Coping with NP-hardness

CSE 421 Fall22
Lecture 28

Outline For Today/Wednesday

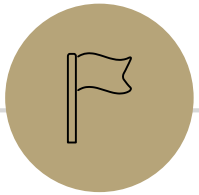
Some Loose Ends

- Transitivity of reductions
- An edge case of the definitions

What do you do when your problem is NP-hard?

Approximation Algorithms for NP-complete problems

- How do we measure quality?
- Another example for vertex cover
- An example for TSP.



Some Loose Ends

Loose End 1: Transitivity

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

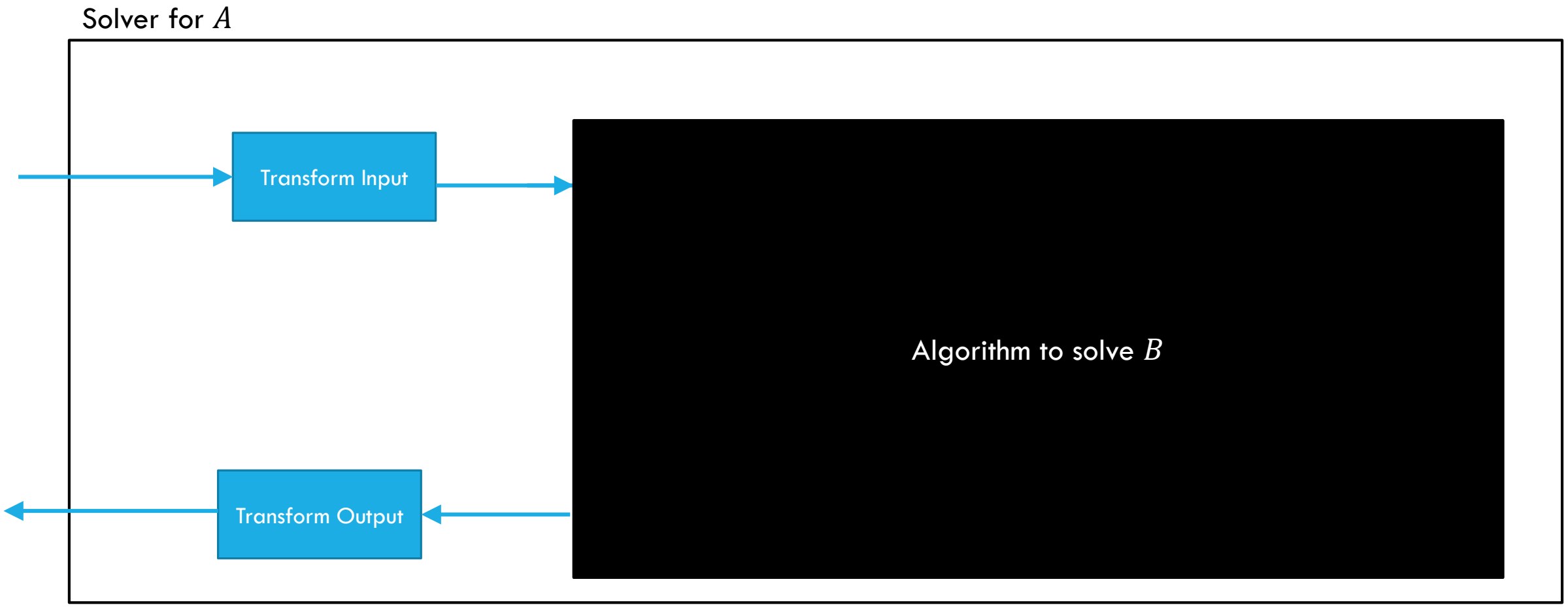
So difficult, it's probably NP-hard. How do I show it?

What does it mean to be NP-hard?

