Thread # dad-jokes

Jennifer Brennan 17 days ago

I'd tell you an NP-complete joke.

...But if you've heard one, you've heard them all.

😂 19

4 replies

Ben Kushigian 🎧 17 days ago

That's very reductionist

⬆ 7

Even More Reductions

CSE 421 Winter 23
Lecture 23

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 $x_1, \ldots, x_n$

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

NP-complete: 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 even the really different looking ones?

Yes! Really all of them!*

*There's a tiny corner-case; take complexity to learn more.

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…

$(\neg x_1 \vee x_3 \vee x_5)$
$\wedge (\neg x_1 \vee \neg x_3 \vee \neg x_5)$
$\wedge (x_1 \vee x_2 \vee x_3)$
$\wedge (\neg x_2 \vee x_3 \vee x_4)$

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.”

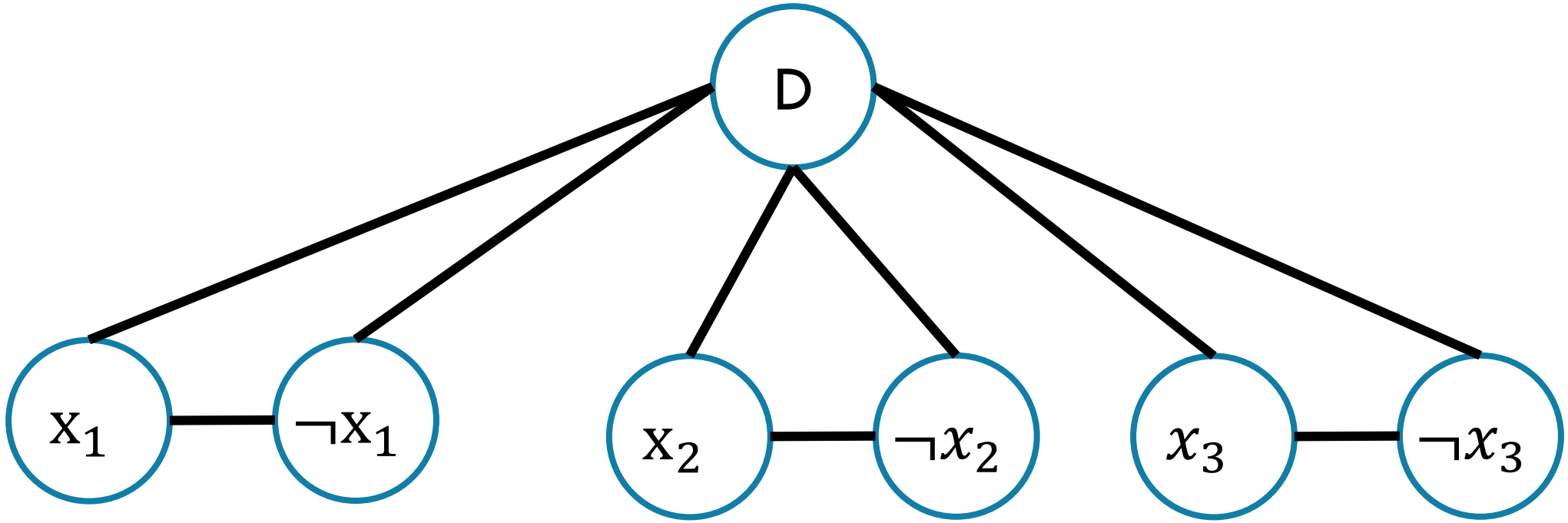
Gadget 1

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)



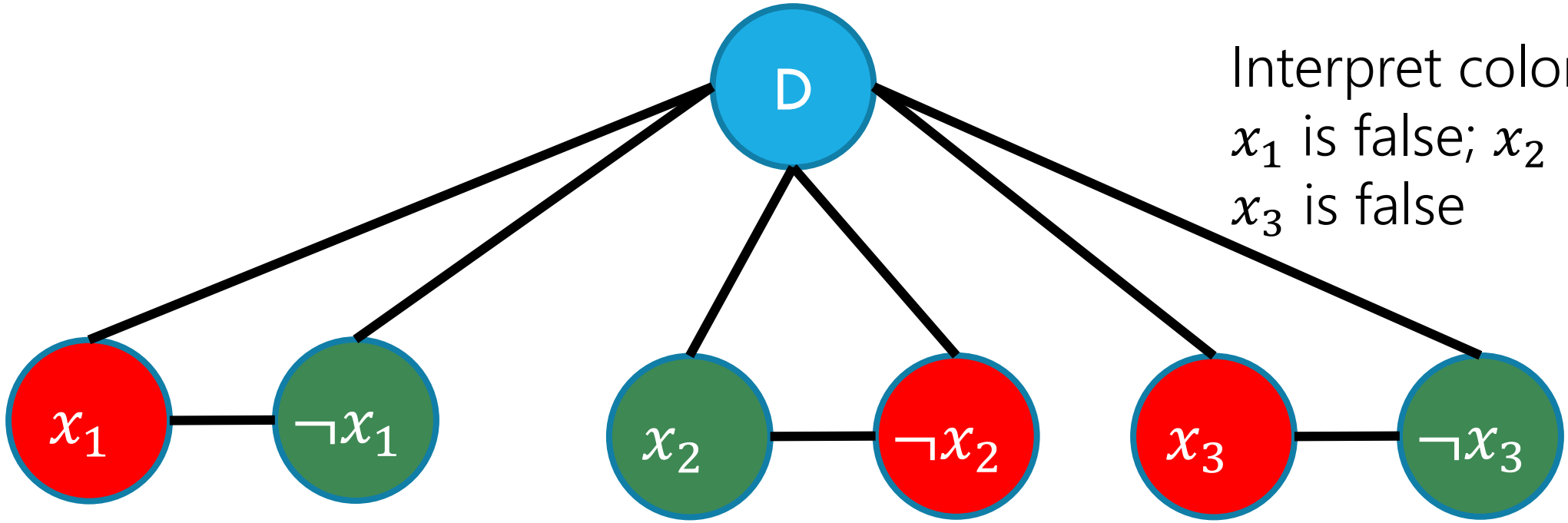
Gadget 1

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

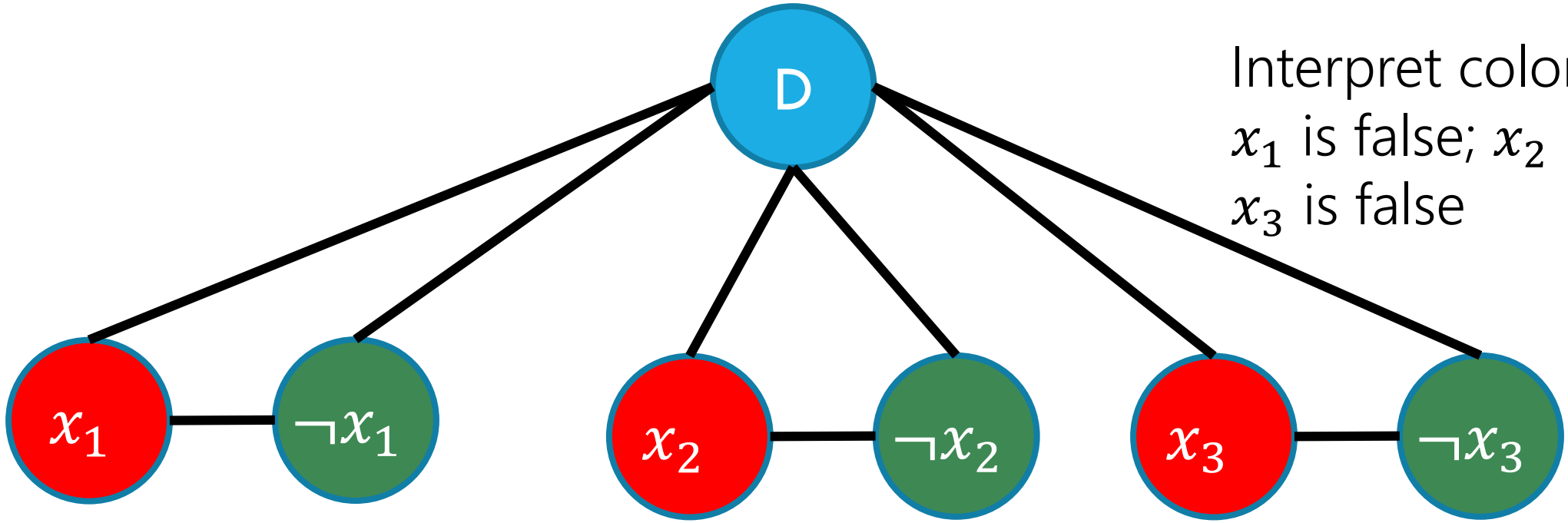
Gadget 1

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?

