Coping With NP-Completeness

CSE 417 Winter 24
Lecture 22

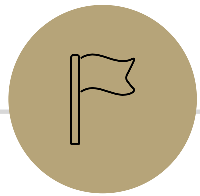
So Far

Reductions “rank” the difficulty of problems

The “easily solvable” problems are problems in P (well, the decision versions anyway).

The “probably should give up” problems are NP-hard problems.

We are trying to convince you that it’s reasonable to believe that an NP-hard problem exists. Then we’ll talk about how you show a problem is NP-hard.



Wrapping up 3-Color ≤ 3-SAT

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 ≤ 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 (u, 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	$x_{u,r} == False \,\|\|\, x_{v,b} == \text{True} \,\|\|\, x_{v,g} == True$
If u is blue, then v is red or green	$x_{u,b} == False \|\| x_{v,r} == \text{True} \,\|\|\, x_{v,g} == True$
If u is green, then v is red or blue	$x_{u,g} == False \|\| x_{v,r} == \text{True} \,\|\|\, x_{v,b} == True$
If v is red, then u is blue or green	$x_{v,r} == False \,\|\|\, x_{u,b} == \text{True} \,\|\|\, x_{u,g} == True$
If v is blue, then u is red or green	$x_{v,b} == False \|\| x_{u,r} == \text{True} \,\|\|\, x_{u,g} == True$
If v is green, then u is red or blue	$x_{v,g} == False \|\| x_{u,r} == \text{True} \,\|\|\, x_{u,b} == True$

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	$x_{u,r} == False \ \| \| \ x_{u,b} ==$ False
If u is red, then u cannot be green	$x_{u,r} == False \ \| \| \ x_{u,g} ==$ False
If u is blue, then u cannot be red	$x_{u,b} == False \ \| \| \ x_{u,r} ==$ False
If u is blue, then u cannot be green	$x_{u,b} == False \ \| \| \ x_{u,g} ==$ False
If u is green, then u cannot be red	$x_{u,g} == False \ \| \| \ x_{u,r} ==$ False
If u is green, then u cannot be blue	$x_{u,g} == False \ \| \| \ x_{u,b} ==$ False
u gets a color!	$x_{u,r} == True \ \| \| \ x_{u,g} == True \| \| \ x_{u,b} == True$

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

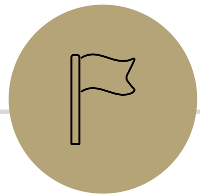
If the reduction returns YES, then G was 3-colorable.

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.



More Reduction Facts

One More Thought

On HW6, you might have done a problem that sounded a lot like 3-COLOR…but 3-COLOR is NP-complete. What's going on?

The algorithm only handled a special case – the coloring problem **on a tree**. We didn't prove $P = NP$. We carved off part of the problem that was easy and solved that (solved only the "easy" instances).

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!

Want to prove your problem is hard?

