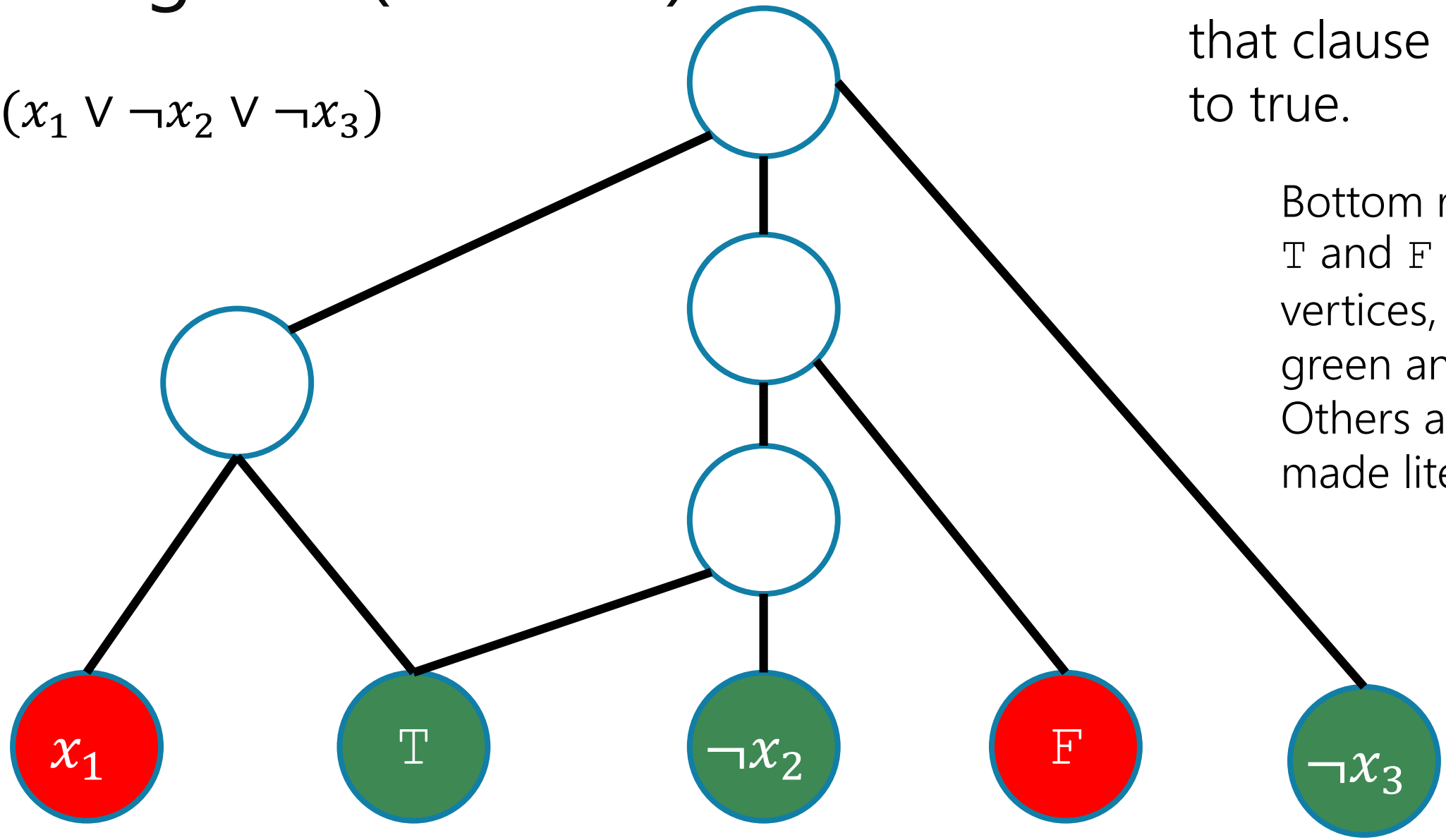
We can interpret a 3-coloring as a setting of the variables!

But, we're not done. The goal is to say the 3-coloring corresponds to a satisfying assignment. One that makes the CNF expression true!

Need to handle each clause

Gadget 2 (clauses)