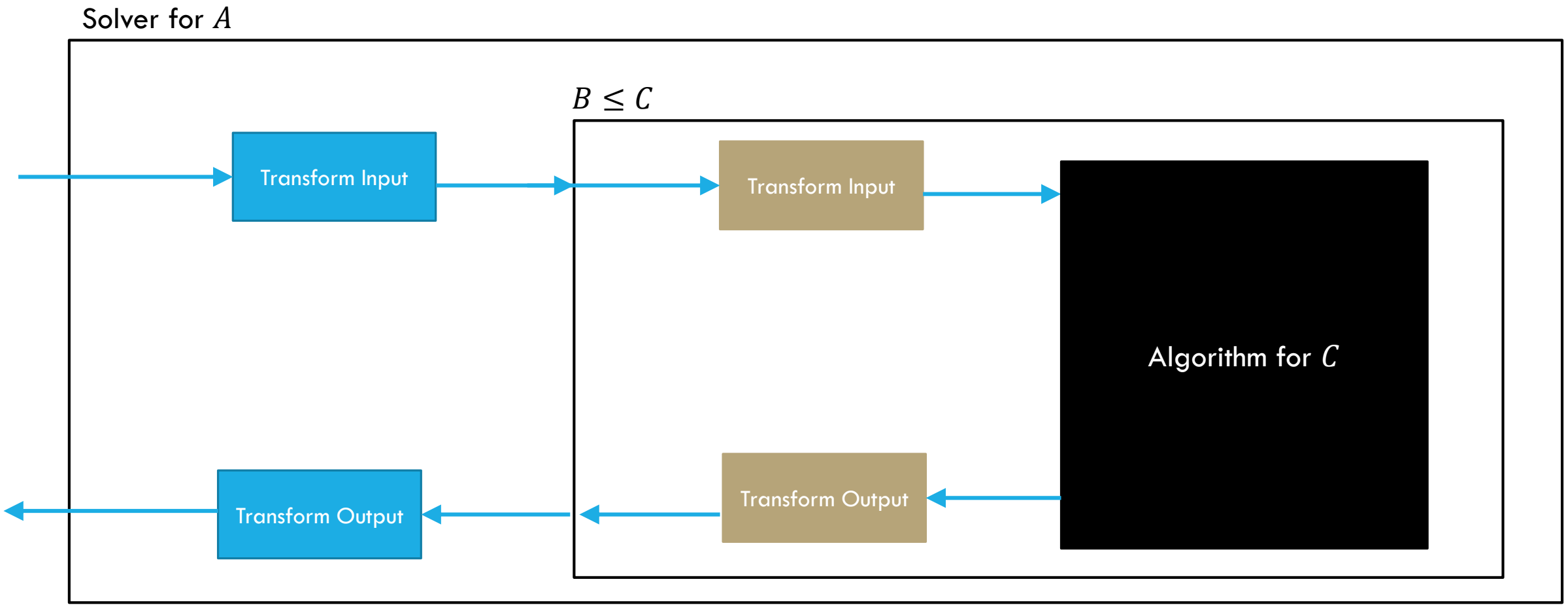
We need to be able to reduce any problem A in $\mathbf{NP}$ to C.

