Even More Reductions

CSE 417 Winter 24
Lecture 22

Where are we?

Last week: What do we do when we don’t know how to solve a problem?

$A \leq B$ (“A reduces to B” “we reduced from A to B”)

If we write a polynomial-time algorithm for B then we get a polynomial time algorithm for A for free!

Definitions of P, NP.

Today

Today:
More definitions: what are the “hard” problems we want to avoid?
More reduction examples.

Overarching Goal:

Even problems that look very different can be reduced to each other.

We’ll do a couple examples together, so you find it plausible.

What’s the significance of these reductions?

Later this week – what do you do when a problem you’re trying to solve 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.

NP (stands for "nondeterministic polynomial")

The set of all decision problems such that if the answer is YES, there is a proof of that which can be verified in polynomial time.

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.

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.

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 $x_1, \ldots, x_n$

A list of constraints, all of which must be met.

Each constraint is of the form:

```
( (xi == <T,F>) || (xj == <T,F>) || (xk == <T/F>) )
```

ORed together, always exactly three variables, you can choose T/F independently for each.

Output: true if there is a setting of the variables where all constraints are met, false otherwise.

Why is it called 3-SAT? 3 because you have 3 variables per constraint
SAT is short for “satisfiability” can you satisfy all of the constraints?

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.

NP (stands for "nondeterministic polynomial")

The set of all decision problems such that if the answer is YES, there is a proof of that 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

More Starting Points

We have one NP-hard problem (3-SAT). It'd be nice if we had more…

I'm just going to give us more (if you're interested in proving these NP-complete, many are here)

3-coloring is NP-complete.

Hamiltonian Path (given a directed graph, is there a path that visits every vertex exactly once?) is NP-complete.

More Reduction Facts

I have a problem

My problem C is hard.

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

What does it mean to be NP-hard?

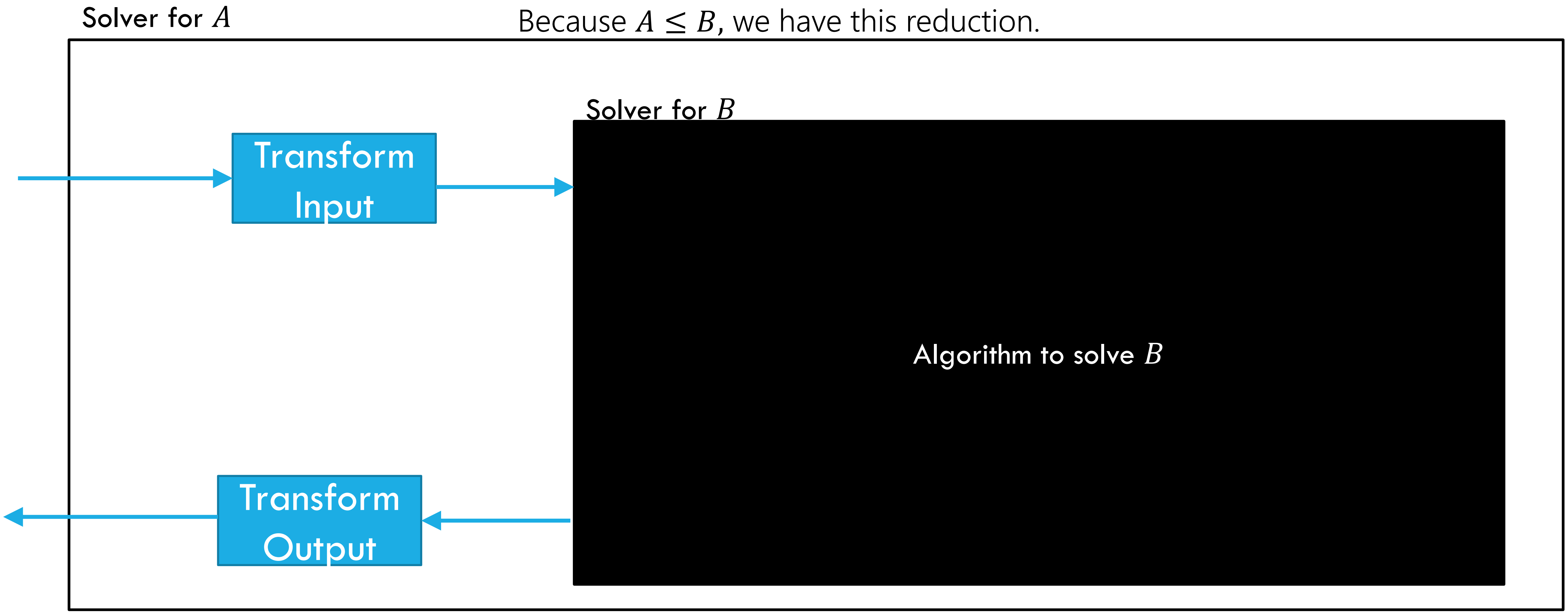
We need to be able to reduce any problem A to C.

