Coping with NP-completeness

CSE 421 Winter 23
Lecture 24

Announcements

Review session: Saturday (3/11) 1:30-3 in CSE2 G01

We'll try to make a recording, but it won't be high quality

Final exam study resources will be up tonight

If you need a conflict exam (pre-existing conflict)

Email Robbie and Allie (both of us, in one email) asap.

If you end up ill/need to isolate the day of the exam

Email Robbie and Allie as soon as you know (both of us, in one email)

What's Coming Up?

Today:

Wrap-up the reduction

What do you do when

What is an approximation algorithm?

Wednesday

What is a "linear program"?

Friday

Using LPs for approximation (very fast)

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.”

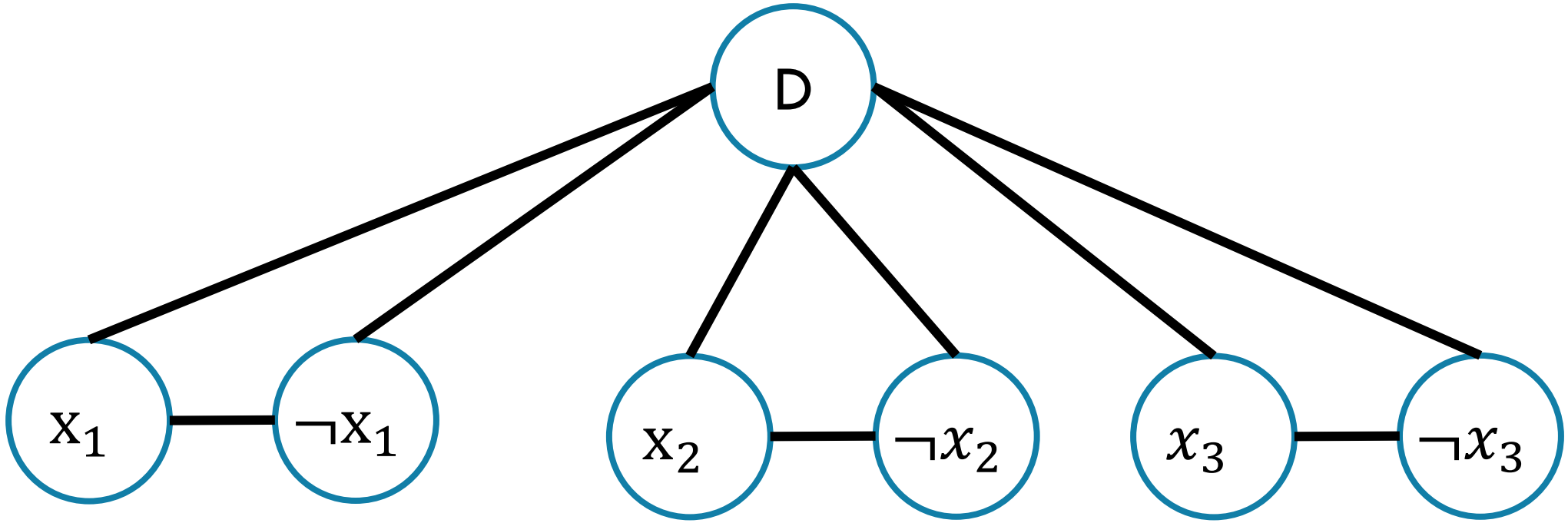
Gadget 1

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)



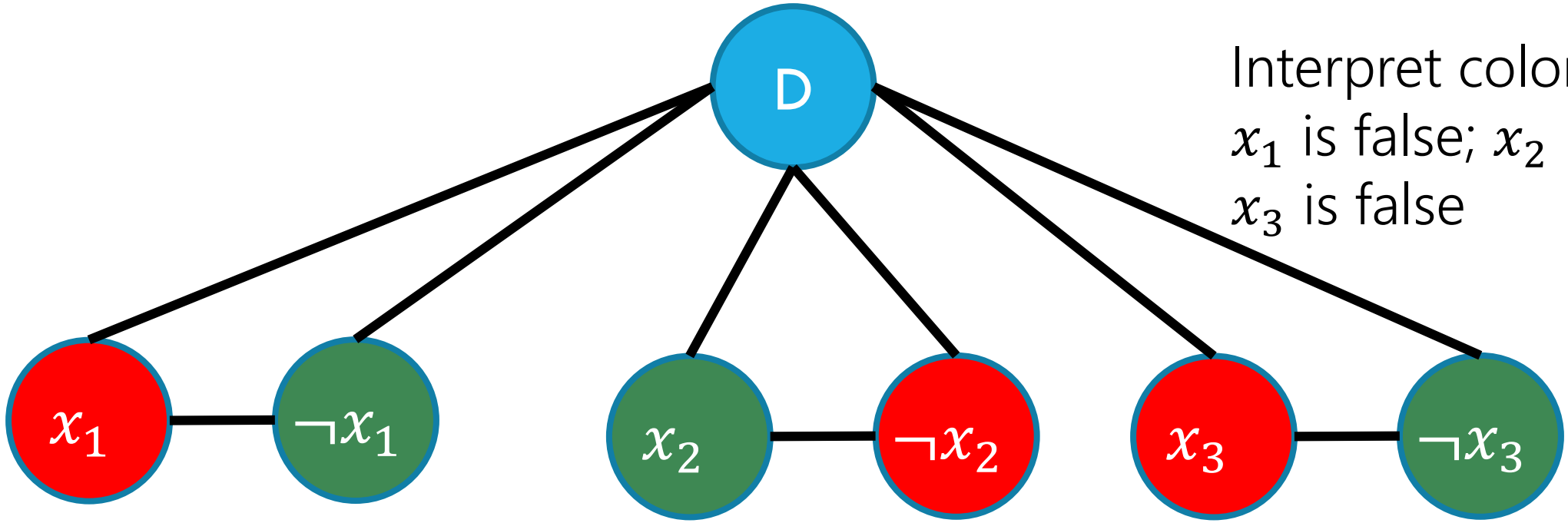
Gadget 1

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

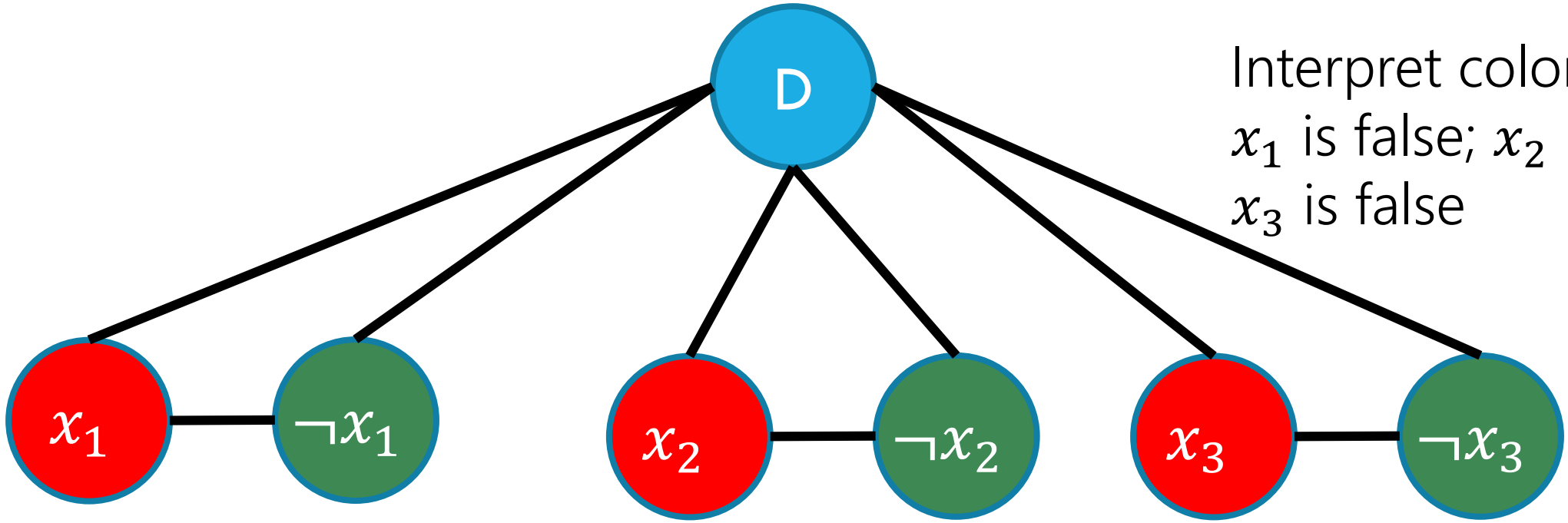
Gadget 1

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?