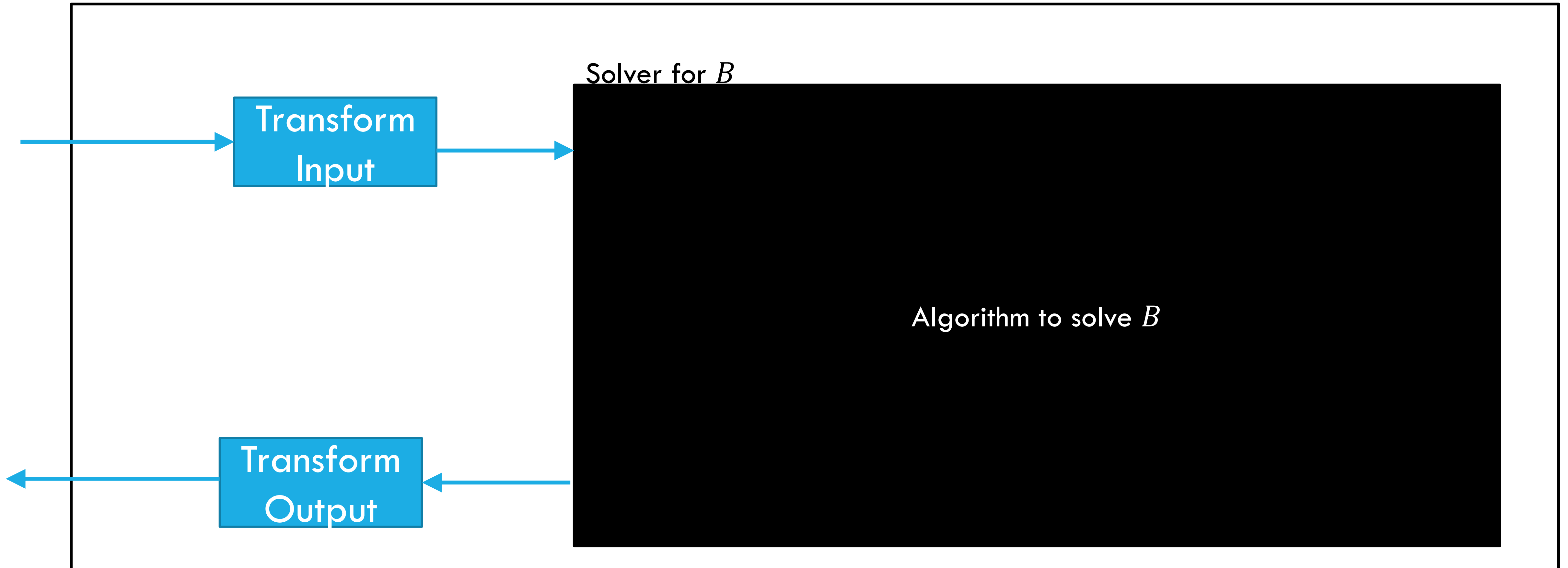
To show B is hard,

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

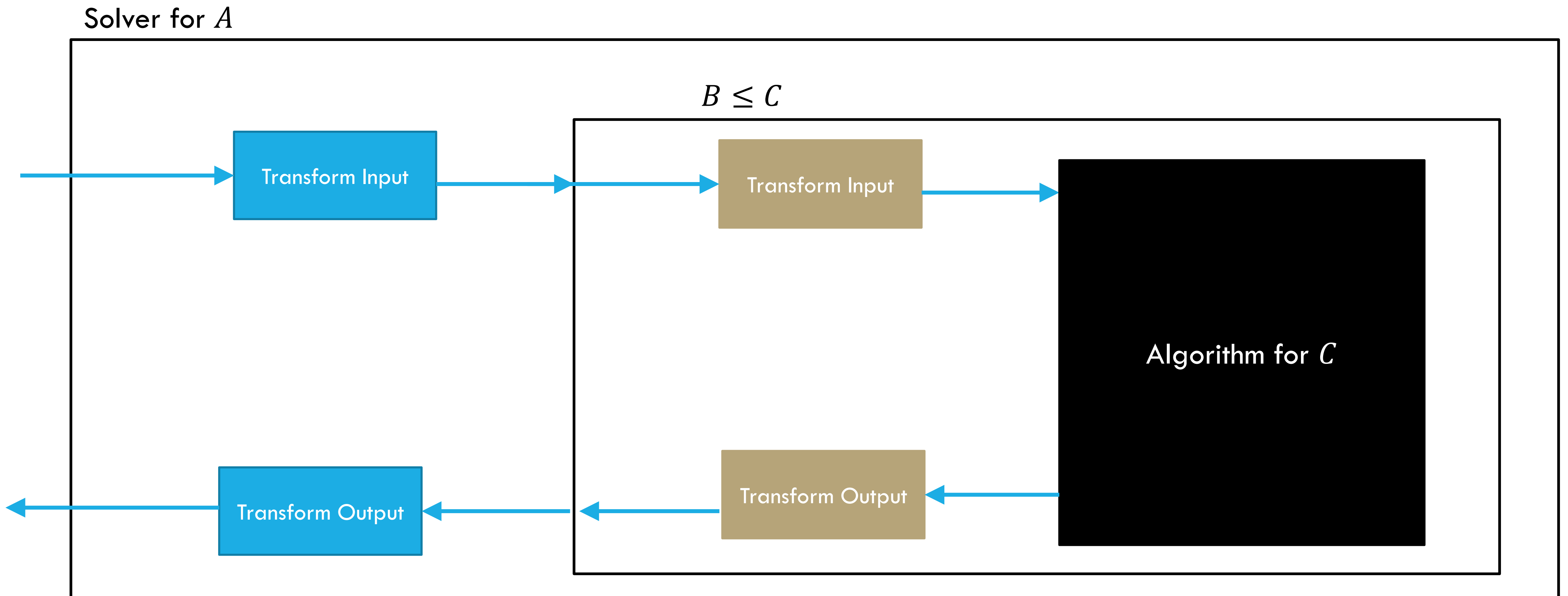
A reduction **From** an NP-hard problem A to B, shows B is also NP-hard.

