CSE 421 Winter 2025
Lecture 26:
Coping with NP-Hardness

Nathan Brunelle

http://www.cs.uw.edu/421

What to do if the problem you want to solve is NP-hard

This isn’t the only answer!

Things to consider

- Suppose you find that you need to solve problem A, but it's NP-Hard
- There could be some hope!
  - Are there properties of your input that will cause you to avoid the worst case?
  - Do to specifically need the "best" answer, or would "pretty good" suffice?
    - How good does "pretty good" NEED to be for you?

Case study: Minimum Vertex Cover

Vertex-Cover:

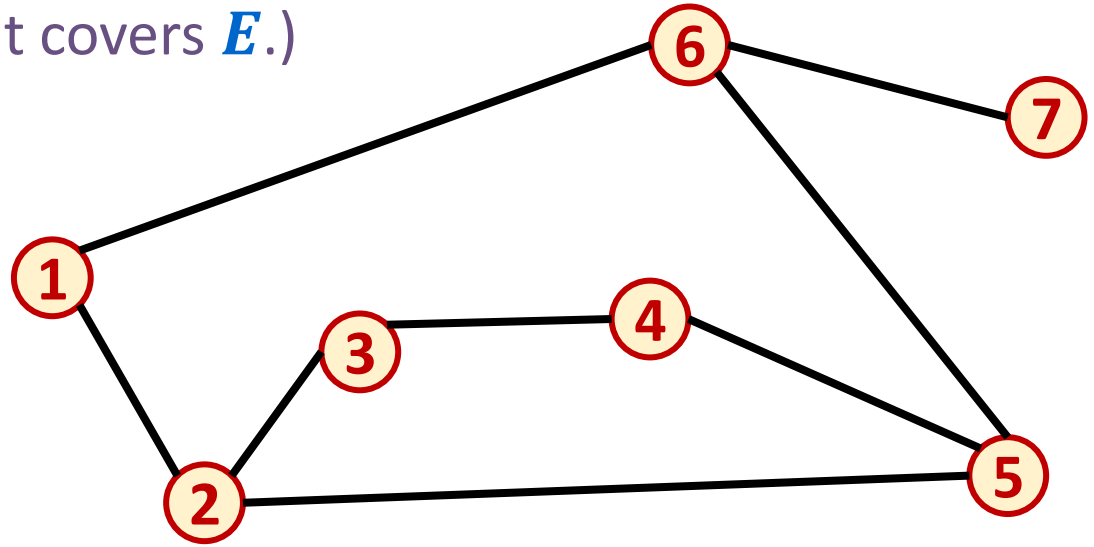
Given a graph $G = (V, E)$ and an integer k

Is there a $W \subseteq V$ with $|W| \leq k$ such that every edge of G has an endpoint in W? (W is a vertex cover, a set of vertices that covers E.)

Minimum Vertex-Cover:

Given a graph $G = (V, E)$

Find the largest $W \subseteq V$ with $|W|$ such that every edge of G has an endpoint in W? (W is a vertex cover, a set of vertices that covers E.)



How “Hard” is Minimum Vertex Cover?

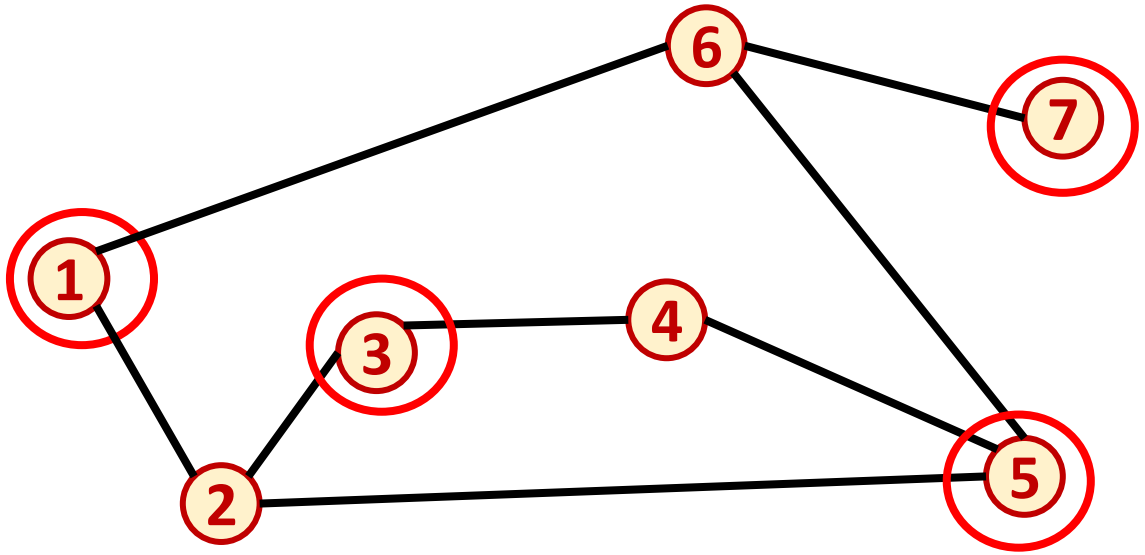
- If I could solve **Vertex-Cover** in polynomial time, could I then solve **Minimum Vertex-Cover** in polynomial time?