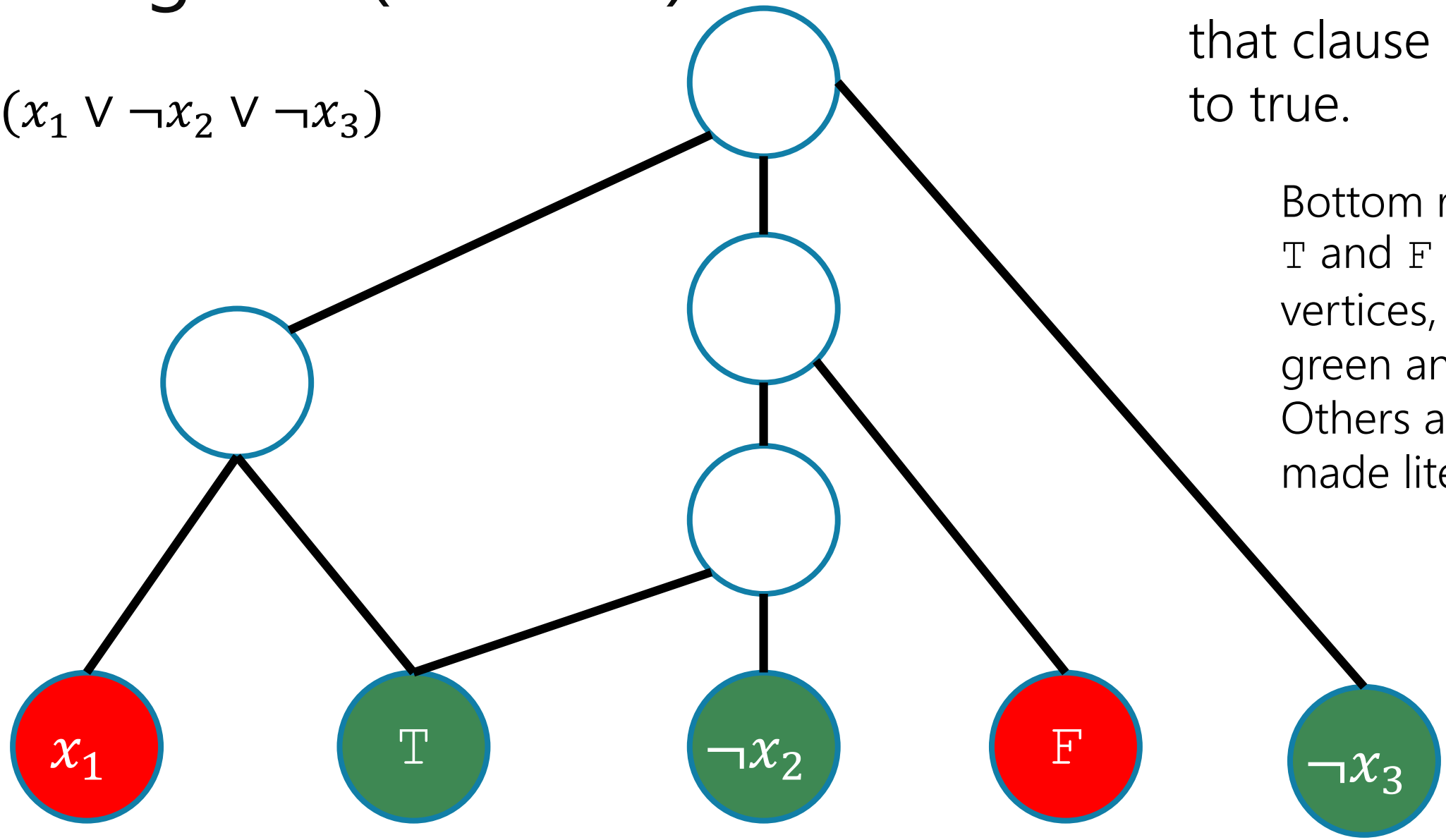
We can interpret a 3-coloring as a setting of the variables!

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)

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

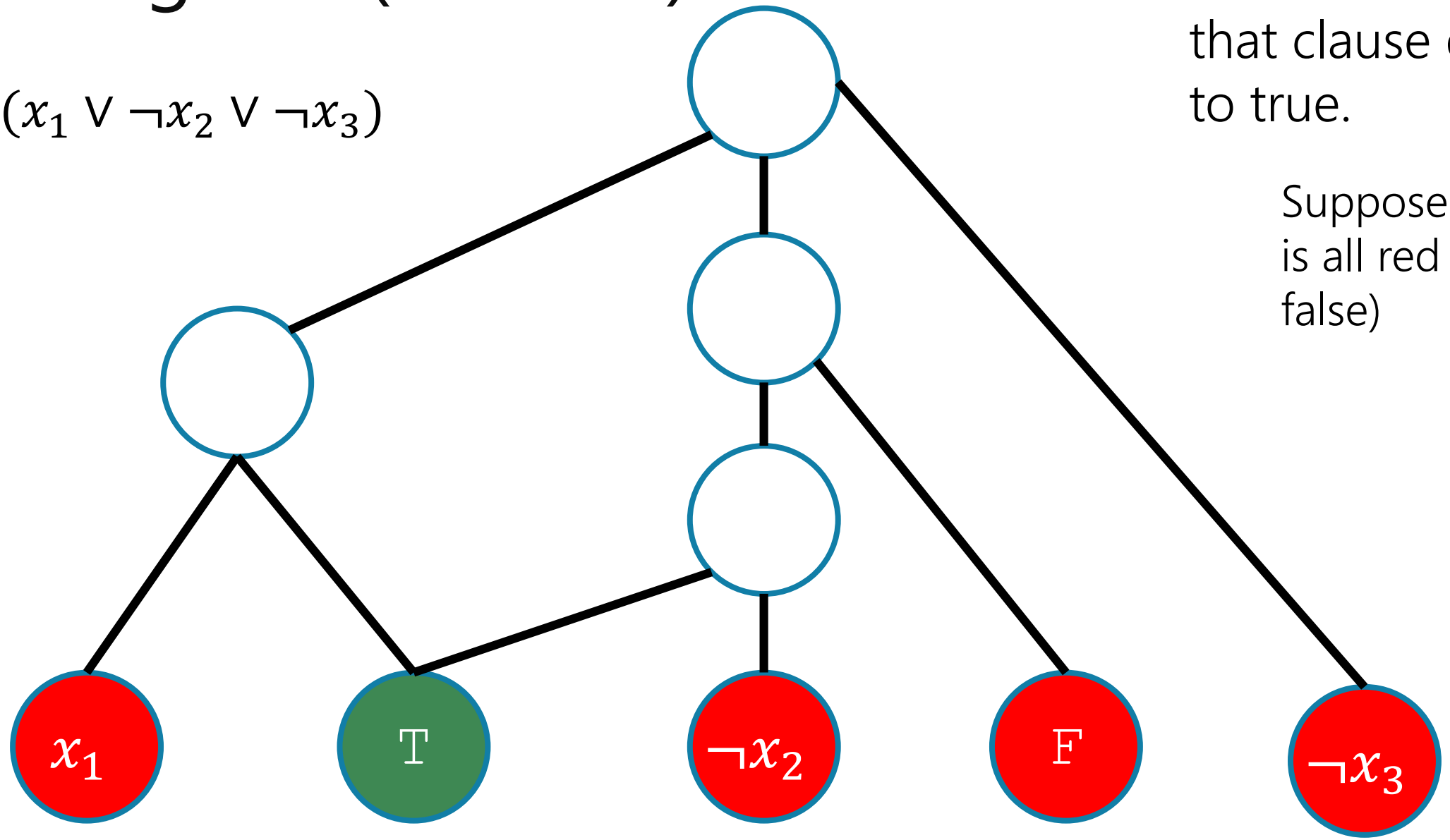


This tricky little graph can be 3-colored iff that clause evaluates to true.

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

Gadget 2 (clauses)

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

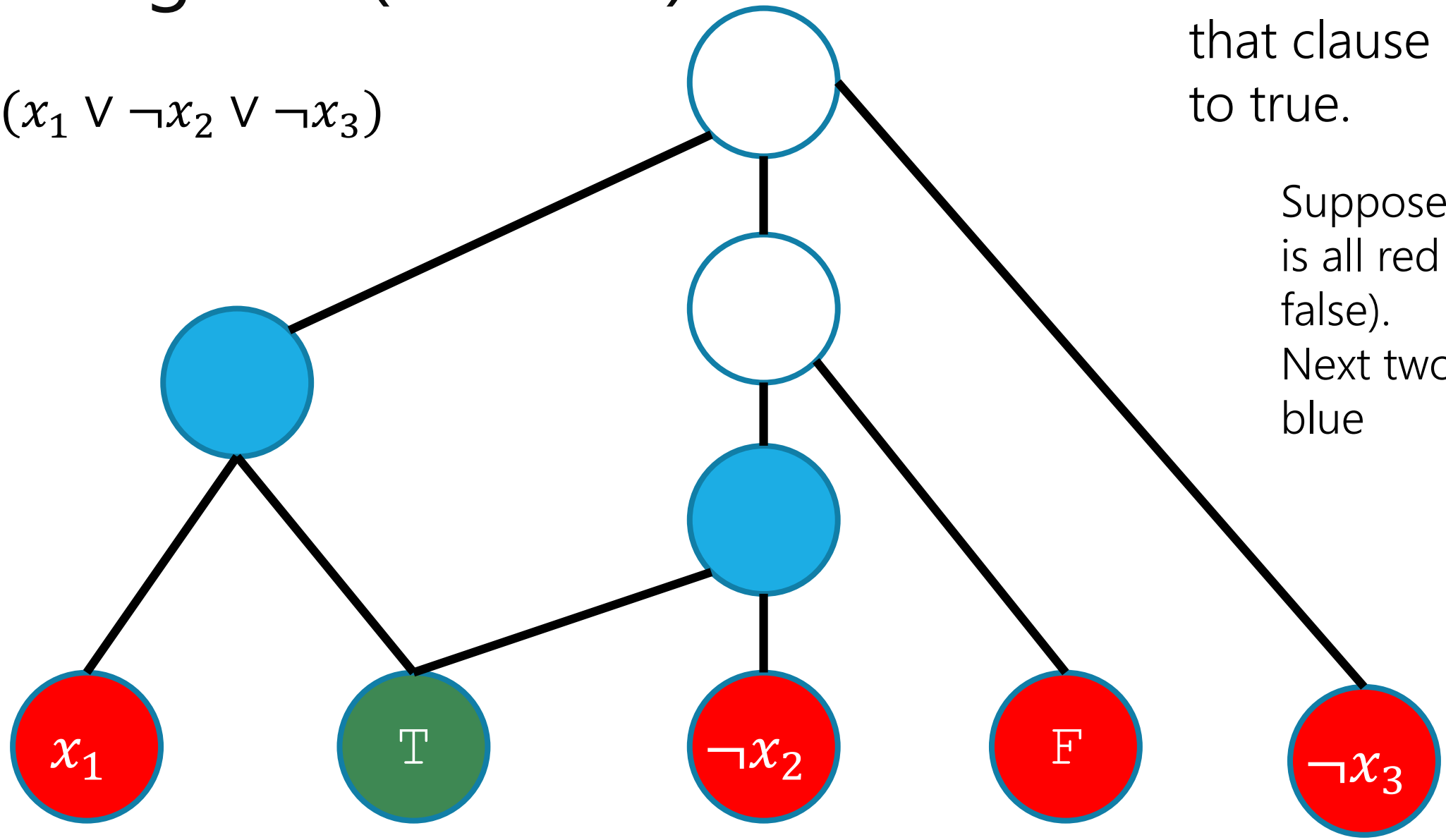


This tricky little graph can be 3-colored iff that clause evaluates to true.