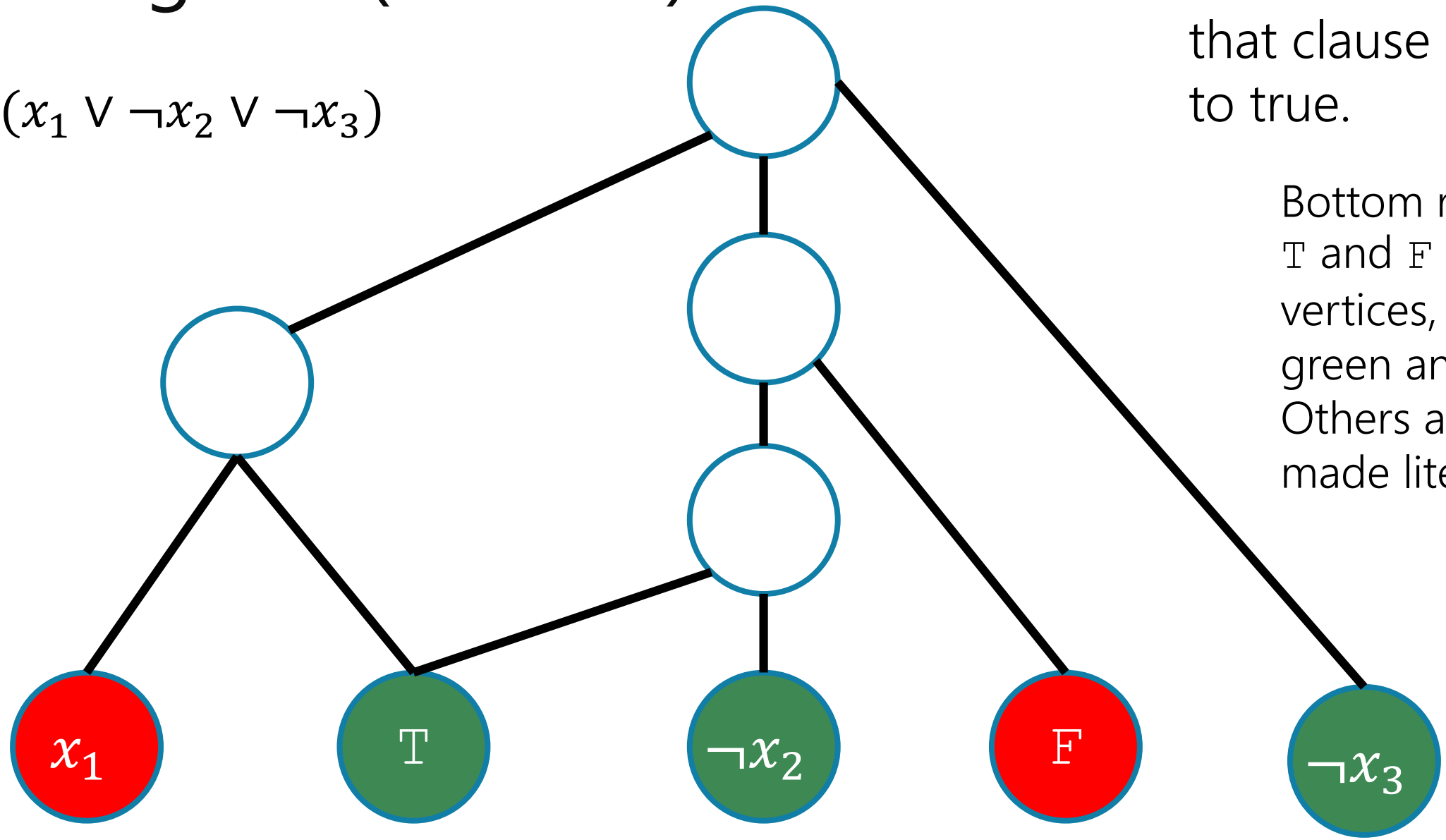
We can interpret a 3-coloring as a setting of the variables!

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)

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

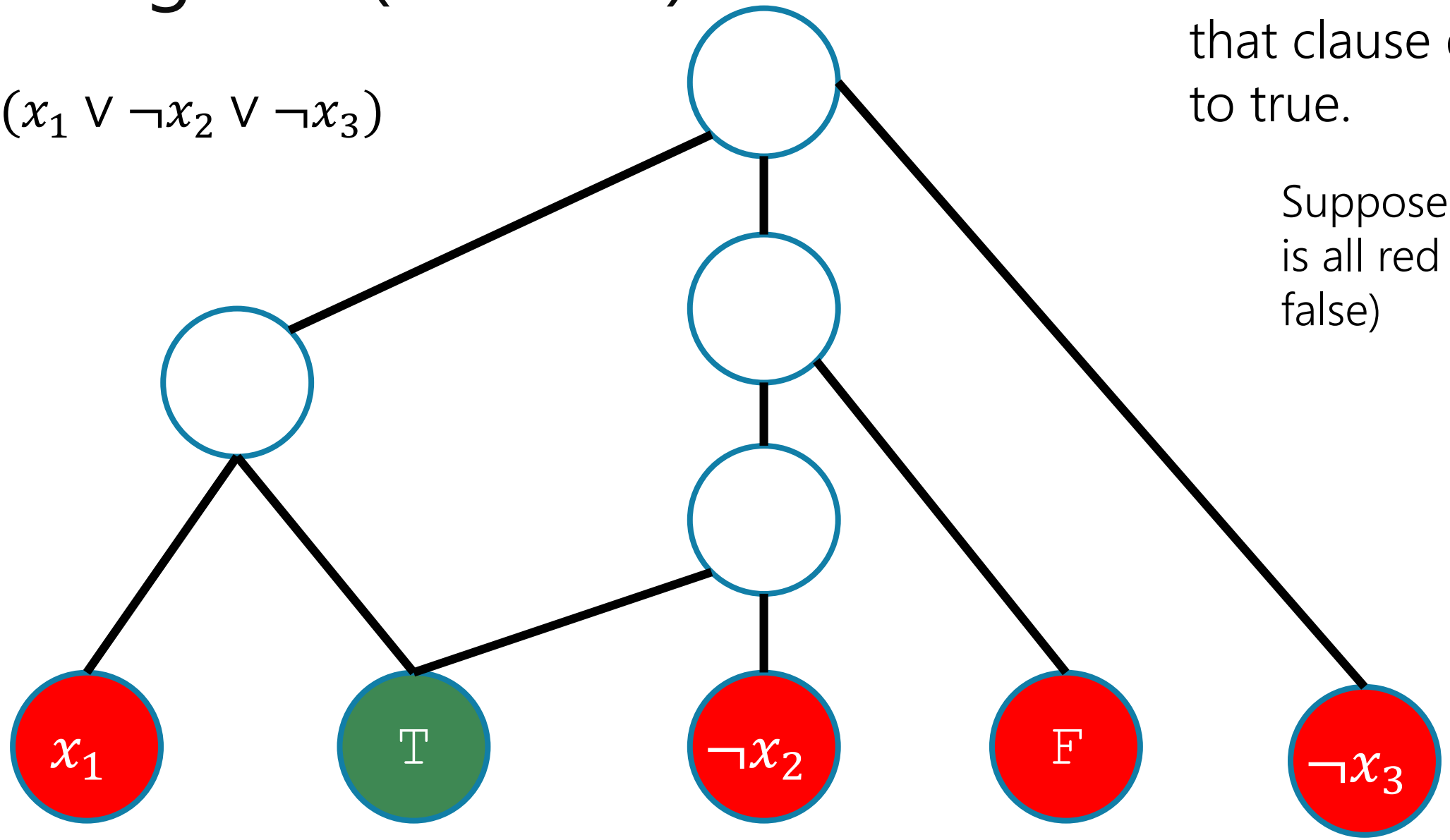


This tricky little graph can be 3-colored iff that clause evaluates to true.