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



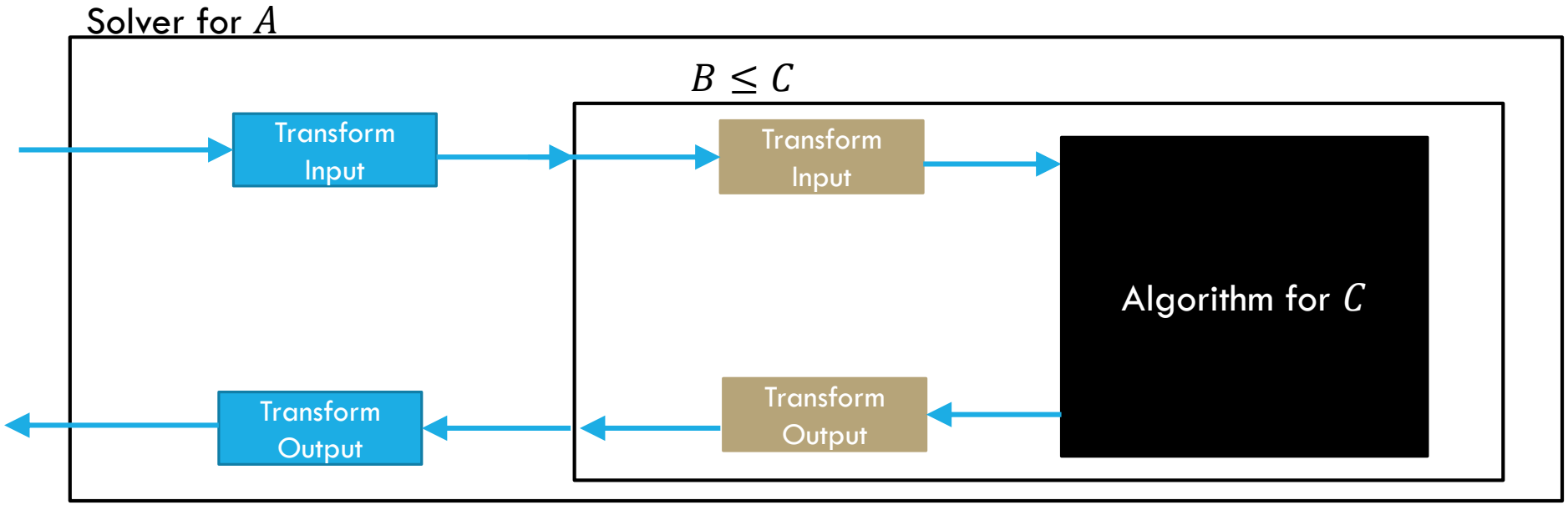
Is the implication true? $A \leq B \leq C \rightarrow A \leq C$



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.



Loose End 2: Input Size

The definitions of both $\mathbf{P}$ and $\mathbf{NP}$ 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?

PRIMES

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 in the input size.

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

Knapsack

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

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

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

Running time: $\Theta(Wn)$

What's the input size?

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

Knapsack

Running time: $\Theta(Wn)$

Input size?

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

That isn't a polynomial time algorithm.

Weakly NP-hard

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

It's only (known to be) NP-hard if you represent the input in binary.

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 $\mathbf{P}$.

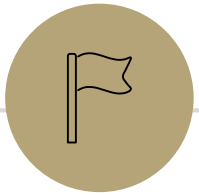
Be sure you've accounted for the fact that numbers are in binary.

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.



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

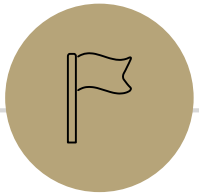
Thousands of times someone has wanted to find an efficient algorithm for a problem…

…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

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.

Step 4 – Permanent Solutions

One good option (others strategies exist):

Approximation algorithms

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

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: 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!

Another Approximation Algorithm

Let's look at another approximation algorithm for vertex cover.

Remember the linear program for vertex cover?

Vertex Cover LP

Minimize $\sum x_u$

Subject to:

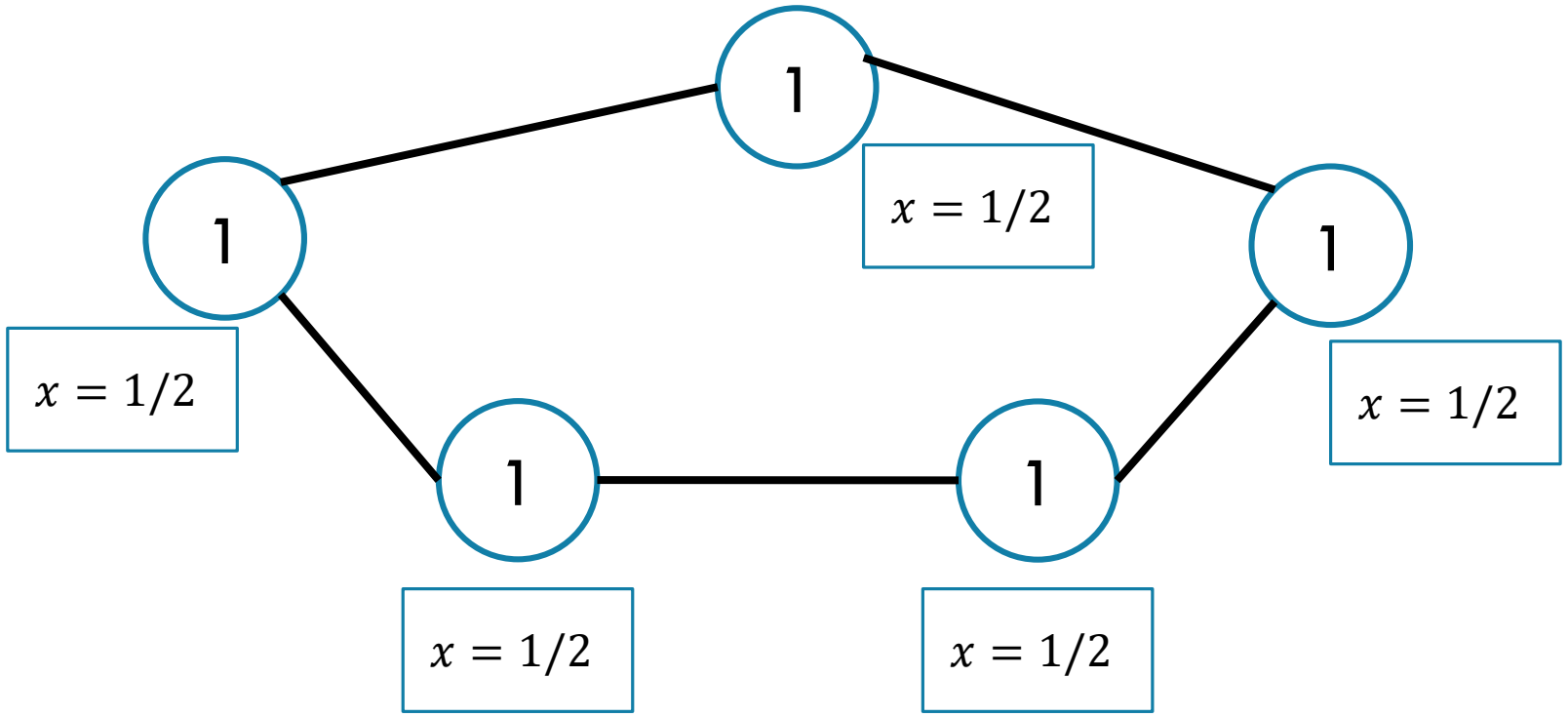
$x_u + x_v \geq 1$ for all $(u, v) \in E$

$0 \leq x_u \leq 1$ for all u.

Don't worry about the weights for today.

Non-Bipartite

What if our original graph isn't bipartite?



The LP finds a fractional vertex cover of weight 2.5

There's no "real"/integral VC of weight 2.5. – lightest is weight 3.

There's a "gap" between integral and fractional solutions.

So, what if the graph isn't bipartite?

Big idea:

Just round!

Pollev.com/robbie

If $x_u \geq \frac{1}{2'}$ round up to 1.

If $x_u < \frac{1}{2}$, round down to 0

Minimize $\sum w(u) \cdot x_u$

Subject to:

$x_u + x_v \geq 1$ for all $(u, v) \in E$

$0 \leq x_u \leq 1$ for all u.

Two questions – is it a vertex cover? How far are we from the true minimum?

Is it a vertex cover?

Every edge was covered in the fractional matching

i.e. for every edge (u, v)

$x_u + x_v \geq 1$.

At least one of those is getting rounded up!

So every edge is covered.

And we've rounded to integers, so we have a "real" vertex cover.

How good of an approximation is it?

Well, we might have doubled the value of the LP when we rounded. But we definitely didn't do any more than that.

$2 \cdot LP \geq ALG$

And the value of the LP is definitely not bigger than the true size of the vertex cover (because otherwise the LP would have found that).

