Coping with NP-hardness

CSE 417 Fall 22
Lecture 27

Announcements

Don’t be scared when you open up HW8

It’s longer, and we’re letting you submit more problems, but the expectation when I’m making grades is the same as the other homeworks.

- You can do more if you want to review and/or catch-up, but you don’t have to.

The last resubmission cycle is due Monday Dec 12 (i.e., right after HW8)

- That’s the Monday of finals week.
- Part of the reason for having so many problems this week is to give you room to do an “extra” one if you’re really close to a grade-guarantee and want some insurance on getting a particular score.

Where Are We?

Any NP-hard problem has the property that if you find a polynomial time algorithm for it, you find a polynomial time algorithm for all problems in NP.

There are thousands of NP-hard problems! We don't think any of them have polynomial time algorithms (but we haven't proven it).

To show your problem B is NP-hard, reduce a known NP-hard problem A to your problem B.

Direction matters!

Where Are We?

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

Step 1: Understand your problem deeply

2-coloring is easy/3-coloring is hard.
Vertex Cover is easy on trees, bipartite graphs; hard on general graphs.

Step 2: Verify that your problem is NP-hard with a reduction

Step 3: Try some band-aids

Maybe a SAT-solver or other library function will just happen to do ok.
Maybe you run this code once a month and it doesn't matter if it's slow.

Step 4: Permanent solutions

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?

Step 4 – Permanent Solutions

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!

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

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

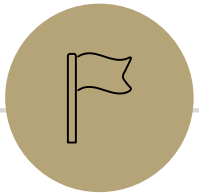
Measuring the complexity in terms of something other than the size of the input is called “parameterized complexity”

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

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.

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!

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 $\sum w(u) \cdot x_u$

Subject to:

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

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

Don't worry about the weights for today.

We got an exact solution for bipartite graphs….

What do we do

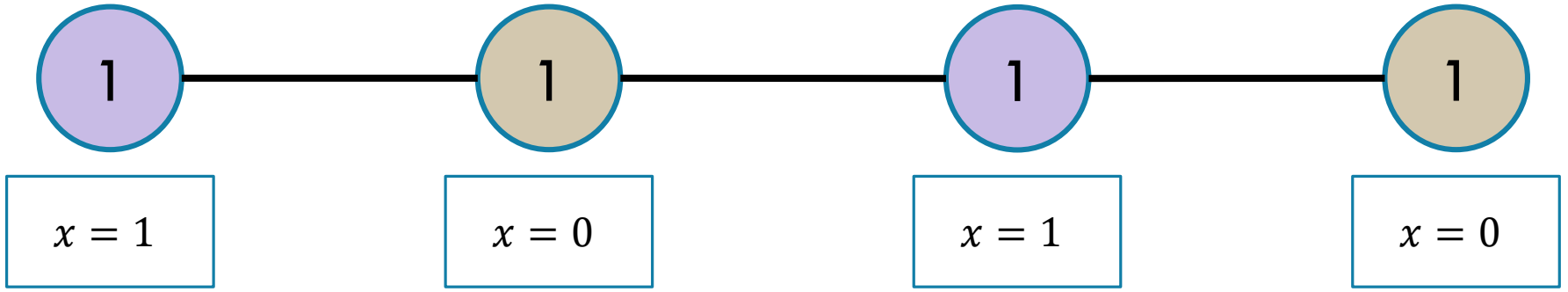
Increase x for the purple vertices, and decrease x for the gold vertices.

(at the same time at the same rate)

Every edge (in our example) has a purple and gold endpoint, so every constraint is still satisfied.

The objective (in our example) increases and decreases at the same rate.

So we still have an optimal (minimum) vertex cover



In General…

2-color the graph (call the vertices “purple” or “gold”)