- If I could solve **Minimum Vertex-Cover** in polynomial time, could I then solve **Vertex-Cover** in polynomial time?

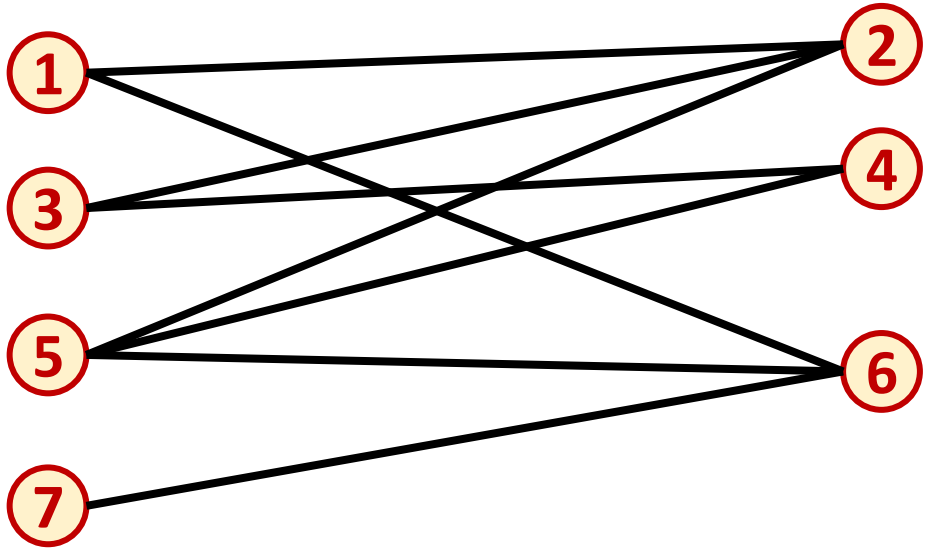
How “Hard” is Minimum Vertex Cover?

- If I could solve **Vertex-Cover** in polynomial time, could I then solve **Minimum Vertex-Cover** in polynomial time?
  - Call **Vertex-Cover** on the input $G, 1$
  - If that returns “No”, call it on $G, 2$
  - If that returns “No”, call it on $G, 3$
  - … first “Yes” must be the minimum
- If I could solve **Minimum Vertex-Cover** in polynomial time, could I then solve **Vertex-Cover** in polynomial time?
  - Call **Minimum Vertex-Cover** on the input G, suppose it returns W
  - Check if $|W| \leq k$
- Conclusion: We can’t expect to solve either in polynomial time….

This graph is bipartite!

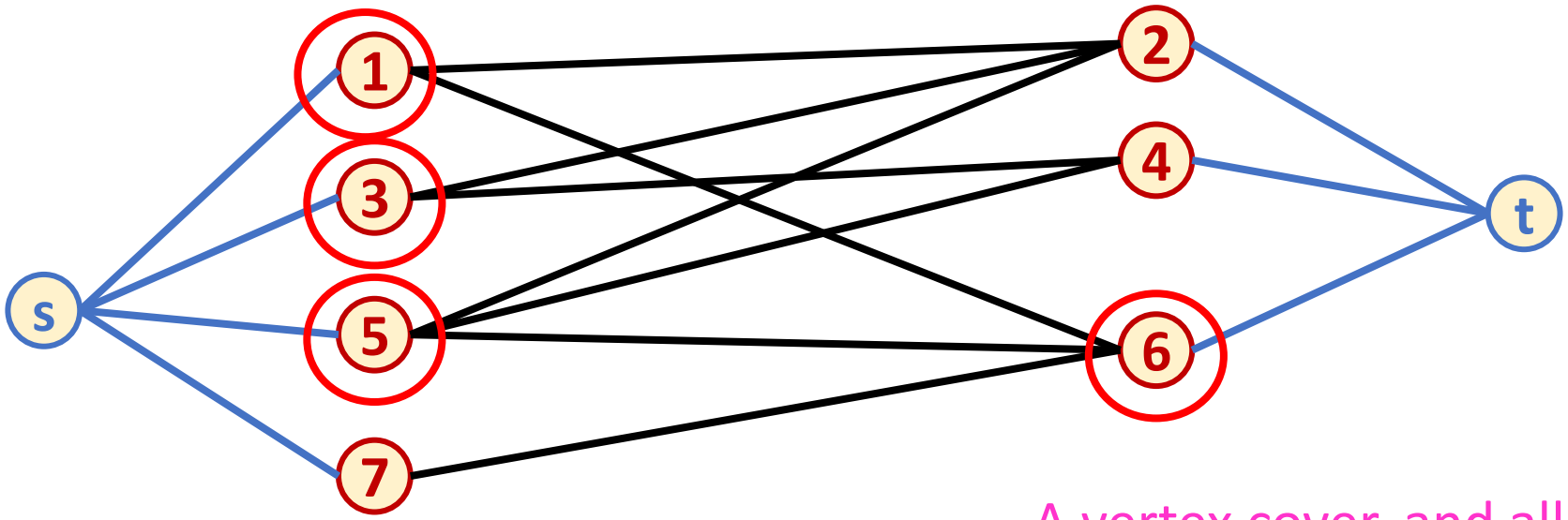


This graph is bipartite!



