# Even More Reductions

CSE 421 Autumn 22
Lecture 27

# Today

Reduction between problems that *look* *very* different.

A few small things to wrap up

Are reductions really transitive?

Some edge cases to the definitions of P, NP, etc.

# 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

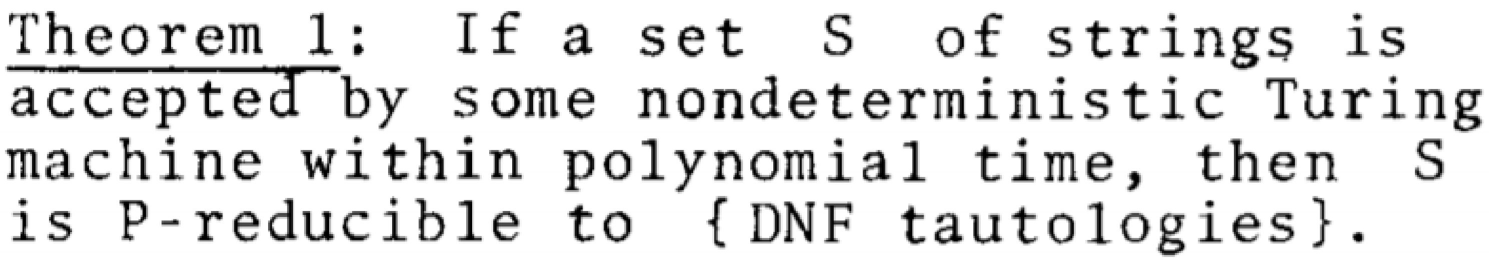
Takes at most polynomial-time otherwise excluding the calls to the library.

# 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 $x_1, \dots, x_n$

“AND” of “ORs”
∧ outside parens
∨ inside parens

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

One of the subexpressions inside parens

Something like:

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

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?

# 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!

# 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; $3\text{-SAT} \leq_P 3\text{-COLOR}$

# 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…

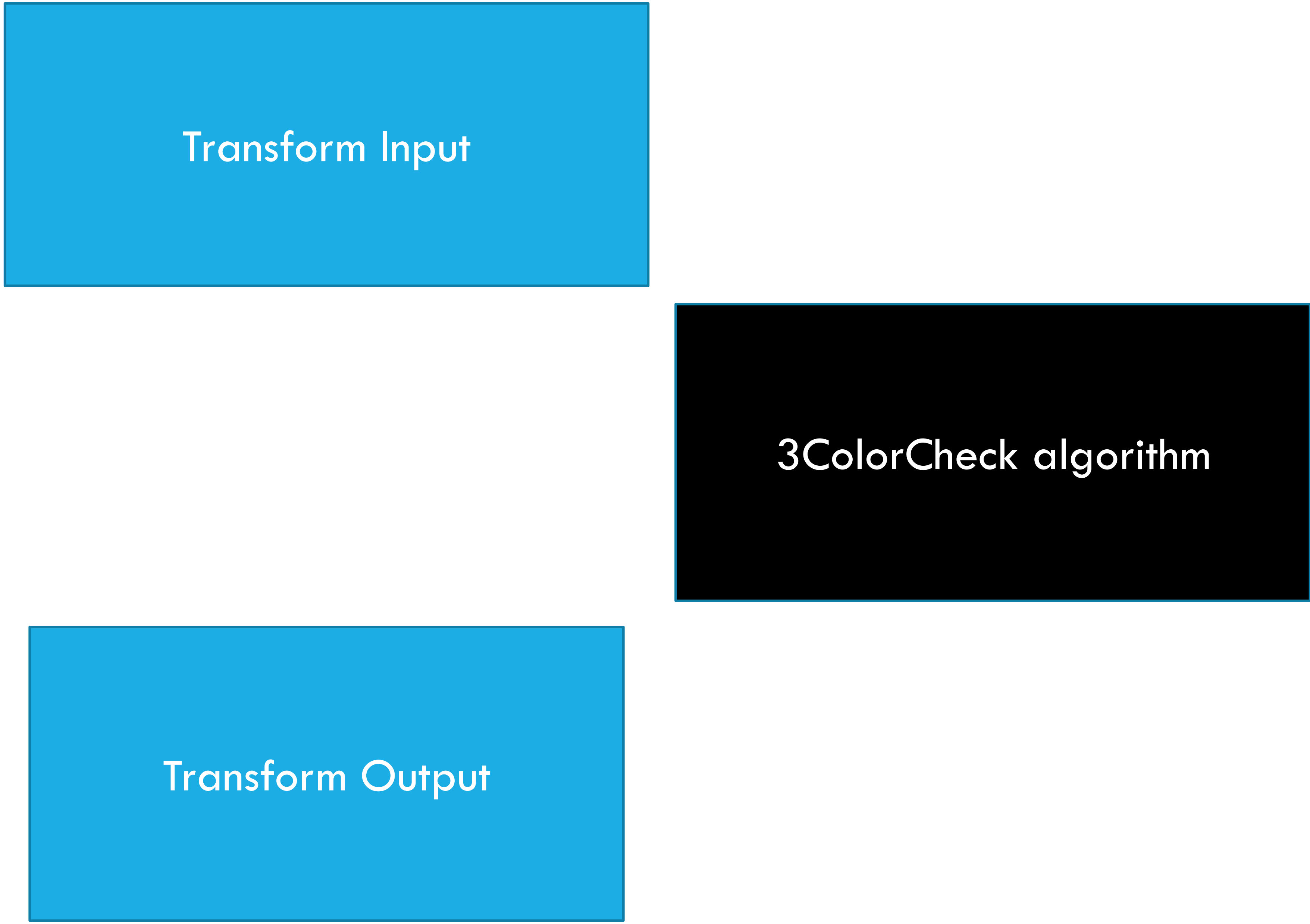
In $3$-SAT we assign variables to true and false.

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

How could we do that?

# Idea…

$$
\begin{aligned}
&(\neg x_1 \lor x_3 \lor x_5) \\
&\land (\neg x_1 \lor \neg x_3 \lor \neg x_5) \\
&\land (x_1 \lor x_2 \lor x_3) \\
&\land (\neg x_2 \lor x_3 \lor x_4)
\end{aligned}
$$



Transform Input

3ColorCheck algorithm

Transform Output

# 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.

Transform certificate into certificate

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.”

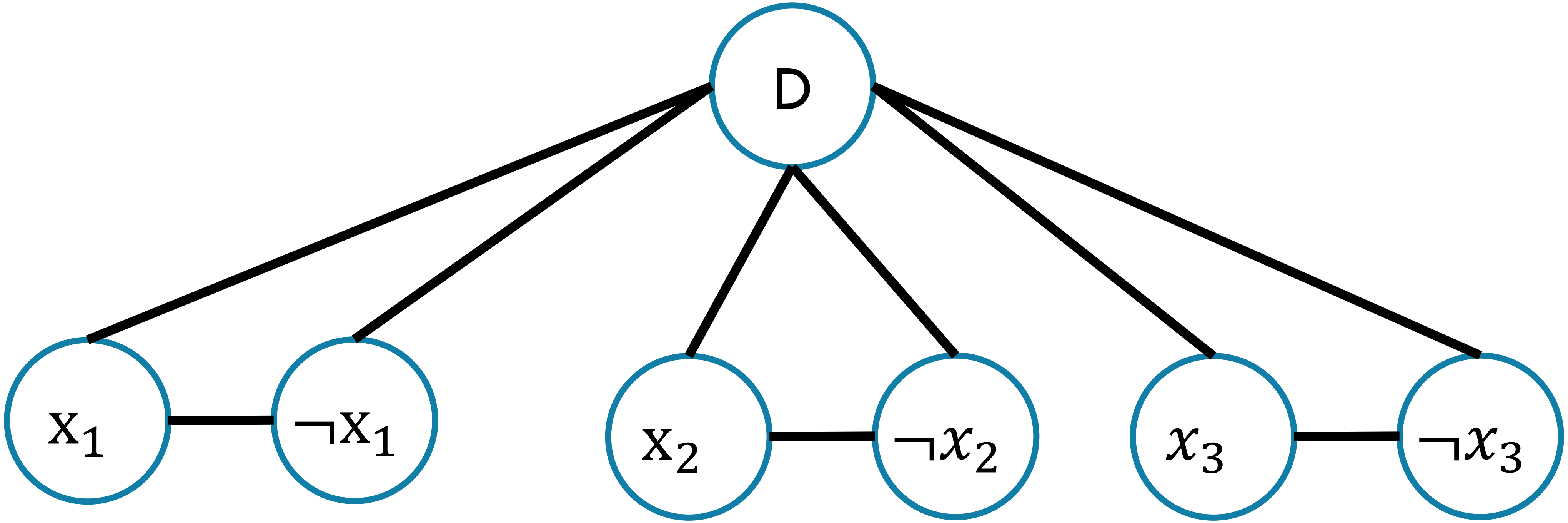
# Gadget 1

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)