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!

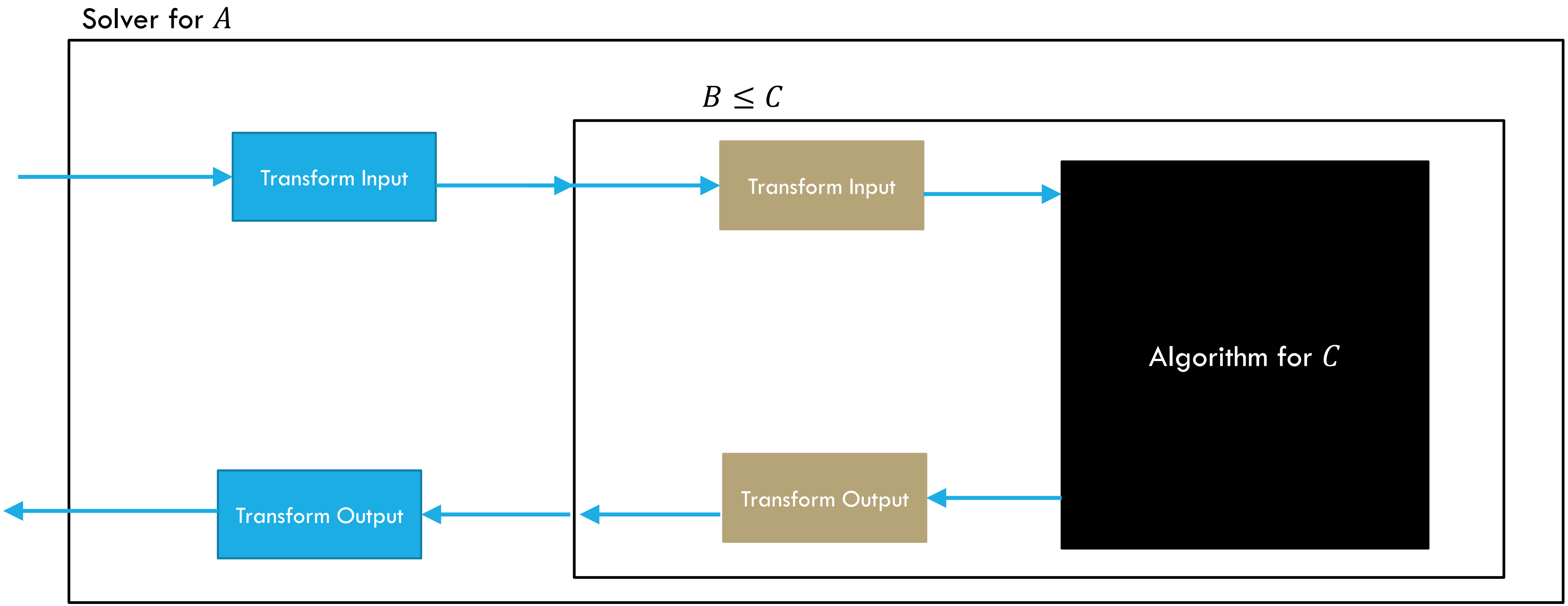
$$A \leq B \leq C \rightarrow A \leq C$$

Is that true?

Because $A \leq B$, we have this reduction.