Increase all the purple vertices by some value δ

And decrease all the gold vertices by the same value δ

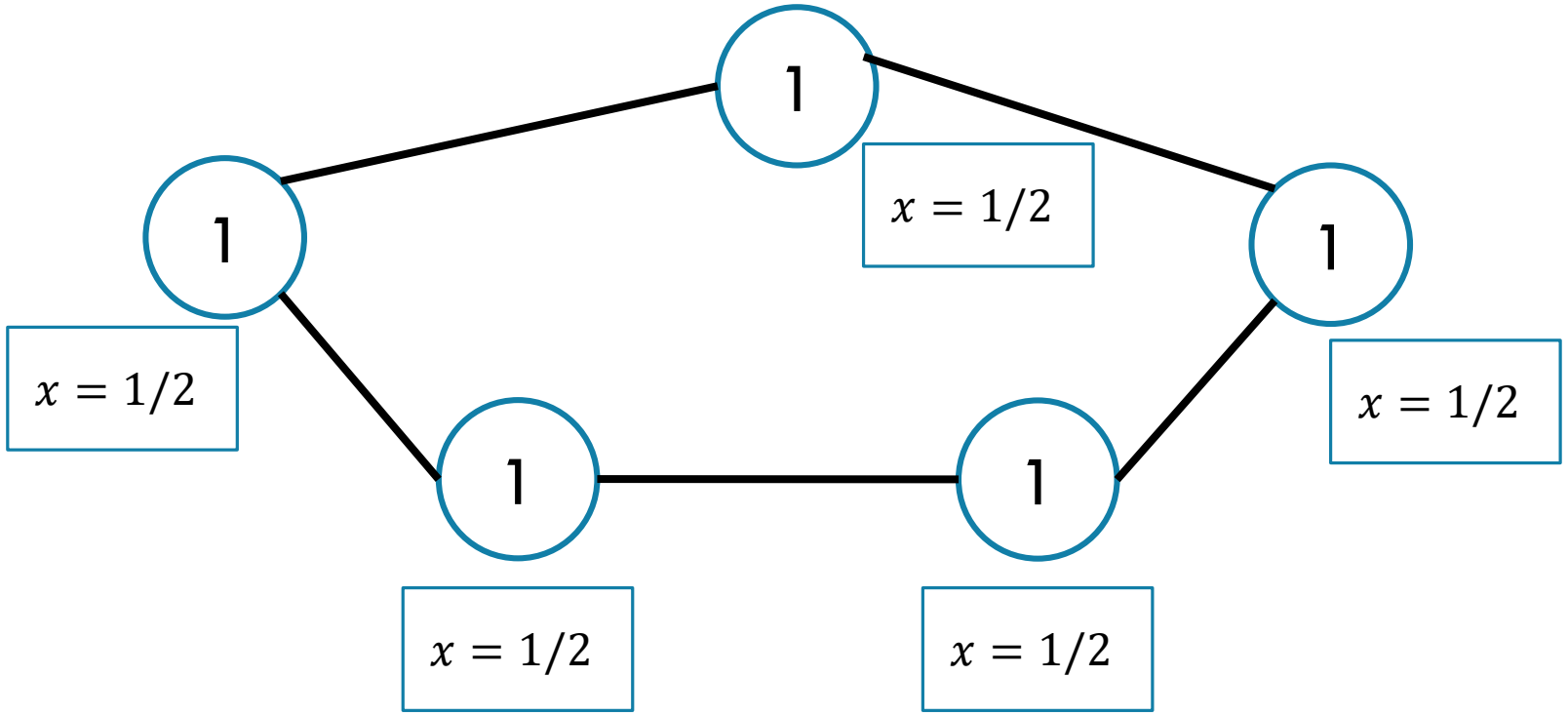
Choose δ so that we set at least one variable to 0 or 1 (but don’t move any variables outside the $[0,1]$ range allowed.

Those vertices that just got set to 0 or 1 can be deleted. Start over with the remaining graph.

Non-Bipartite

We needed the graph to be bipartite to be able to 2-color it.

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!

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.

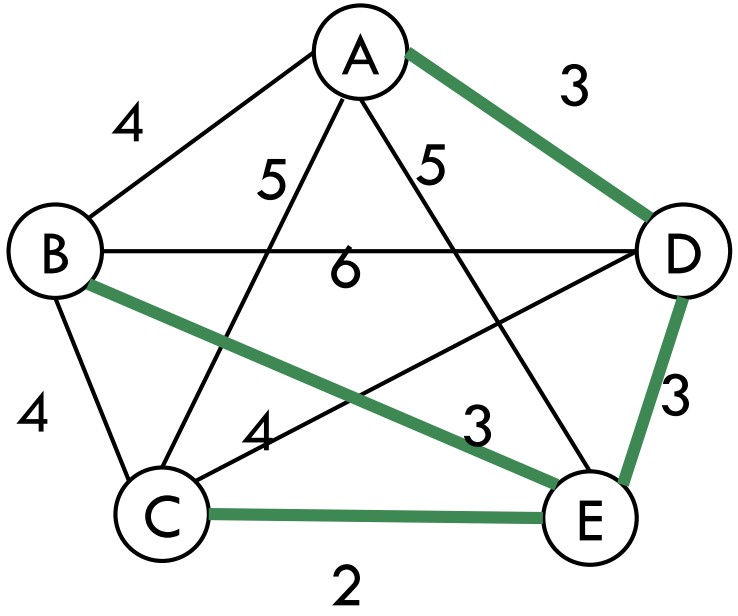
2. The weights satisfy the "triangle inequality" (it's faster to go from x to y directly than it is to go from x to y through x).