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

Gadget 2 (clauses)

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

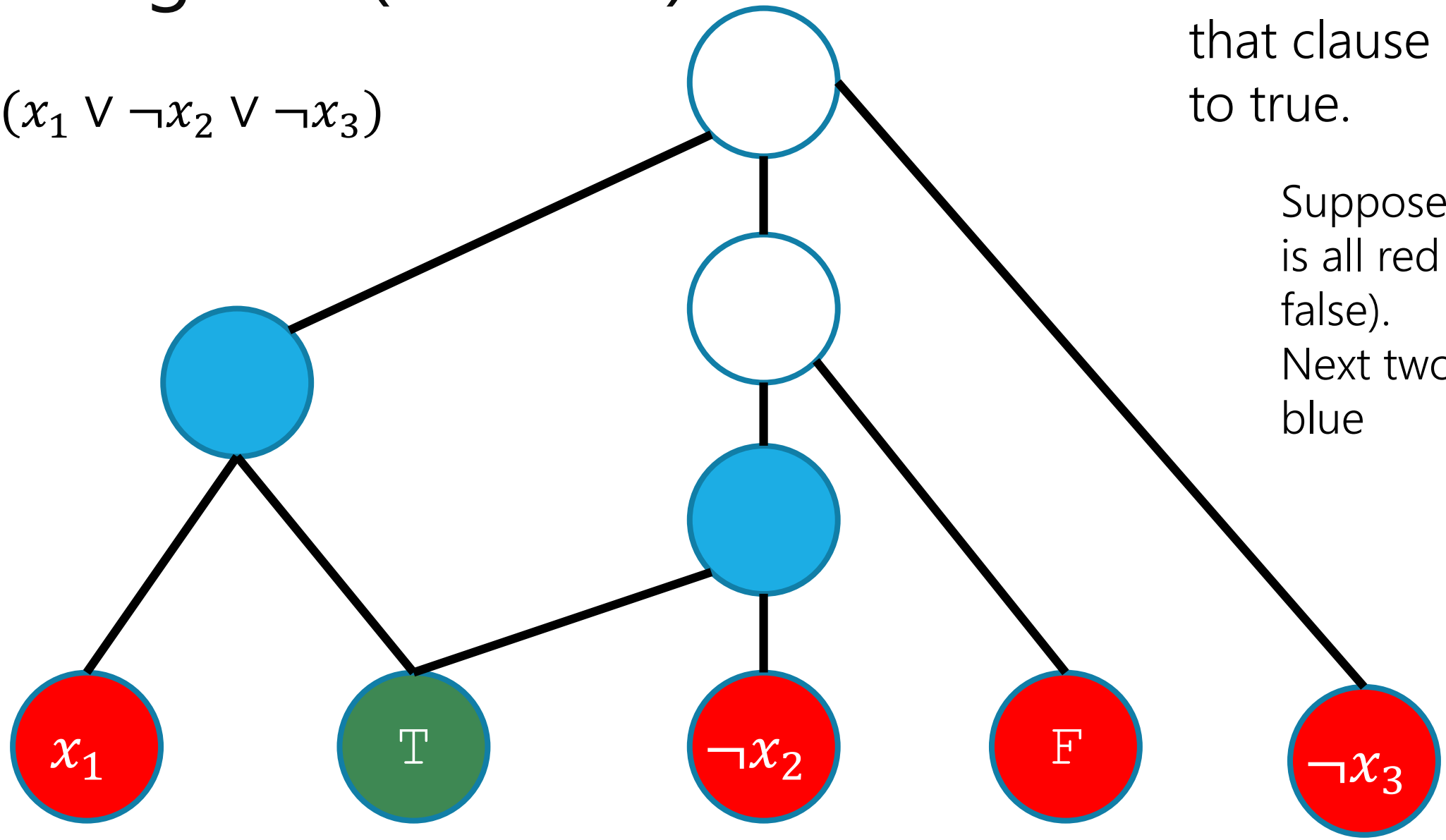


This tricky little graph can be 3-colored iff that clause evaluates to true.

Suppose bottom row is all red (clause is false)

Gadget 2 (clauses)

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

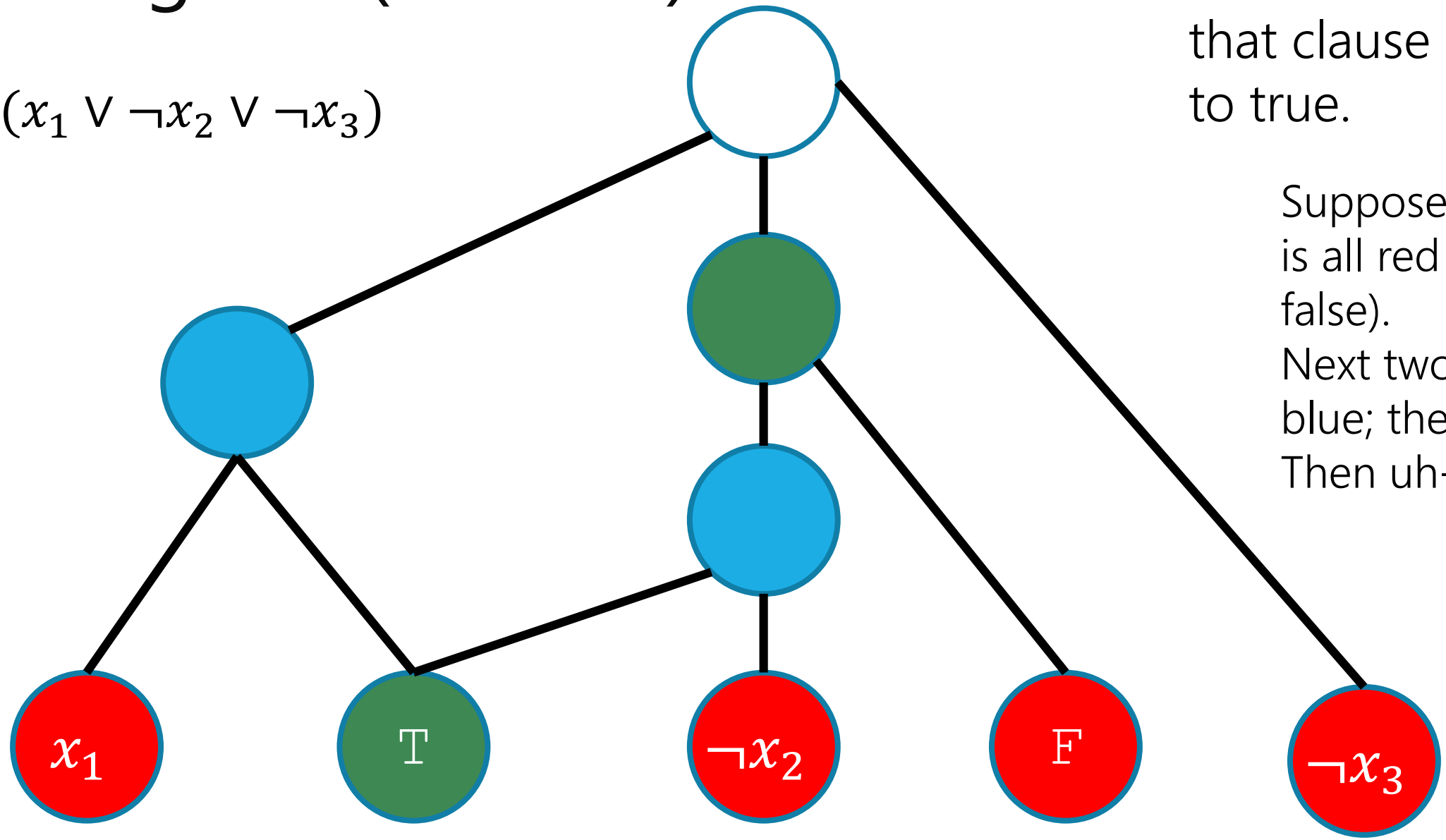


This tricky little graph can be 3-colored iff that clause evaluates to true.