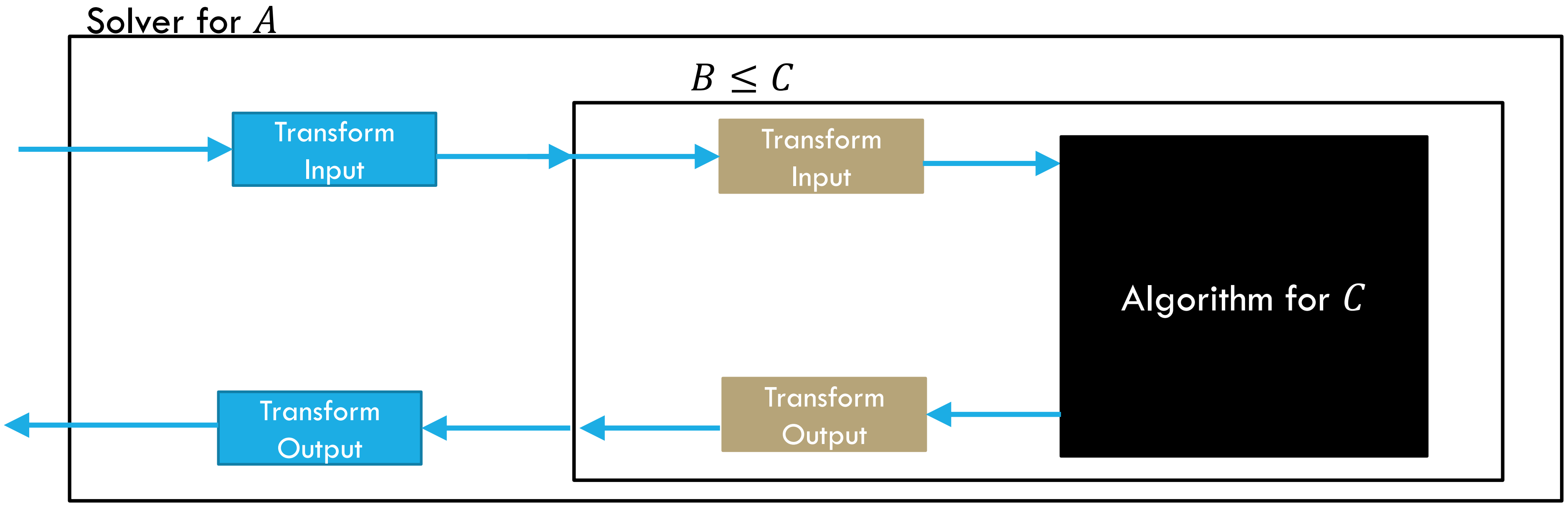
$$A \leq B \leq C \rightarrow A \leq C$$



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

Why does it work? Because our reductions work!

How long does it take? Still polynomial time! (Even if the input gets bigger at each step, it still can't get bigger than a polynomial). And we don't need a B solver, the reduction is the solver! We only use a C solver so it's "really" a reduction.



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.

More Reductions

Another Reduction

More reductions between different looking problems.

We'll show 3-Coloring ≤ 3-SAT.

🚨 This reduction does not show 3-coloring is NP-hard. 🚨

It is, but we'd need the reduction to go the other direction to demonstrate it.

3-Coloring $\leq$ 3-SAT

Need to transform a 3-coloring instance (a problem about a graph)

To a 3-SAT instance (a problem about variables and constraints)

Those look very different!!

It's going to take some creativity to make the conversion.

Your main takeaway from this lecture is **not** these particular reductions or these particular techniques.

Your takeaway is "wow, even if problems can look pretty different, they can be closely related!"

3-Coloring $\leq$ 3-SAT

Need to transform a 3-coloring instance (a problem about a graph)

To a 3-SAT instance (a problem about variables and constraints).

3-SAT talks about Boolean variables and constraints.

What variables could we use to describe coloring?

What constraints would the coloring impose?

3-Coloring $\leq$ 3-SAT

Variables: is this vertex red? Blue? Green? (can't have just one variable, let's just have three).

Constraints?

If (u, v) is an edge, then u and v are different colors.

u gets exactly one color.

3-Coloring $\leq$ 3-SAT

Variables: is this vertex red? Blue? Green? (can't have just one variable, let's just have three).

$x_{u,r}, x_{u,b}, x_{u,g}$

Constraints?

If (u, v) is an edge, then u and v are different colors.

u gets exactly one color.

These are going to take a bit of work:

Edge Requirements

We need to make sure the edges are different colors.

As an example

If u is red, and (u, v) is an edge, then v is blue OR v is green.

$$x_{u,r} == False \,||\, x_{v,b} == True || \, x_{v,g} == True$$

Law of implication: "if p then q" is equivalent to $!p||q$.

Edge Constraints

All combinations constraints:

English – for each edge $(\boldsymbol{u}, \boldsymbol{v})$	SAT
If u is red, then v is blue or green	$x_{u,r} == False \mid\mid x_{v,b} == \text{True} \mid\mid x_{v,g} == True$
If u is blue, then v is red or green	$x_{u,b} == False \mid\mid x_{v,r} == \text{True} \mid\mid x_{v,g} == True$
If u is green, then v is red or blue	$x_{u,g} == False \mid\mid x_{v,r} == \text{True} \mid\mid x_{v,b} == True$
If v is red, then u is blue or green	$x_{v,r} == False \mid\mid x_{u,b} == \text{True} \mid\mid x_{u,g} == True$
If v is blue, then u is red or green	$x_{v,b} == False \mid\mid x_{u,r} == \text{True} \mid\mid x_{u,g} == True$
If v is green, then u is red or blue	$x_{v,g} == False \mid\mid x_{u,r} == \text{True} \mid\mid x_{u,b} == True$

Some of these aren't strictly necessary (are implied by the others) but better safe than sorry.