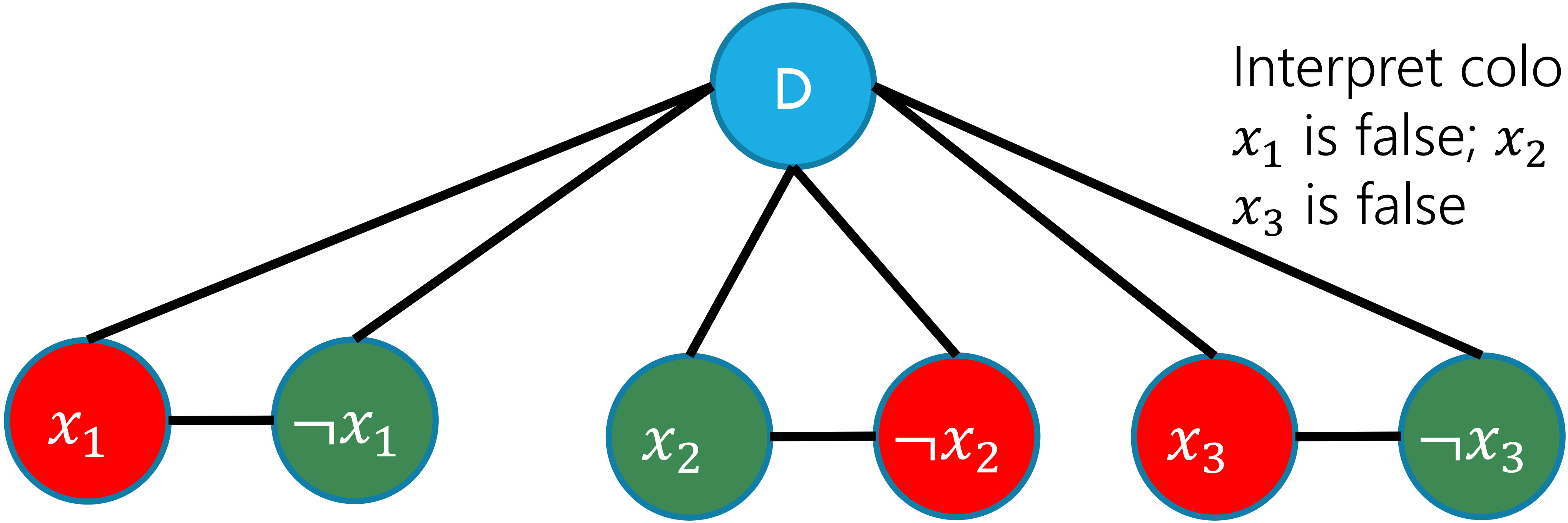
# Gadget 1

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

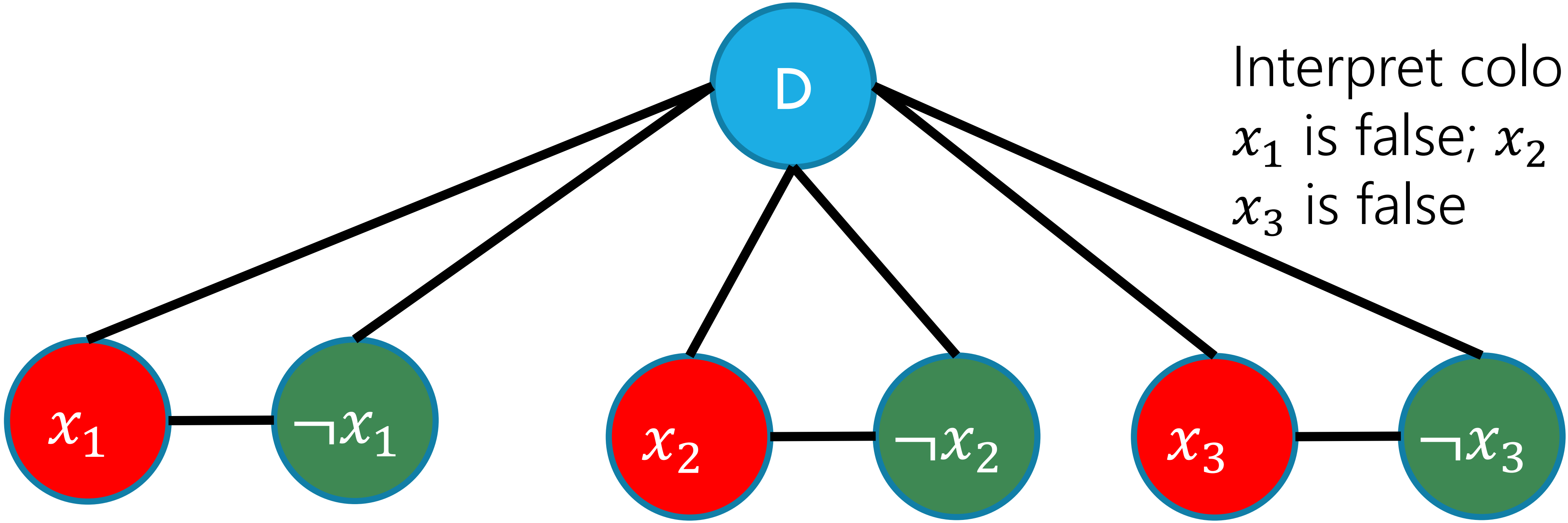
# Gadget 1

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?

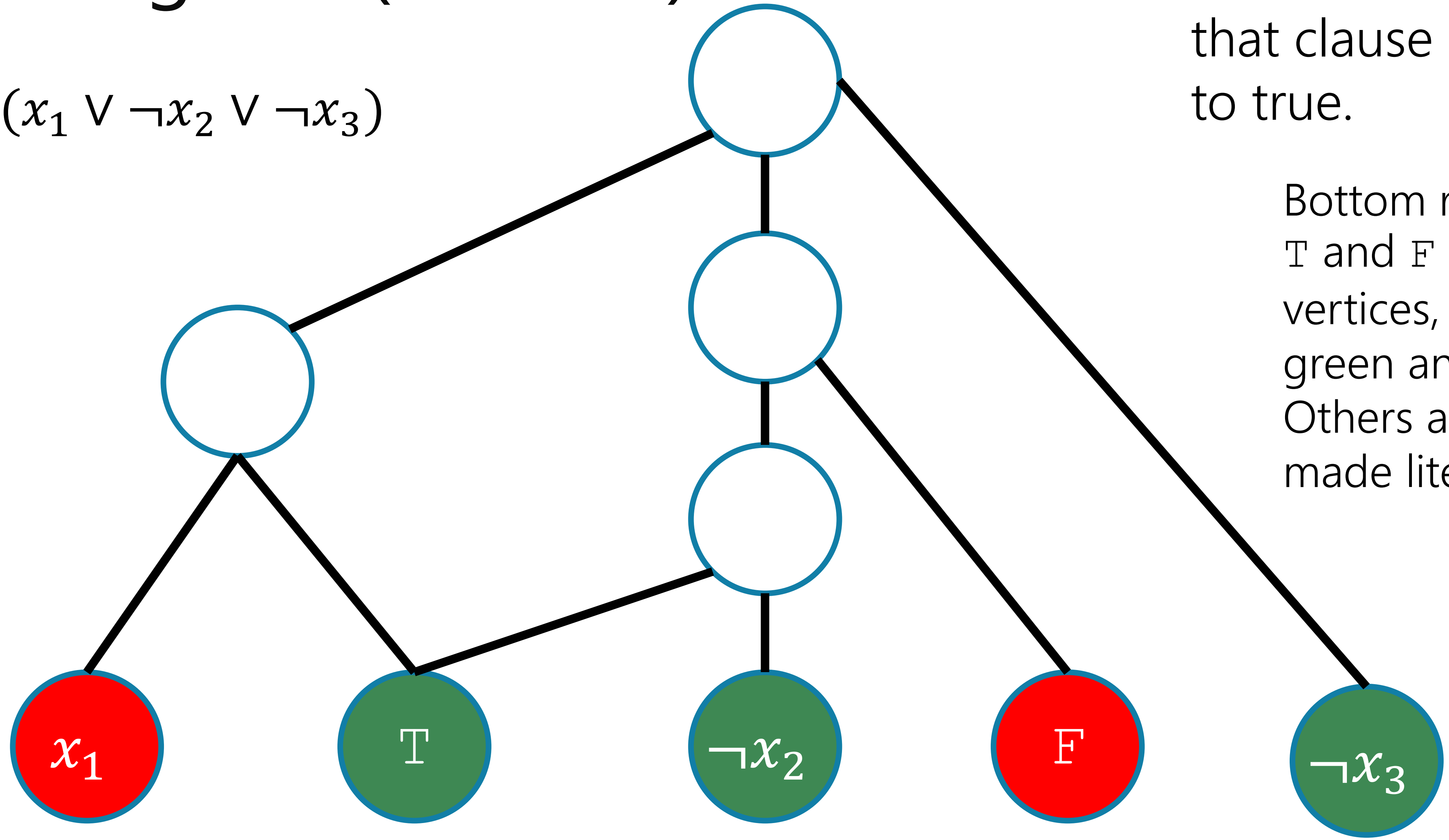
We can interpret a 3-coloring as a setting of the variables!