Suppose bottom row is all red (clause is false)

Gadget 2 (clauses)

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

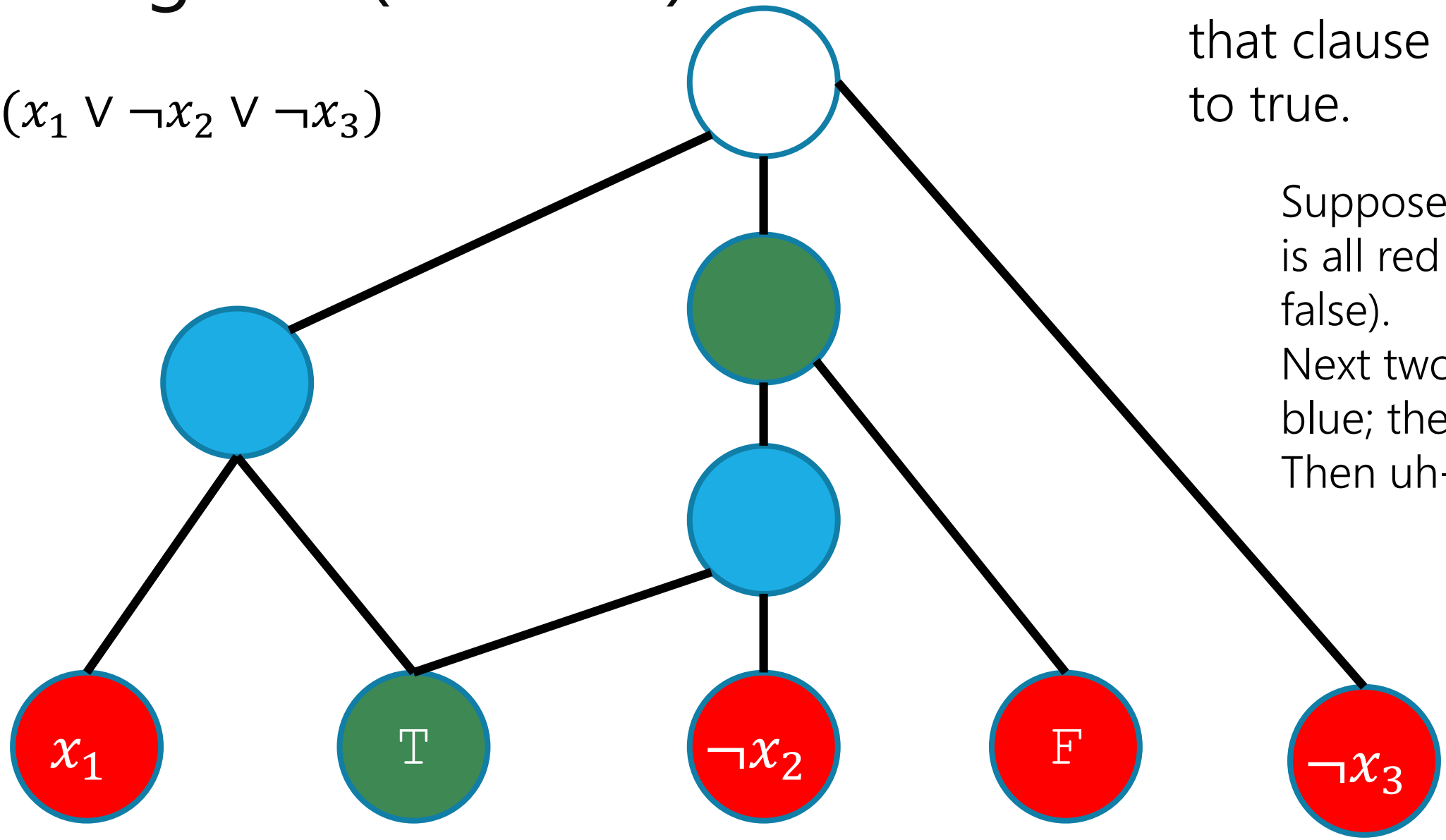


This tricky little graph can be 3-colored iff that clause evaluates to true.

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

Gadget 2 (clauses)

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

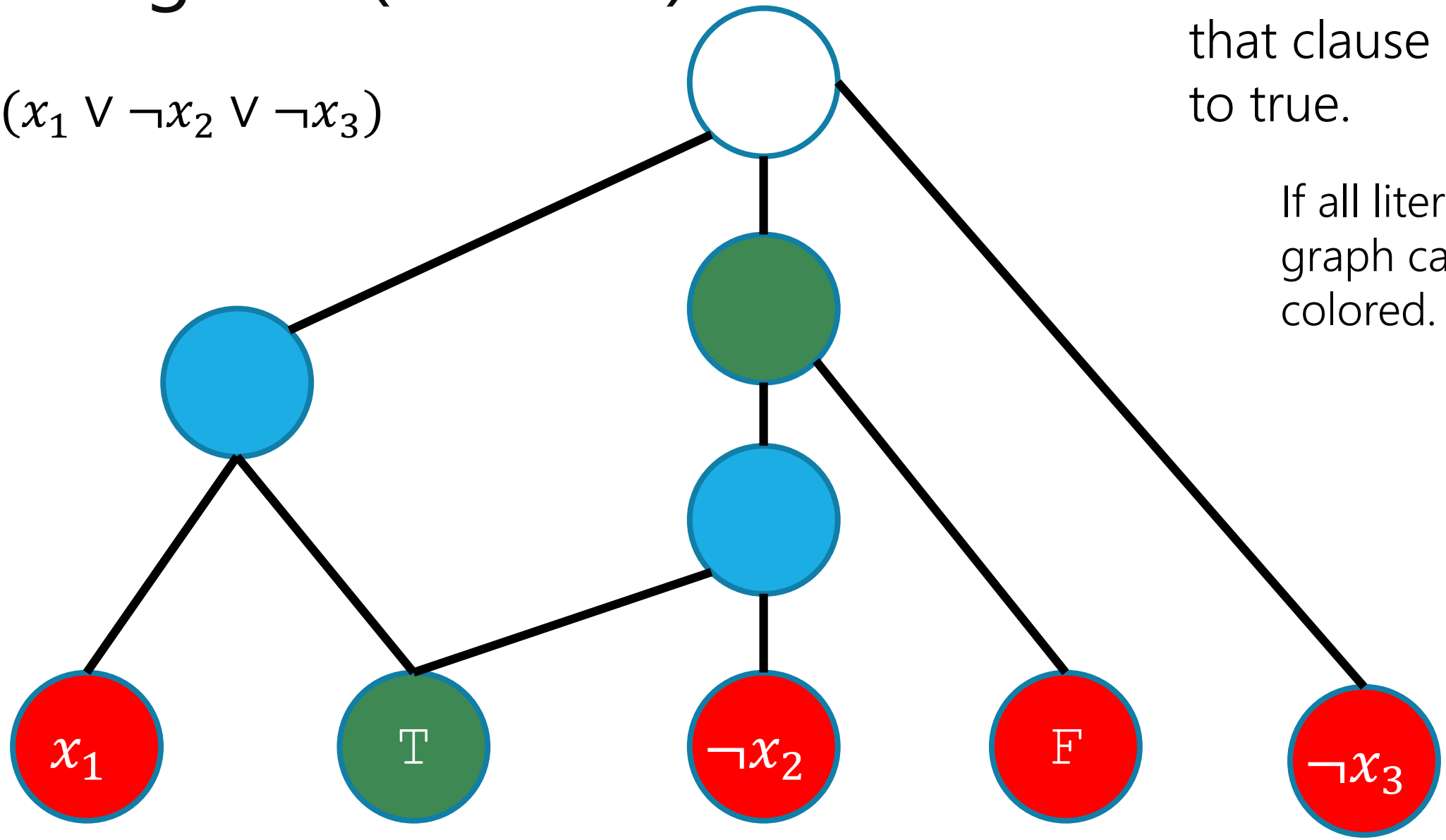


This tricky little graph can be 3-colored iff that clause evaluates to true.

