+ practice material for final up now (i.e. 2 minutes ago)
  - some old solutions available, some not.
  - some differences in content; if you don't know how to approach a problem, we might not have taught you

+ final logistics
  - Are allowed to talk in small groups, but submission is still individual
  - Will have an Ed post to find extra people.

"Exams"

Coping With NP-Completeness

CSE 417 Winter 21
Lecture 25

Announcements

HW7 out tonight

Three questions:

Problem One asks you to take a practice exam.

Problem Two is DP practice (for more review)

Problem Three is an approximation algorithm.

You get to choose which of 2,3 you'd prefer.

If you do both, you can get extra credit.

But problem 3 will use material from Monday

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)?

$x_i = \neg x_j \vee \neg x_k$.

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…band-aids

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 ($2n^2$ 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:

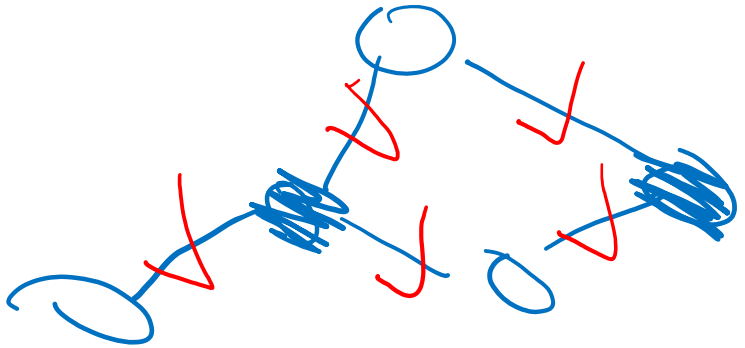
Exponential algorithms that aren't-as-slow-as-others

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.

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!

$\binom{n}{3}$

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.

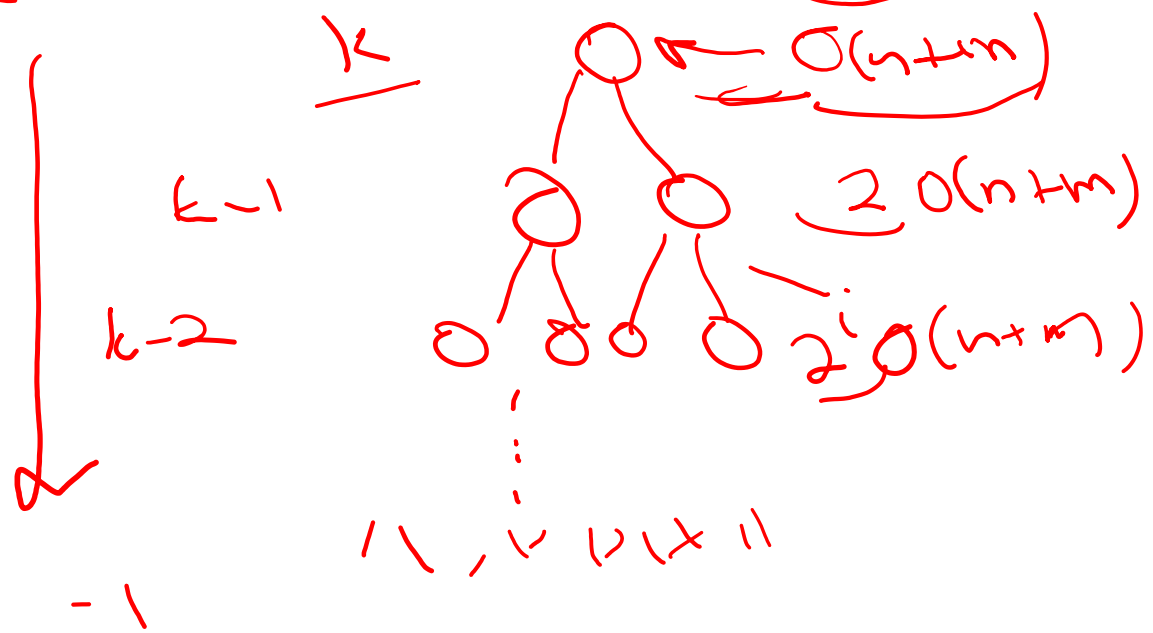
Algorithm

```
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



$$\sum_{i=0}^{k} 2^i O(n+m) \approx O(2^k (n+m))$$

Fill out the poll everywhere for Activity Credit!
Go to pollev.com/cse417 and login with your UW identity

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

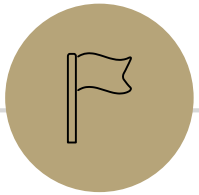
Measuring the complexity in terms of something other than the size of the input is called "parameterized complexity"

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!