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)

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

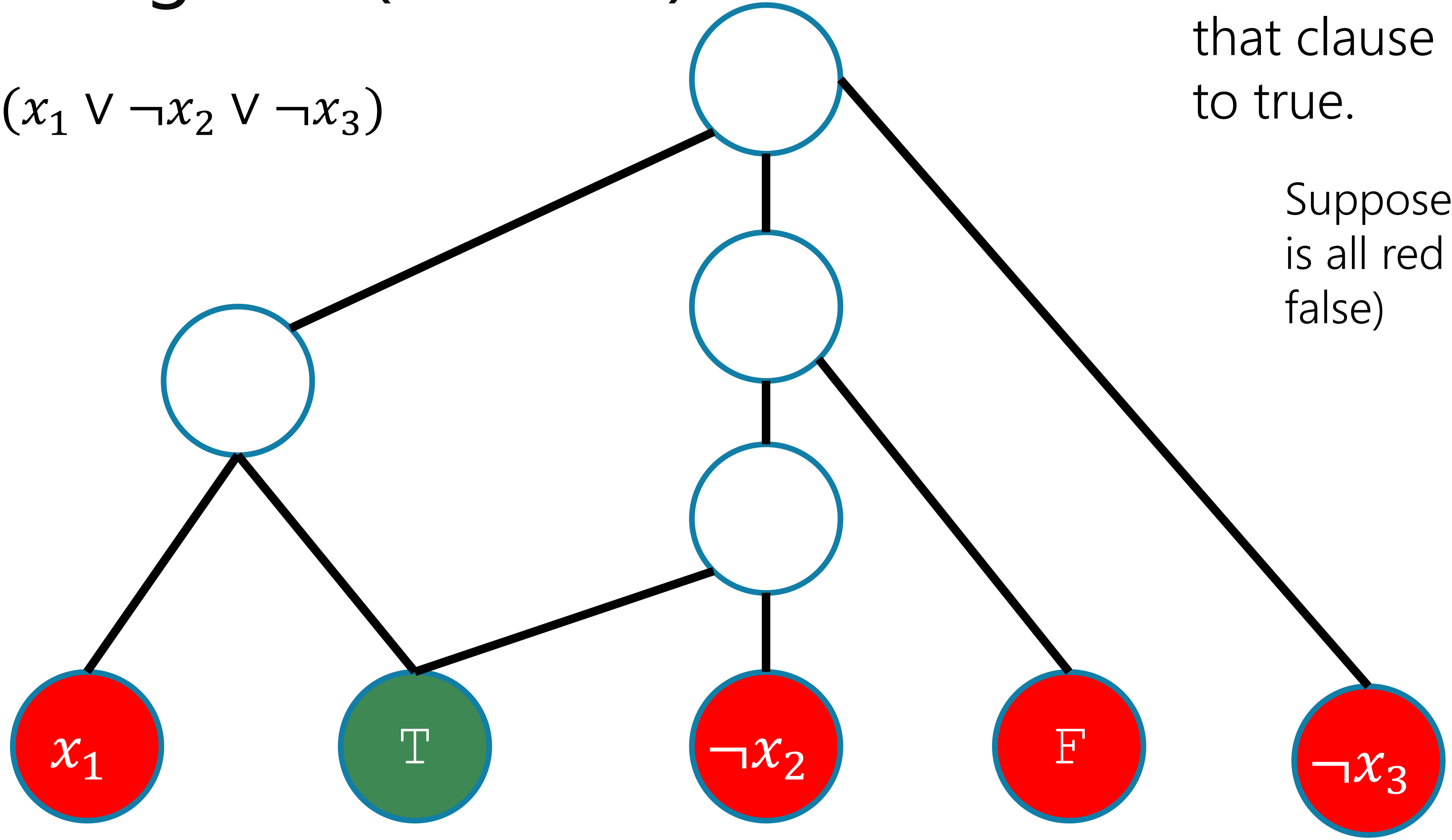


This tricky little graph can be 3-colored iff that clause evaluates to true.

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

# Gadget 2 (clauses)

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

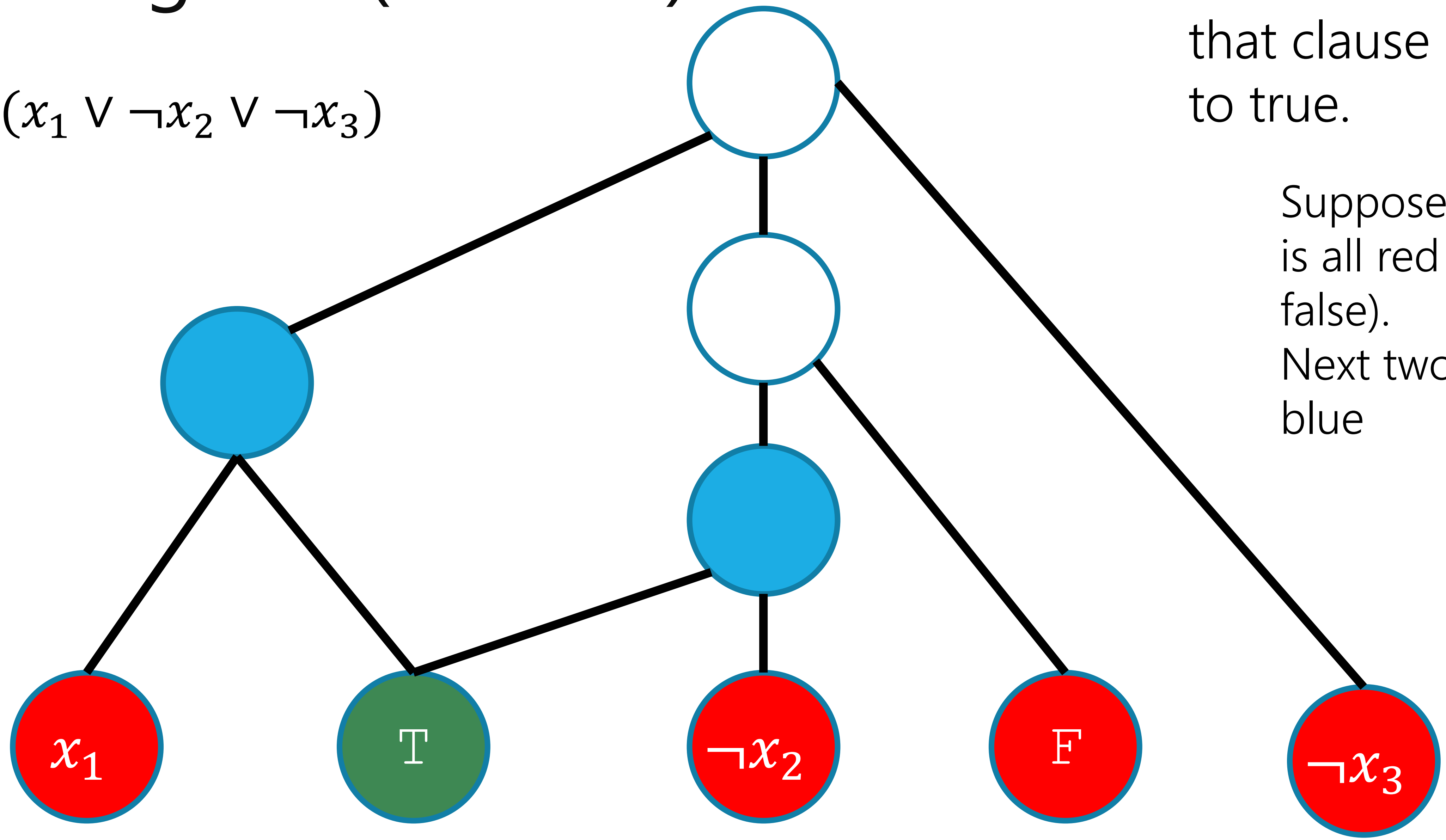


This tricky little graph can be 3-colored iff that clause evaluates to true.