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

Gadget 2 (clauses)

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

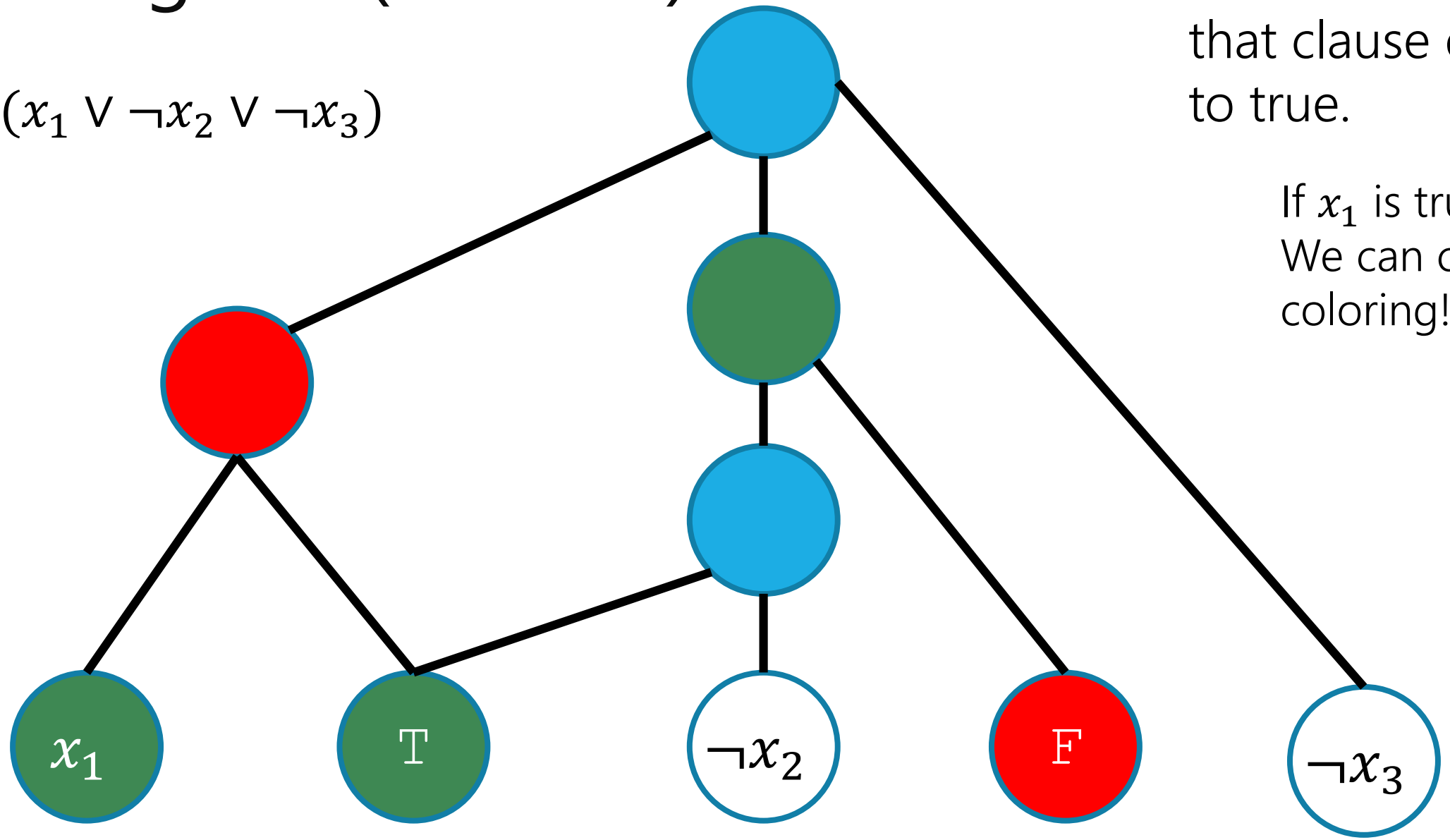


This tricky little graph can be 3-colored iff that clause evaluates to true.

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

Gadget 2 (clauses)

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

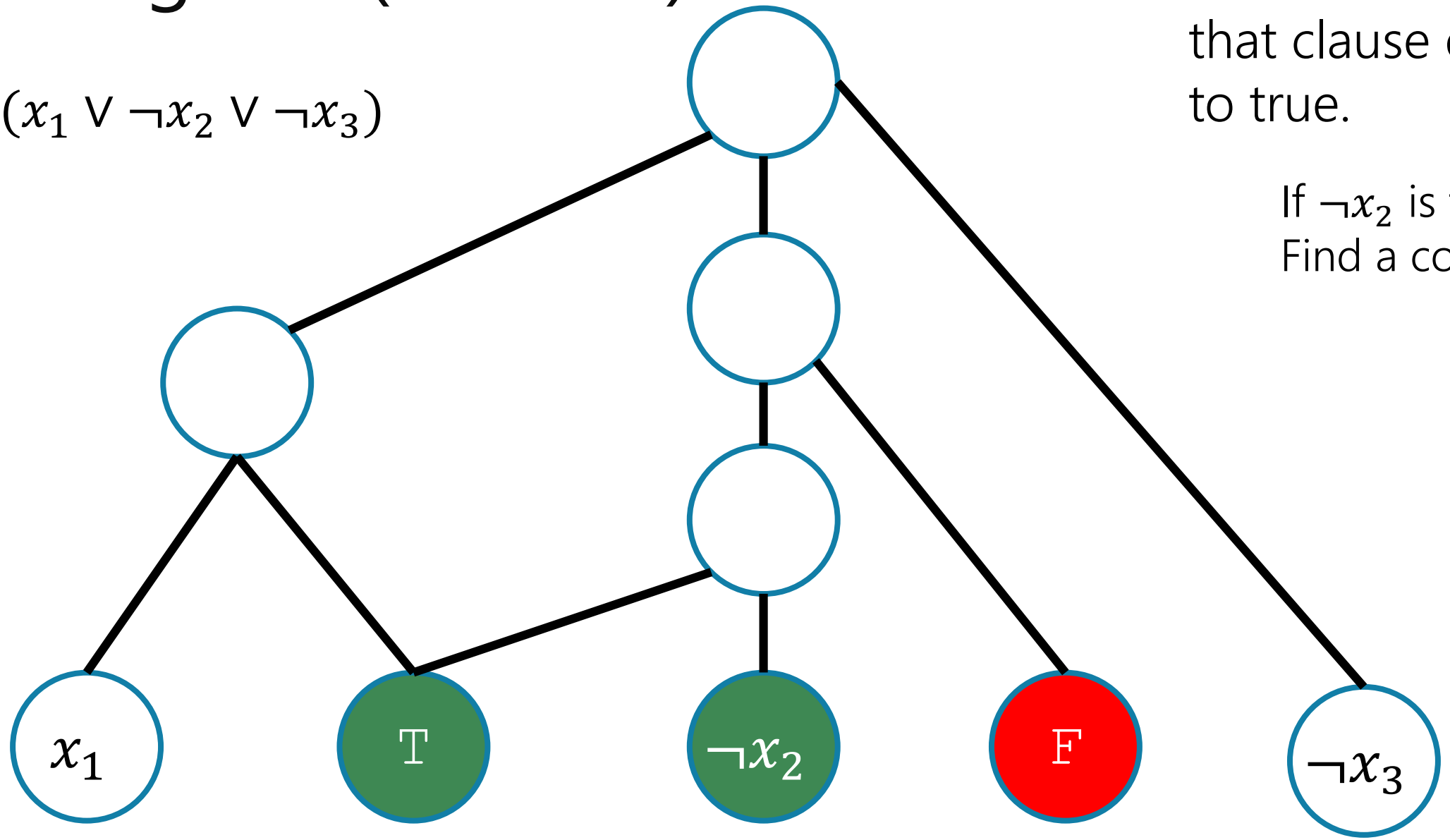


This tricky little graph can be 3-colored iff that clause evaluates to true.

If x_1 is true…
We can complete the coloring!

Gadget 2 (clauses)

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

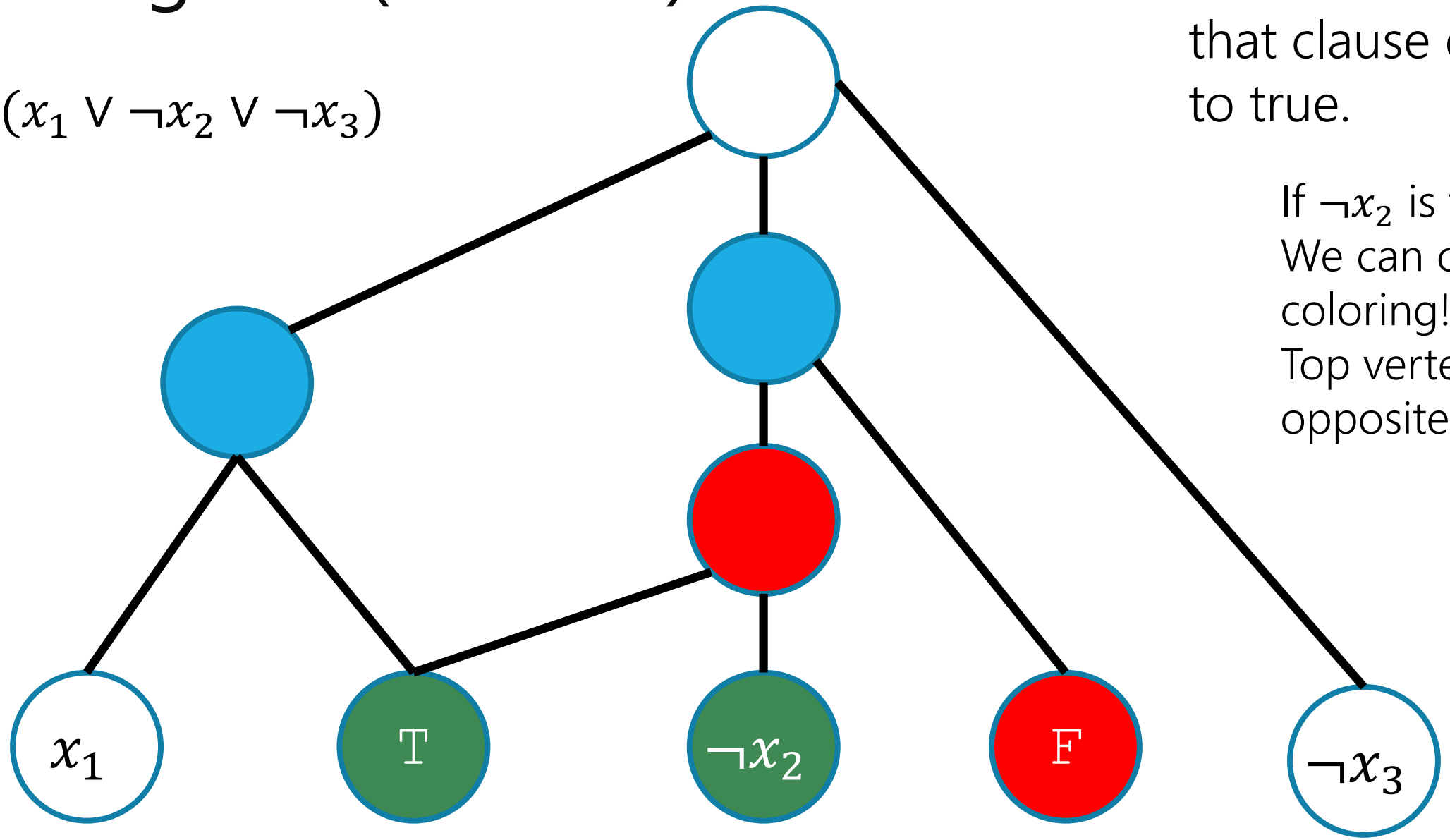


This tricky little graph can be 3-colored iff that clause evaluates to true.