Vertex Covers Block Flow from s to t.

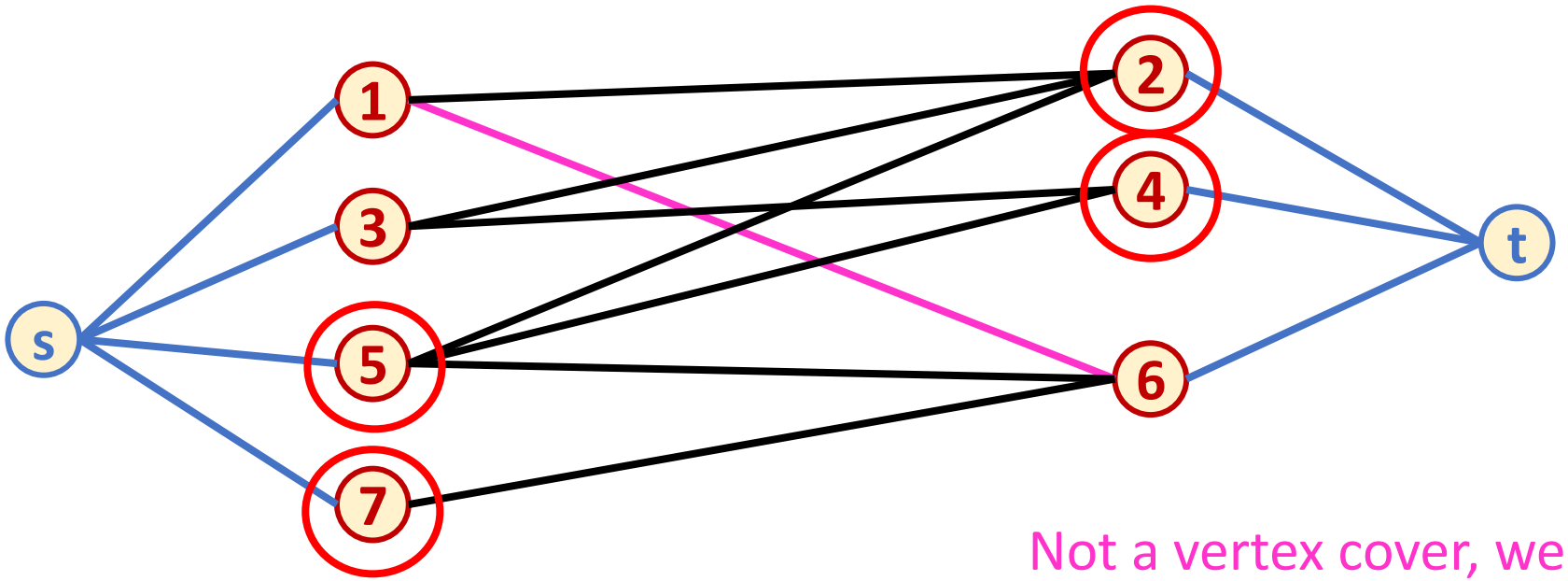
C is a vertex cover of G iff all flow from **s** to **t** must go through C.



A vertex cover, and all flow must pass through a selected node. "Clogging up" those nodes would stop the flow

Vertex Covers Block Flows from s to t.