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

Gadget 2 (clauses)

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

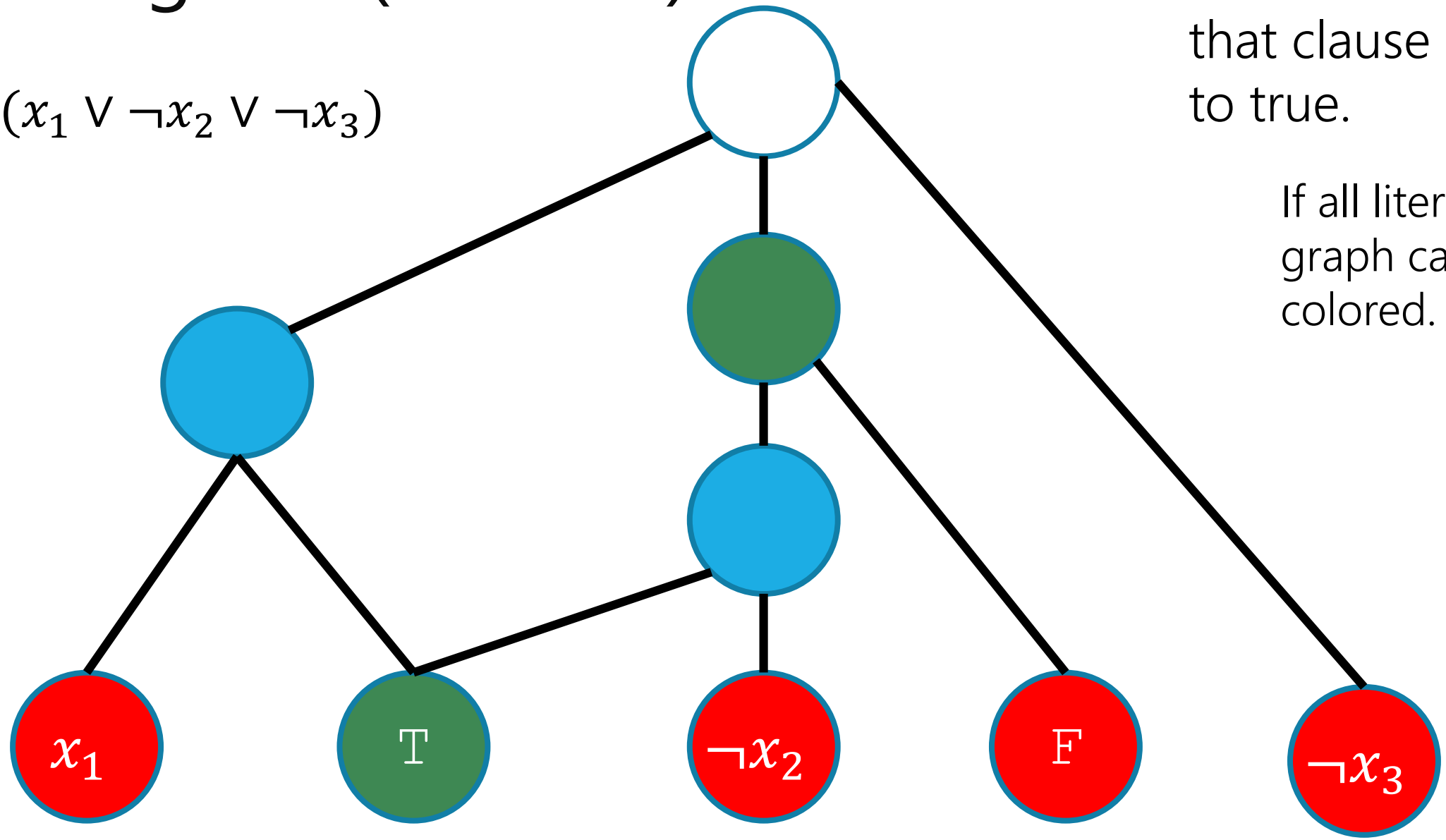


This tricky little graph can be 3-colored iff that clause evaluates to true.

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

Gadget 2 (clauses)

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

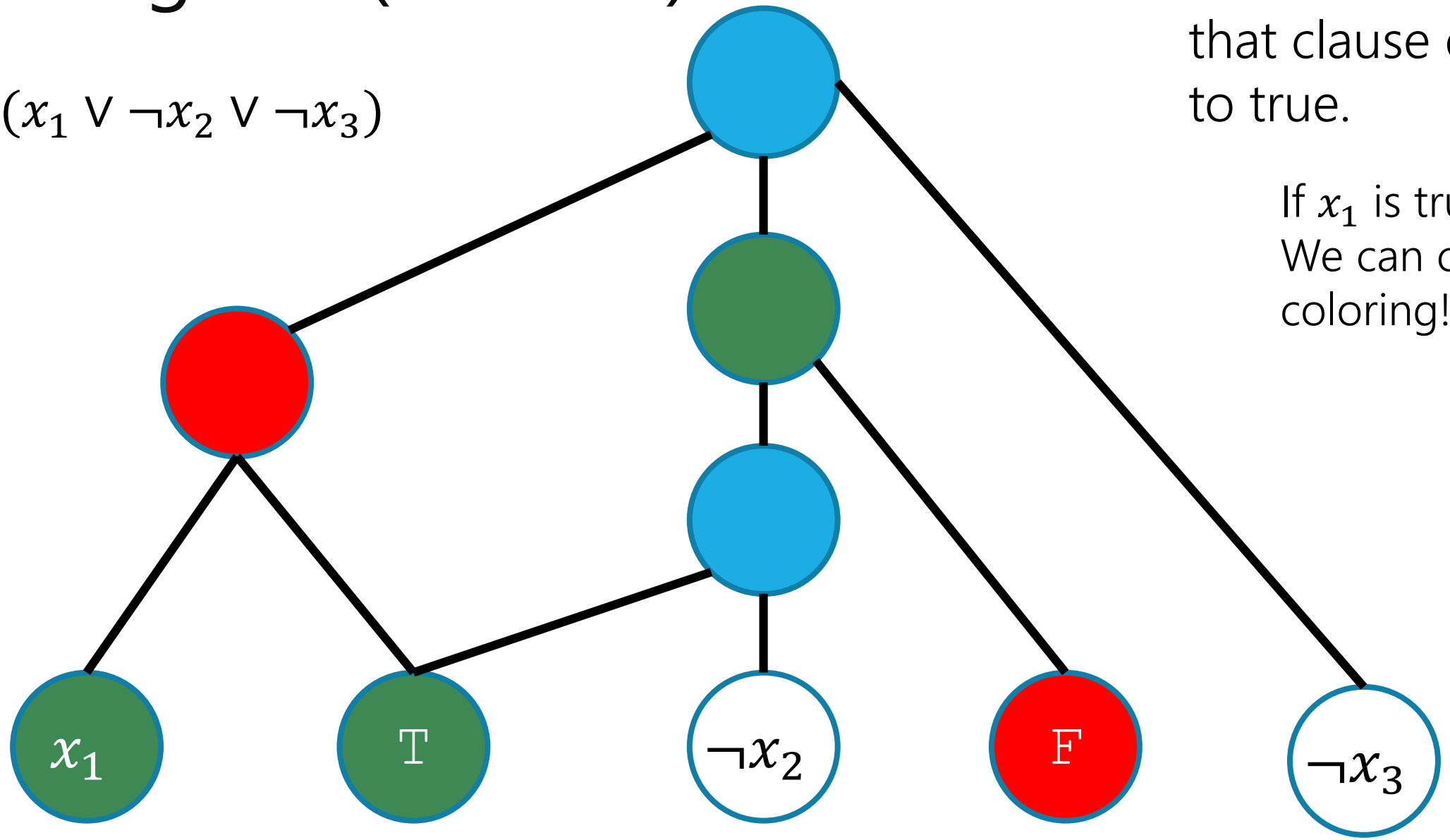


This tricky little graph can be 3-colored iff that clause evaluates to true.