TSP starting point

What would be a good baseline?

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

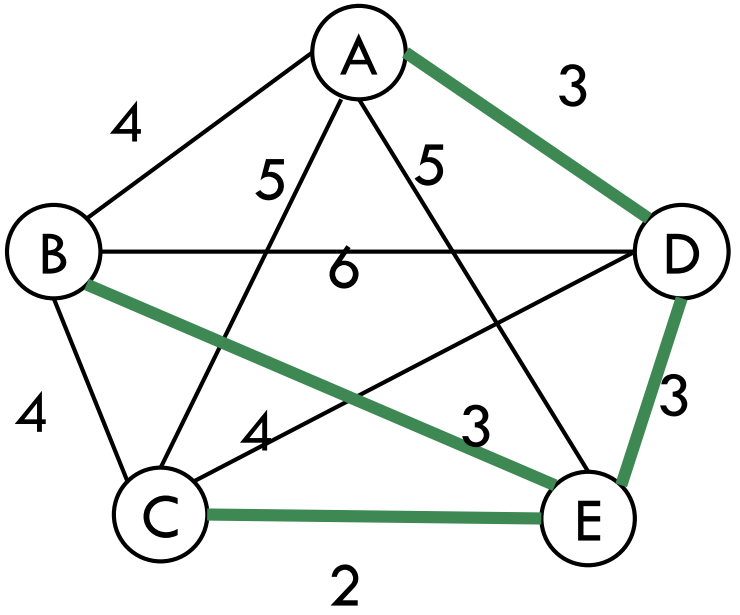
A Minimum Spanning Tree!



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.

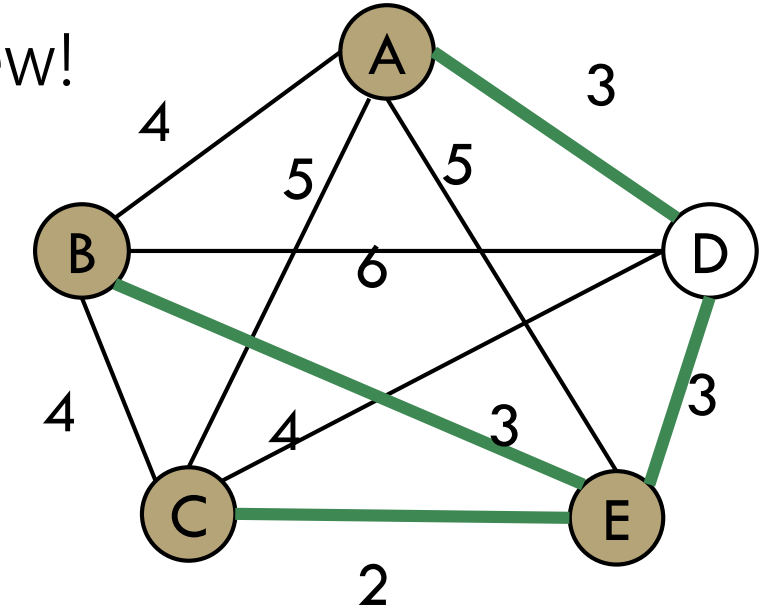
Doing a Little Better

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!

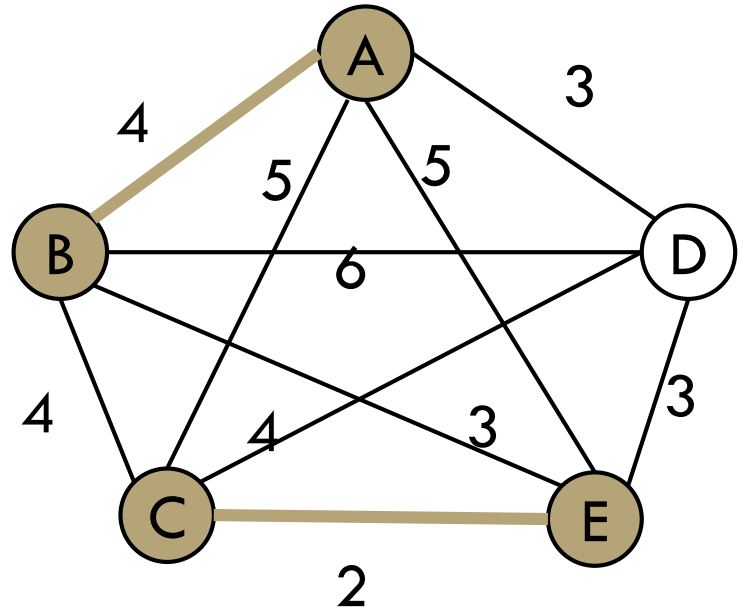
So lets add some possible ways out.