Suppose bottom row is all red (clause is false)

# Gadget 2 (clauses)

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

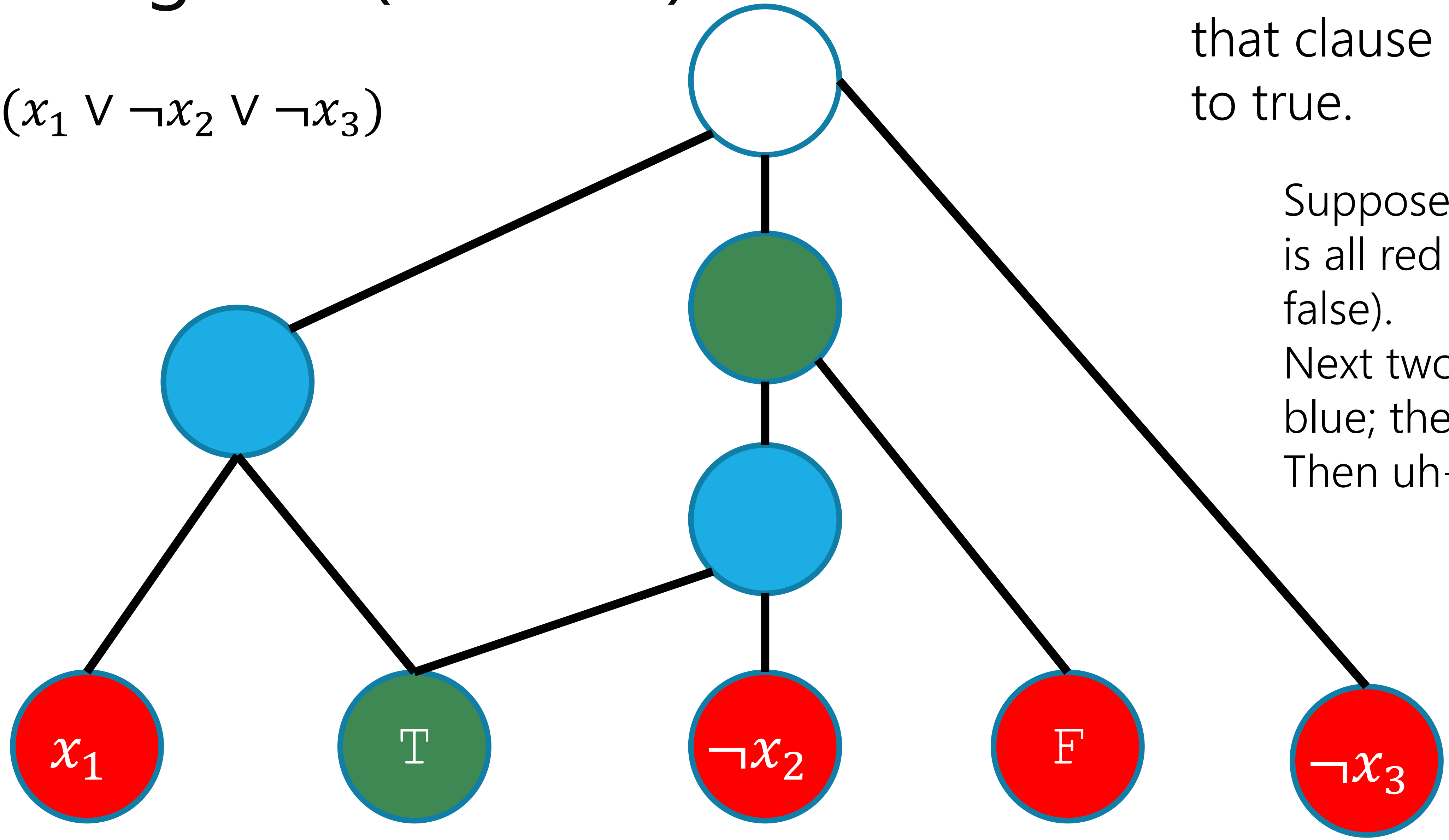


This tricky little graph can be 3-colored iff that clause evaluates to true.

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

# Gadget 2 (clauses)

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

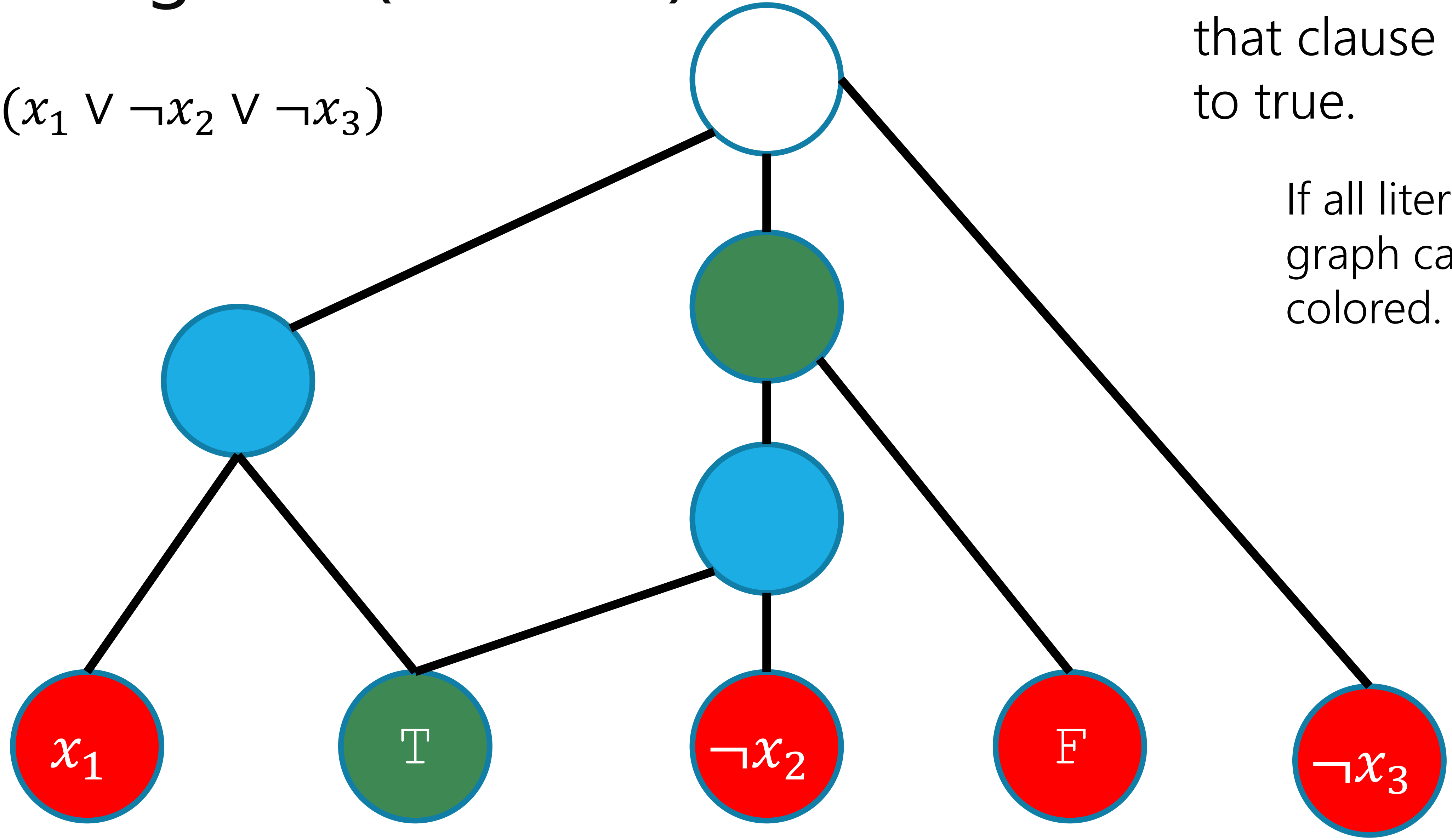


This tricky little graph can be 3-colored iff that clause evaluates to true.