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

Gadget 2 (clauses)

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

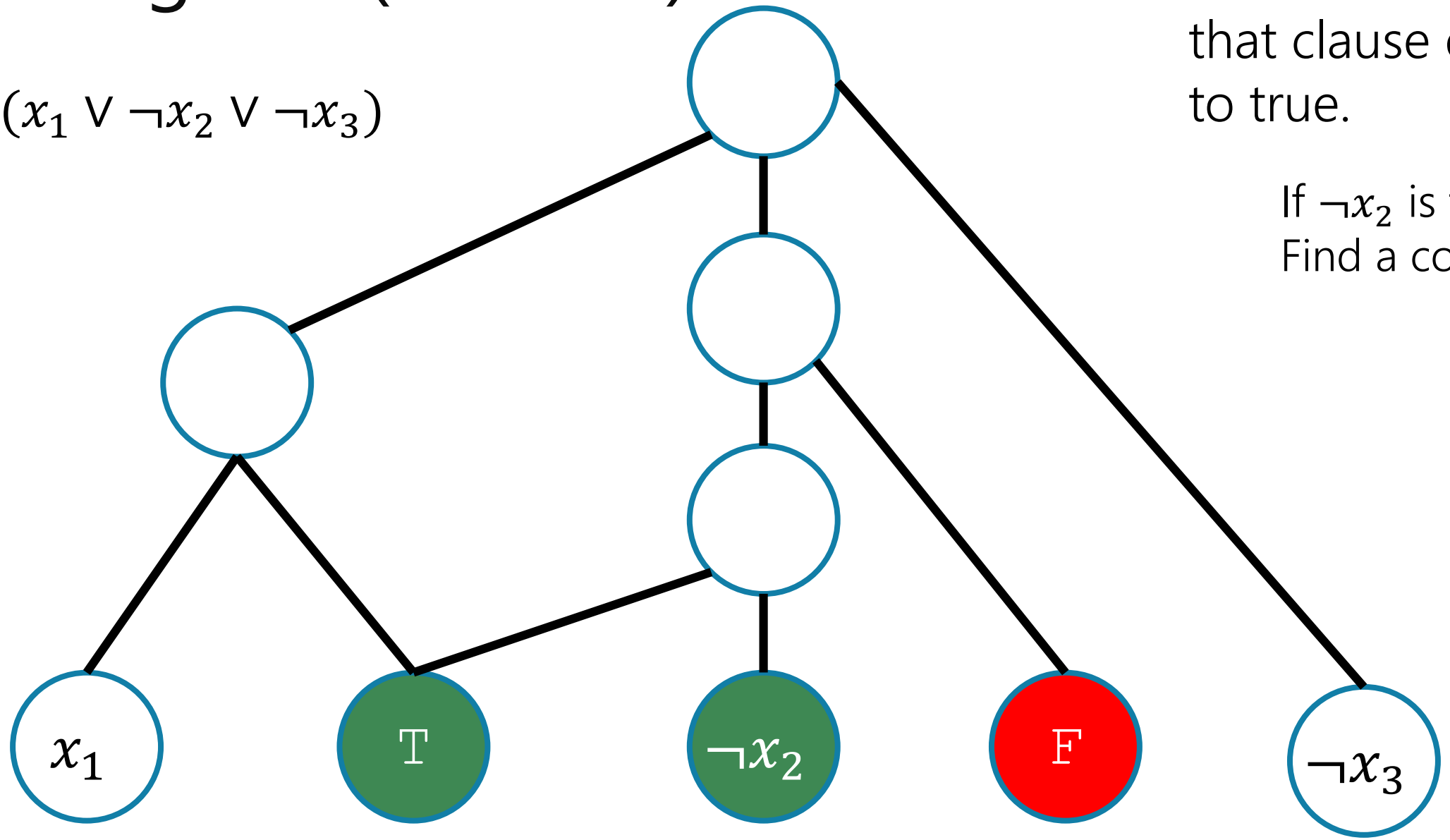


This tricky little graph can be 3-colored iff that clause evaluates to true.

If x_1 is true…
We can complete the coloring!

Gadget 2 (clauses)

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

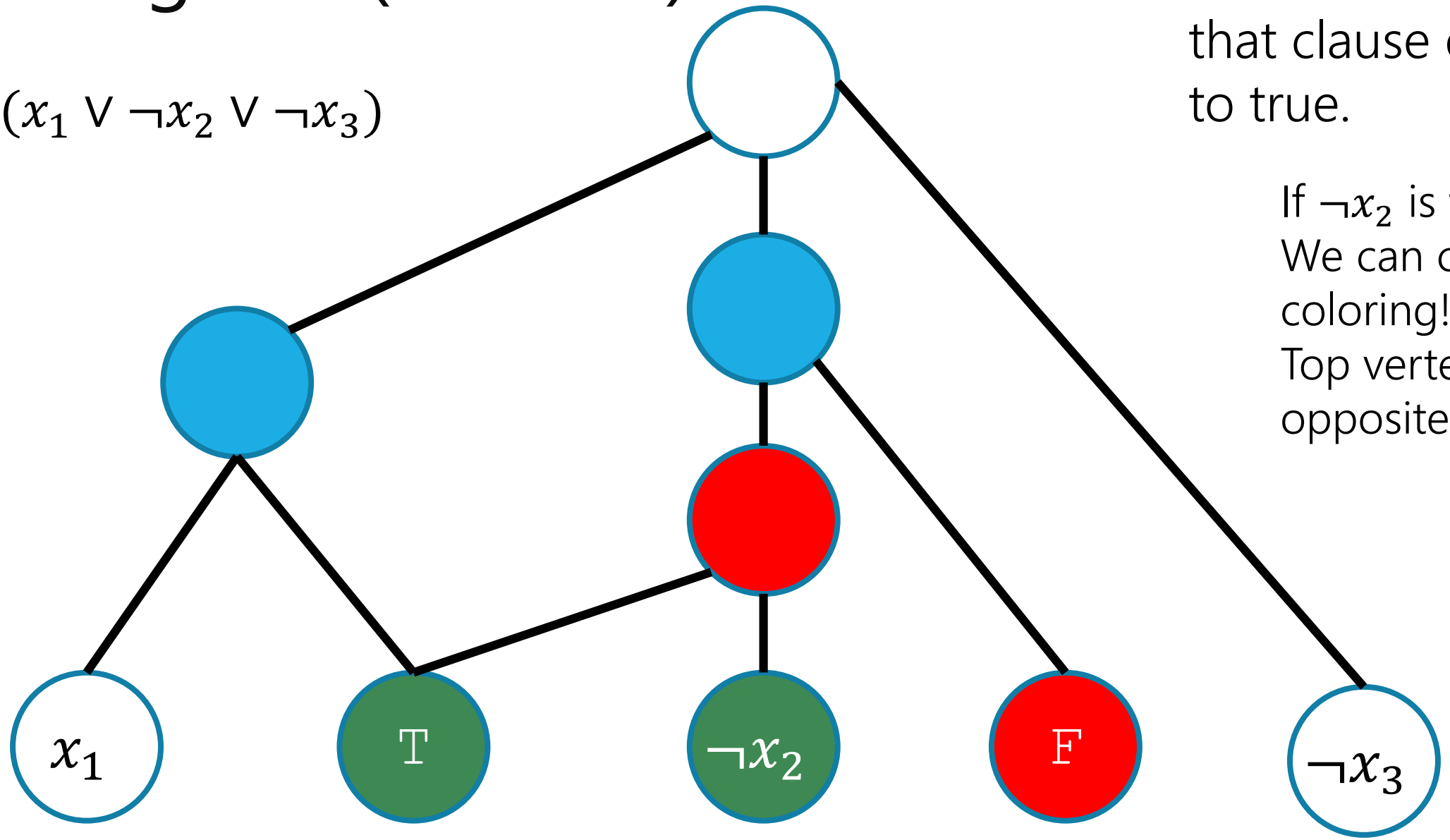


This tricky little graph can be 3-colored iff that clause evaluates to true.