$(x_1 \vee \neg x_2 \vee \neg x_3)$

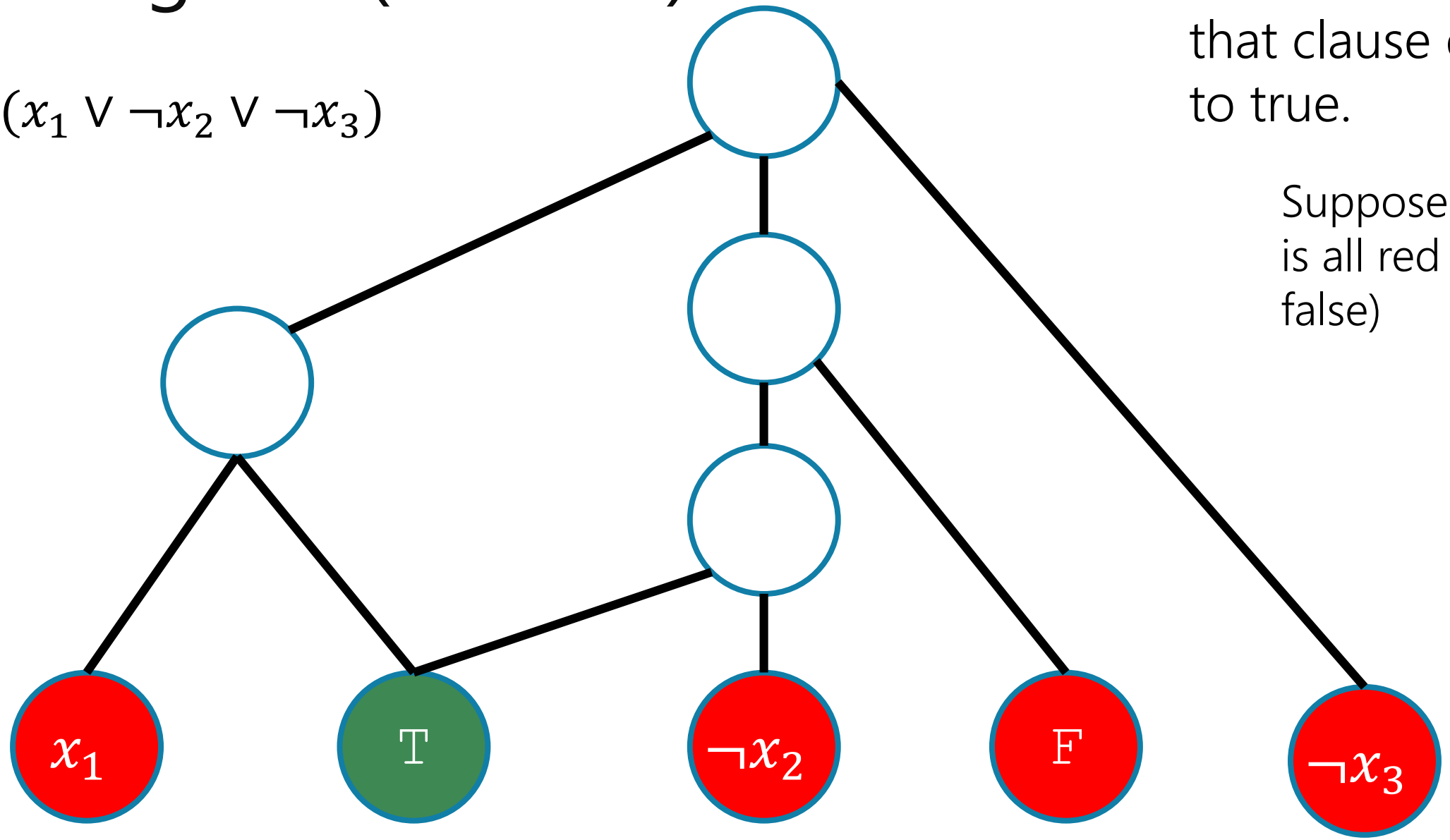


This tricky little graph can be 3-colored iff that clause evaluates to true.

Bottom row:
T and F are new vertices, colored green and red.
Others are already-made literal vertices

Gadget 2 (clauses)

$(x_1 \vee \neg x_2 \vee \neg x_3)$

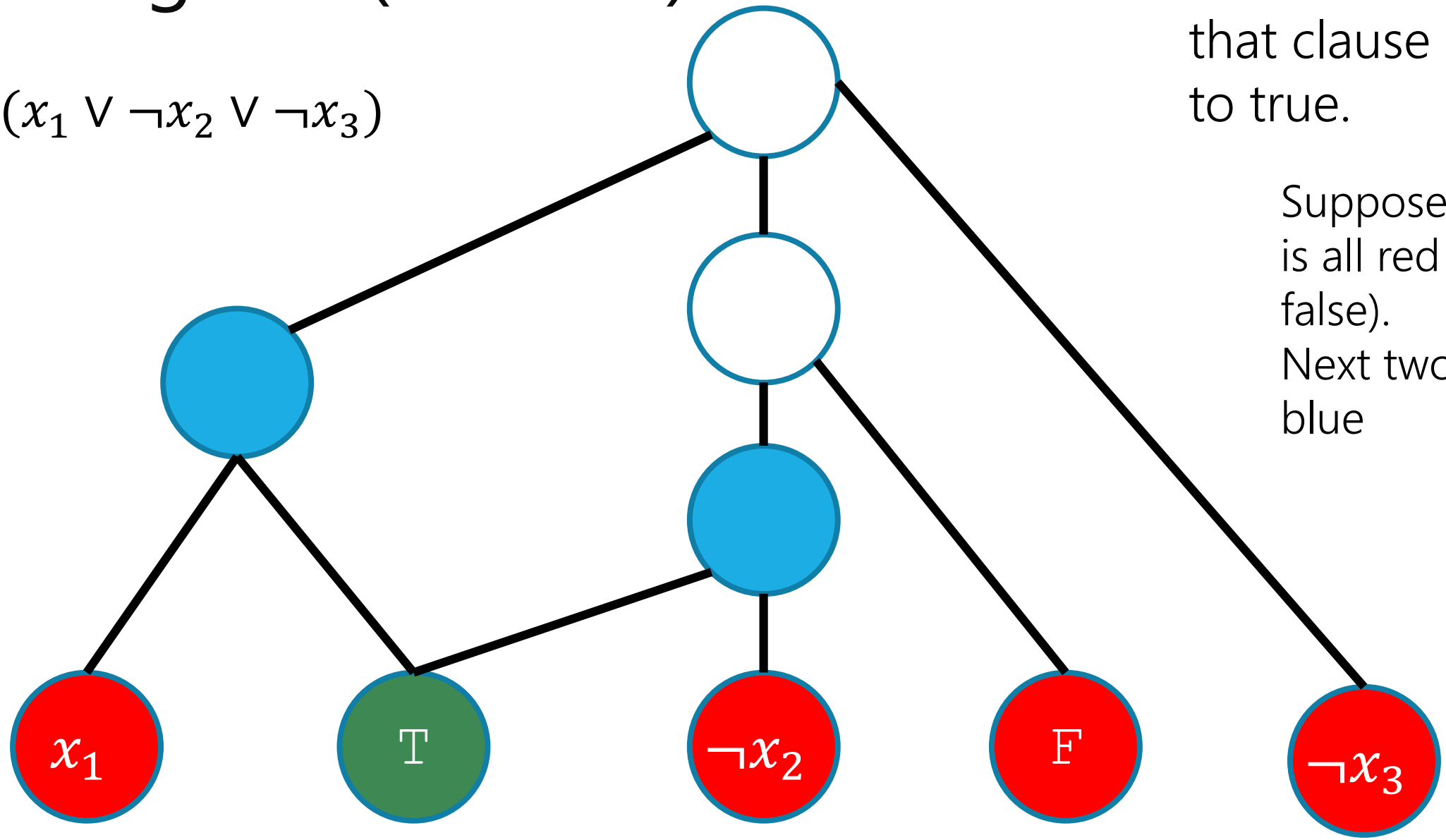


This tricky little graph can be 3-colored iff that clause evaluates to true.

Suppose bottom row is all red (clause is false)

Gadget 2 (clauses)