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

# Gadget 2 (clauses)

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

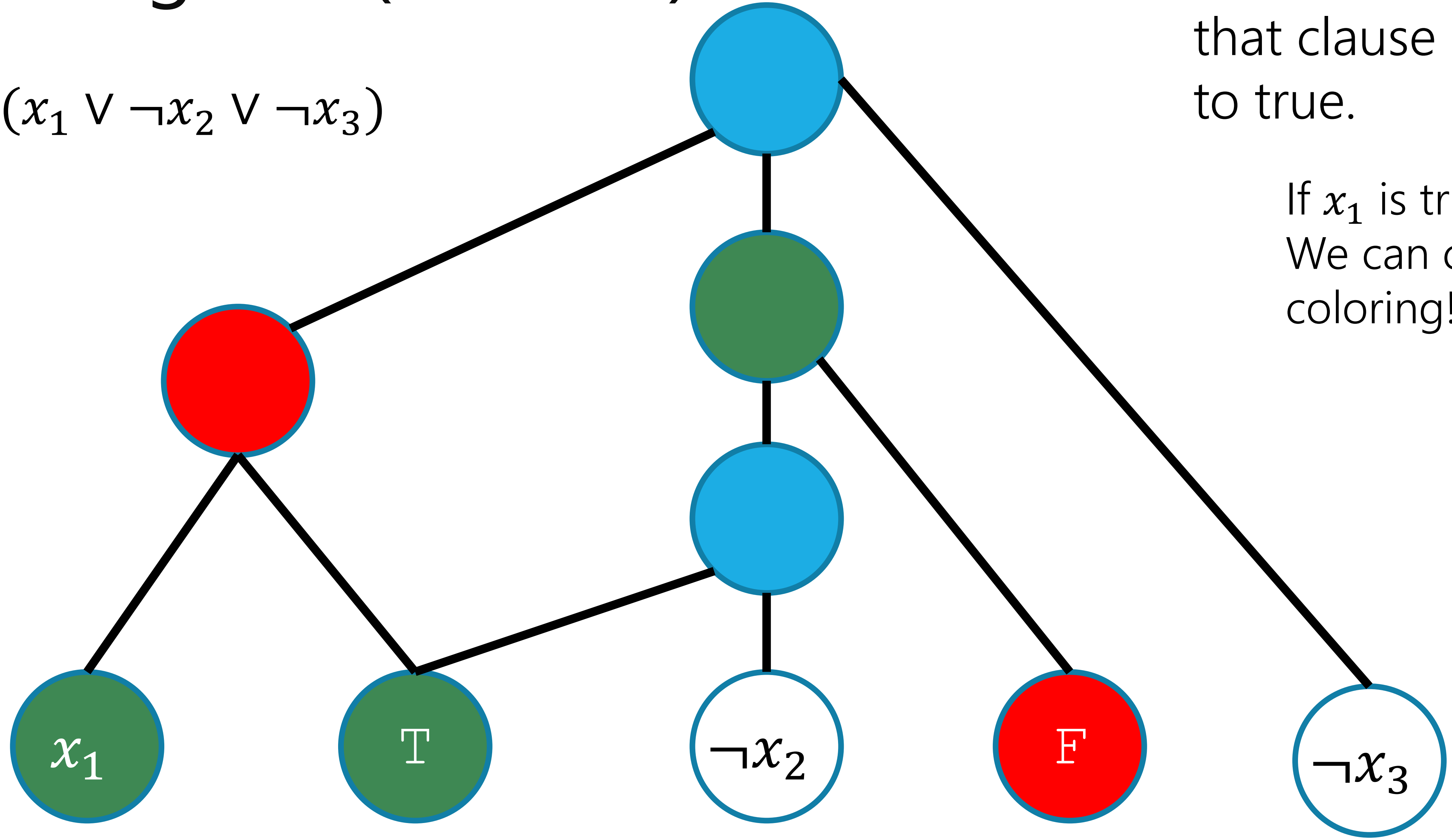


This tricky little graph can be 3-colored iff that clause evaluates to true.

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

# Gadget 2 (clauses)

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



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

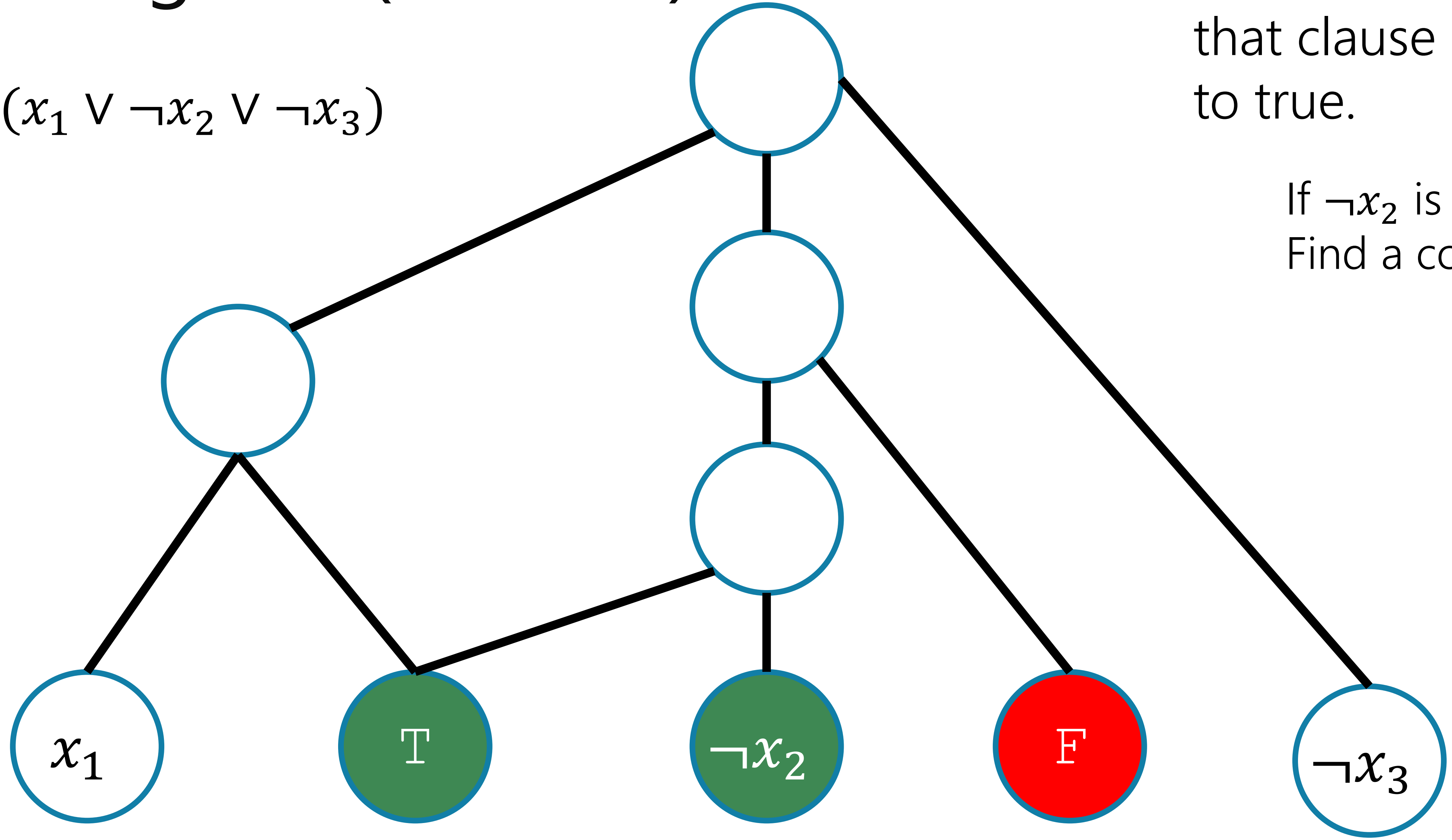
If $x_1$ is true…
We can complete the coloring!

# Gadget 2 (clauses)

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

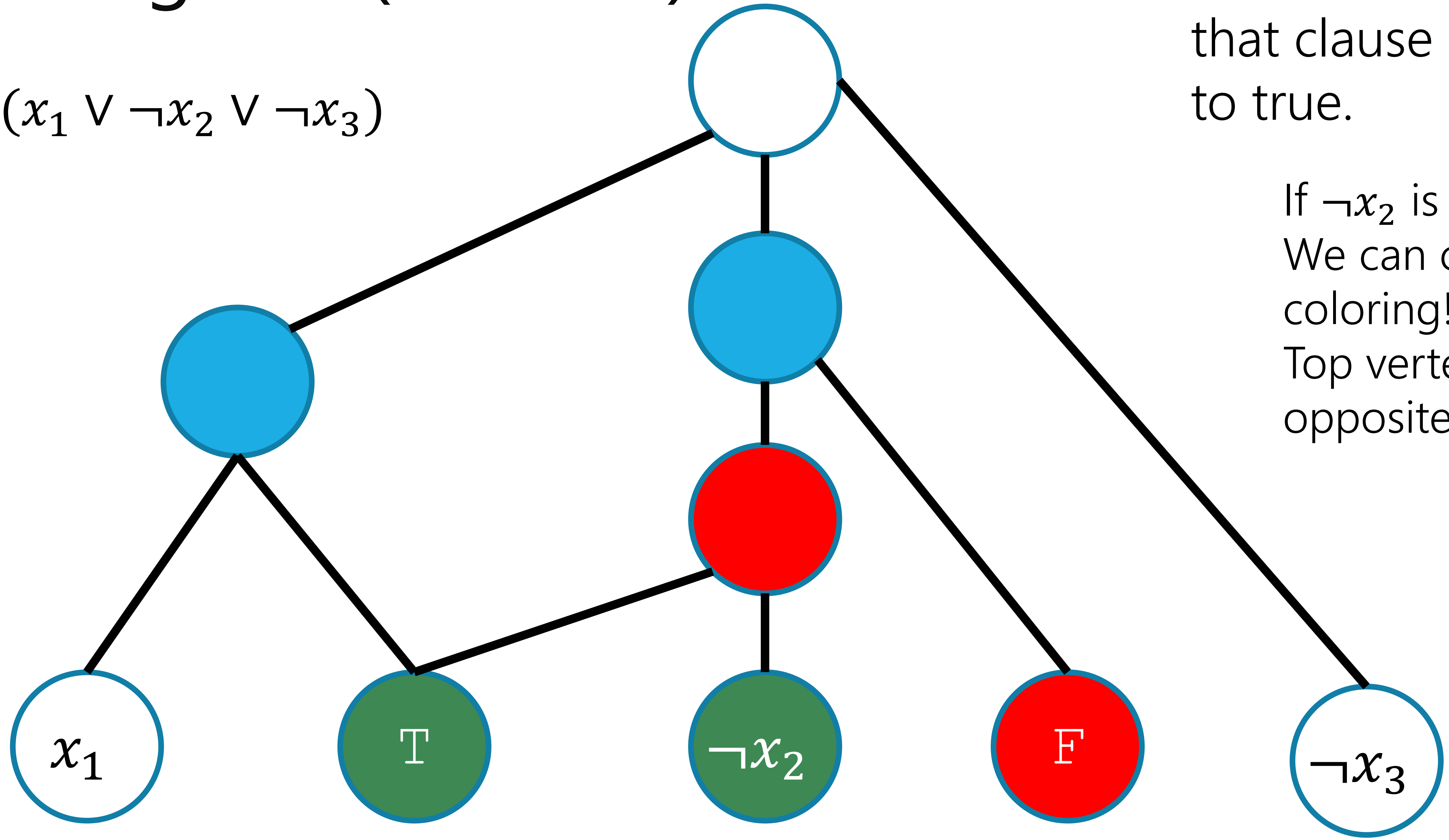
This tricky little graph can be 3-colored iff that clause evaluates to true.