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

Gadget 2 (clauses)

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

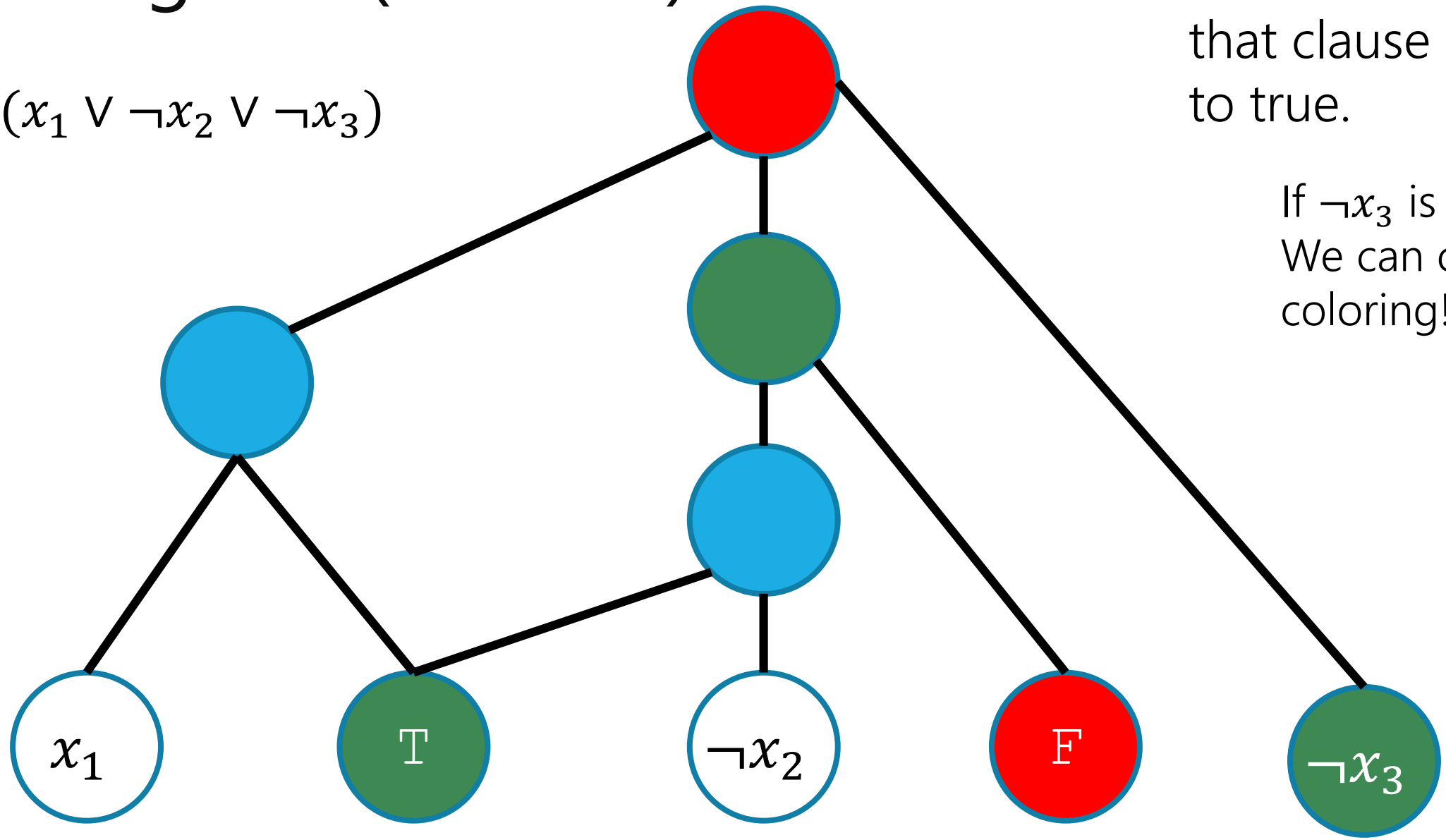


This tricky little graph can be 3-colored iff that clause evaluates to true.

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

Gadget 2 (clauses)

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

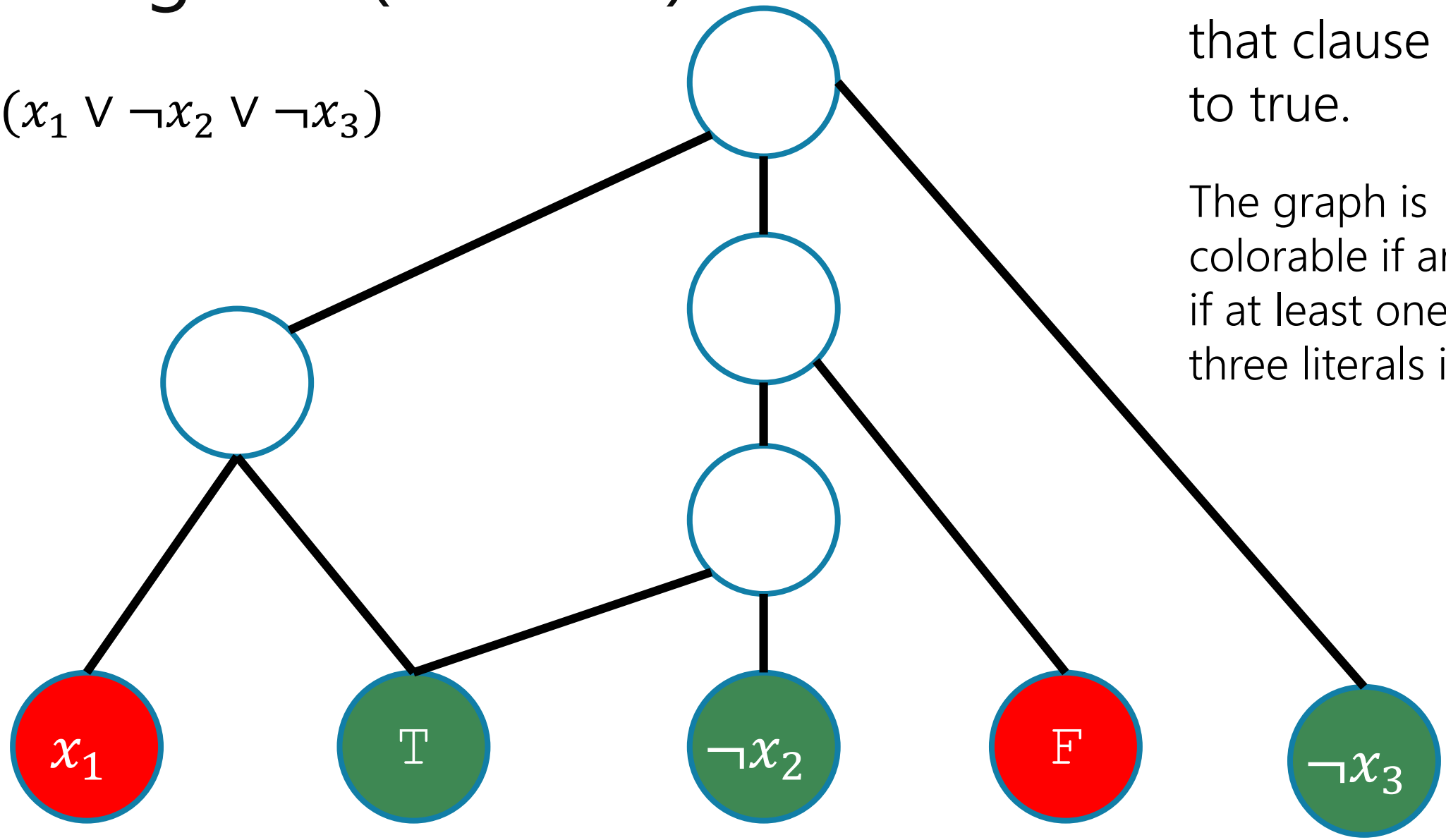


This tricky little graph can be 3-colored iff that clause evaluates to true.

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.