$(x_1 \vee \neg x_2 \vee \neg x_3)$

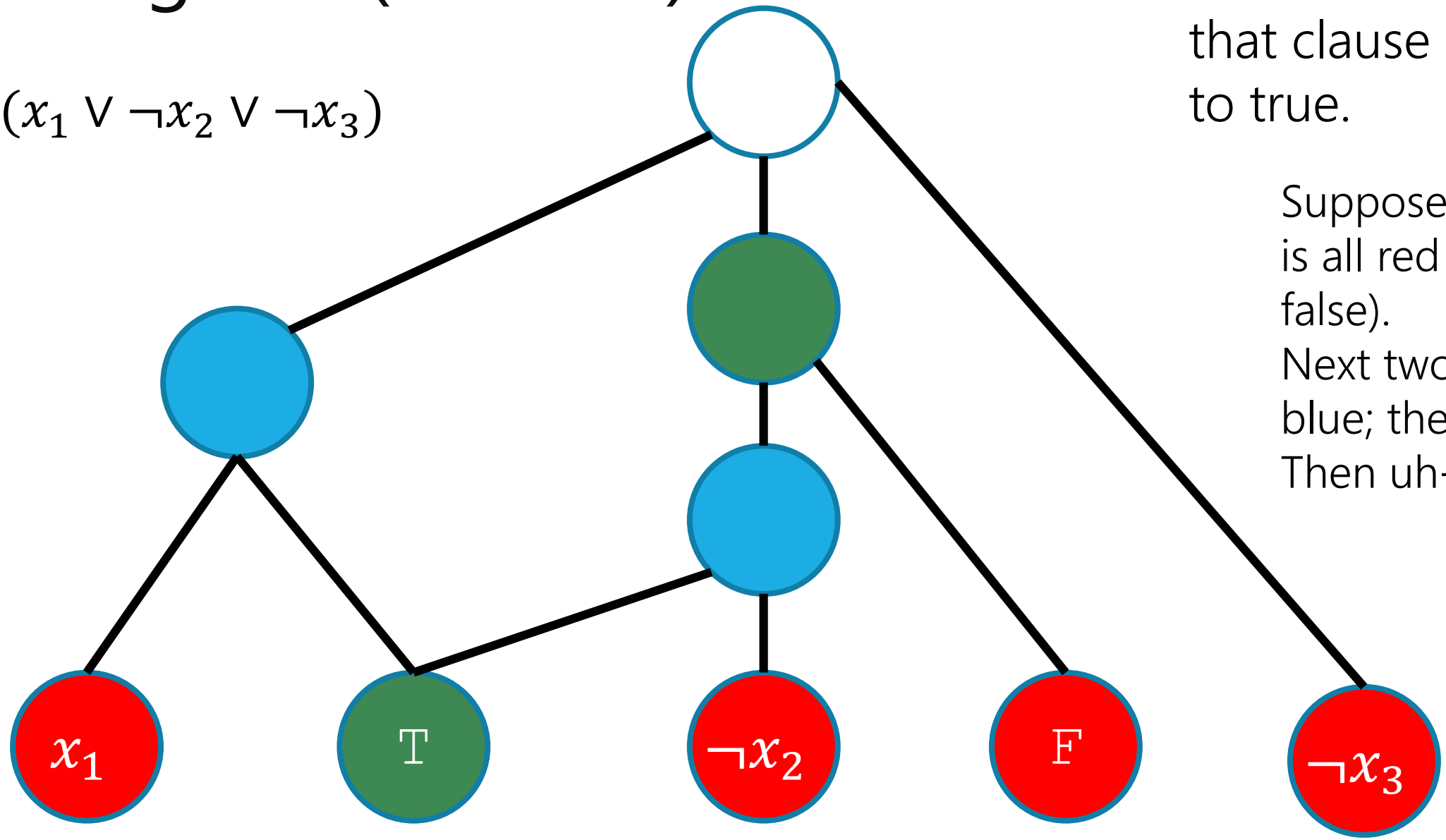


This tricky little graph can be 3-colored iff that clause evaluates to true.

Suppose bottom row is all red (clause is false).
Next two must be blue

Gadget 2 (clauses)

$(x_1 \vee \neg x_2 \vee \neg x_3)$

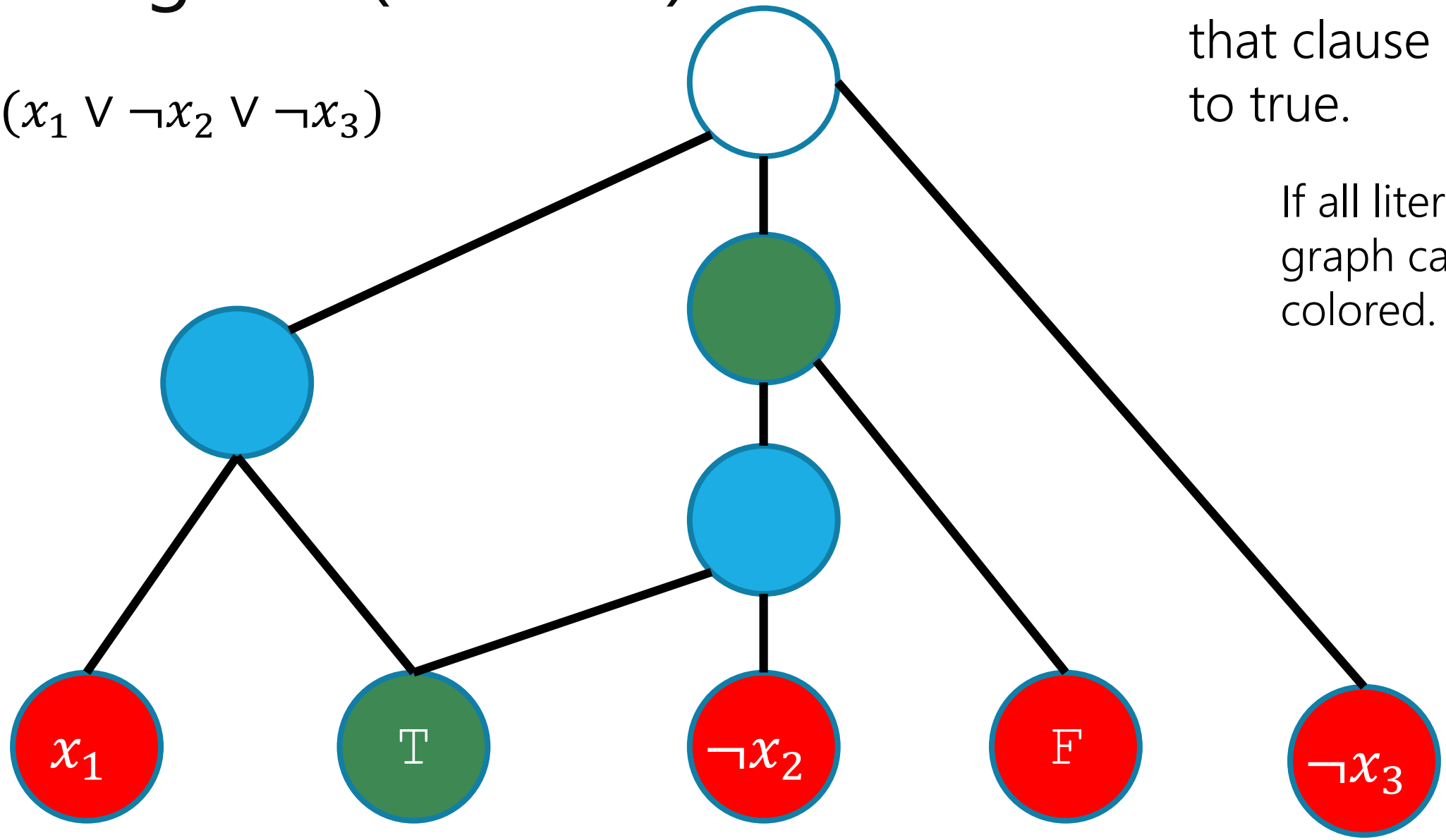


This tricky little graph can be 3-colored iff that clause evaluates to true.