$OPT \geq LP$

Combining:

$2 \cdot OPT \geq ALG$

So we're safe in calling this a 2-approximation.

Comparing to the LP value

We did a weird thing on that last slide.

We were supposed to compare the value of our vertex cover to the best vertex cover.

But instead we compared it to the value of the LP…which we know isn't always the value of the vertex cover!

That wasn't laziness, it's a very common technique. We know very little about the true value of the vertex cover (if we knew what it looked like VERY VERY precisely, why couldn't we just write an algorithm to find it? We actually won't know much). So we start with what the algorithm gave us (that we do understand).

Side Note

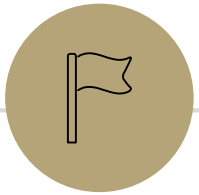
Could we do better?

Not just with the LP.

If you take a graph with n vertices and every possible edge, the LP's minimum is $n/2$, the true minimum vertex cover is size $n - 1$.

The ratio is $2 - 1/n$. So if we don't at least double the value **sometimes** we won't get a vertex cover at all.

Getting a 1.9999999 approximation is an open problem!



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.

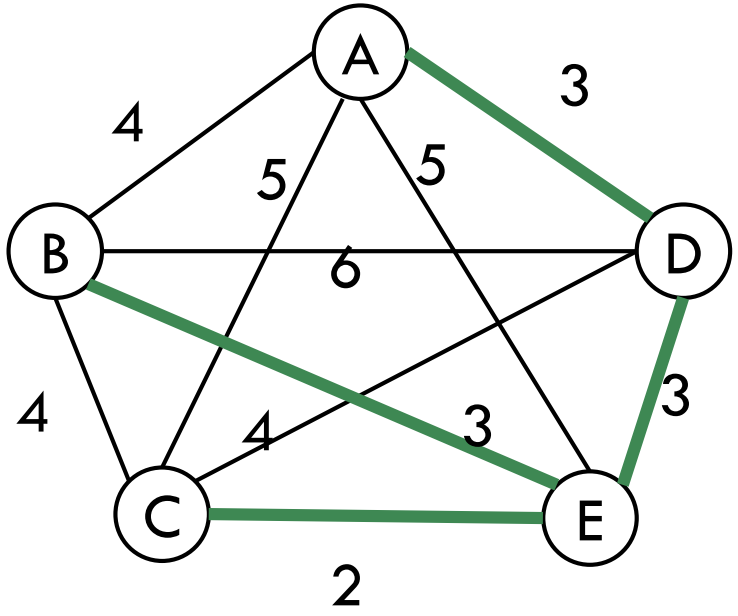
2. The weights satisfy the "triangle inequality" (it's faster to go from x to y directly than it is to go from x to y through x).

TSP starting point

What would be a good baseline?

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

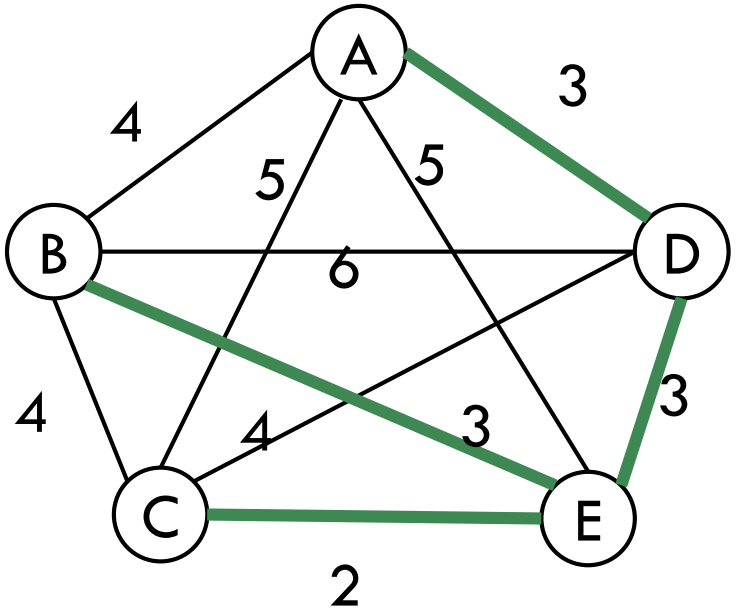
A Minimum Spanning Tree!