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

Is that true?



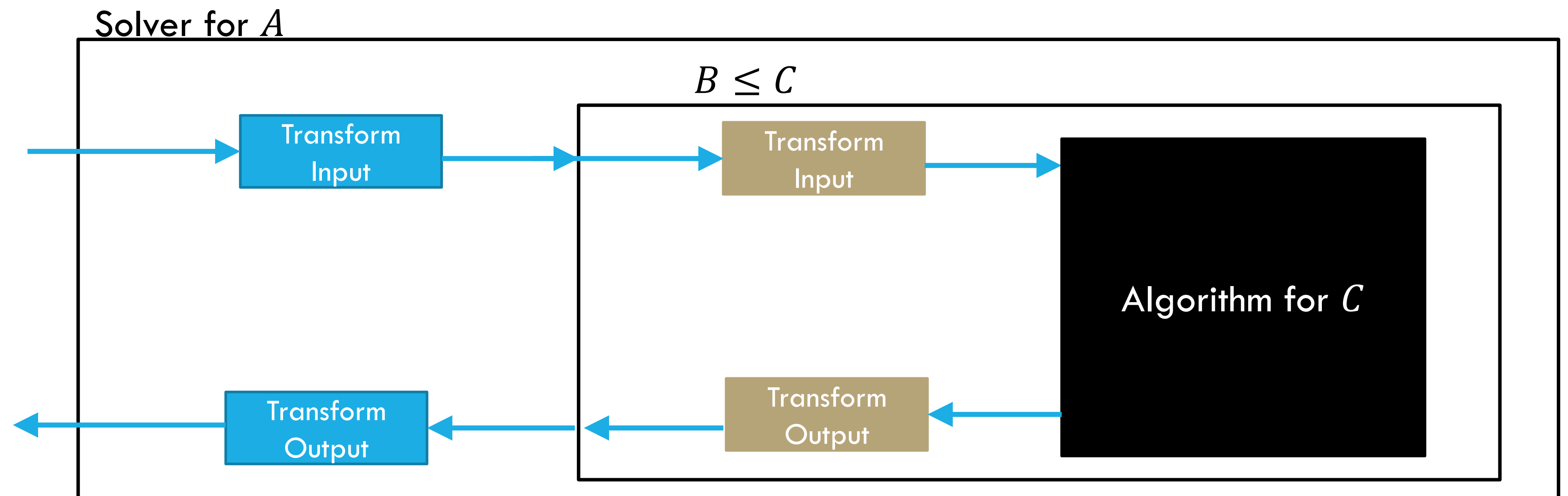
$$A \leq B \leq C \;\rightarrow A \leq C$$



$A \le B \le C \;\rightarrow A \le C$

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

$A \leq B$

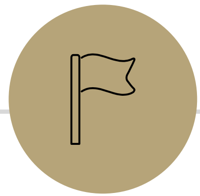
If I know B is not hard [I have an algorithm for it] then A is also not hard.

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)



Using Reductions

How do you show a problem is NP-hard/-complete?

Let B be the new problem you are interested in.

To show B is NP-complete

Reduce from a known NP-hard problem A to B.

Show that B is in NP.

To show B is NP-hard

Reduce from a known NP-hard problem A to B.

Why that direction?

The known NP-hard problem is known to be hard.

We're pretty sure we're not going to find a polynomial time algorithm for A.

We're not 100% sure, but we're like 99% sure.

Suppose, for the sake of "contradiction", you could find a polynomial time algorithm for problem B. Then with the reduction, you'd have a polynomial time algorithm for A too! But A is known to be an NP-hard problem. So finding a polynomial time algorithm for it would be shocking. A "contradiction."