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

Gadget 2 (clauses)

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

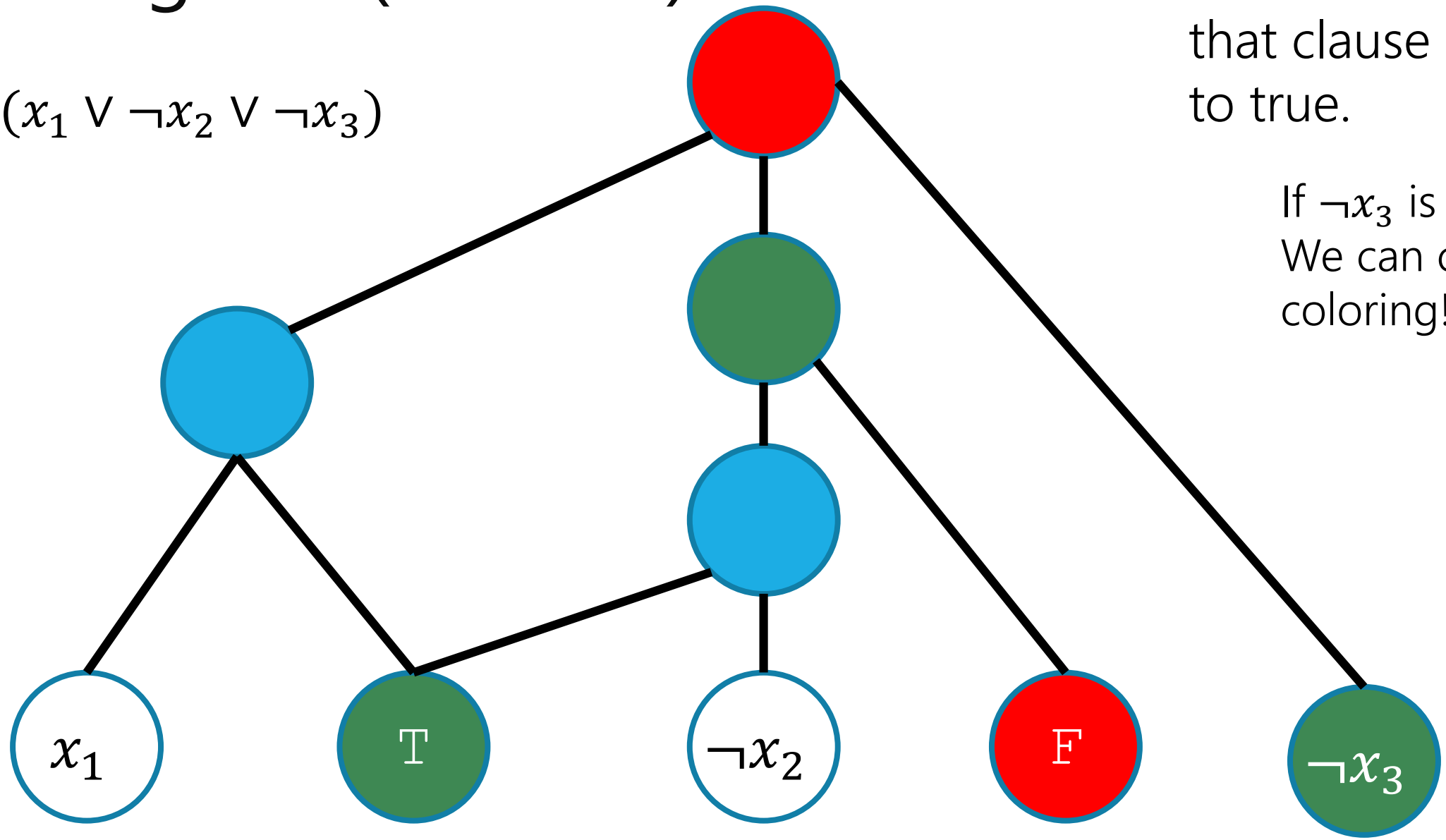


This tricky little graph can be 3-colored iff that clause evaluates to true.

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

Gadget 2 (clauses)

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

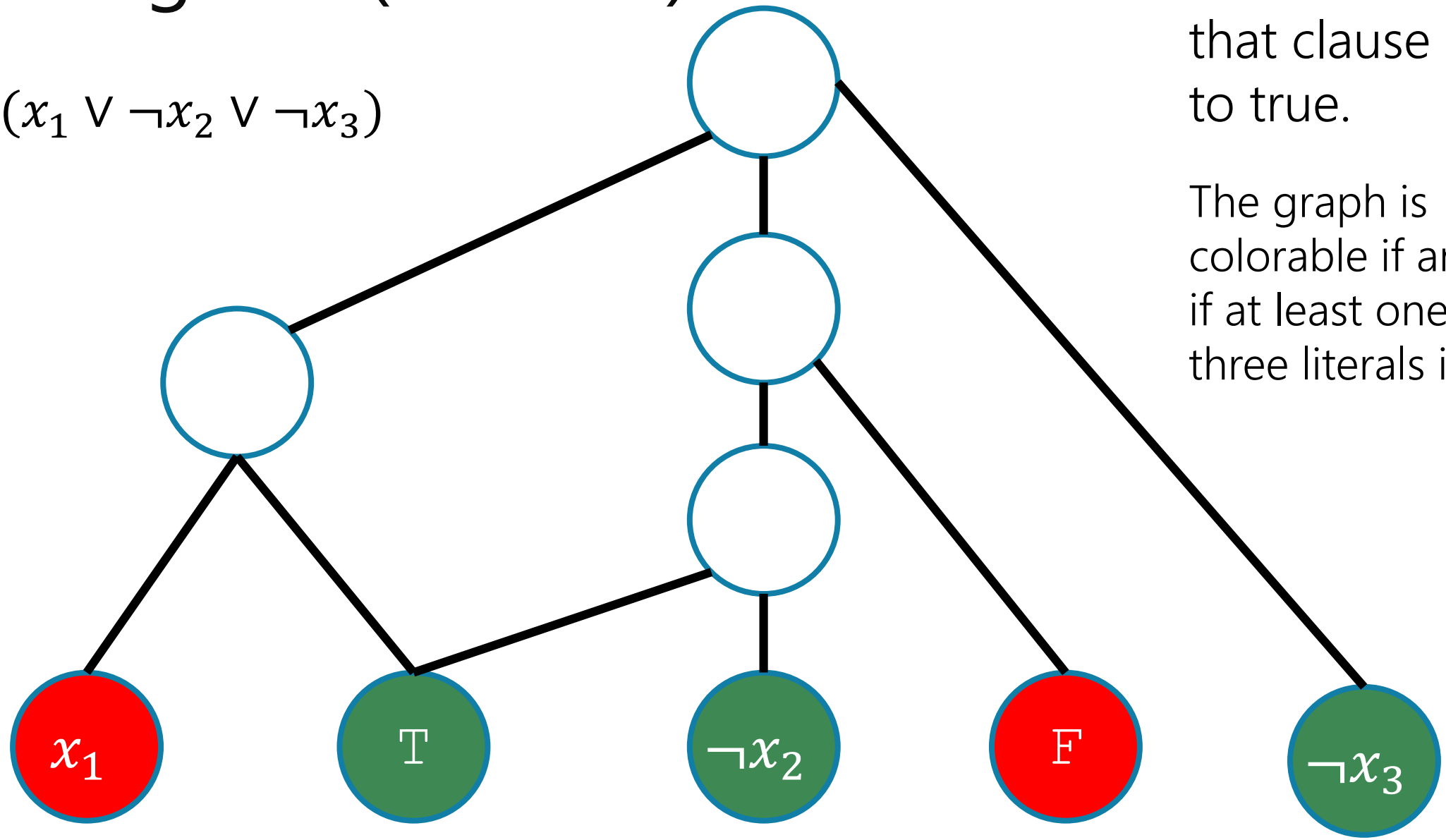


This tricky little graph can be 3-colored iff that clause evaluates to true.

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.