Are those constraints enough?

Suppose we used those constraints, ran the 3-SAT solver on these constraints, and just return what it says.

Are we done? If this reduction is correct, explain to each other why! If it's not correct explain why not.

English – for each edge (u, v)	SAT
If u is red, then v is blue or green	$x_{u,r} == False \mid\mid x_{v,b} ==$ True $\mid\mid x_{v,g} == True$
If u is blue, then v is red or green	$x_{u,b} == False \mid\mid x_{v,r} ==$ True $\mid\mid x_{v,g} == True$
If u is green, then v is red or blue	$x_{u,g} == False \mid\mid x_{v,r} ==$ True $\mid\mid x_{v,b} == True$
If v is red, then u is blue or green	$x_{v,r} == False \mid\mid x_{u,b} ==$ True $\mid\mid x_{u,g} == True$
If v is blue, then u is red or green	$x_{v,b} == False \mid\mid x_{u,r} ==$ True $\mid\mid x_{u,g} == True$
If v is green, then u is red or blue	$x_{v,g} == False \mid\mid x_{u,r} ==$ True $\mid\mid x_{u,b} == True$

Are those constraints enough?

Suppose we used those constraints, ran the 3-SAT solver on what we got.

If the graph is 3-colorable, then the 3-SAT instance has a solution (pick your favorite coloring and set the variables to match that coloring).

If the graph is not 3-colorable

The 3-SAT solver will still say there's a solution for this instance. Just set every variable to false!

Consistency Constraints

Reductions often need extra constraints/structures.

When you say “I want this variable to mean X” you really need to force the variable to mean X.

So if you want a coloring, you need to make sure even “well, yeah of course that’s what I meant” requirements are explicit.

What are we missing? Every vertex needs exactly one color.

Consistency

More constraints:

English – for each vertex	SAT
If u is red, then u cannot be blue	$x_{u,r} == False \mid\mid x_{u,b} ==$ False
If u is red, then u cannot be green	$x_{u,r} == False \mid\mid x_{u,g} ==$ False
If u is blue, then u cannot be red	$x_{u,b} == False \mid\mid x_{u,r} ==$ False
If u is blue, then u cannot be green	$x_{u,b} == False \mid\mid x_{u,g} ==$ False
If u is green, then u cannot be red	$x_{u,g} == False \mid\mid x_{u,r} ==$ False
If u is green, then u cannot be blue	$x_{u,g} == False \mid\mid x_{u,b} ==$ False
u gets a color!	$x_{u,r} == True \mid\mid x_{u,g} == True \mid\mid x_{u,b} == True$

From 2 to 3.

Hang on! Is this allowed in 3-SAT?

$x_{u,r} == False \,||\, x_{u,b} == \text{False}$

The definition said 3 items each…

A trick to fix it. Make two copies, or in a dummy variable d being True in one and false in the other.

$x_{u,r} == False \,||\, x_{u,b} == \text{False} \,||\, d == True$

$x_{u,r} == False \,||\, x_{u,b} == \text{False} \,||\, d == False$

d will make one of the two true. The other copy is satisfied if and only if the original one was.

Reduction

Given a graph G, we make the following 3-SAT instance

Variables: $x_{u,r}, x_{u,g}, x_{u,b}$ for each vertex u

Constraints: As described on the last few slides.

Run a 3SATSolver.

Return whatever it returns.

Running Time?

We need n variables and $6m + 13n$ constraints.

Making them is mechanical, definitely polynomial time.

Correctness

Our correctness proofs are usually:

Certificate for 3-coloring becomes a certificate for 3-SAT

The only certificates for 3-SAT come from certificates for 3-coloring

Let's start with

If G is 3-colorable, then the reduction says YES.

Correctness

If G is 3-colorable, then the reduction says YES.

If G is 3-colorable, then there is a 3-coloring. From any 3-coloring, set $x_{u,r}$ to be true if u is red and false otherwise.

$x_{u,g}$ to be true if u is green and false otherwise.

$x_{u,b}$ to be true if u is blue and false otherwise.

The constraints are satisfied (for the reasons listed on the prior slides)

So the 3-SAT algorithm must say the constraints are satisfiable, and the reduction returns true!

Correctness

If the reduction returns YES, then G was 3-colorable.

Correctness

If the reduction returns YES, then G was 3-colorable.

If the reduction returns YES, then the 3-SAT algorithm returned YES, so the 3-SAT instance had a satisfying assignment.

We can convert the variables to a coloring:

For every u, exactly one of $x_{u,r}, x_{u,g}, x_{u,r}$ is true. We have a constraint requiring at least one, and constraints preventing more than one variable for the same vertex being true.