Suppose bottom row is all red (clause is false).
Next two must be blue; then green;
Then uh-oh

Gadget 2 (clauses)

$(x_1 \vee \neg x_2 \vee \neg x_3)$

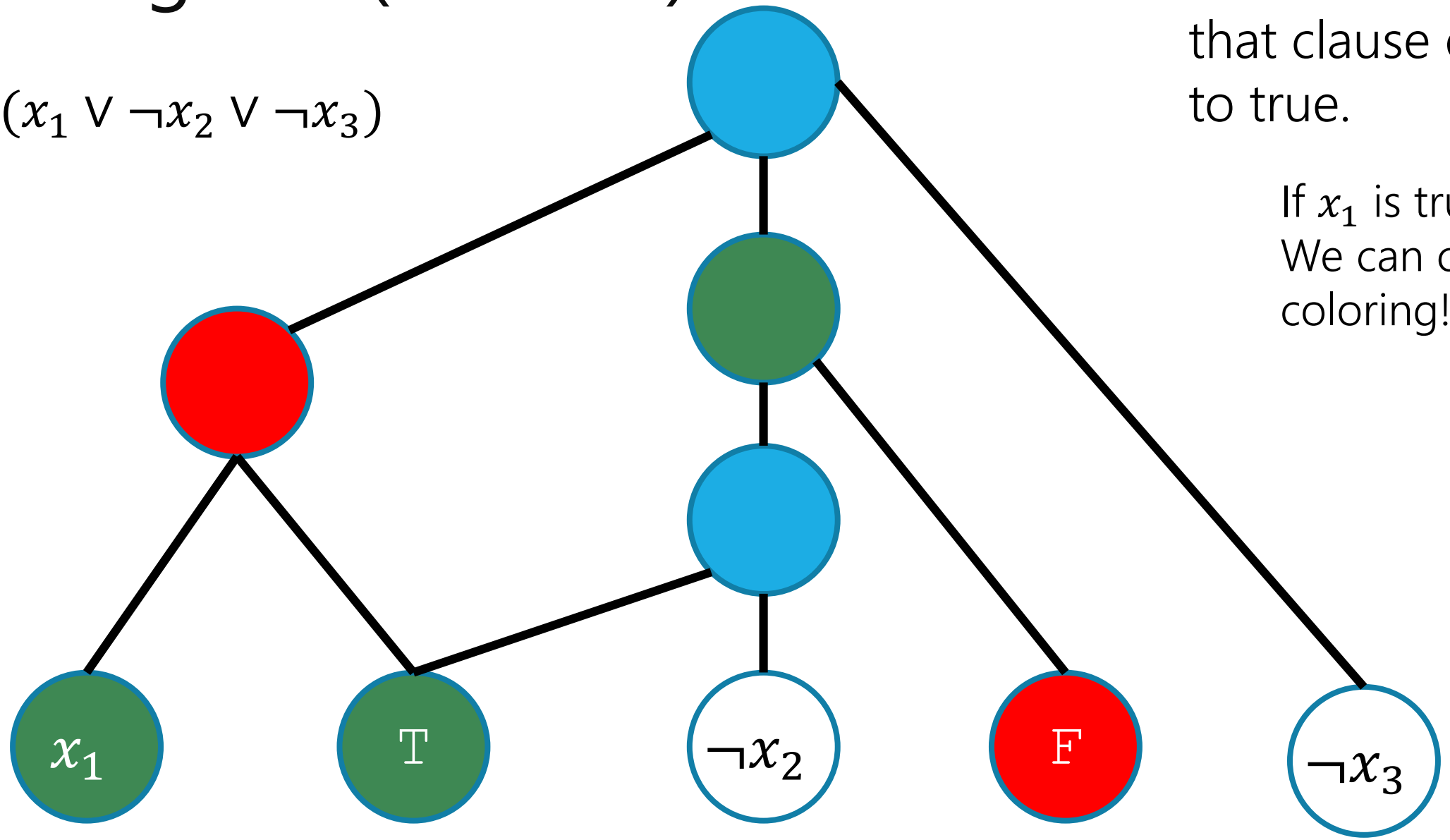


This tricky little graph can be 3-colored iff that clause evaluates to true.

If all literals are false, graph can't be 3-colored.

Gadget 2 (clauses)