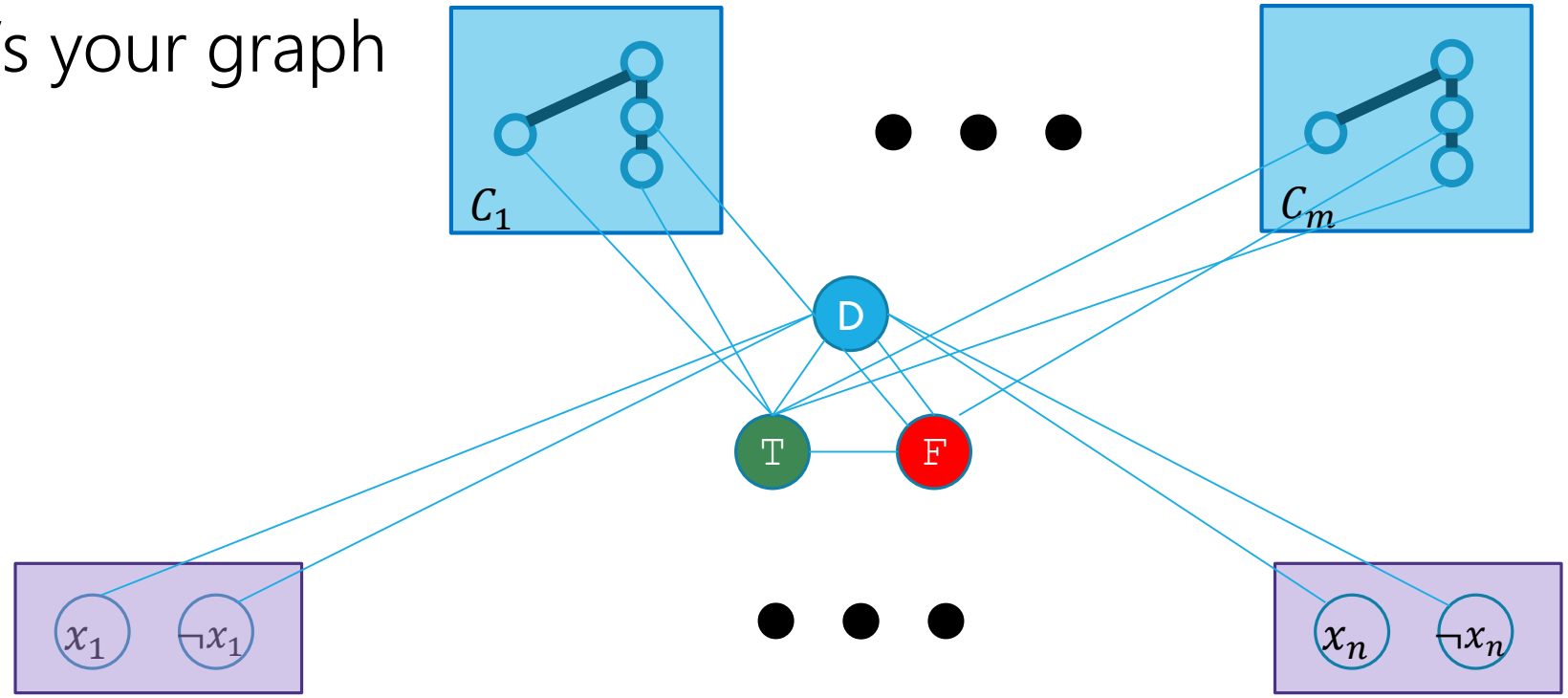
Putting it together

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

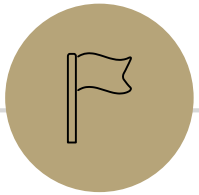
If the graph is 3-colorable, then there must be a satisfying assignment.

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.



Some Loose Ends

I have a problem

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

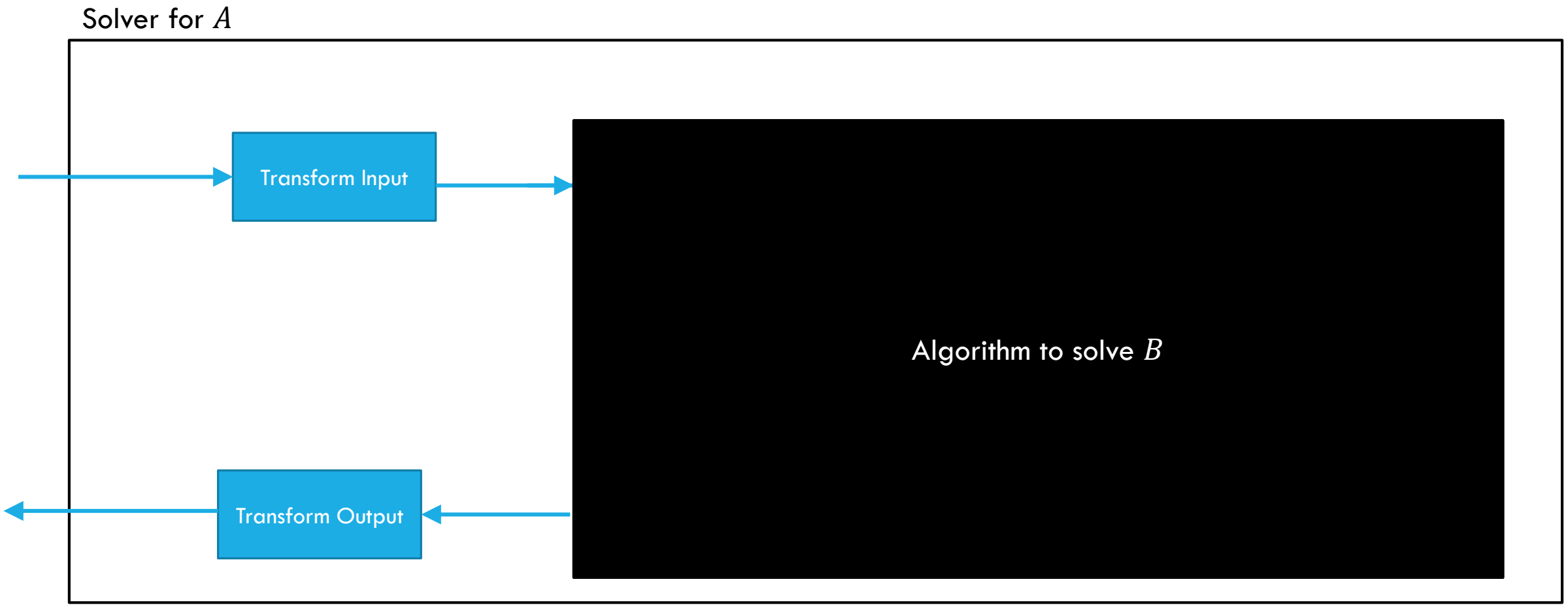
So difficult, it's probably NP-hard. How do I show it?

What does it mean to be NP-hard?

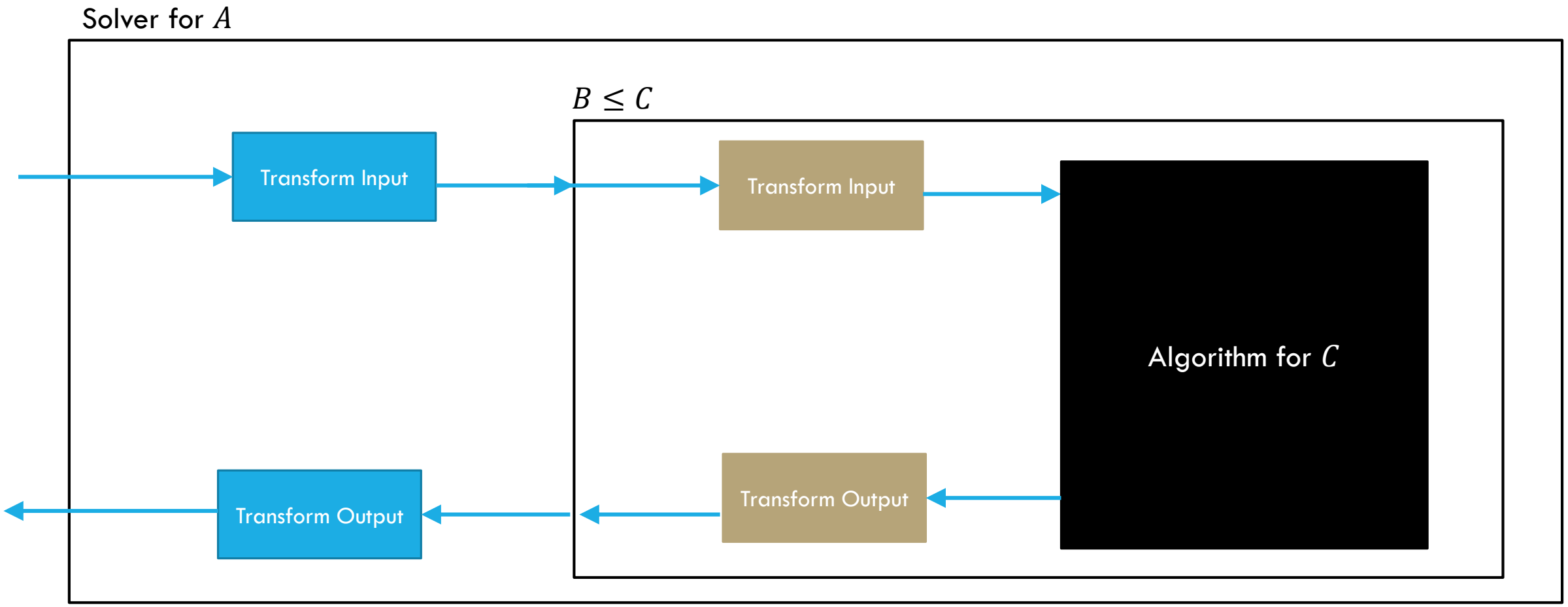
We need to be able to reduce any problem A in $\mathbf{NP}$ to C.