Color the vertices the associated colors. Since every vertex is colored, at least one of the constraints is active for each edge, so we have a valid coloring.

One More Thought

Vertex Cover is NP-complete (you can do a reduction from independent set. It's good practice!)

But we wrote a polynomial time algorithm for vertex cover didn't we? We wrote two– a DP one and an LP one. What's going on?

The algorithms we saw only handled special cases – Vertex cover on trees or vertex cover on bipartite graphs. We didn't prove $P = NP$. We carved off part of the problem that was easy and solved that (solved only the "easy" instances).

Hamilton

On a directed graph G:

A Hamiltonian Path is a path that visits every vertex exactly once.

A Hamiltonian Cycle is a Hamiltonian Path with an extra edge connecting the first vertex to the last vertex.

Assume that Hamiltonian Path is NP-hard (it is)

Use that to prove Hamiltonian Cycle is NP-hard.

Pollev.com/robbie

Which direction?

Reduce FROM the known hard problem TO the new problem.

Want to show Hamiltonian Path $\leq$ Hamiltonian Cycle.

Reduction

Let G be the instance for Hamiltonian Path

Make H a copy of G with an extra vertex u added.

For every vertex v, add an edge from v to u and from u to v

Run the Hamiltonian Cycle Solver on H

Return what it returns.

Correctness

If G has a Hamiltonian Path,

Then there is a Hamiltonian Cycle in H by following the path in G going to u and going back to the start.

So we correctly return YES.

Correctness

If our reduction returns YES, then H had a Hamiltonian Cycle.

Delete u (and its edges from the cycle)

Since a Hamiltonian Cycle visits each vertex exactly once, what remains is a path that visits each vertex (except u) exactly once.

That’s a Hamiltonian Path!

So G has a Hamiltonian Path.

Reductions

We saw a reduction between two very similar (on the surface) problems when we reduced from 2-coloring to 3-coloring.

The real power of reductions is when problems look very different on the surface but you can still reduce from one to the other.

We're going to do a couple more reductions with varying levels of differences between the problems.

3-SAT $\leq$ Independent Set

Independent Set: Input: an undirected graph G, and an integer k
Output: true is there an independent set of size at least k and false otherwise.

An independent set is a set of vertices so that there are no edges directly connecting them (i.e. no edge has both endpoints in the set).

This reduction will show Independent Set is NP-complete!

The Reduction

What do we do with our 3-SAT instance?

High level idea: we want the independent set to correspond to the things that make the constraints true.

An independent set of size at least "number of constraints" will hopefully correspond to a setting of the variables.

Reduction Idea

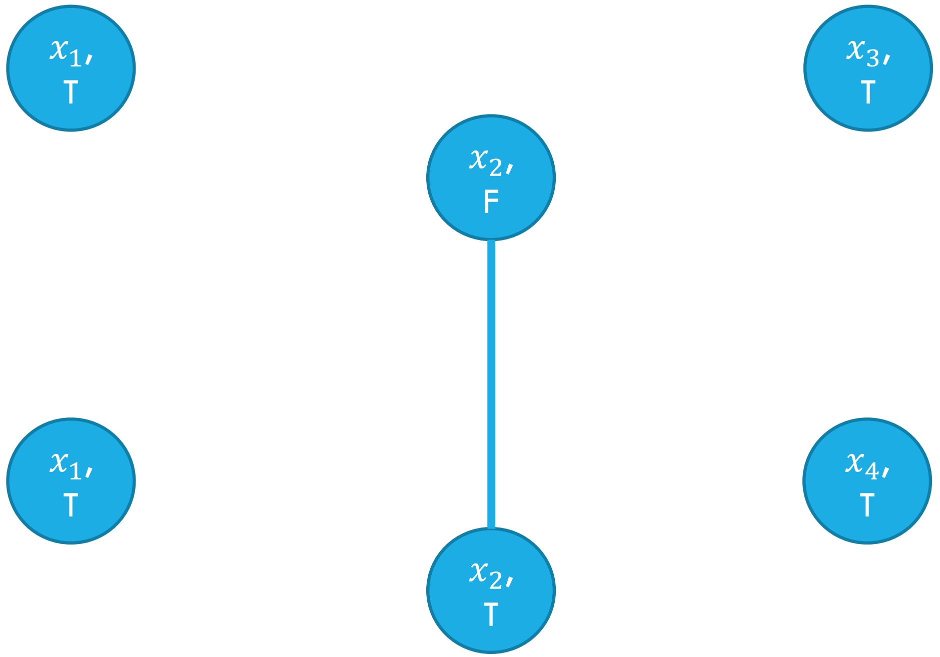
Connecting two vertices by an edge means we can have at most one in our independent set.