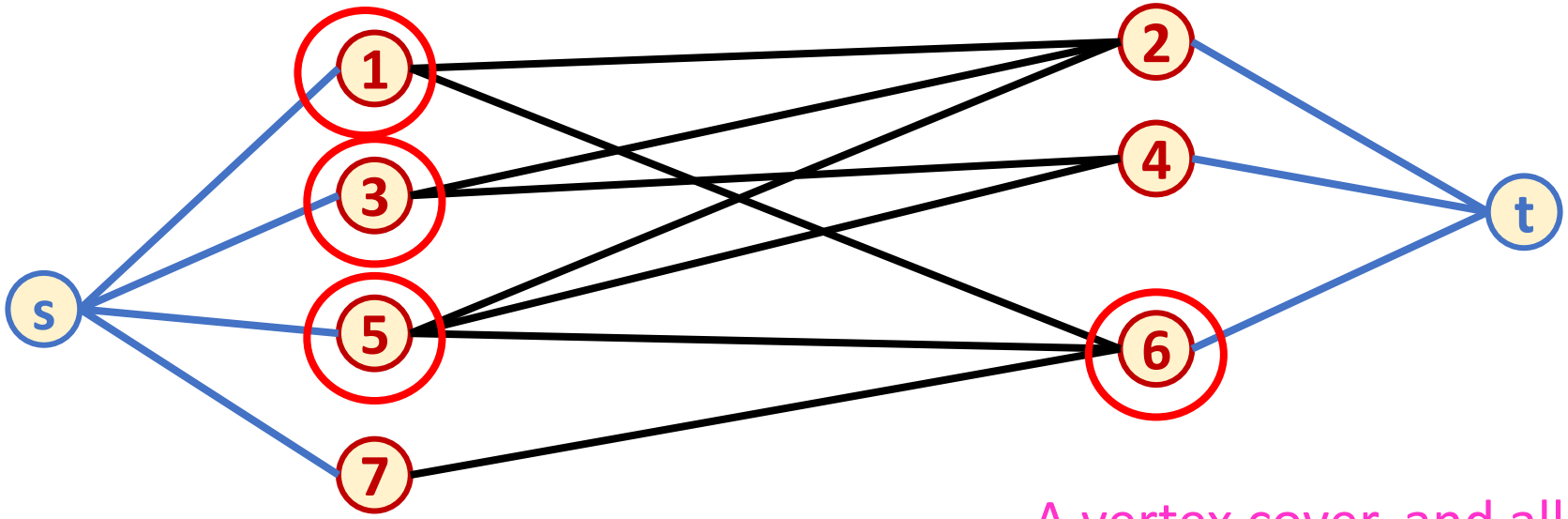
C is a vertex cover of G iff all flow from **s** to **t** must go through C.



Not a vertex cover, we can still have flow through edge (1,6)

Vertex Covers Block Flows from s to t.

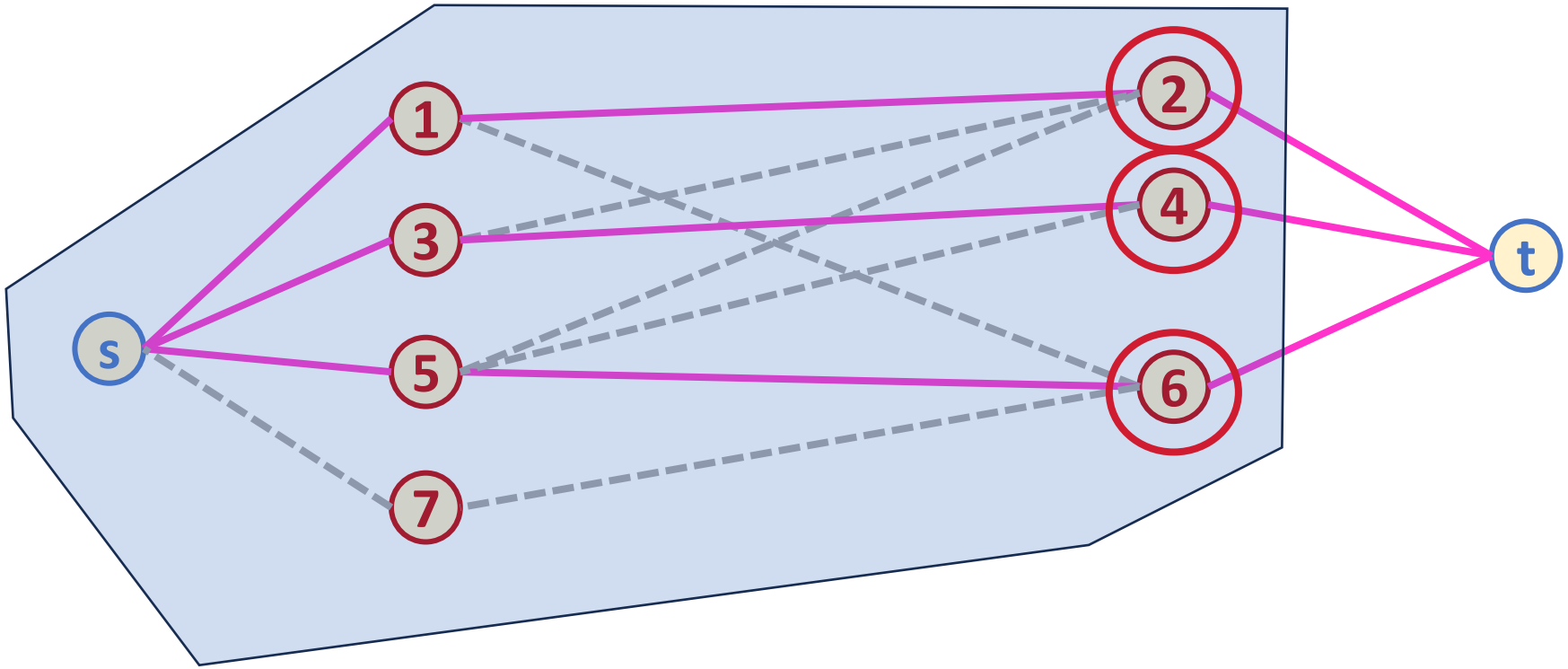
Recall that for any s-t cut in the graph, “clogging up” the crossing edges will stop the flow as well. This means we could stop flow by finding a cut, then selecting one end point for each edge in the cut.



