Coping With NP-Completeness

CSE 417 Fall 22
Lecture 26

Announcements

We spammed your inbox last night (sorry) via Canvas.

We wrote a script to count up your E and S scores.

It should show the state of gradescope as of about 9 PM last night.

This is the first time we've used this script! It might be buggy; if you see something you think is wrong, please ask us (private post on ed).

Suggested use: look for the 4 "total" columns on canvas; see if that matches your count. If not, then use the homework-by-homework columns to see where the discrepancy is.

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.

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

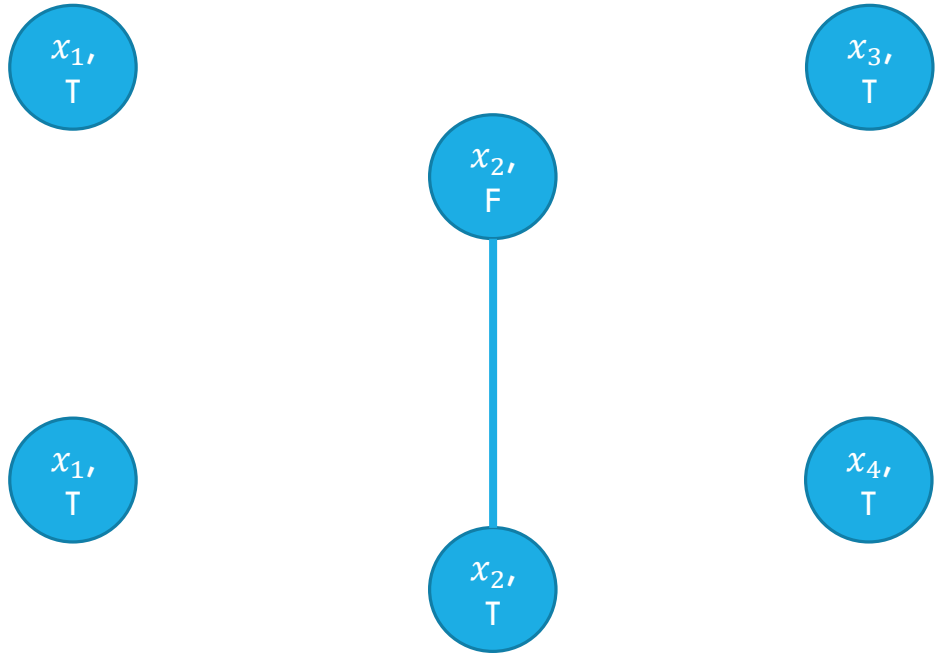
Connecting two vertices by an edge means we can have at most one in our independent set.

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

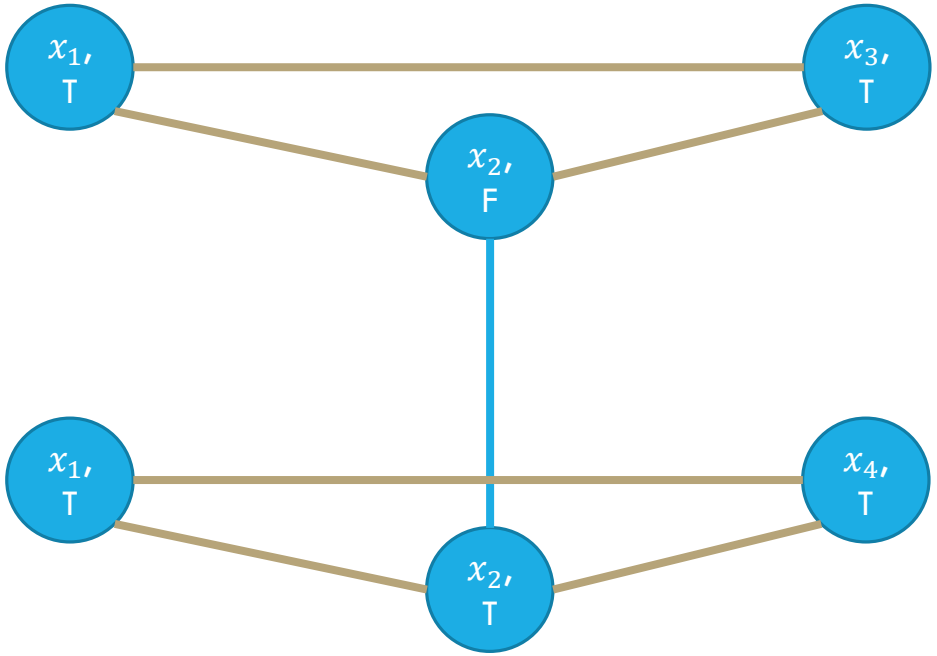
Connecting two vertices by an edge means we can have at most one in our independent set.

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!

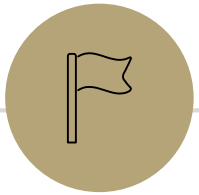
So…

3-SAT $\leq$ INDEPENDENT SET

That means that INDEPENDENT SET is NP-hard.