What would help?

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

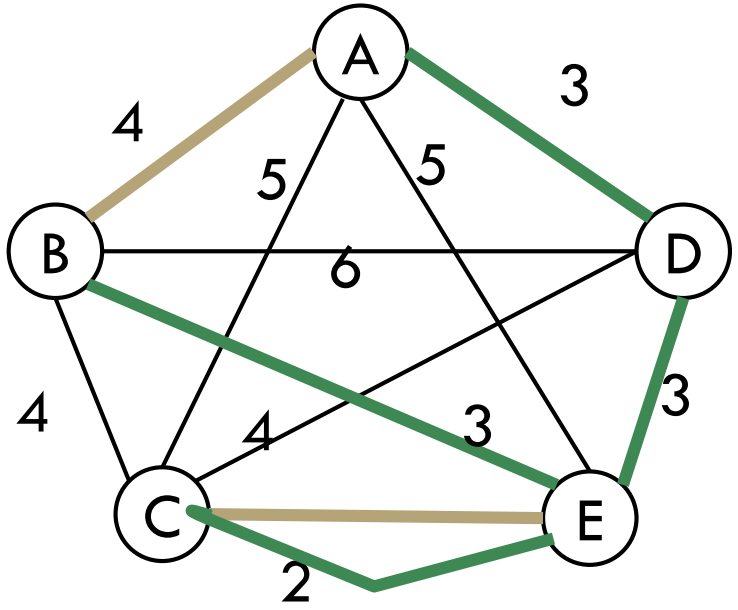
Specifically a minimum weight matching.



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)!

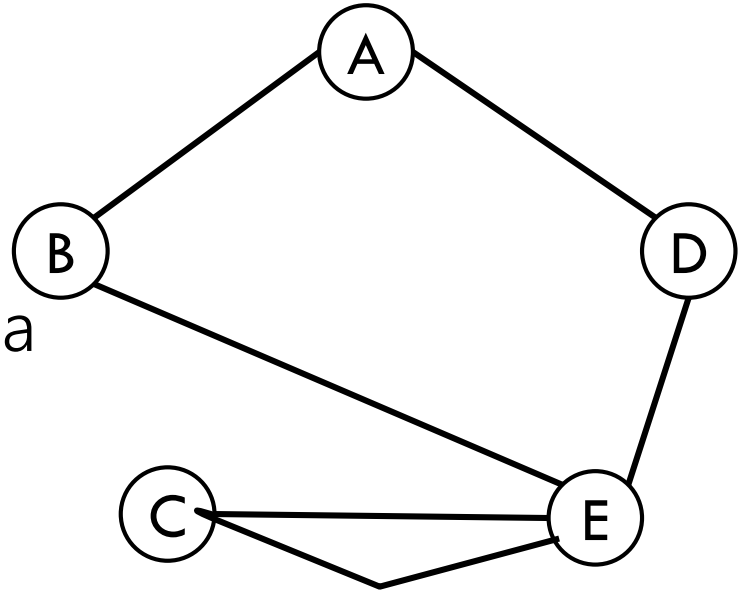
So we now have the MST AND the minimum weight matching on the odd edges.



Did It Help?

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

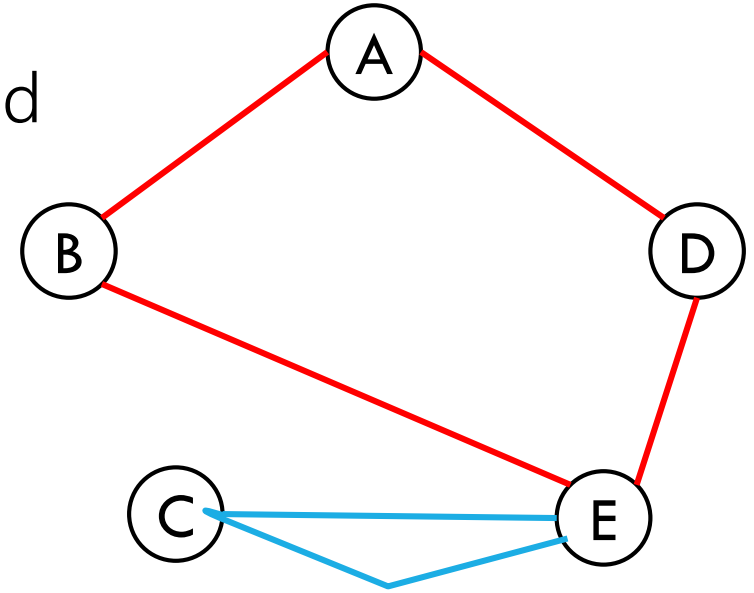
We'll have to find one.



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 **last year**.

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.