$(x_1 \vee \neg x_2 \vee \neg x_3)$

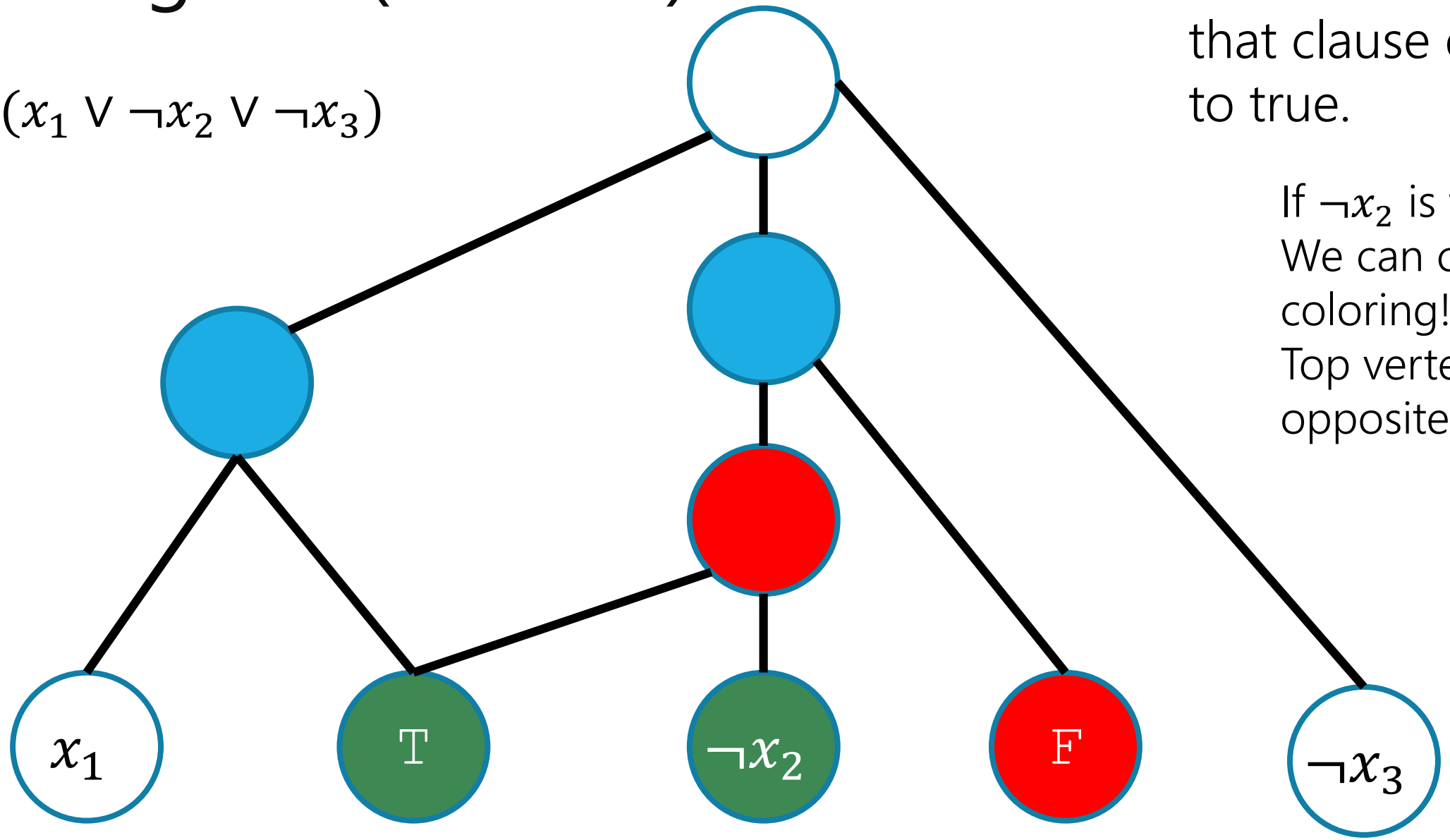


This tricky little graph can be 3-colored iff that clause evaluates to true.

If x_1 is true…
We can complete the coloring!

Gadget 2 (clauses)

$(x_1 \vee \neg x_2 \vee \neg x_3)$

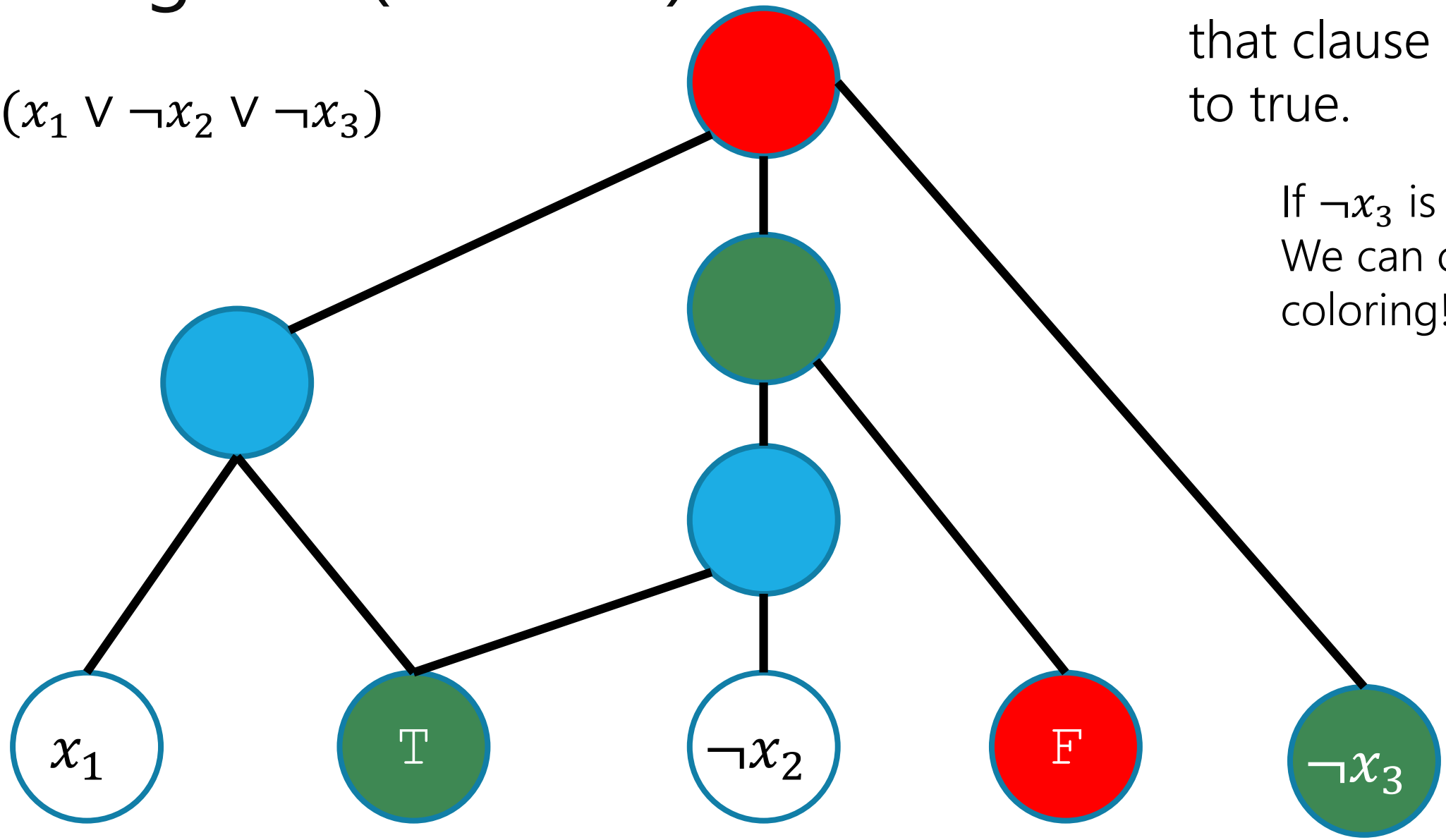


This tricky little graph can be 3-colored iff that clause evaluates to true.