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

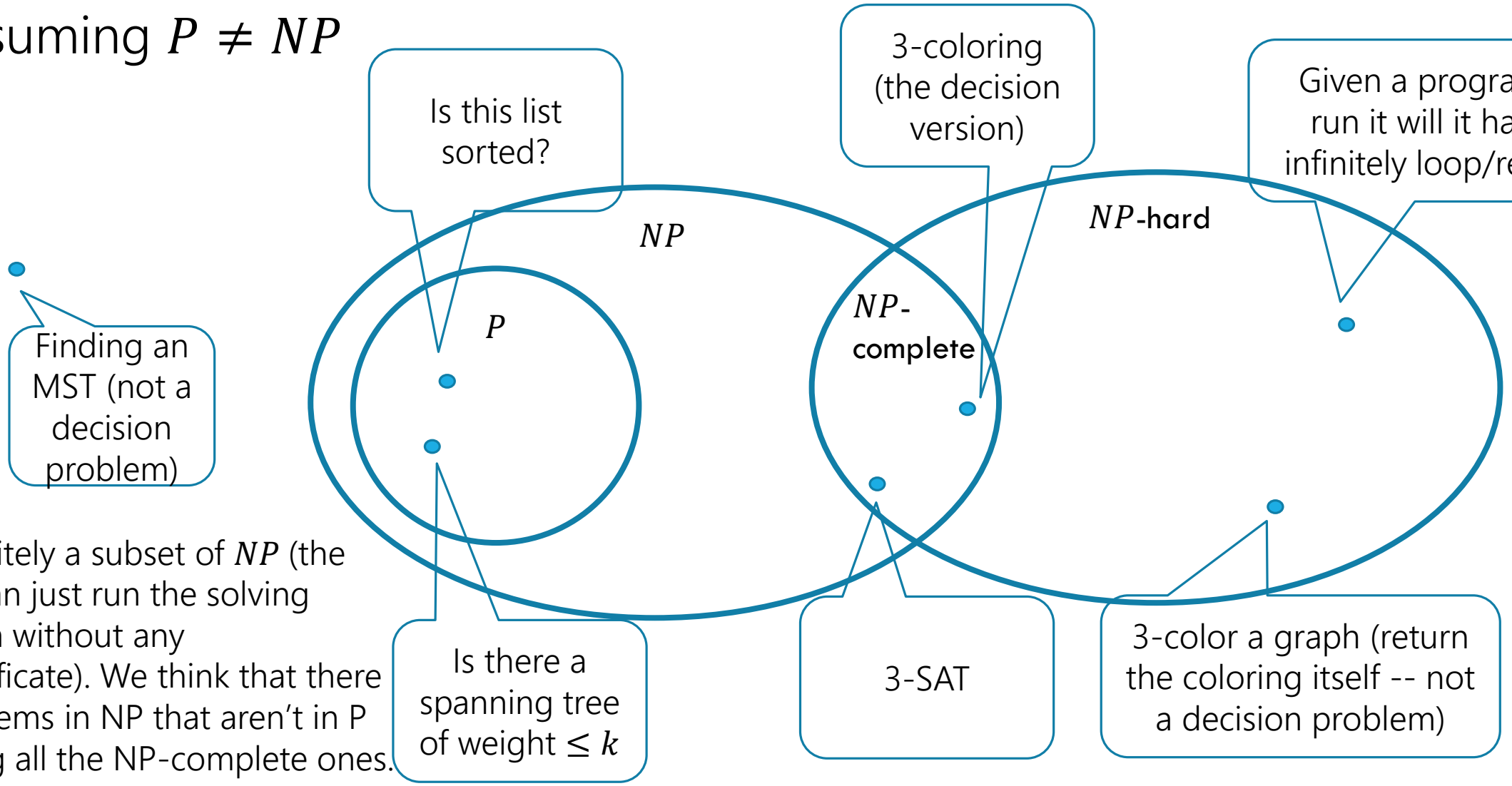
Even though they look very different, the tasks "find an efficient algorithm to solve 3-SAT" and "find an efficient algorithm to solve INDEPENDENT SET" are equivalent!



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

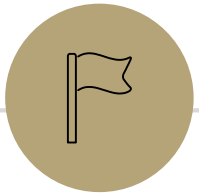
Thousands of times someone has wanted to find an efficient algorithm for a problem…

…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?

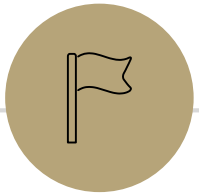
To prove P≠NP we need to better understand the differences between problems.

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?

Vertex Cover

Input: Graph G, integer k

Output: Is there a set of at most k vertices such that every edge has at least one endpoint in the set?

The problem is NP-complete.

In the worst-case, we need exponential time.

```
For every subset S of vertices
    Check if every edge has at least one endpoint
in the set
```

Time? $O(2^n(n+m))$

Vertex Cover

We can do better (sometimes)!

Don't check every subset, just the biggest allowed subsets. How does the running time change as k changes?