Not really a contradiction (we don't know as a mathematical fact that A can't be solved in polynomial-time) but that's the high-level idea.

Double check the direction!!

Seriously.

It’s the easiest way to solve the wrong problem.

If both problems *really are* NP-complete, then both reductions exist! You won’t even notice anything is amiss.

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)

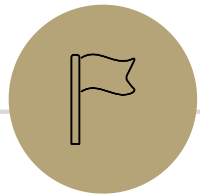
Use that to prove Hamiltonian Cycle is NP-hard.

Pollev.com/robbie

Which direction?

Reduce FROM the known hard problem TO the new problem.

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



More Practice

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.

More Practice

Suppose you know that 3-SAT is NP-complete and you want to show that Independent Set is NP-complete.

Independent Set: Given a graph $G = (V, E)$ and in integer k, return true if there is a set S of k vertices such that for all $u, v \in S$ $(u, v) \notin E$

What reduction do you show?

Do you need to show anything else?

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-hard!

To show it's complete, we also need to show it's in NP. What's our certificate? The independent set itself!

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

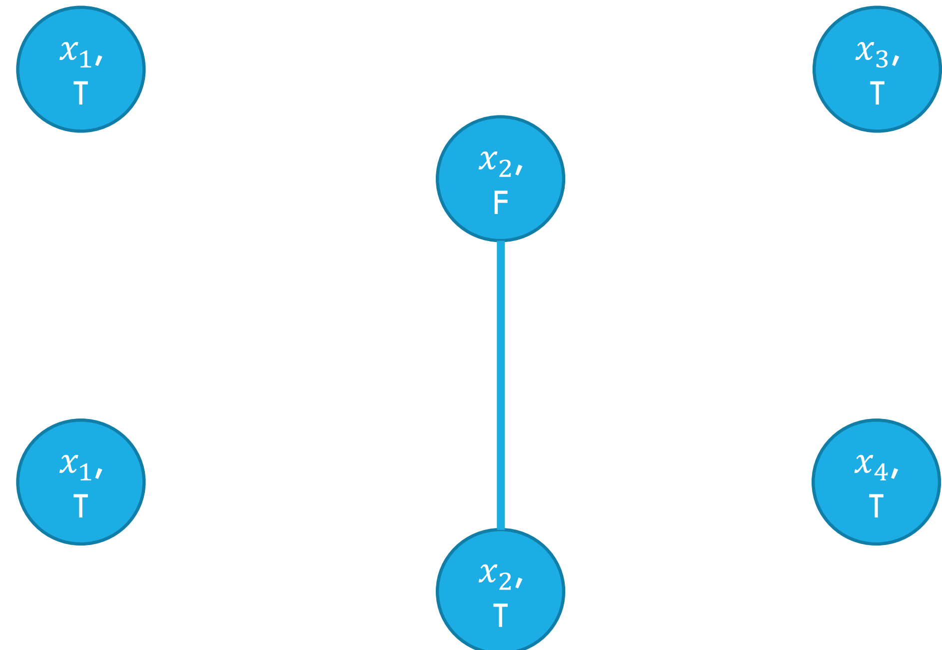
Connecting two vertices by an edge means we can have at most one in our independent set.