If $\neg x_2$ is true…
Find a coloring!



# Gadget 2 (clauses)

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

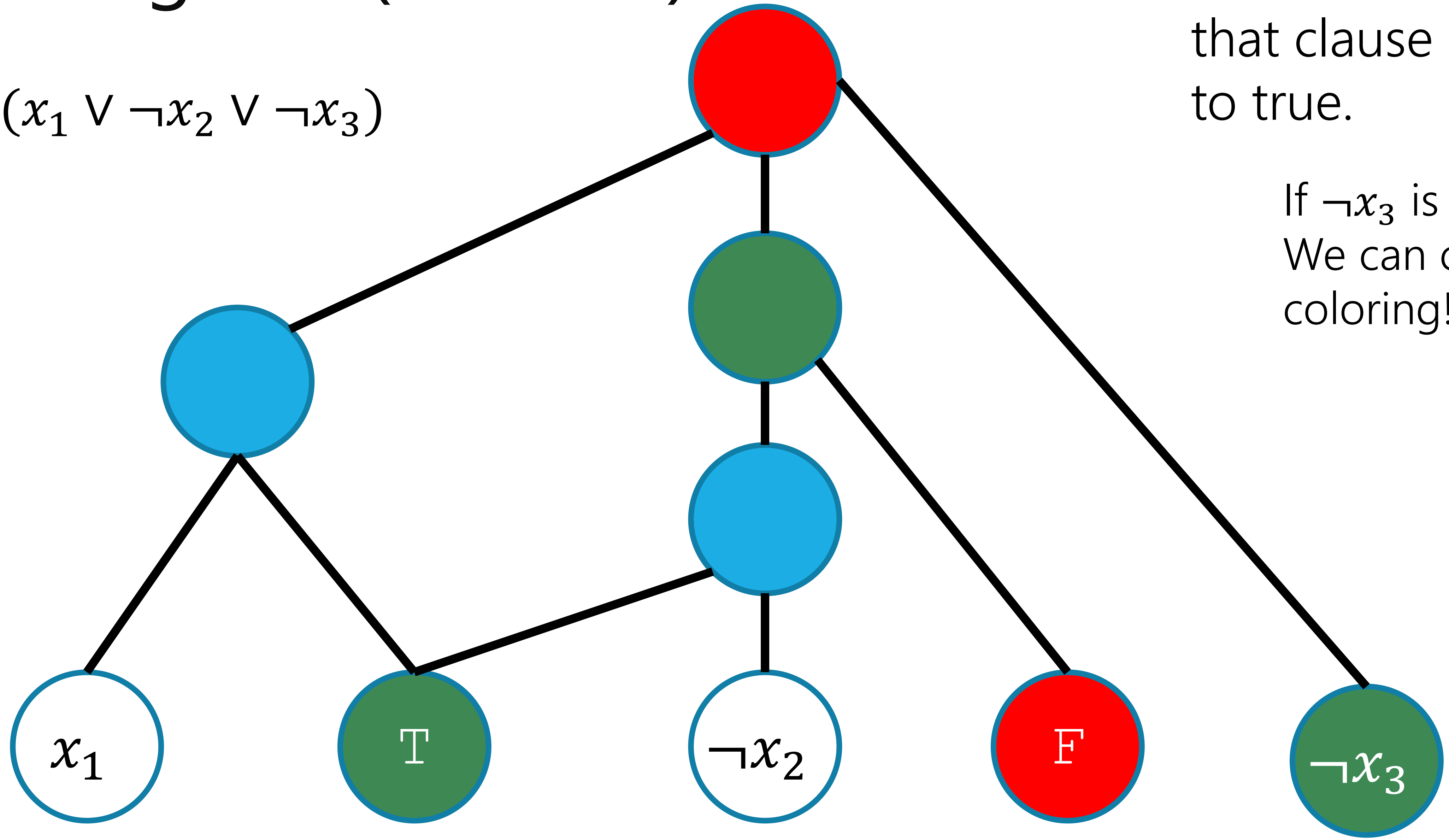


This tricky little graph can be 3-colored iff that clause evaluates to true.

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

# Gadget 2 (clauses)

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

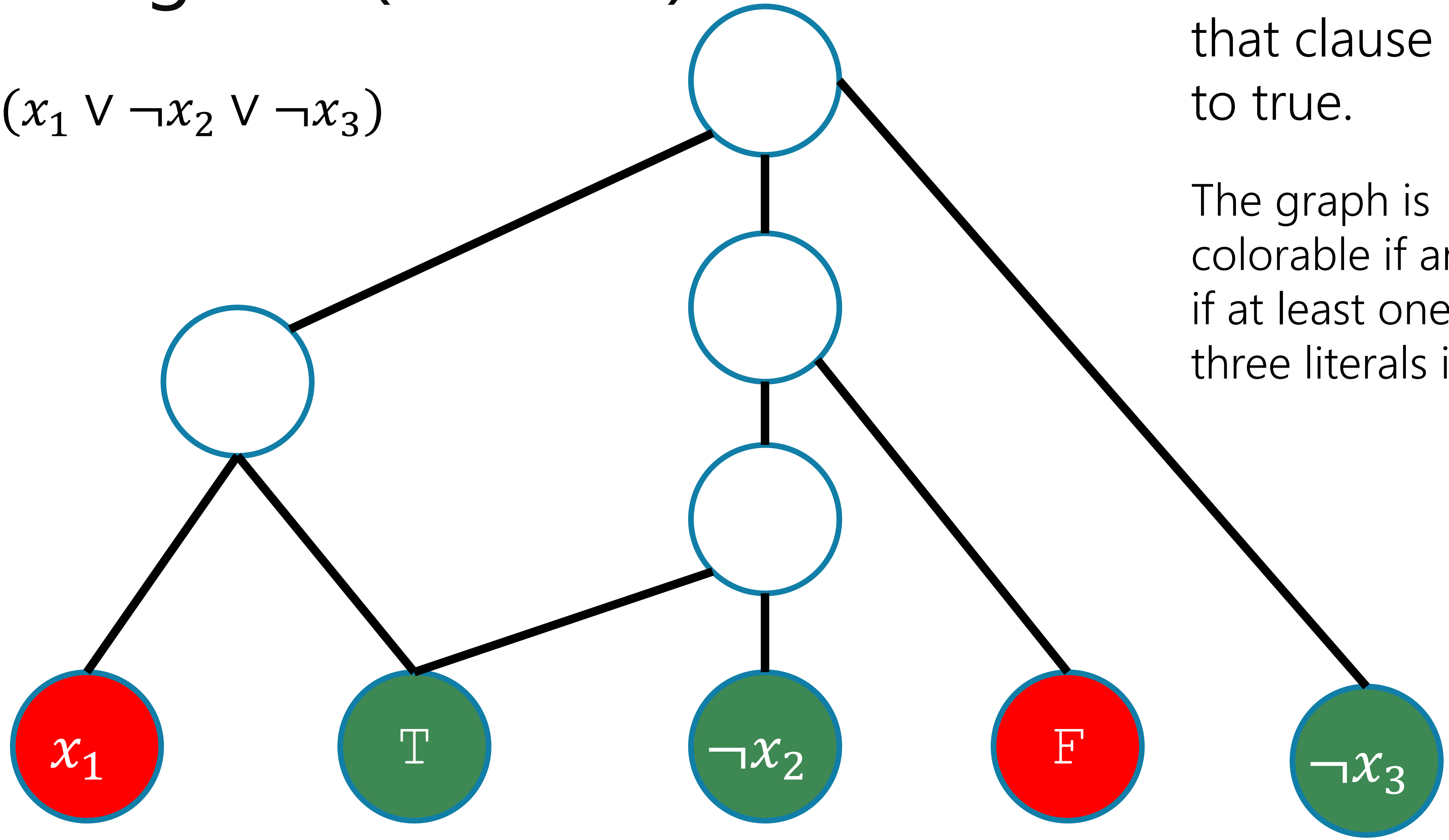


This tricky little graph can be 3-colored iff that clause evaluates to true.