Have the vertices correspond to the pieces of the constraints.

$x_1 == True \,||\, x_2 == False \,||\, x_3 == True$
$x_1 == True \,||\, x_2 == True \,||\, x_4 == True$

Which Booleans can't we have both of?
I.e. which pairs don't make sense together?

Add edges between the same variable set to opposite values.



Reduction Idea Step 2

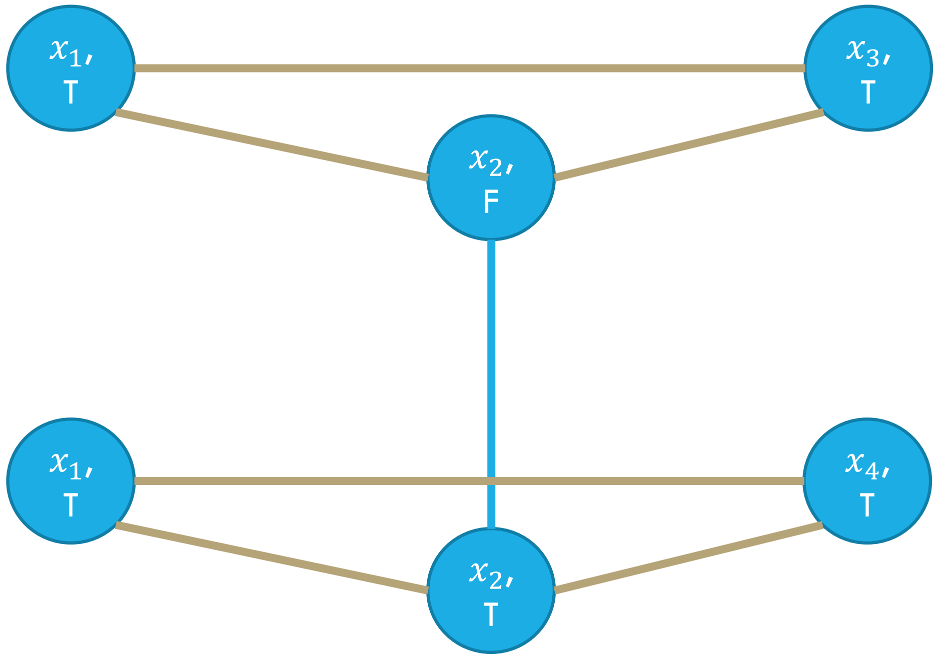
Connecting two vertices by an edge means we can have at most one in our independent set.

How big of an independent set do we want? Would be nice to count how many constraints are satisfied…need to make sure we take only one vertex per constraint.

$x_1 == True \,||\, x_2 == False \,||\, x_3 == True$
$x_1 == True \,||\, x_2 == True \,||\, x_4 == True$

Need only one vertex per constraint.

Add edges between all vertices coming from one constraint.



Reduction

Given a 3-SAT instance, make the graph G described on the last slide.

Ask the IND-SET library if there is an independent set of size at least (number of constraints of the 3-SAT instance) in G.

Return what the IND-SET library says.

Correctness

If there is a satisfying assignment for the 3-SAT instance, then there is a way to set the variables so that:

1. At least one part of every constraint is true

2. Every variable is set to true or false, not both.

In the graph, there is a large-enough independent set:

For each constraint, choose one of the true pieces (if there's more than one), and take the corresponding vertex. Is it an independent set?

We take only one per group, so the within group edges aren't included.

Each variable is only true or false, so we don't include any of the other edges.

Correctness, Part 2

Suppose there is an independent set of size at least (number of constraints)

Because of the "in-group" edges, an independent set has at most one per group. Thus every group has exactly one vertex in the independent set.

Set variables of the 3-SAT instance to match the chosen variables. We won't try to set variables "inconsistently" (i.e. no variable is both true and false) because of the edges we added between groups.

And we satisfy every constraint (because we chose a good setting of one piece with the independent set). So the indpendent set was satisfiable!

So…

3-SAT $\leq$ INDEPENDENT SET

That means that INDEPENDENT SET is NP-hard.

(And, since it's also in NP, it's NP-complete.)

Even though they look very different, the tasks "find an efficient algorithm to solve 3-SAT" and "find an efficient algorithm to solve INDEPENDENT SET" are equivalent!

Why are P and NP interesting?

Why do we care?

We've seen a few NP-complete problems.

But why should we care about those few?

Just memorize them and avoid them, right?

It's more than just a few…

NP-Complete Problems

But Wait! There's more!

Main Theorem. All the problems on the following list are complete.

1. SATISFIABILITY
COMMENT: By duality, this problem is equivalent to determining whether a disjunctive normal form expression is a tautology.