If $\neg x_2$ is true…
We can complete the coloring!
Top vertex is opposite of $\neg x_3$

Gadget 2 (clauses)

$(x_1 \vee \neg x_2 \vee \neg x_3)$

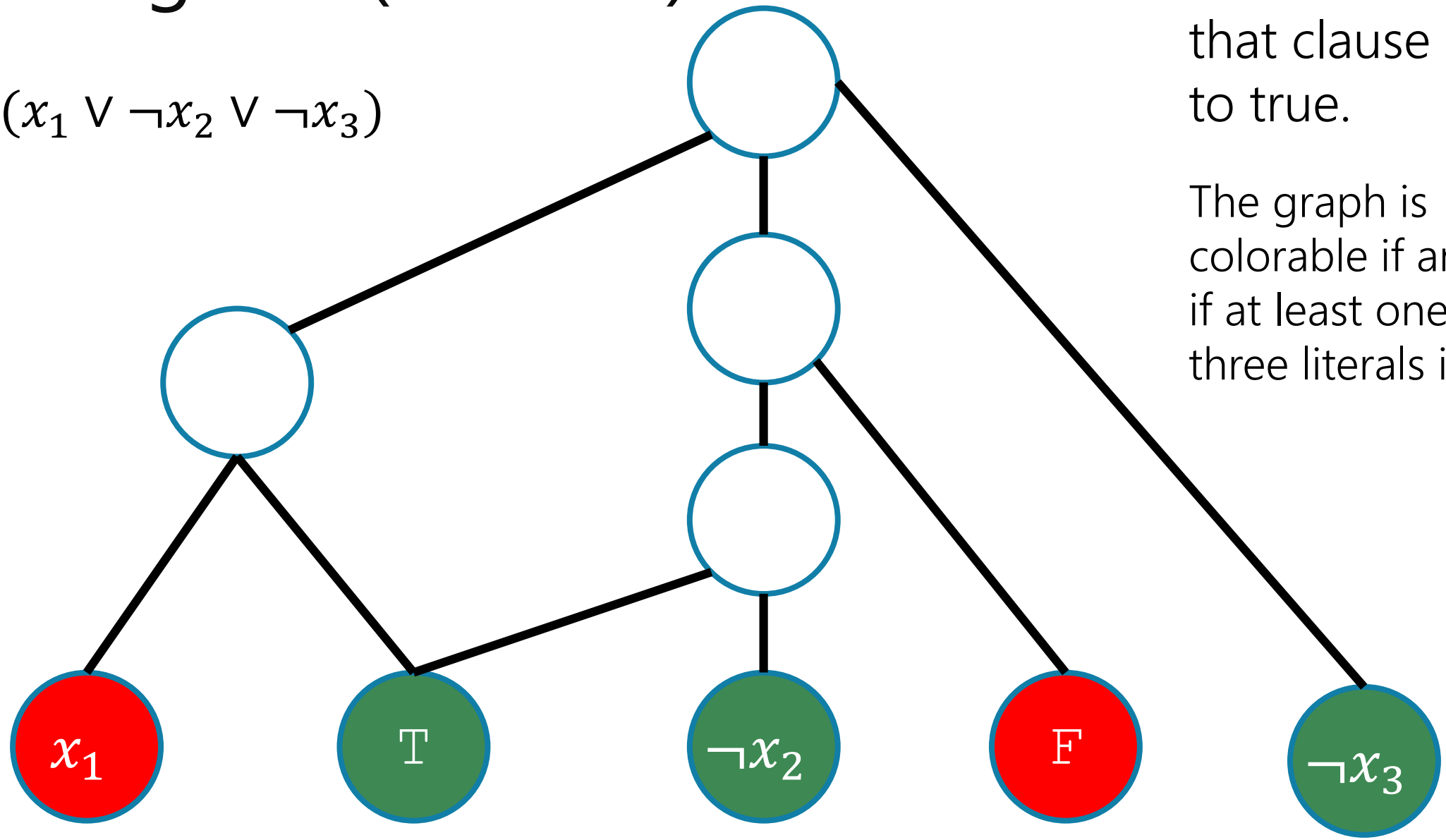


This tricky little graph can be 3-colored iff that clause evaluates to true.

If $\neg x_3$ is true…
We can complete the coloring!

Gadget 2 (clauses)

$(x_1 \vee \neg x_2 \vee \neg x_3)$



This tricky little graph can be 3-colored iff that clause evaluates to true.

The graph is colorable if and only if at least one of the three literals is green.

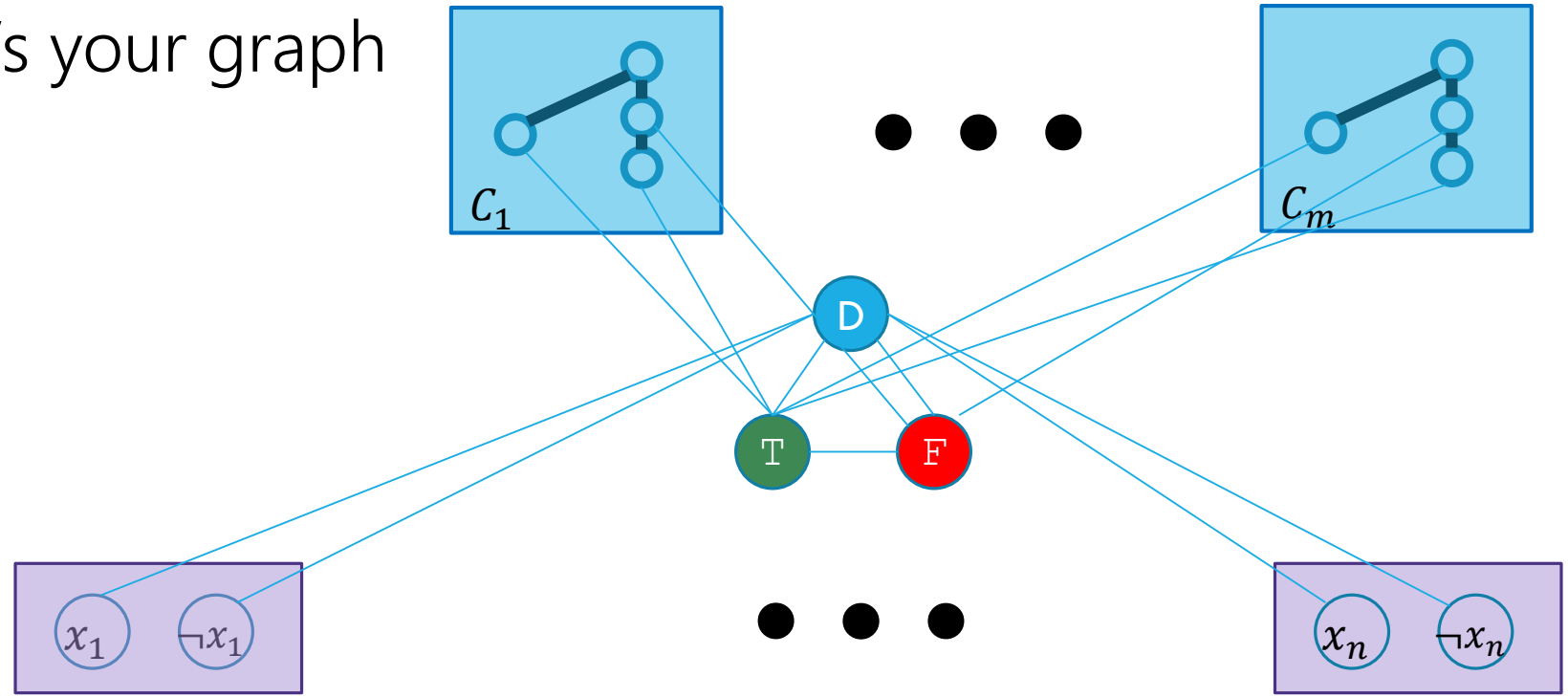
Putting it together

Make a vertex for every literal

Make one of those subgraphs for **every** clause

Make T,F, Dummy vertices and connect them as shown.