When k is a constant (say, $k \leq 3$)

How many subsets are there of size 3? n^3, running time: $O(n^3(n+m))$

That's not too terrible!

Vertex Cover

When k is a constant (say, $k \leq 3$)

Running time $O(n^3(n+m))$

k is a little bigger (say, $k = \log_2 n$)

Running time $O\left(n^{\log n}(n+m)\right)$ not polynomial anymore

Worst value of k ($k = n/2$)

Running time $O\left(2^{\frac{n}{2}}(n+m)\right)$ VERY SLOW

k very very big ($k = n - 3$)

Running time $O(n^3(n+m))$ (not many very large vertex sets)

We can do better

When k is big, not much we can do. What about when it's small?

Our running time depends on k anyway, let's focus in on making our algorithm better when k is small.

Key idea: pick an edge (u, v)

There is a vertex cover of size k if and only if

There is a vertex cover of size $k-1$ in $G-u$ or $G-v$.

i.e. at least one of u, v in the minimum vertex cover.

Key Idea – Let's Prove it!

If there is a vertex cover of size k, then there is a vertex cover of size $k - 1$ in $G - u$ or $G - v$.

Every vertex cover has to cover (u, v). So at least one of u or v is included. Delete that vertex (one arbitrarily if both are in the vertex cover) and all edges that touch it. Every other edge was covered by another vertex (since we deleted all the edges touching the deleted vertex). What remains is a vertex cover of size $k - 1$ on $G - u$ or $G - v$.

Key Idea – Let's Prove it!

If there is a vertex cover of size $k-1$ in $G-u$ or $G-v$ then there is a vertex cover of size k in G.

Assume that the vertex cover of size $k-1$ is in $G-u$ (the argument is the same if it's in $G-v$ instead). Take the vertex cover of $G-u$ and add in u. Every edge of $G-u$ is covered by the vertex cover. The only other edges in G touch u, so u covers them.

Algorithm

```
VertexCover(graph G, int k)
     if(G has no edges) //we've covered them all!
          return true
     if(k < 0)
          return false
     H1 = copy of G
     H2 = copy of G
     pick any edge (u,v)
     H1 = H1.remove(u) //removes u and all edges with u as
an endpoint
     H2 = H2.remove(v) //removes u and all edges with u as
an endpoint
     return VertexCover(H1,k-1) || VertexCover(H2, k-1)
```

Running Time

Recurrence: $T(k) = \begin{cases} 2T(k-1) + O(n+m) & \text{if } k \geq 1 \\ O(1) & \text{if } k < 0 \end{cases}$

Running time? Unroll or use recursion tree

Pollev.com/robbie

Running Time

Recurrence: $T(k) = \begin{cases} 2T(k-1) + O(n+m) & \text{if } k \geq 1 \\ O(1) & \text{if } k < 0 \end{cases}$

Running time? Unroll or use recursion tree

$$O\left((n+m) \cdot 2^k\right)$$

Vertex Cover

When k is a constant (say, $k \leq 3$)

Running time $O\left(2^3(n+m)\right) = O(n+m)$

k is a little bigger (say, $k = \log_2 n$)

Running time $O\left(2^{\log n}(n+m)\right) = O\left(n(n+m)\right)$ still polynomial!

$k = n/2$

Running time $O\left(2^{n/2}(n+m)\right)$ very slow

k very very big ($k = n-3$)

Running time $O(2^{n-3}(n+m))$ very very slow

Comparison

Sample values of k	Brute Force	Recurse by edge
3	$O(n^3(n+m))$	$O(n+m)$
$\log n$	$O\left(n^{\log n}(n+m)\right)$	$O(n(n+m))$
$n/2$	$O\left(2^{\frac{n}{2}}(n+m)\right)$	$O\left(2^{\frac{n}{2}}(n+m)\right)$
$n-3$	$O(n^3(n+m))$	$O(2^{n-3}(n+m))$

Takeaway

If your vertex cover is small you can get a pretty efficient algorithm.

For k at most $O(\log n)$ it even becomes polynomial.

A “simple case” you can carve off.

More Generally

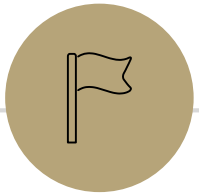
Measuring the complexity in terms of something other than the size of the input is called “parameterized complexity”

Common parameters:

The answer (like Ford-Fulkerson! And vertex cover)

How “complicated” the input is (e.g. for graphs, do you have a tree, something very close to a tree, or nothing like a tree).

Another example: SAT – an instance with few variables and many constraints is very different from an instance with many variables and few constraints.



Approximation Algorithms

Decision 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.

Finding an approximation for Vertex Cover

Take the idea from the clever exponential time algorithm.

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
```

Does it work?

Do we find a vertex cover?

Is it close to the smallest one?

But first, let's notice – we're back to polynomial time algorithms!

If we're going to take exponential time, we can get the exact answer. We want something fast if we're going to settle for a worse answer.

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!