2. 0-1 INTEGER PROGRAMMING
INPUT: integer matrix C and integer vector d
PROPERTY: There exists a 0-1 vector x such that $Cx = d$.

3. CLIQUE
INPUT: graph G, positive integer k
PROPERTY: G has a set of k mutually adjacent nodes.

4. SET PACKING
INPUT: Family of sets $\{S_j\}$, positive integer ℓ
PROPERTY: $\{S_j\}$ contains ℓ mutually disjoint sets.

5. NODE COVER
INPUT: graph G', positive integer ℓ
PROPERTY: There is a set $R \subseteq N'$ such that $|R| \leq \ell$ and every arc is incident with some node in R.

6. SET COVERING
INPUT: finite family of finite sets $\{S_j\}$, positive integer k
PROPERTY: There is a subfamily $\{T_h\} \subseteq \{S_j\}$ containing $\leq k$ sets such that $\cup T_h = \cup S_j$.

7. FEEDBACK NODE SET
INPUT: digraph H, positive integer k
PROPERTY: There is a set $R \subseteq V$ such that every (directed) cycle of H contains a node in R.

8. FEEDBACK ARC SET
INPUT: digraph H, positive integer k
PROPERTY: There is a set $S \subseteq E$ such that every (directed) cycle of H contains an arc in S.

9. DIRECTED HAMILTON CIRCUIT
INPUT: digraph H
PROPERTY: H has a directed cycle which includes each node exactly once.

10. UNDIRECTED HAMILTON CIRCUIT
INPUT: graph G
PROPERTY: G has a cycle which includes each node exactly once.

11. SATISFIABILITY WITH AT MOST 3 LITERALS PER CLAUSE
INPUT: Clauses $D_1, D_2, \ldots, D_r$, each consisting of at most 3 literals from the set $\{u_1, u_2, \ldots, u_m\} \cup \{\bar{u}_1, \bar{u}_2, \ldots, \bar{u}_m\}$
PROPERTY: The set $\{D_1, D_2, \ldots, D_r\}$ is satisfiable.

12. CHROMATIC NUMBER
INPUT: graph G, positive integer k
PROPERTY: There is a function $\phi: N \to Z_k$ such that, if u and v are adjacent, then $\phi(u) \neq \phi(v)$.

13. CLIQUE COVER
INPUT: graph G', positive integer ℓ
PROPERTY: N' is the union of ℓ or fewer cliques.

14. EXACT COVER
INPUT: family $\{S_j\}$ of subsets of a set $\{u_i, i = 1,2,\ldots,t\}$
PROPERTY: There is a subfamily $\{T_h\} \subseteq \{S_j\}$ such that the sets T_h are disjoint and $\cup T_h = \cup S_j = \{u_i, i = 1,2,\ldots,t\}$.

15. HITTING SET
INPUT: family $\{U_i\}$ of subsets of $\{s_j, j = 1,2,\ldots,r\}$
PROPERTY: There is a set W such that, for each i, $|W \cap U_i| = 1$.

16. STEINER TREE
INPUT: graph G, $R \subseteq N$, weighting function $w: A \to Z$, positive integer k
PROPERTY: G has a subtree of weight $\leq k$ containing the set of nodes in R.

17. 3-DIMENSIONAL MATCHING
INPUT: set $U \subseteq T \times T \times T$, where T is a finite set
PROPERTY: There is a set $W \subseteq U$ such that $|W| = |T|$ and no two elements of W agree in any coordinate.

18. KNAPSACK
INPUT: $(a_1, a_2, \ldots, a_r, b) \in Z^{n+1}$
PROPERTY: $\Sigma a_j x_j = b$ has a 0-1 solution.

19. JOB SEQUENCING
INPUT: "execution time vector" $(T_1, \ldots, T_p) \in Z^p$,
"deadline vector" $(D_1, \ldots, D_p) \in Z^p$
"penalty vector" $(P_1, \ldots, P_p) \in Z^p$
positive integer k
PROPERTY: There is a permutation π of $\{1,2,\ldots,p\}$ such that

$$\left(\sum_{j=1}^{p} [\text{if } T_{\pi(1)} + \cdots + T_{\pi(j)} > D_{\pi(j)} \text{ then } P_{\pi(j)} \text{ else } 0]\right) \leq k .$$

20. PARTITION
INPUT: $(c_1, c_2, \ldots, c_s) \in Z^s$
PROPERTY: There is a set $I \subseteq \{1,2,\ldots,s\}$ such that $\sum_{h \in I} c_h = \sum_{h \notin I} c_h$.