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

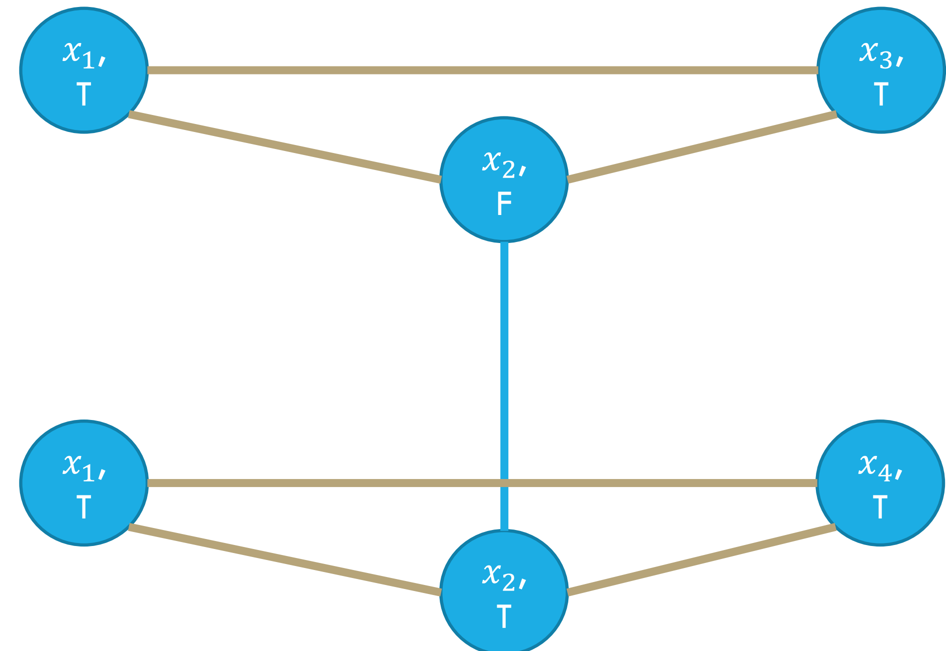
Connecting two vertices by an edge means we can have at most one in our independent set.

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!

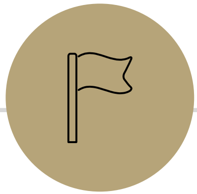
So…

3-SAT $\leq$ INDEPENDENT SET

That means that INDEPENDENT SET is NP-hard.

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

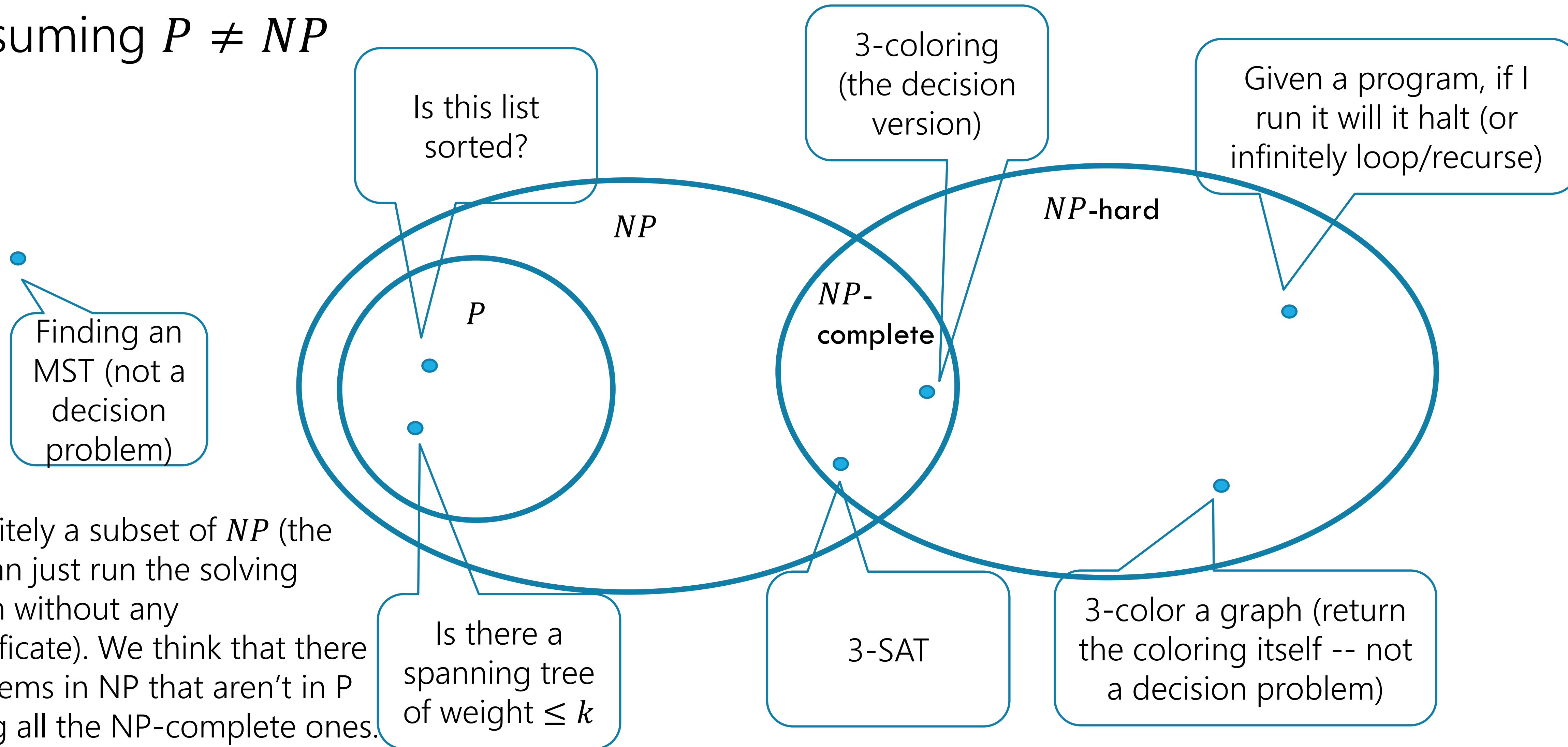
Even though they look very different, the tasks "find an efficient algorithm to solve 3-SAT" and "find an efficient algorithm to solve INDEPENDENT SET" are equivalent!