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.

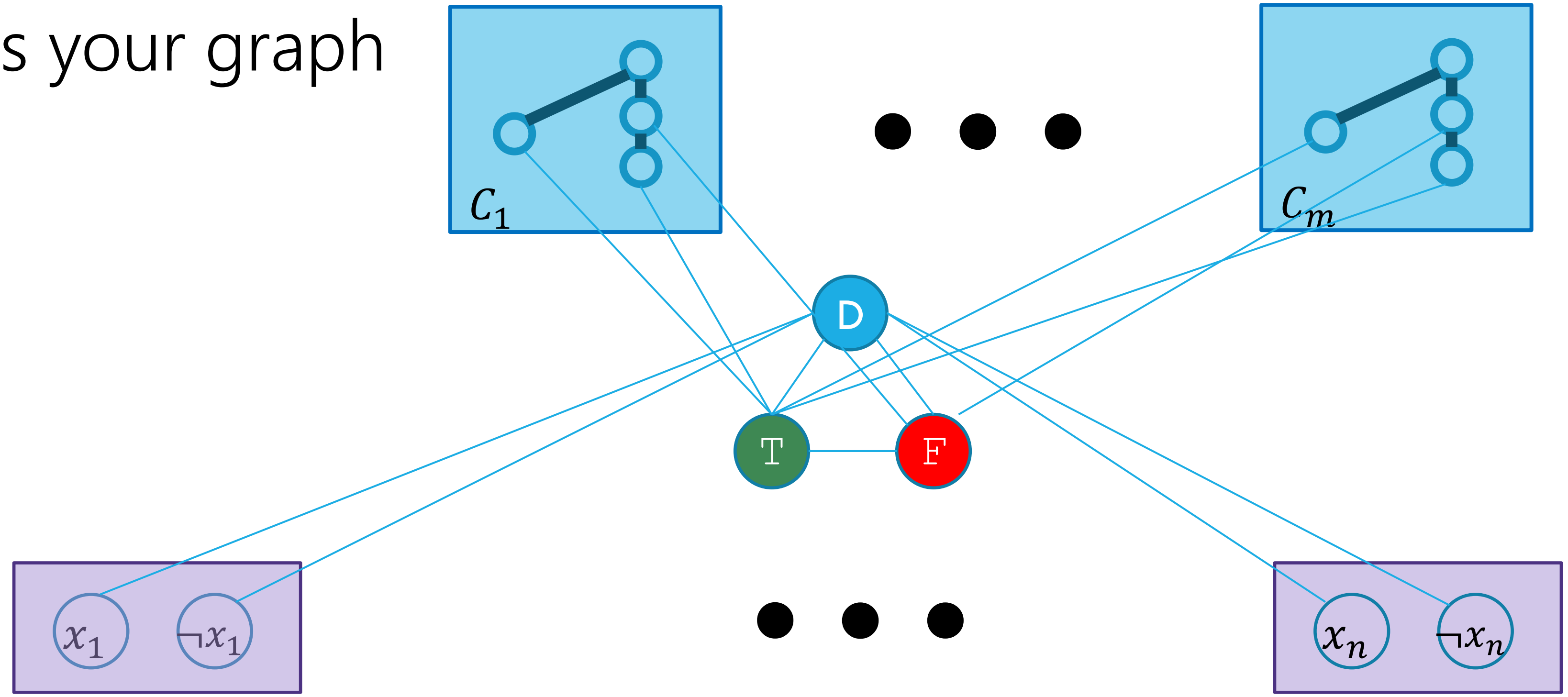
# Putting it together

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 `T,F`,D all must have different colors (since they are all adjacent). Call `T`'s color "green", `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.

# Some Loose Ends

# I have a problem

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

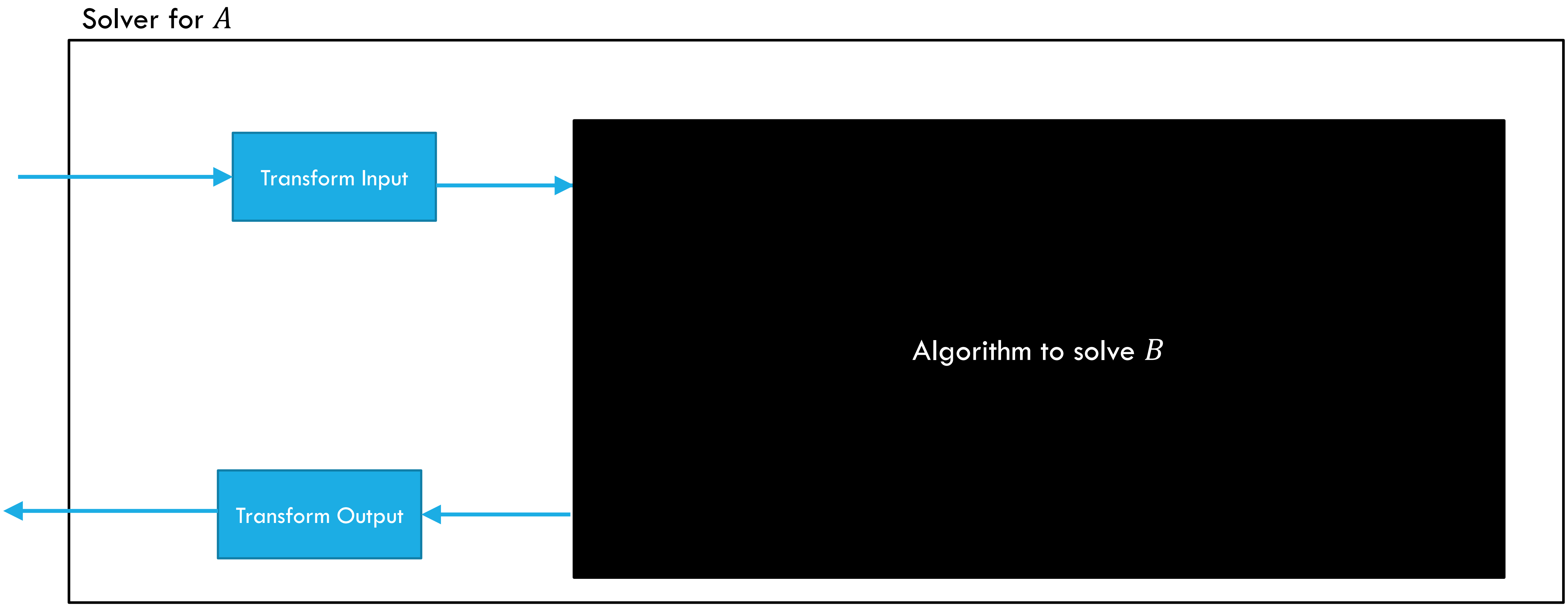
So difficult, it's probably NP-hard. How do I show it?

What does it mean to be NP-hard?