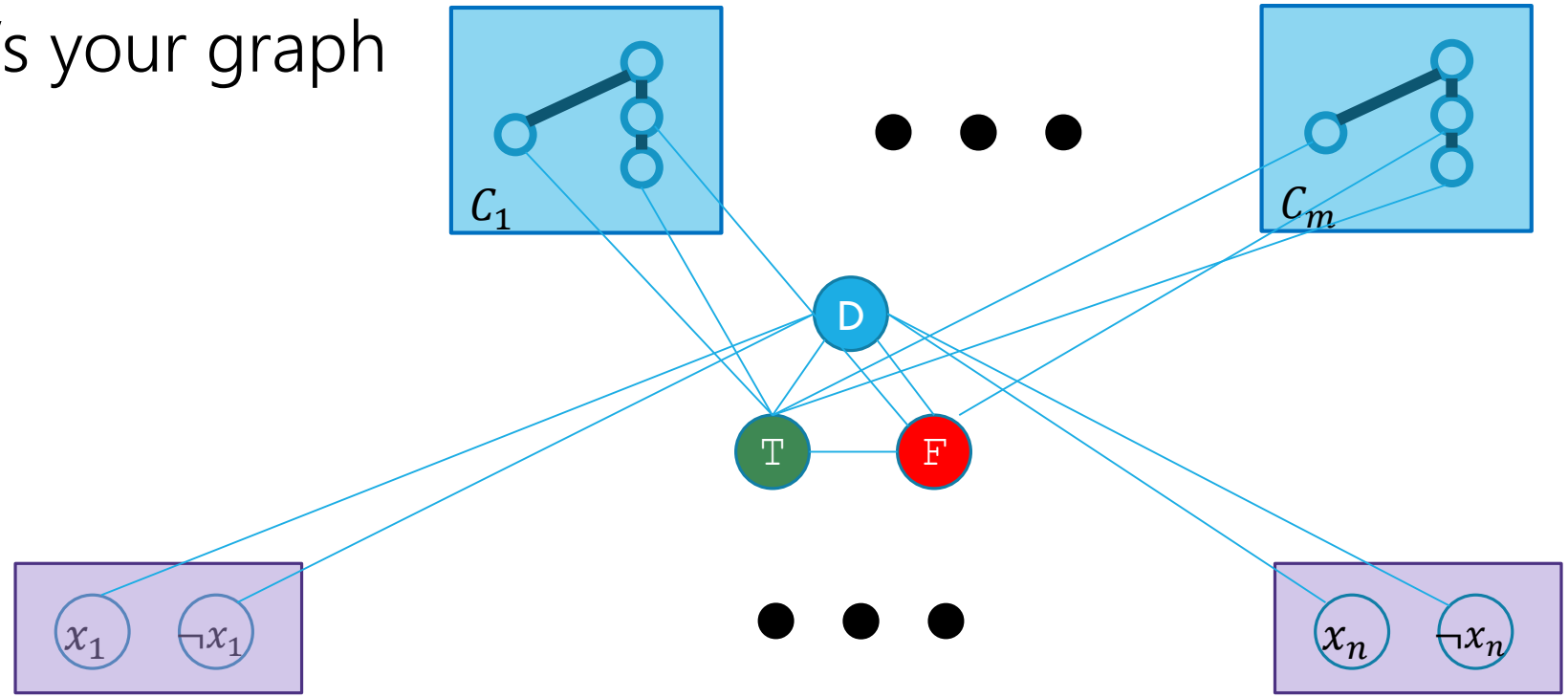
Putting it together

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.

Tips

Remember to write two separate arguments

- Too easy to lose details of the “hard” direction if you’re doing two at once
- And avoid contradiction (easy to do the same direction twice)

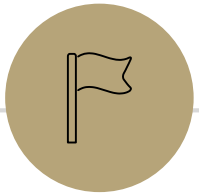
Usually: focus on the certificates

- That’s probably how you came up with the reduction in the first place.

Take a contrapositive (carefully!) if you feel stuck

Remember you can always change the reduction if the proof is too hard!

- Optimize for the proof.
- No one will ever run your reduction on a real computer, right?



Some Loose Ends

I have a problem

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

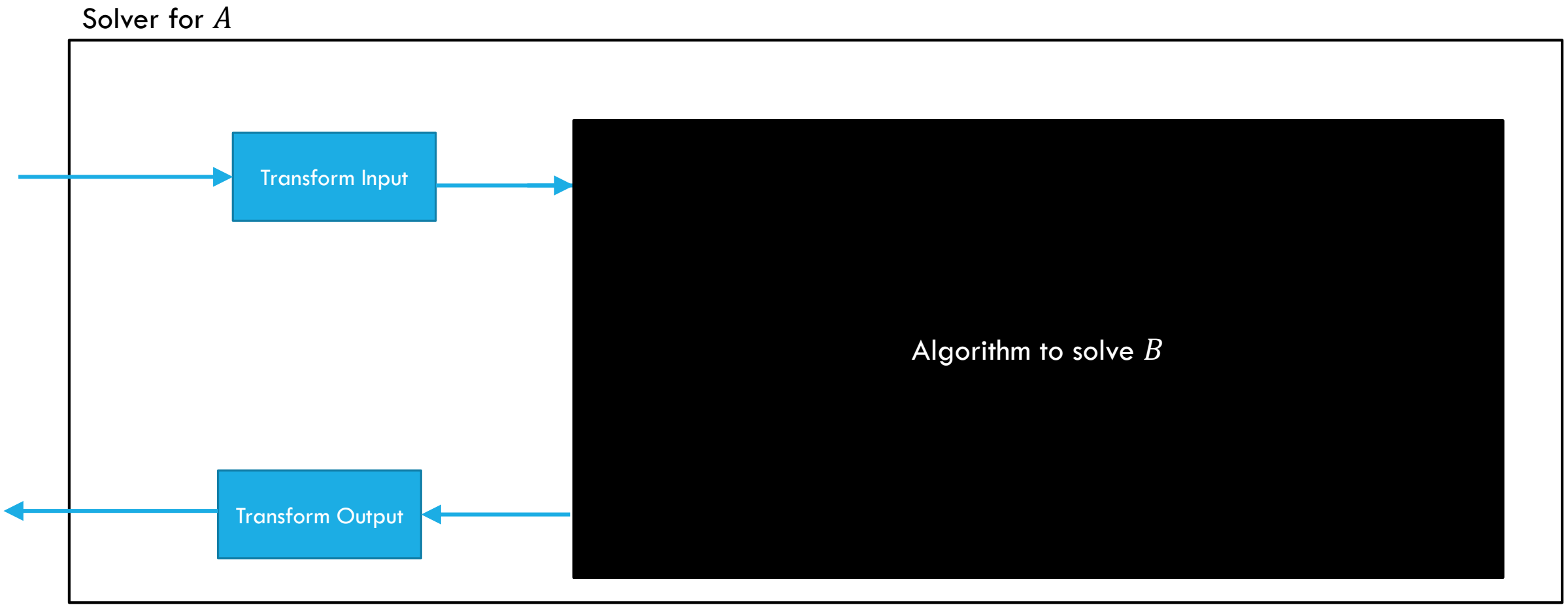
So difficult, it's probably NP-hard. How do I show it?

What does it mean to be NP-hard?

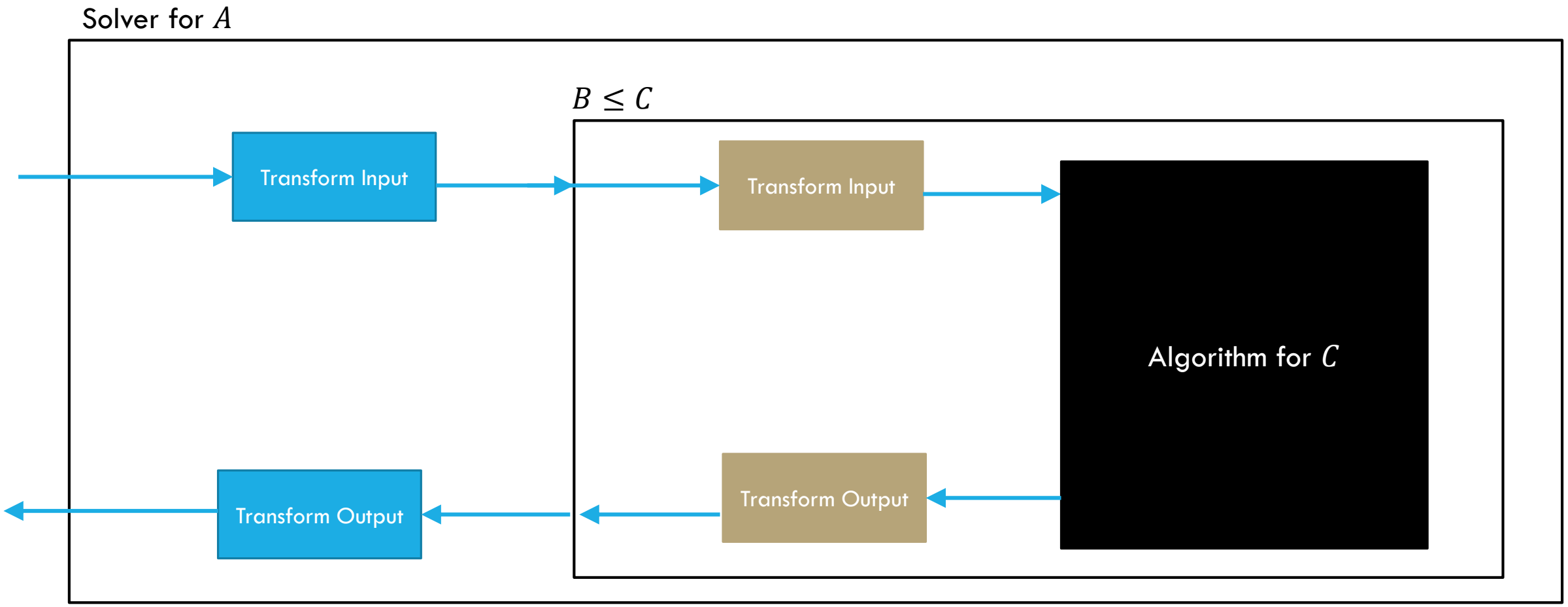
We need to be able to reduce any problem A in $\mathbf{NP}$ to C.