More Context; Coping

What (we think) the world looks like

Assuming $P \neq NP$



P is definitely a subset of NP (the verifier can just run the solving algorithm without any hint/certificate). We think that there are problems in NP that aren't in P (including all the NP-complete ones.

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

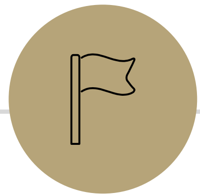
Thousands of times someone has wanted to find an efficient algorithm for a problem…

…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.



Why Should you care?

Why should you care about P vs. NP

Most computer scientists are convinced that P≠NP.

- A survey of experts (PhDs in CS) found 98% of them thought P≠NP.
- And the median guess was that we're at least 50 years from getting the answer.

Why should you care about this problem?

It's your chance for:

$1,000,000. The Clay Mathematics Institute will give $1,000,000 to whoever solves P vs. NP (or any of the 5 remaining mathematical conjectures they listed)

To get a Turing Award

Why should you care about P vs. NP

Most computer scientists are convinced that P≠NP.

- A survey of experts (PhDs in CS) found 98% of them thought P≠NP.
- And the median guess was that we're at least 50 years from getting the answer.

Why should you care about this problem?

It's your chance for:

$1,000,000. The Clay Mathematics Institute will give $1,000,000 to whoever solves P vs. NP (or any of the 5 remaining mathematical conjectures they listed)

To get ~~a Turing Award~~ the Turing Award renamed after you.

Why Should You Care if P=NP?

Suppose P=NP.

Specifically that we found a genuinely in-practice efficient algorithm for an NP-complete problem. What would you do?

$1,000,000 from the Clay Math Institute obviously, but what's next?

Why Should You Care if P=NP?

We found a genuinely in-practice efficient algorithm for an NP-complete problem. What would you do?

- Another $5,000,000 from the Clay Math Institute
- Put mathematicians out of work?
- Decrypt (essentially) all current internet communication. See the real world question.

A world where P=NP is a very very different place from the world we live in now.

Why Should You Care if P≠NP?

We already expect P≠NP. Why should you care when we finally prove it?

P≠NP says something fundamental about the universe.

For some questions there is not a clever way to find the right answer

- Even though you'll know it when you see it.

There is actually a way to obscure information, so it cannot be found quickly no matter how clever you are.

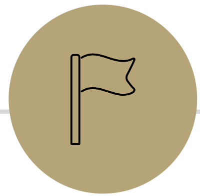
Why Should You Care if P≠NP?

To prove P≠NP we need to better understand the differences between problems.

- Why do some problems allow easy solutions and others don't?
- What is the structure of these problems?

We don't care about P vs NP just because it has a huge effect about what the world looks like.

We will learn a lot about computation along the way.



Dealing With NP-hardness

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-completeness

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.

Step 4 – Permanent Solutions

Two good options:

Exponential algorithms that aren't-as-slow-as-others

Give you an exact answer; won't take polynomial time but will be guaranteed take you less time than brute force.

Approximation algorithms

Don't give you the best answer, but guarantees a reasonable amount of time, and a guaranteed-pretty-good-answer.

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?