21. MAX CUT
INPUT: graph G, weighting function $w: A \to Z$, positive integer W
PROPERTY: There is a set $S \subseteq N$ such that

$$\sum_{\substack{\{u,v\} \in A \\ u \in S \\ v \notin S}} w(\{u,v\}) \geq W .$$

Karp's Theorem (1972)

A lot of problems are NP-complete

NP-Complete Problems

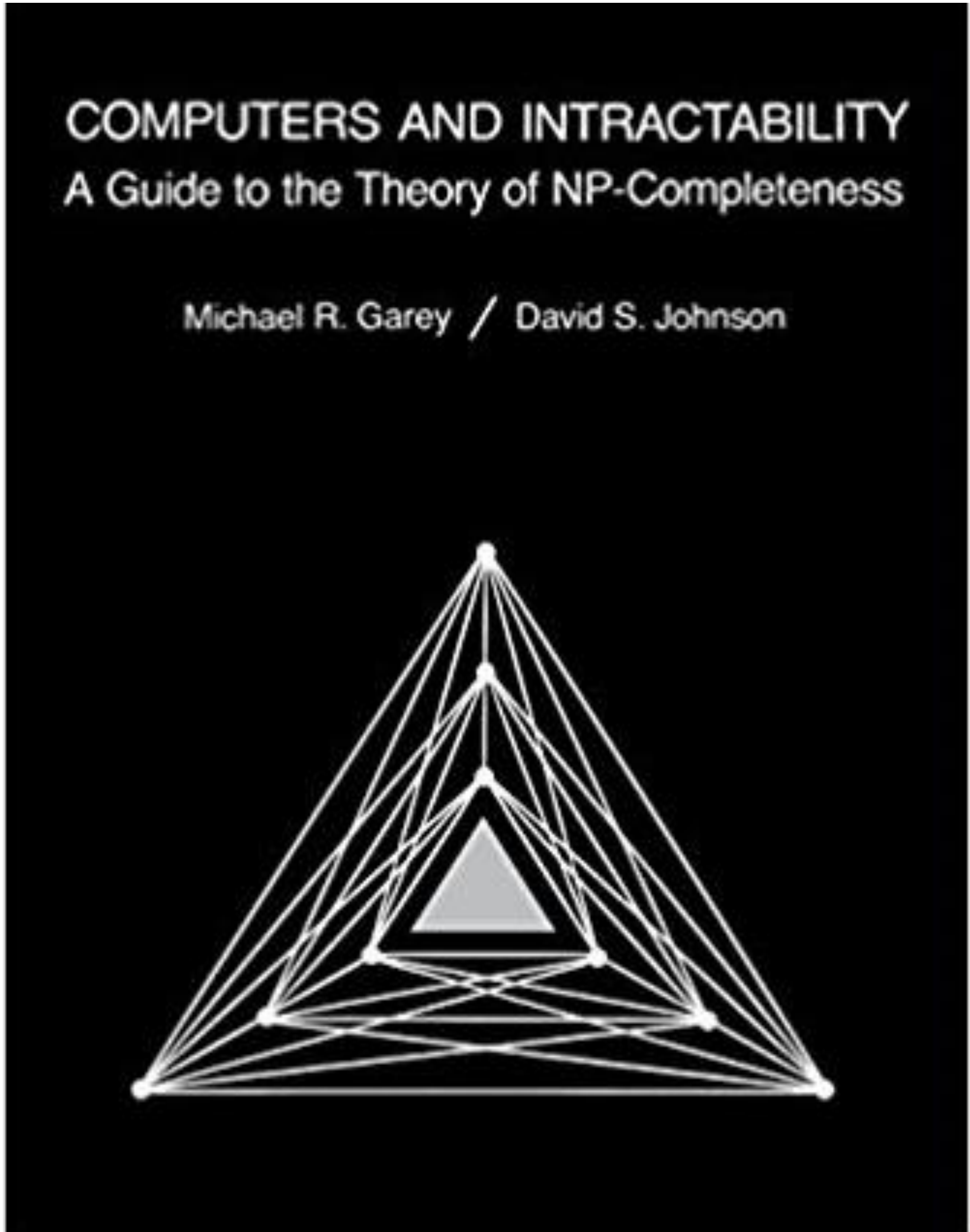
But Wait! There's more!

By 1979, at least 300 problems had been proven NP-complete.

Garey and Johnson put a list of all the NP-complete problems they could find in this textbook.

Took almost 100 pages to just list them all.

No one has made a comprehensive list since.



NP-Complete Problems

But Wait! There's more!

In December 2018, mathematicians and computer scientists put papers on the arXiv claiming to show (at least) 25 more problems are NP-complete.

There are literally thousands of NP-complete problems known.

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-completeness

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

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

Sooner or later it will happen to one of you.

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

We'll discuss options next week!