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.

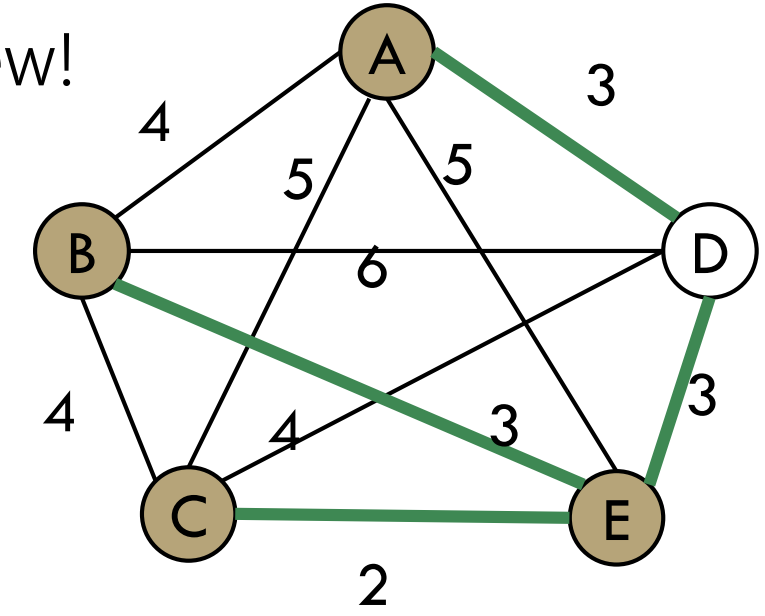
Doing a Little Better

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!

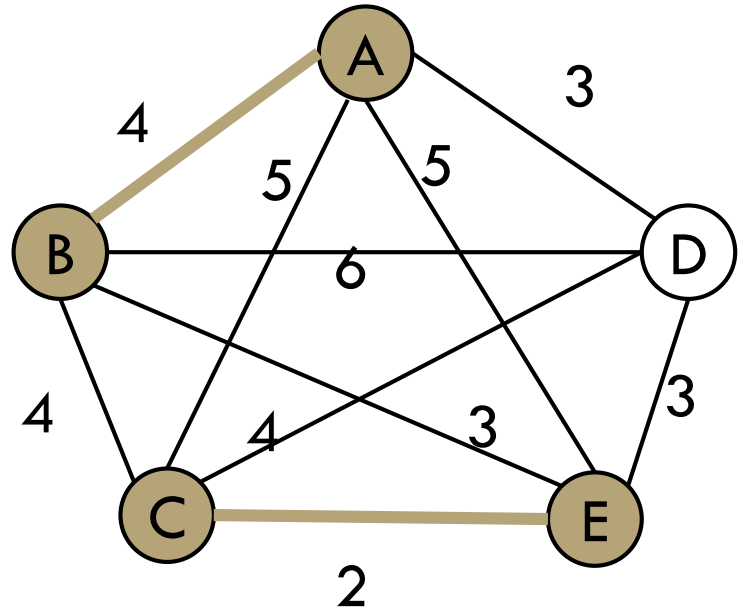
So lets add some possible ways out.



What would help?