A vertex cover, and all flow must pass through a selected node. “Clogging up” those nodes would stop the flow

Max Vertex On Bipartite Graphs

So… vertices of G involved in Min Cut (one per edge crossing the cut) form a minimum vertex cover of G.



Takeaway

- Check to see if there are properties of your inputs you assume
  - Will your graph always be a tree?
  - Will it always be bipartite?
  - Do you know the maximum degree of each node?
  - Will it be a DAG?
  - Any of these may enable you to write an efficient algorithm to solve the problem for the specific instances you may see
- Check to see if there are properties of your inputs that you can add in
  - Can you narrow the scope of your application or make other design decisions so that you can assume something above?

What to do if the problem you want to solve is NP-hard

2nd thing to try if your problem is a minimization or maximization problem

- Try to find a polynomial-time worst-case approximation algorithm
  - For a minimization problem
    - Find a solution with value $\leq \boldsymbol{K}$ times the optimum
  - For a maximization problem
    - Find a solution with value $\geq \mathbf{1}/\boldsymbol{K}$ times the optimum

Want $\boldsymbol{K}$ to be as close to $\mathbf{1}$ as possible.

Approximation Algorithm for Min Vertex Cover

$W = \emptyset$

$uncovered = E$

While($uncovered \neq \emptyset$){

  consider any $(u, v) \in uncovered$

  add u and v to W

  remove any edge connected to u or v

    from $uncovered$

}

return W

Final W is guaranteed to be a vertex cover, because edges could only be removed from $uncovered$ when they were covered by W.

Claim: At most a factor 2 larger than the optimal vertex-cover size.

Proof: Edges "considered" don't share any vertices. For any alternative vertex-cover, it must choose at least one of $\boldsymbol{u}$ or $\boldsymbol{v}$ to cover that "considered" edge. This means we didn't do worse that twice as many edges as necessary

Travelling-Salesperson Problem (TSP)

Travelling-Salesperson Problem (TSP):

Given: a set of n cities $v_1, \ldots, v_n$ and distance function d that gives distance $d(v_i, v_j)$ between each pair of cities

Find the shortest tour that visits all n cities.

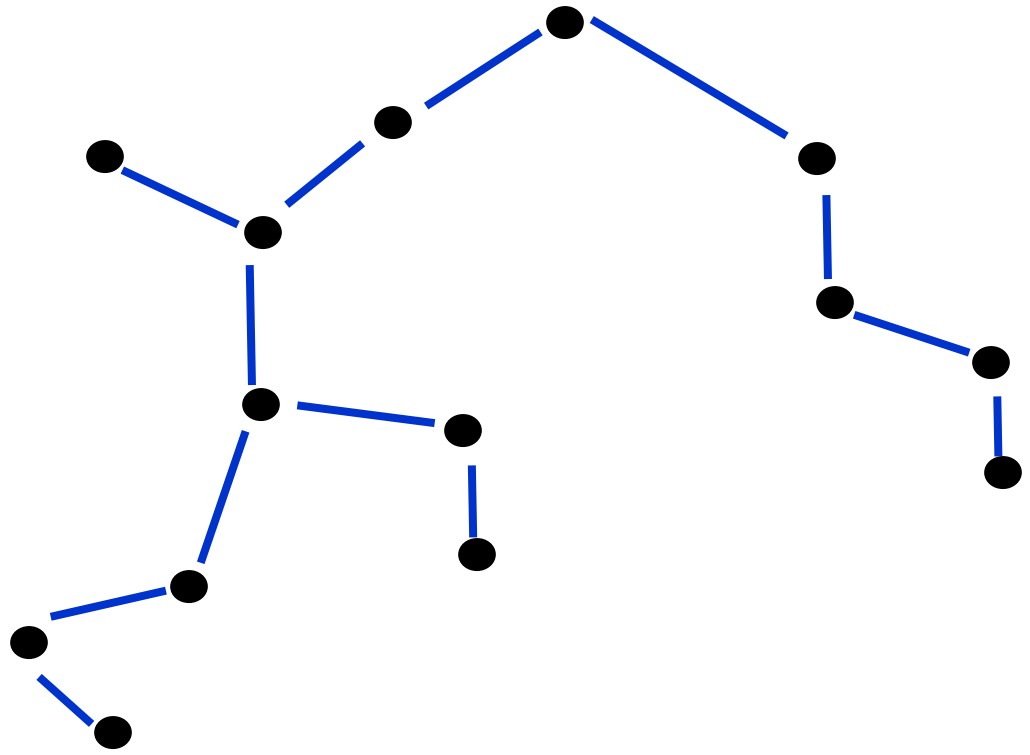
MetricTSP:

The distance function d satisfies the triangle inequality:

$$d(u, w) \leq d(u, v) + d(v, w)$$

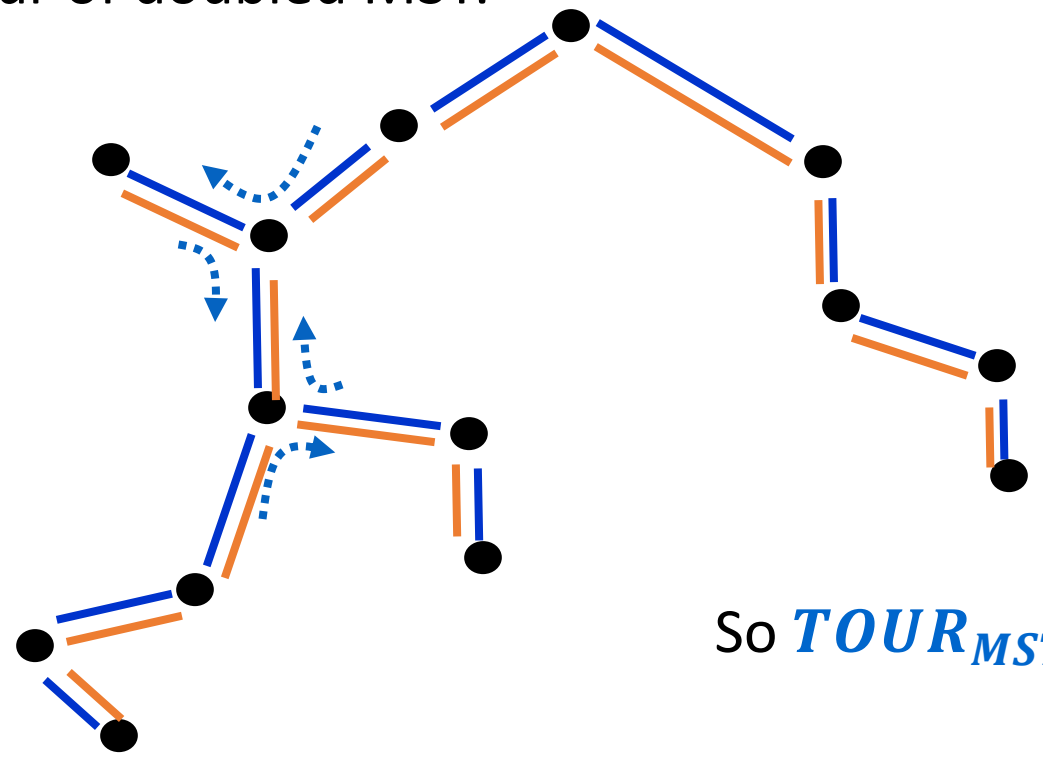
Proper tour: visit each city exactly once.

Minimum Spanning Tree Approximation: Factor of 2



TSP: Minimum Spanning Tree Factor 2 Approximation

Euler Tour of doubled MST:



Euler tour covers each edge twice so $\boldsymbol{TOUR_{MST}(G) = 2\,MST(G)}$

Any tour contains a spanning tree so $\boldsymbol{MST(G) \leq TOUR_{OPT}(G)}$

So $\boldsymbol{TOUR_{MST}(G) = 2\,MST(G) \leq 2\,TOUR_{OPT}(G)}$

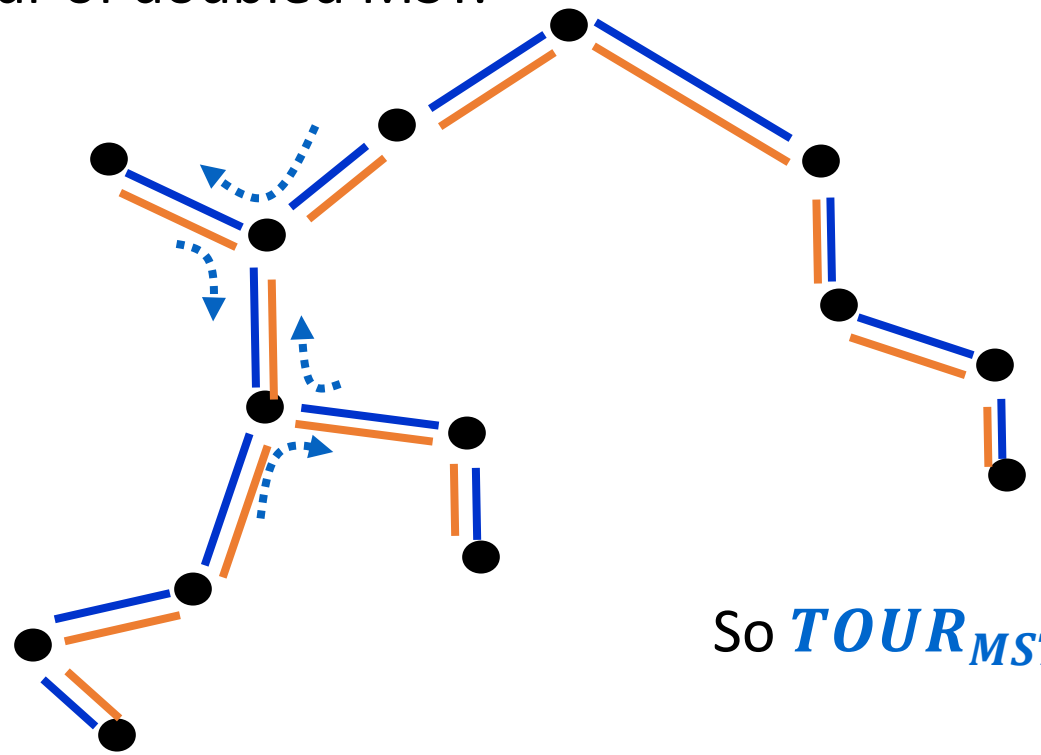
This visits each node more than once, so not a proper tour.