That's your graph



Putting it together

If there is a satisfying assignment, then the graph is 3-colorable.

Putting it together

If there is a satisfying assignment, then the graph is 3-colorable.

Consider a satisfying assignment. Assign all true literals and T to be green, assign all false literals and F to be red, assign D to be blue.

Now consider the clause gadgets. We saw that if at least one literal vertex is green, we can color the remaining vertices via case analysis. Since we have a satisfying assignment, each clause gadget has a green colored node, so we can complete the coloring. This is a 3-coloring of the full graph.

Putting it together

If the graph is 3-colorable, then there must be a satisfying assignment

Putting it together

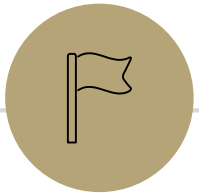
If the graph is 3-colorable, then there must be a satisfying assignment.

Consider a valid 3-coloring. The three vertices $\mathtt{T}, \mathtt{F}$,D all must have different colors (since they are all adjacent). Call $\mathtt{T}$'s color "green", $\mathtt{F}$'s "red" and D's "blue." Since we put edges between x_i and $\neg x_i$, literals always get opposite colors, and since all are attached to D each gets red or green. Observe that every gadget is properly colored (as we colored the full graph), thus by our case analysis, each gadget must have at least one green vertex among the three literals. Set the variables to be true if their vertex is green and false if red (since we put edges between opposites this is consistent). Since each clause has a green vertex, every clause has a true literal and the assignment is satisfying for the 3-SAT instance.

Putting it together

The graph can be constructed in polynomial time.

There are a constant number of vertices per clause or variable of the SAT instance, and it's a mechanical process to create the edges, so the total time is polynomial. We call the library only once, which is polynomial as well.



More Reduction Facts

Some New Problems

Here are some new problems. Are they in NP?

If they're in NP, what is the "certificate" when the answer is yes?

COMPOSITE – given an integer n is it composite (i.e. not prime)?

MAX-FLOW – find a maximum flow in a graph.

VERTEX-COVER – given a graph G and an integer k, does G have a vertex cover of size at most k?

NON-3-Color – given a graph G, is it not 3-colorable?

Some New Problems

COMPOSITE – given an integer n is it composite (i.e. not prime)?

In NP (certificate is factors).

MAX-FLOW – find a maximum flow in a graph.

Not in NP (not a decision problem)

VERTEX-COVER – given a graph G and an integer k, does G have a vertex cover of size at most k?

In NP (certificate is cover)

NON-3-Color – given a graph G, is it not 3-colorable?

Not known to be in NP.

I have a problem

My problem C is too difficult to solve (at least for me).

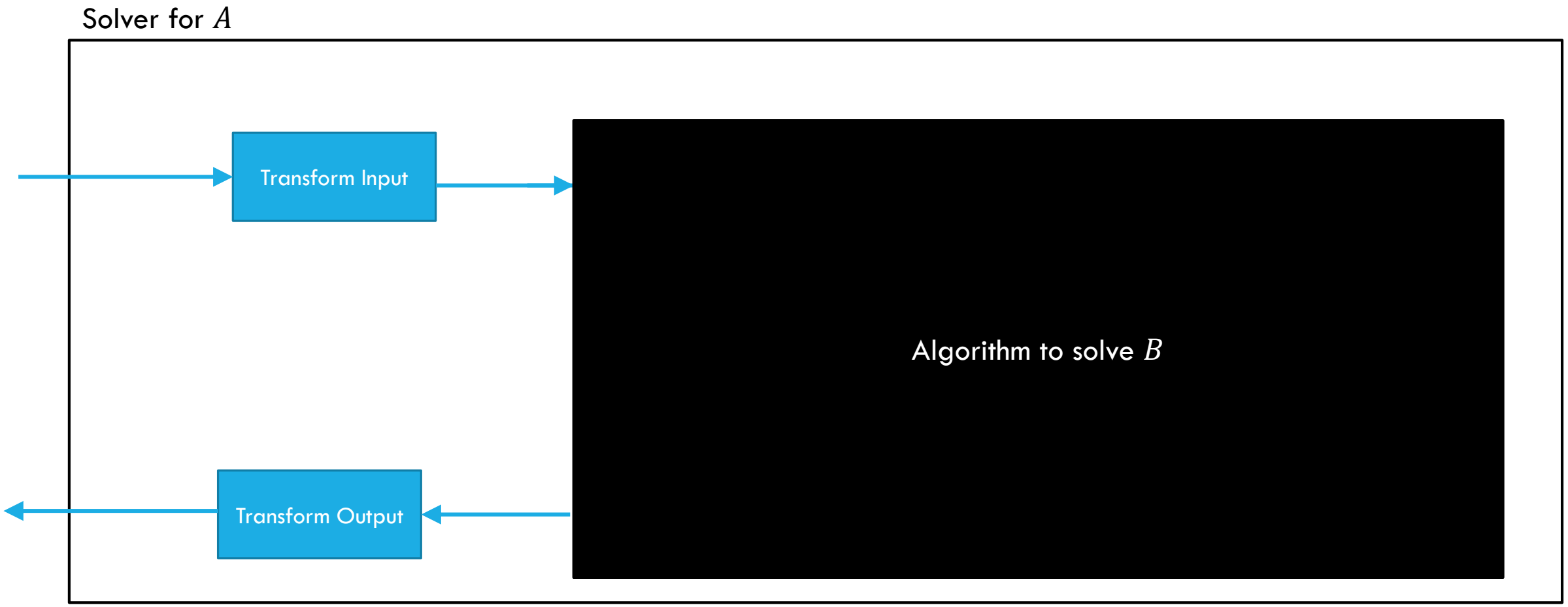
So difficult, it's probably NP-hard. How do I show it?

What does it mean to be NP-hard?