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



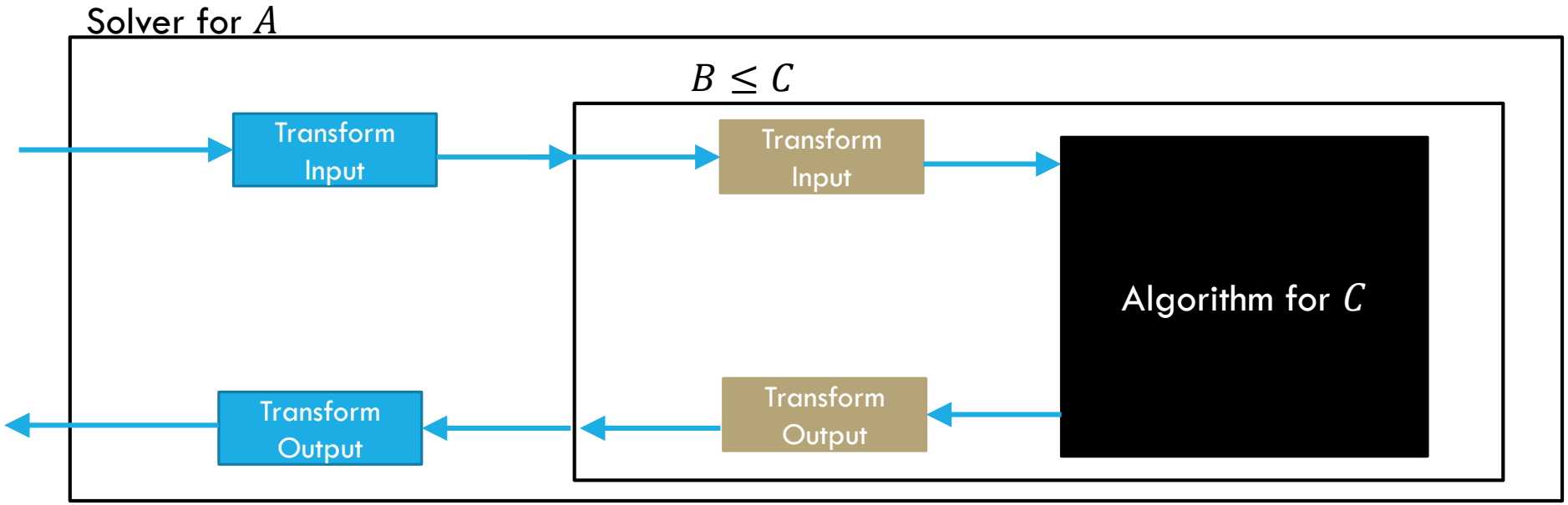
Is the implication true? $A \leq B \leq C \rightarrow A \leq C$



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

$A \leq B$

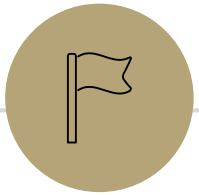
If I know B is not hard [I have an algorithm for it] then A is also not hard.

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! (more Friday)

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.

Optimization Problems

Putting away decision problems, we're now interested in optimization problems.

Problems where we're looking for the "biggest" or "smallest" or "maximum" or "minimum" or some other "best"

Vertex Cover (Optimization Version)

Given a graph G find the **smallest** set of vertices such that every edge has at least one endpoints in the set.

Much more like the problems we're used to!

What does NP-hardness say?

NP-hardness says:

We can't tell (given G and k) if there is a vertex cover of size k.

And therefore, we can't find the minimum one (write the reduction! It's good practice. Hint: binary search over possible values of k).

It doesn't say (without thinking more at least) that we couldn't design an algorithm that gives you an independent set that's only a tiny bit worse than the optimal one. Only 1% worse, for example.

How do we measure worse-ness?

Approximation Ratio

For a minimization problem (find the shortest/smallest/least/etc.)

If $OPT(\mathrm{G})$ is the value of the best solution for G, and $ALG(G)$ is the value that your algorithm finds, then ALG is an α approximation algorithm if for every G,

$$\alpha \cdot OPT(G) \geq ALG(G)$$

i.e. you're within an α factor of the real best.

Approximation Ratio

For a maximization problem (find the longest/biggest/most/etc.)

If $OPT(\mathrm{G})$ is the value of the best solution for G, and $ALG(G)$ is the value that your algorithm finds, then ALG is an α approximation algorithm if for every G,

$$OPT(G) \leq \alpha \cdot ALG(G)$$

i.e. you're within an α factor of the real best.

α switched sides! We want $\alpha \geq 1$ for both maximization and minimization to make it easier to think about.
If your maximization solution is "half-as-good" it's a 2-approximation.

Approximation Algorithms

Can easily fill an entire course…

Two prototypical examples (there are others!):

Combinatorial approaches

Techniques we've used much of this quarter!

But instead of focusing on the best aim for simple, and pretty good.

LP-based approaches

Write an LP

"round" to a 'pretty good' solution.

Recall: Vertex Cover

Given an (undirected) graph G, find a (small) set of vertices such that each edge has at least one endpoint in the set.

For trees: can use DP

For bipartite graphs: can use flow

For general graphs?: ummmmmm…

The problem is NP-complete “in general”

Best we’ve seen so far is that greedy doesn’t work!

Recall: Finding an approximation for VC

For every edge, at least one of u,v is in the minimum vertex cover.

But instead of checking which of u, v a good idea to add, just add them both!

```
While(G still has edges)
     Choose any edge (u,v)
     Add u to VC, and v to VC
     Delete u v and any edges touching them
EndWhile
```

We talked about this before.
During greedy algorithms week!

Does it work?

Do we find a vertex cover?

Is it close to the smallest one?

Does it run in polynomial time?

Do we find a vertex cover?

When we delete an edge, it is covered (because we added both u and v. And we only stop the algorithm when every edge has been deleted. So every edge is covered (i.e. we really have a vertex cover).

How big is it?

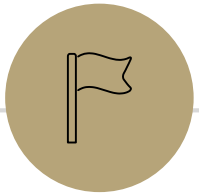
Let OPT be a **minimum** vertex cover.

Key idea: when we add u and v to our vertex cover (in the same step), at least one of u or v is in OPT.

Why? (u, v) was an edge! OPT covers (u, v) so at least one is in OPT.

So how big is our vertex cover? At most twice as big!

This is a 2-approximation for vertex cover!



One More Problem

At a VERY high-level

Another Algorithm

Lets try to approximate Travelling Salesperson.

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.

Some assumptions:

1. The graph is undirected.

2. The graph is complete (every edge is there) – the edges might represent series of roads rather than individual streets. Weight is how much gas you need to travel.

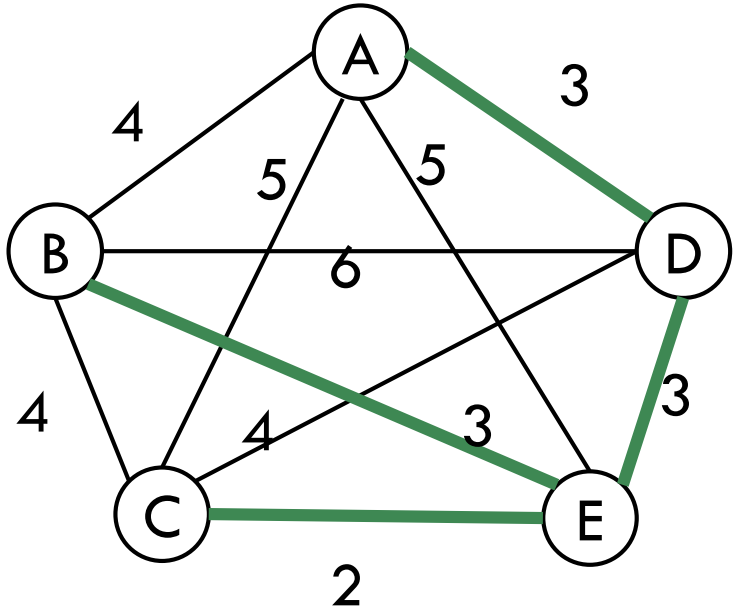
2. The weights satisfy the "triangle inequality" (it's faster to go from x to y directly than it is to go from x to y through x).

TSP starting point

What would be a good baseline?

Something we **can** calculate that would at least connect things up for us.

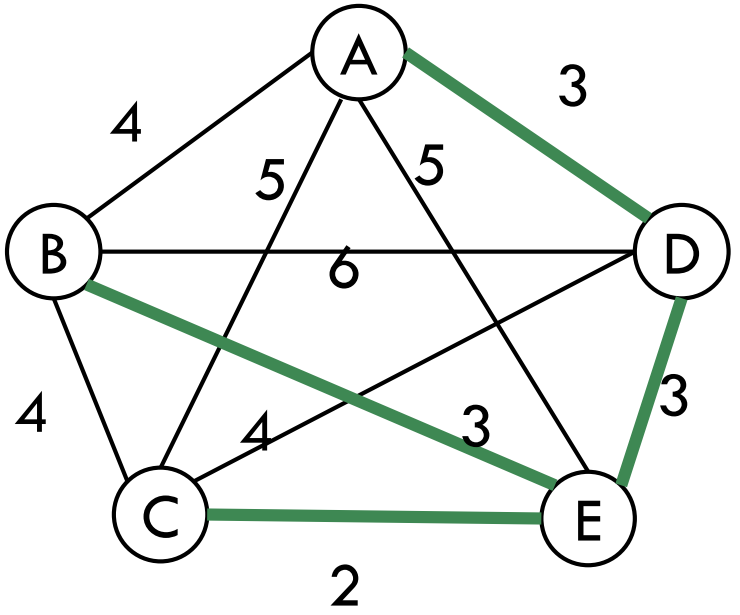
A Minimum Spanning Tree!



From MST to Tour

How do we get from start to every vertex and back?

Make the starting point the root, do a traversal (DFS) of the graph!



Why not BFS? Because the "next vertex" isn't always right next to you! (not a problem in this example, but very bad if you have a very tall tree)

How much gas do we use in DFS? We use each edge twice

If D is the starting point:
Go from D to A, back to D
To E Down to B back to E to C
Back to E back to D.