Why did this work?

- We found an Euler tour on a graph that used the edges of the original graph (possibly repeated).
- The weight of the tour was the total weight of the new graph.
- Suppose now
  - All edges possible
  - Weights satisfy the triangle inequality (MetricTSP)

MetricTSP: Minimum Spanning Tree Factor 2 Approximation

Euler Tour of doubled MST:



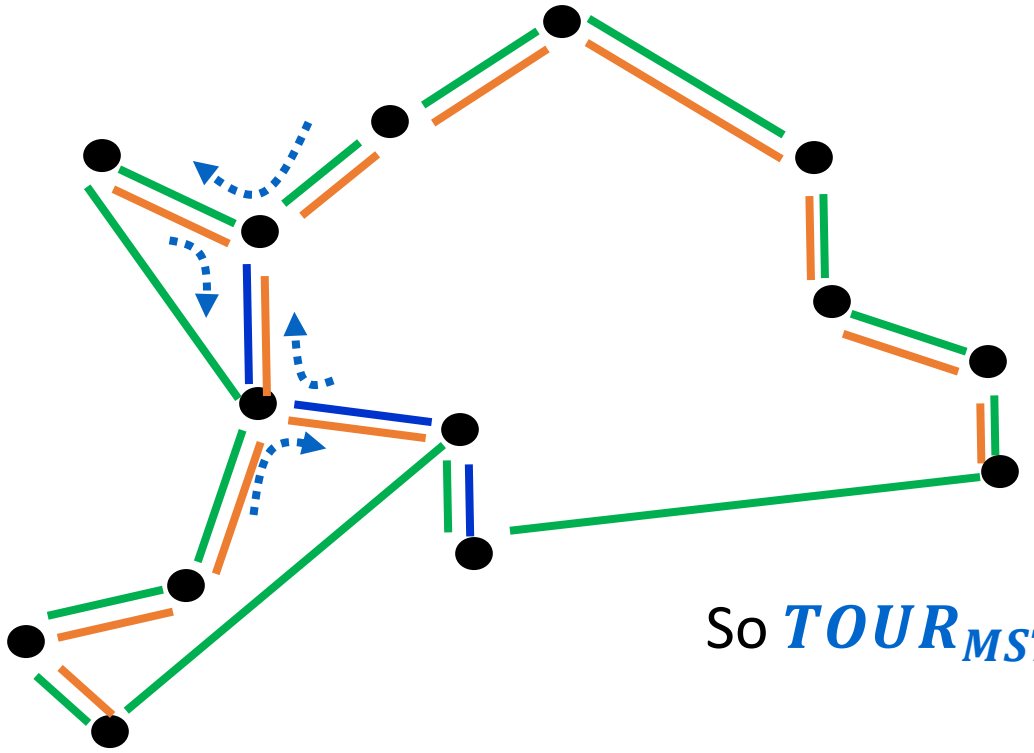
Euler tour covers each edge twice
so $\boldsymbol{TOUR_{MST}(G) = 2\,MST(G)}$

Any tour contains a spanning tree
so $\boldsymbol{MST(G) \leq TOUR_{OPT}(G)}$

So $\boldsymbol{TOUR_{MST}(G) = 2\,MST(G) \leq 2\,TOUR_{OPT}(G)}$

Instead: take shortcut to next unvisited vertex on the Euler tour
By triangle inequality this can only be shorter.