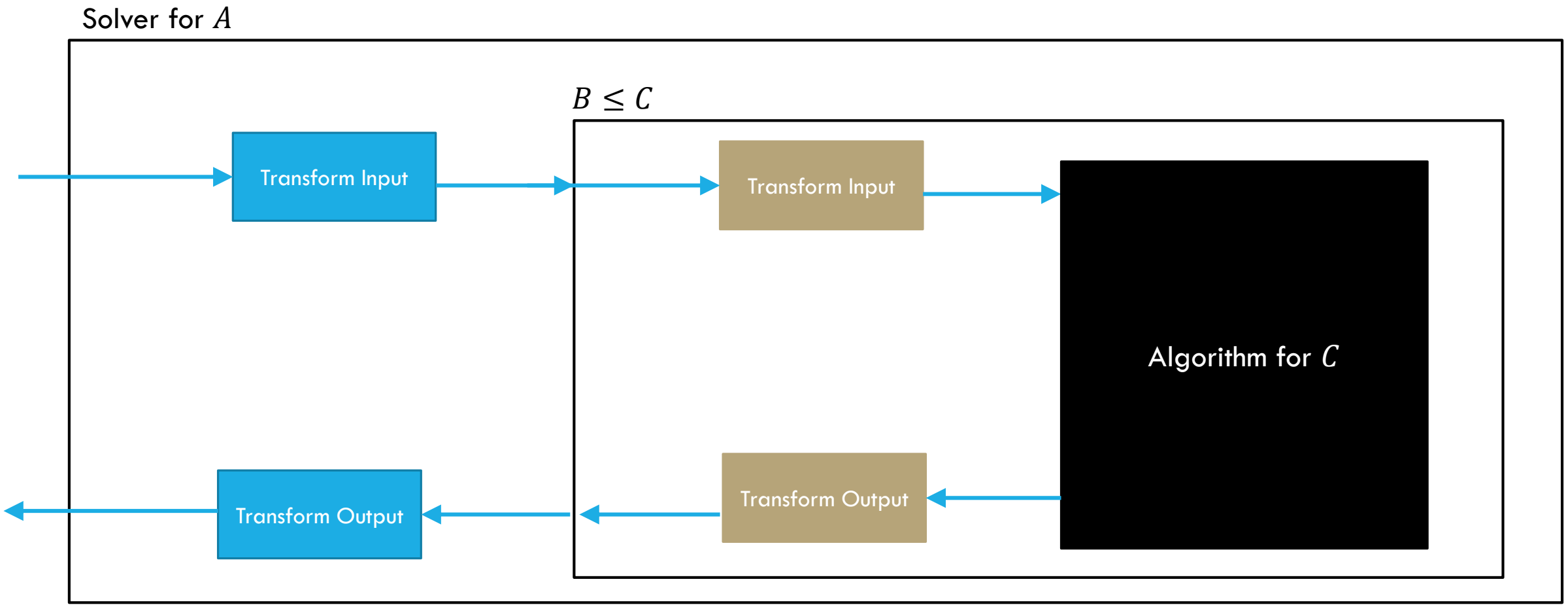
We need to be able to reduce any problem A in $\mathbf{NP}$ to C.

Let's choose B to be a **known** NP-hard problem. Since B **is known** to be NP-hard, $A \leq B$ for every possible A. So if **we show** $B \leq C$ too then $A \leq B \leq C \rightarrow A \leq C$ so every NP problem reduces to C!

Is the implication true? $A \leq B \leq C \rightarrow A \leq C$



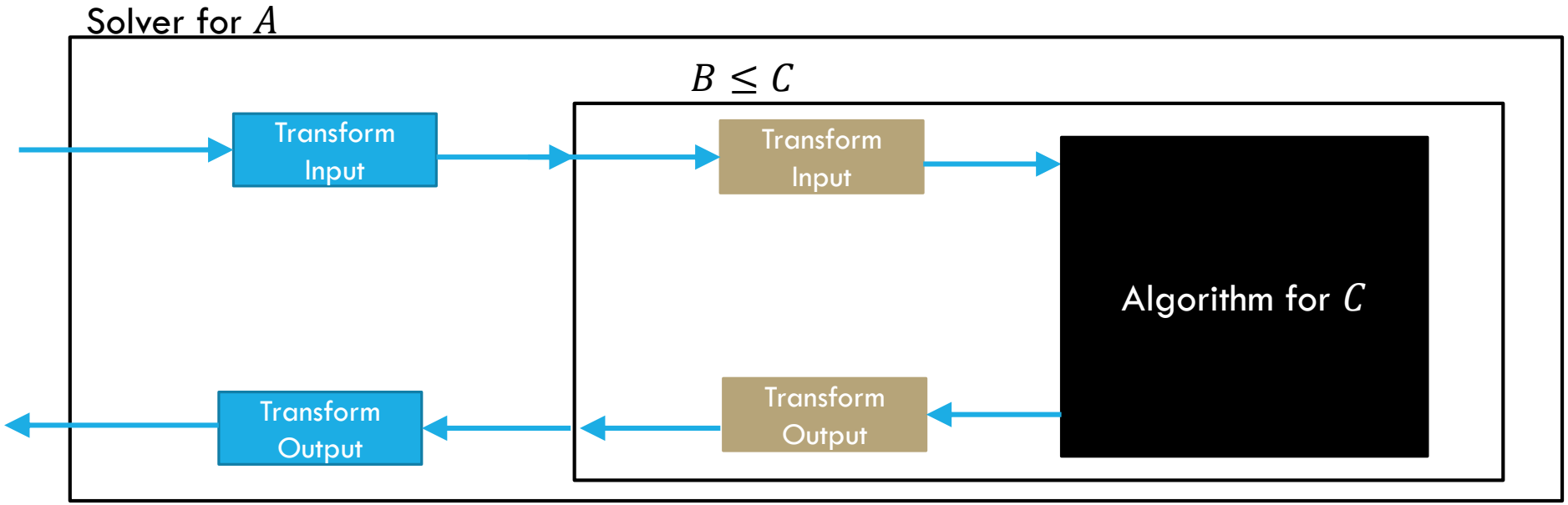
Is the implication true? $A \leq B \leq C \rightarrow A \leq C$



Is the implication true? $A \leq B \leq C \rightarrow A \leq C$

Why does it work? Because our reductions work!

How long does it take? We need polynomially many calls to B, each requires polynomially many calls to C. That's still polynomial. Similarly running time is polynomial times a polynomial, so a polynomial.



Said Differently

$A \leq B$

If I know B is not hard [I have an algorithm for it] then A is also not hard.

This is how we usually use reductions

$A \leq B$

If I know A is hard, then B also must be hard.

(contrapositive of the last statement)

Want to prove your problem is hard?

To show B is hard,

Reduce **FROM** the known hard problem **TO** the problem you care about

A reduction **From** an NP-hard problem A to B, shows B is also NP-hard.

P (stands for "Polynomial")

The set of all decision problems that have an algorithm that runs in time $O(n^k)$ for some constant k (on input of size n).

NP (stands for "nondeterministic polynomial")

The set of all decision problems such that for every YES-instance (of size n), there is a certificate (of size $O(n^k)$) for that instance which can be verified in polynomial time.

NP-hard

The problem B is NP-hard if
for all problems A in NP, A reduces to B.

NP-Complete

The problem B is NP-complete if B is in NP
and B is NP-hard