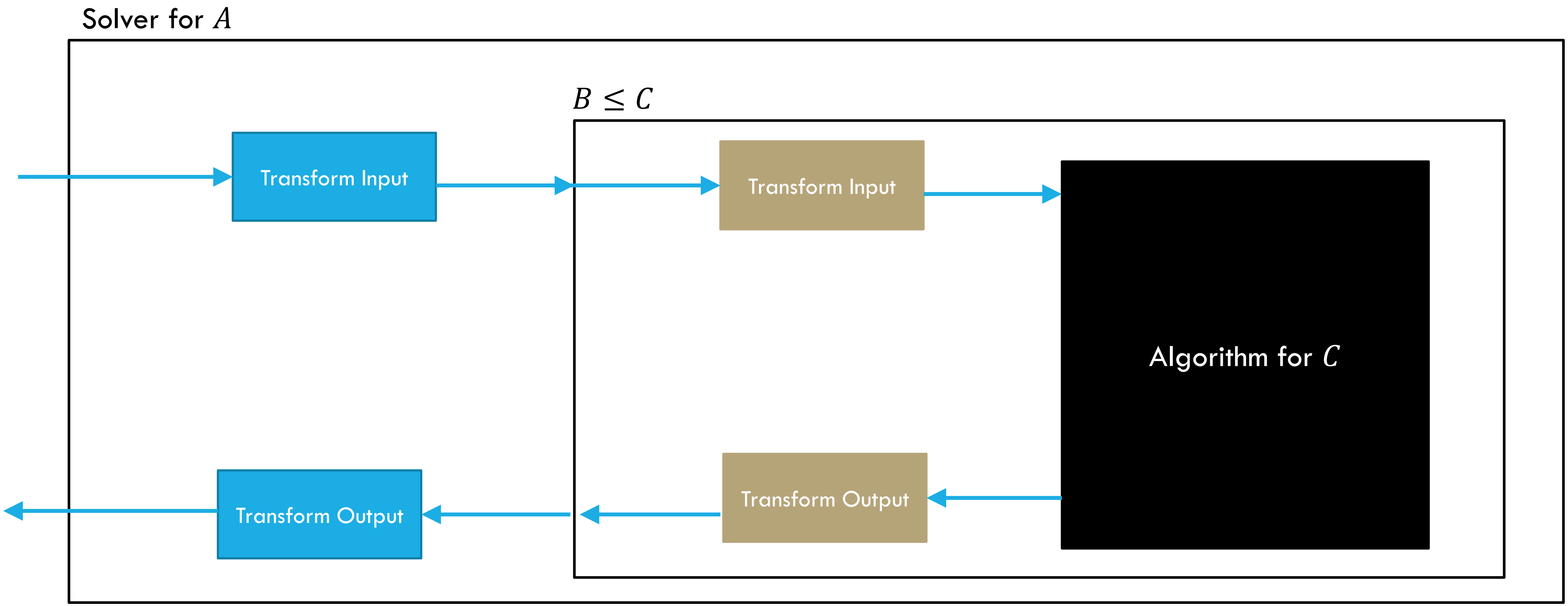
We need to be able to reduce any problem $A$ in $\mathbf{NP}$ to $C$.

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? $A \leq B \leq C \ \rightarrow A \leq C$



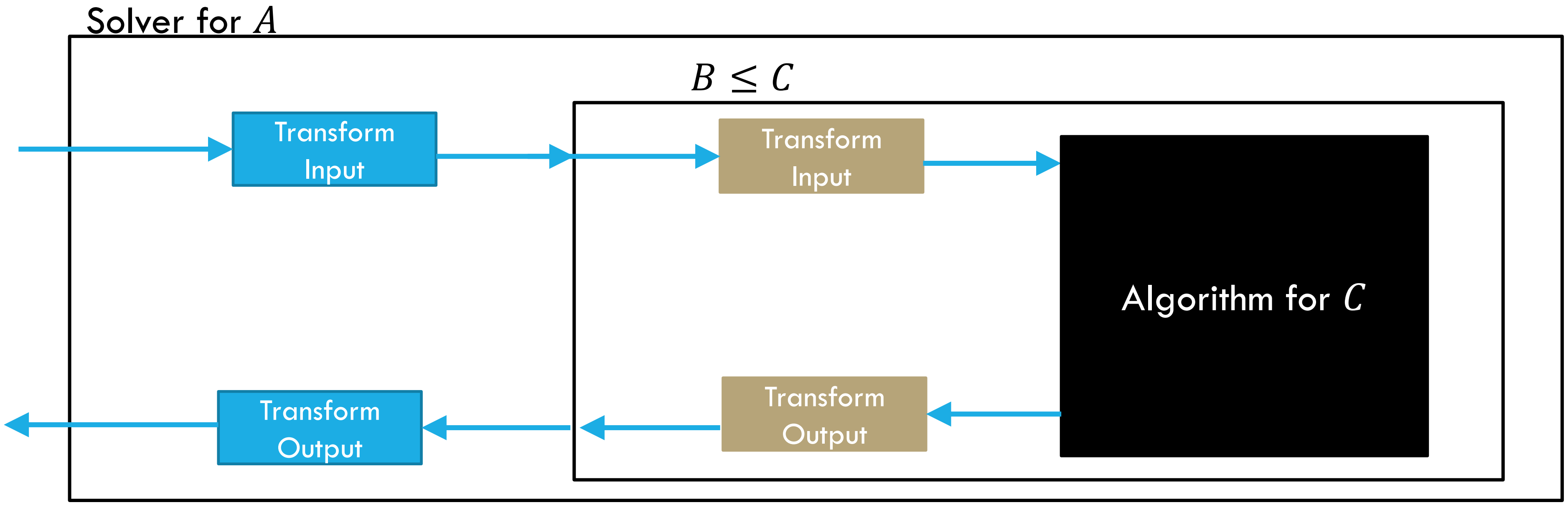
# Is the implication true? $A \leq B \leq C \ \rightarrow A \leq C$



# Is the implication true? $A \le B \le C \ \rightarrow A \le C$

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.



# Two Uses of Reductions

$A \leq B$

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

This is how you're used to using reductions

$A \leq B$

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

contrapositive of the last statement; the way we've used them this week.

# Be careful with your input

The definitions of both $\mathrm{P}$ and $\mathrm{NP}$ refer to the "size" of the input.

**P (stands for "Polynomial")**

The set of all decision problems that have an algorithm that runs in time $O\left(n^k\right)$ 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.

What does "size" mean?

# “Size”

“Size” is “number of bits used to represent the input.”

But I’ve never told you how to represent anything…

*Normally* all (reasonable) representations give you the same behavior.

Whether you represent a graph with

an adjacency list: $O(m + n)$ bits

A less efficient adjacency list: still $O(m + n)$ bits

An adjacency matrix: $O(n^2)$ bits

Your $O(m^5 n^{13})$ algorithm is still polynomial time.

# “Size”

*Normally* all (reasonable) representations give you the same behavior.

But *occasionally* it really matters.

The most common time where it matters is when (potentially very large) integers are a part of the input.

Consider the following problem:

PRIMES (on input $n$, $n$ represented in binary, return true if $n$ is prime)

Your algorithm? Trial division (is it divisible by 2, 3, 4, 5,…)

What’s the running time? Is it polynomial?

# “PRIME

Trial division:

There are like $\sqrt{n}$ divisions to try.

Division? It’s not a constant anymore! $n$ is too big! It isn’t an `int`, it’s an arbitrarily large integer.

Repeated subtraction will work though

We’re just trying to check if it’s polynomial. We don’t need the fastest algorithm.

So $O(\sqrt{n})$ divisions, each taking $O(n^k)$ time, where $k$ is a constant.

Sounds polynomial to me, right?

# Representing an integer

We are supposed to be looking at the running time based on the size of the input.

How many bits does it take to represent the number $n$?

What if $n$ is $2^5$?

# Representing an integer

We are supposed to be looking at the running time based on the size of the input.

How many bits does it take to represent the number $n$?

What if $n$ is $2^5$?

Only 6 bits! $100000_2$

In general it's $\Theta(\log n)$ bits.

So is $\sqrt{n} \cdot n^k$ polynomial time?

No! It's exponential.

Side note:
There is a polynomial time algorithm for PRIMES. That is, a $\Theta(\log^k n)$ algorithm for telling whether $n$ is prime.
It uses some fancy number theory and modular arithmetic.

# Knapsack

On HW5, you solved (a non-decision-version of) the knapsack problem.

Input: A list of $n$ objects (plants) of value $v_i$ and weight $w_i$,
a max weight $W$, a target value $T$.

Output: `true` if there is a set of objects of total value $T$ (or more) of weight at most $W$.

Running time: $\Theta(Wn)$

What's the input size?

$O(n\,[\log(\max v_i) + \log(\max w_i)] + \log(W) + \log(T))$

# Knapsack

Running time: $\Theta(Wn)$

Input size?

$O(n\,[\log(\max v_i) + \log(\max w_i)] + \log(W) + \log(T))$

That isn't a polynomial time algorithm.

# Weakly NP-hard

You might have heard (in 332 or on Wikipedia) that Knapsack is an NP-hard problem. It is…but that's very dependent on the fact that it takes $\log(W)$ bits to represent $W$.

It's only NP-hard if you represent the input in the usual way.

If you made the input length $O(n+W)$ you'd have a polynomial time algorithm.

The different input gives you a different problem! And the other one isn't known to be NP-hard.

If changing the representation of numbers from binary to unary makes the problem not NP-hard anymore, we call it weakly NP-hard.

# Takeaways

Be careful when deciding if an algorithm shows a problem is in $\mathrm{P}$.

Be sure you've accounted for the fact that numbers are in binary.

Some problems have algorithms that are polynomial *in variables of interest* but not in *the size of the input*

- $\mathrm{P}$ vs. $\mathrm{NP}$ asks about *the size of the input.*
- But you as an algorithm designer probably care about variables of interest.

If you see "$A$ is NP-hard" and you think you have a polynomial-time algorithm for $A$, double check you understand the representation that was used to prove the problem is 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