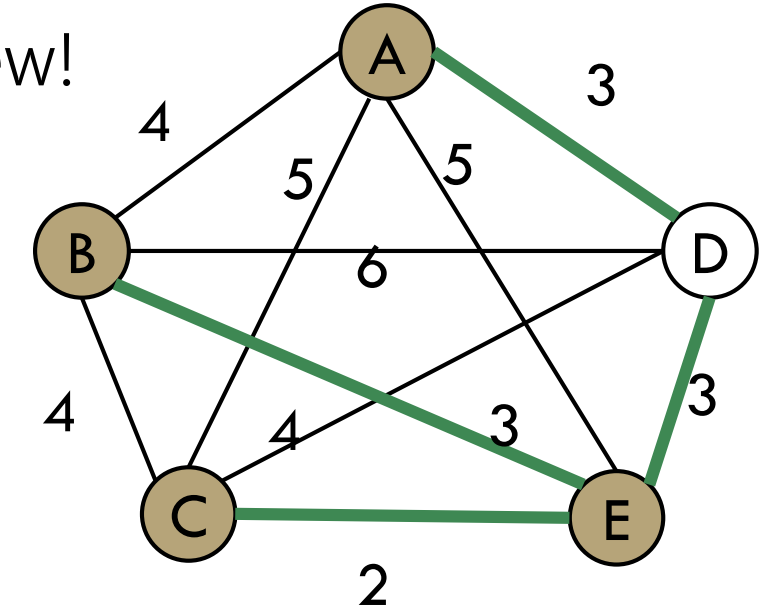
Doing a Little Better

Using each edge twice is potentially a little wasteful. Can we do better?

The biggest problem is vertices of odd degree. The last time we enter that vertex, the only way out is an already used edge.

And that's definitely not taking us somewhere new!

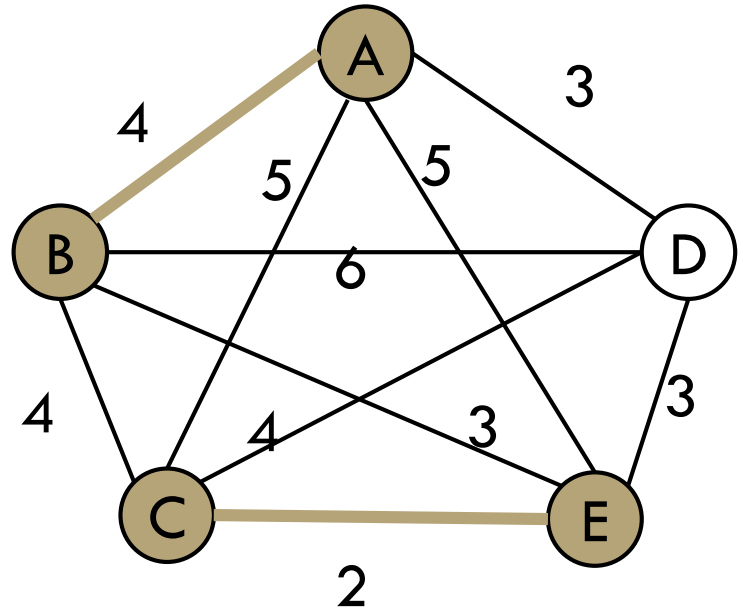
So lets add some possible ways out.



What would help?

A matching would help! (i.e. a set of edges that don't share endpoints)

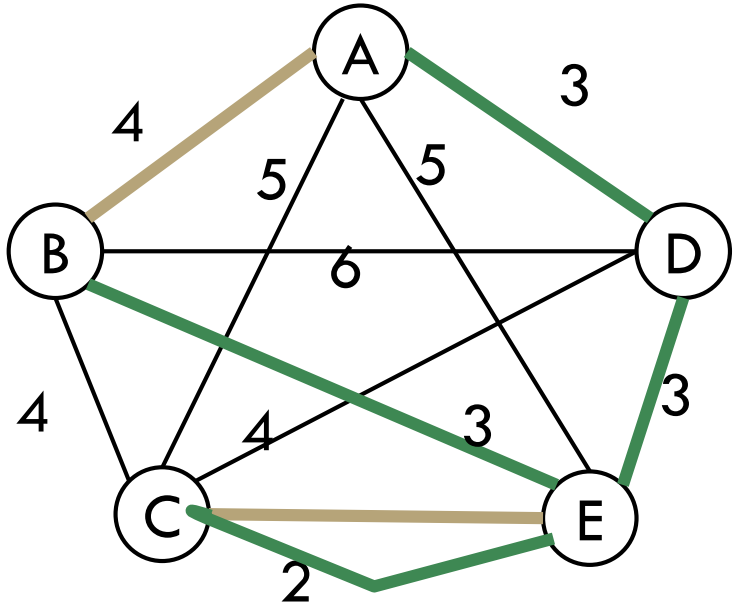
Specifically a minimum weight matching.



You can find one of those efficiently (just trust me)

Add those edges in (if they're already in the MST, make an extra copy)!

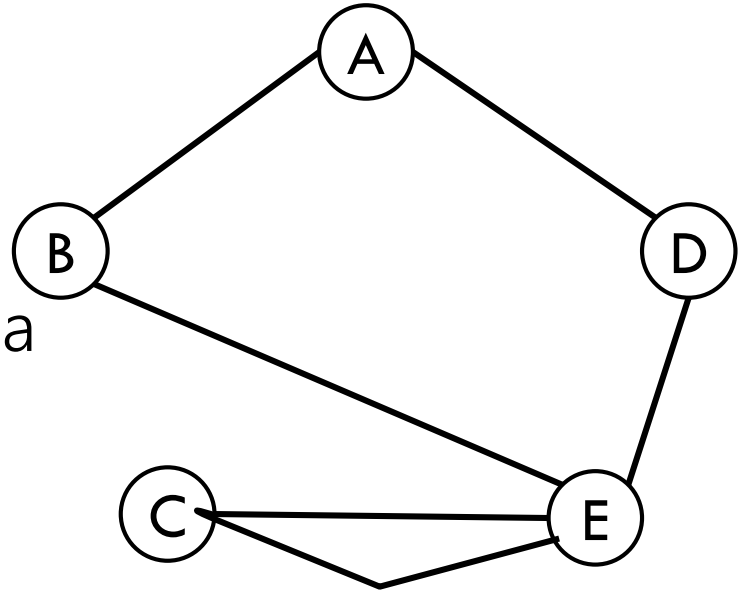
So we now have the MST AND the minimum weight matching on the odd edges.



Did It Help?

So…now every vertex has even degree…but there's not a nice order anymore.

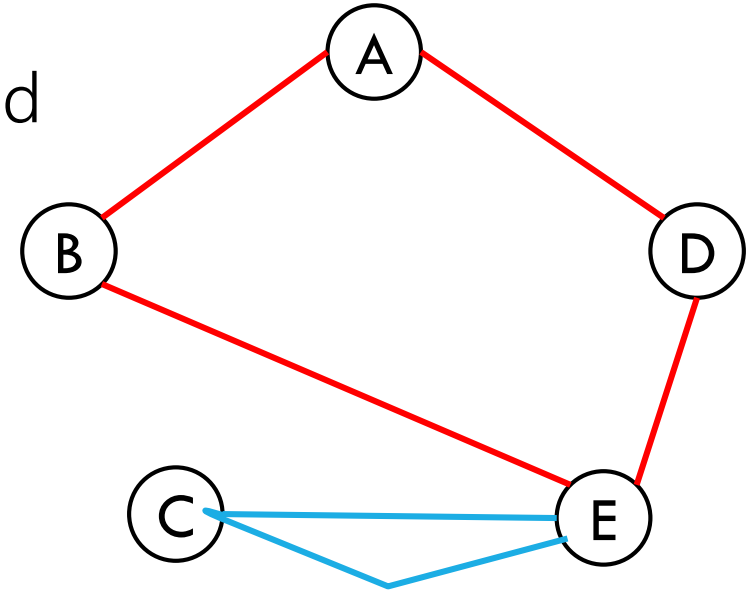
We'll have to find one.



Start from the starting point, and just follow any unused edge!

Because every vertex has even degree, (except for the starting vertex) when you go in, you can come out! So you can't "get stuck"

What if you get back to the start and end up with unused edges? Find a visited vertex one is adjacent to and "splice in" the cycles.



D,A,B,E,D is found first. E,C,E found next. After splicing: D,A,B,E,C,E,D. is the final tour

Is it a good approximation algorithm?

We will visit every vertex at least once!

Every vertex had degree at least one (because we started with an MST!)

So by the end of the process, we had degree at least two on every vertex.

And we go back and use all the edges we selected. So we visit every vertex, and we start and end at the same place.

Is it a good approximation algorithm?

What does our algorithm produce?

At most $\frac{3}{2}OPT$ (at most $\mathbf{1.5}$ times the weight of the optimal tour)

Why? We use every edge once, that's one MST plus the weight of the matching.

How much is the MST? Less than OPT. (OPT has a spanning tree inside it!)

How much is the matching? Less than $\frac{1}{2}OPT$. (OPT is less than a tour on the odd vertices, and a tour on the odd vertices is made up of two matchings)

Approximating TSP

We found a $\frac{3}{2}$-approximation for TSP!

The algorithm is called "Christofides Algorithm"

It's almost 50 years old.

The best approximation is $\frac{3}{2} - \epsilon$ where $\epsilon \approx 10^{-36}$

Developed by three researchers at UW **in the last two years**.

https://arxiv.org/pdf/2007.01409.pdf

Summary

Coping with NP-hardness.

1. Understand your problem really well (make sure you're not solving an easy special case).

2. Prove the problem really is NP-hard.

3. Try a band-aid (SAT library, Integer programming library, etc.)

4. Try to find a good-enough exponential time algorithm or an approximation algorithm.