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



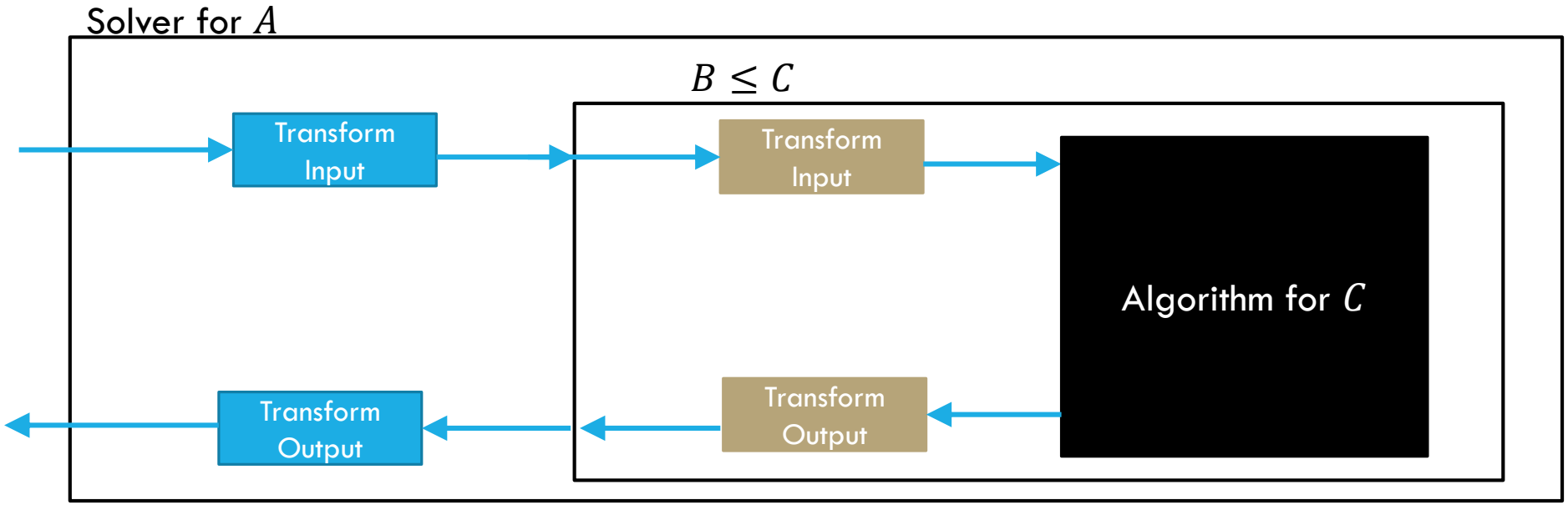
Is the implication true? $A \leq B \leq C \rightarrow A \leq C$



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

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.



Two Uses of Reductions

$A \leq B$

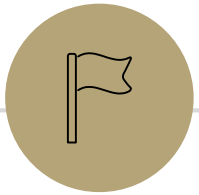
If I know B is not hard [I have an algorithm for it] then A is also not hard.

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.



Coping with NP-completeness

Examples

There are literally thousands of NP-complete problems.
And some of them look weirdly similar to problems we do know efficient algorithms for.

In P

Short Path

Given a directed graph, report if there is a path from s to t of length at most k.

NP-Complete

Long Path

Given a directed graph, report if there is a path from s to t of length at least k.

Examples

In P

Light Spanning Tree

Given a weighted graph, find a spanning tree (a set of edges that connect all vertices) of weight at most k.

NP-Complete

Traveling Salesperson

Given a weighted graph, find a tour (a walk that visits every vertex and returns to its start) of weight at most k.

The electric company just needs a greedy algorithm to lay its wires.
Amazon doesn't know a way to optimally route its delivery trucks.

Examples

In P

2-Coloring

Given an undirected graph, can the vertices be labeled red and blue with no edge having the same colors on both endpoints?

NP-Complete

3-Coloring

Given an undirected graph, can the vertices be labeled red, blue, and green with no edge having the same colors on both endpoints?

Just changing a number by one takes us from one of the first problems we solved (and one of the fastest algorithms we've seen) to something we don't know how to solve efficiently at all.

Dealing with NP-hardness

Thousands of times someone has wanted to find an efficient algorithm for a problem…

…only to realize that the problem was NP-hard.

Takeaway 1:

Even though we haven't proven $P \neq NP$ (i.e. we haven't proven any of these problems **don't** have an efficient algorithm), this is good evidence that we shouldn't by trying to solve NP-hard problems.

It's probably not just a matter of finding the "right representation"/"right angle on the problem" we've tried a few thousand of them.

Dealing with NP-hardness

Thousands of times someone has wanted to find an efficient algorithm for a problem…

…only to realize that the problem was NP-hard.

Takeaway 2:

Sooner or later it will happen to one of you.

What do you do if you think your problem is NP-complete?

Dealing with NP-hardness

You just started your new job at Amazon. Your boss asks you to look into the following problem

You have a graph, each vertex is where a specific truck has to do a delivery. Starting from the warehouse, how do you make all the deliveries and return to the warehouse using the minimum amount of gas.

Traveling Salesperson

Given a weighted graph, find a tour (a walk that visits every vertex and returns to its start) of weight at most k.

Step 1: Make sure your problem is really NP-hard

Understand **exactly** what your inputs and outputs are.

2-coloring and 3-coloring are very different.

Finding a vertex cover of a general graph is NP-hard. Finding a vertex cover of a bipartite graph can be done in multiple very efficient ways.

Understand **exactly** what you're being asked to solve.
Realizing you're trying to solve a problem on a tree instead of a general graph almost always makes DP possible.
Are your constraints linear (can you use an LP)?
Are your constraints simple (are you solving 2SAT instead of 3SAT)?

Step 2: It still looks hard

Now that you know exactly what you're trying to solve, and you still can't solve it…

Next try to prove hardness (i.e. do a reduction).

Usually there's a similar problem you can convert from!

It's easier to do a reduction from 3-coloring to 5-coloring than from Hamiltonian Path to 5-coloring.

Both reductions **exist** but there's no need to flex here, look up a list of NP-complete problems and see what's similar looking.

Step 3: ???

So you go to your boss and say

“Sorry, problem’s NP-hard. I proved it.”

And your boss says:

“that’s a cool proof and all, but really. We need to tell the drivers where to go tomorrow…and we need to use less gas.

Step 3: Bandaids

Can you write your problem as a SAT instance?

Ok, you definitely can if it's in NP, that's what NP-hardness means…can you write it as a reasonably-sized SAT instance (n^3 instead of $1000000n^{100}$)?

There are SAT libraries that **often** run pretty fast. In the worst-case they're still exponential, but you don't always hit the worst case!

Can you write your problem as an integer program?

Run an integer programming library and see what happens!

Can you write your problem as a graph problem?

Many are very well studied for "simple" graphs (e.g. "planar" graphs, ones that can be drawn on a piece of paper without edges crossing).

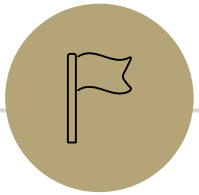
Step 4 – Permanent Solutions

Those exponential time algorithms are great as band-aids.

If it's a one-time thing, or just a "we'll run this about once a week, if it takes too long once in a while no big deal" these are fine.

But what if you need a guarantee!

Your code is running every night, and you need an answer by 6 AM or the delivery trucks don't go out.



Optional: Input Size

Be careful with your input

The definitions of both P and 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 $O(n^k)$ 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.

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

Consider the "knapsack" problem

Input: A list of n objects 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.

There's a DP with running time: $\Theta(Wn)$. But Knapsack is NP-hard.

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 $\mathbf{P}$.

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*

$\mathbf{P}$ vs. $\mathbf{NP}$ 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