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

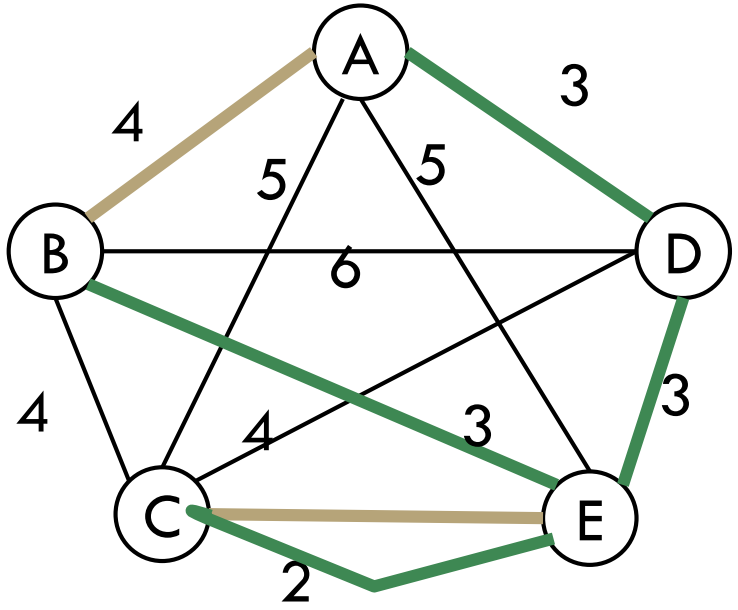
Specifically a minimum weight matching.



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)!

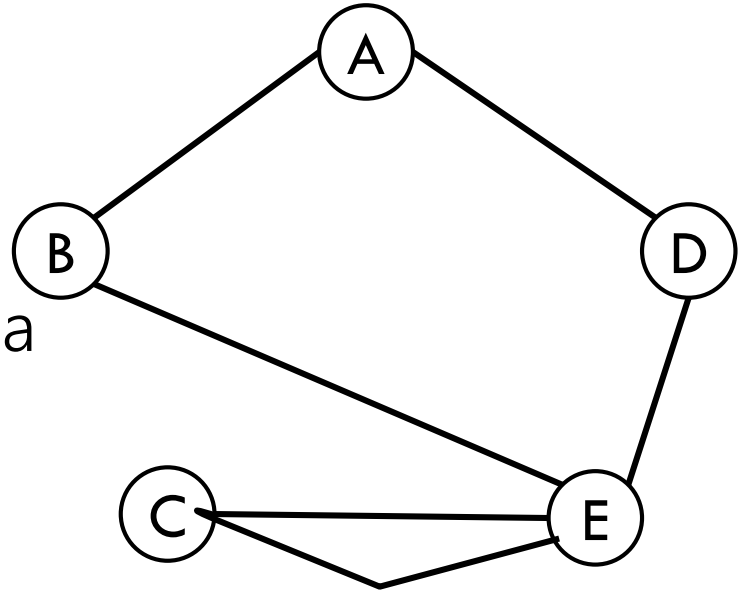
So we now have the MST AND the minimum weight matching on the odd edges.



Did It Help?

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

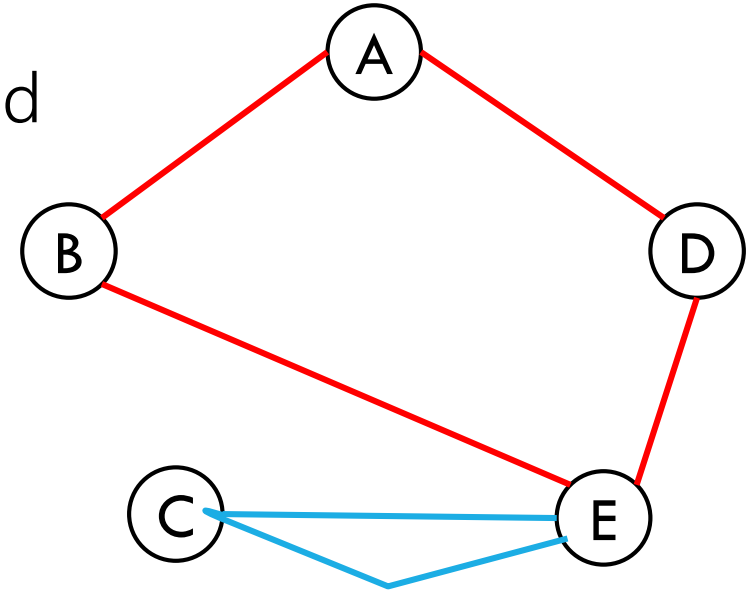
We'll have to find one.



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.