MetricTSP: Minimum Spanning Tree Factor 2 Approximation



Euler tour covers each edge twice
so $\boldsymbol{TOUR_{MST}(G) = 2\,MST(G)}$

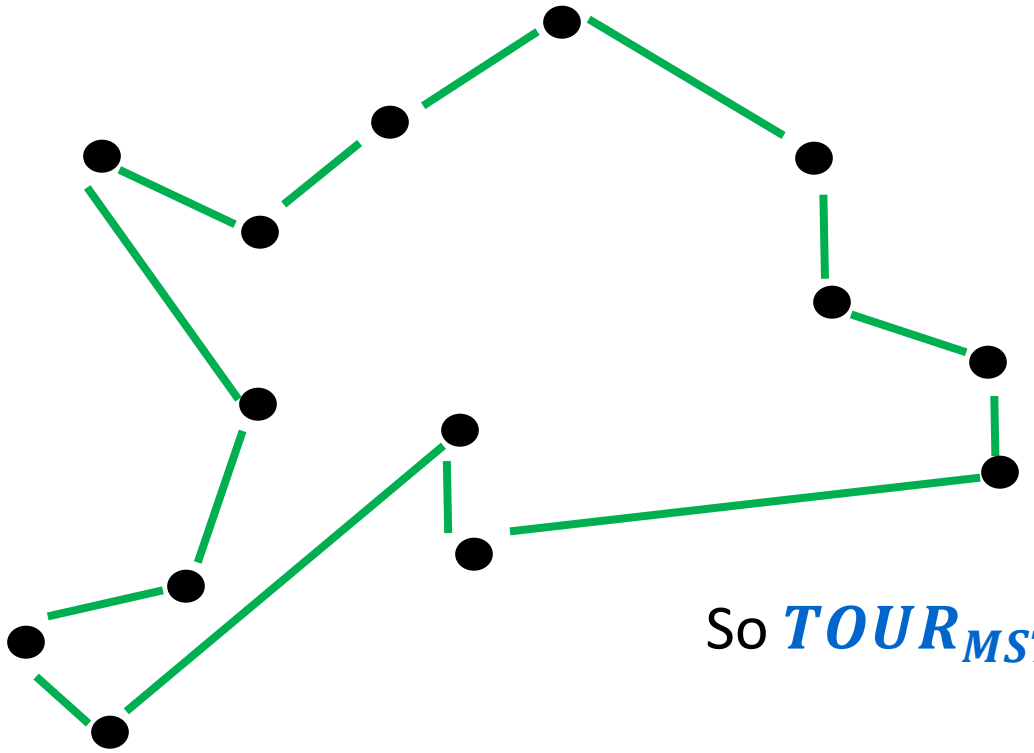
Any tour contains a spanning tree
so $\boldsymbol{MST(G) \leq TOUR_{OPT}(G)}$

So $\boldsymbol{TOUR_{MST}(G) = 2\,MST(G) \leq 2\,TOUR_{OPT}(G)}$

Instead: take shortcut to next unvisited vertex on the Euler tour
By triangle inequality this can only be shorter.

MetricTSP: Minimum Spanning Tree Factor 2 Approximation

Final:



Euler tour covers each edge twice
so $\boldsymbol{TOUR_{MST}(G) = 2\,MST(G)}$

Any tour contains a spanning tree
so $\boldsymbol{MST(G) \leq TOUR_{OPT}(G)}$

So $\boldsymbol{TOUR_{MST}(G) = 2\,MST(G) \leq 2\,TOUR_{OPT}(G)}$

Instead: take shortcut to next unvisited vertex on the Euler tour
By triangle inequality this can only be shorter.

Max-3SAT Approximation

Max-3SAT: Given a 3CNF formula F find a truth assignment that satisfies the maximum possible # of clauses of F.

Observation: A single clause on 3 variables only rules out $1/8$ of the possible truth assignments since each literal has to be false to be ruled out.

$\Rightarrow$ a random truth assignment will satisfy the clause with probability $7/8$.

So in expectation, if F has m clauses, a random assignment satisfies $7m/8$ of them.

A greedy algorithm can achieve this: Choose most frequent literal appearing in clauses that are not yet satisfied and set it to true.

If $\mathbf{P} \neq \mathbf{NP}$ no better approximation is possible

Other ways to approach Vertex Cover

- So Far:
  - Make assumptions about the graph
  - Decide you don’t need to be perfect
- Another Option:
  - For original Vertex Cover (the one with k), will we know k in advance?

What to do if the problem you want to solve is NP-hard

Maybe you only need to solve it if the solution size is small…

- What if you only need to find cliques or vertex covers of constant size?
- For both **Clique** and **Vertex Cover**, the obvious brute force algorithm would have time $\Theta(n^k)$: try all subsets of size k.
- For **Clique** the best algorithms known are all $n^{\Omega(k)}$
- However, **Vertex Cover** has a much better algorithm…

The theory of **fixed parameter tractability** looks at **NP** problems using a second parameter k in addition to input size n and seeks algorithms with running times $f(k) \cdot n^{O(1)}$ where f might be exponential.

Fixed Parameter Algorithms

The theory of **fixed parameter tractability** looks at **NP** problems using a second parameter k in addition to input size n and seeks algorithms with running times $f(k) \cdot n^{O(1)}$ where f might be exponential.

Clique: Extra parameter k for clique size target:

Brute force algorithm: try all subsets of size k and check: $\Theta(k^2 n^k)$ time.

Vertex-Cover: Extra parameter k for clique size target:

Brute force algorithm: try all subsets of size k and check: $\Theta(mn^k)$ time.

- Neither is a good fixed parameter algorithm

Vertex-Cover Fixed Parameter Algorithm

```
Vertex-Cover(C, b) {
    if there is an edge (u, v) not covered by C {
        if b > 0 {
            Vertex-Cover(C ∪ {u}, b − 1)
            Vertex-Cover(C ∪ {v}, b − 1)
        }
    }
    else
        Output YES (and set C) and halt
    }
}
```

Call **Vertex-Cover**$(\emptyset, k)$
if no answer, output **NO**

Analysis:

- Time to identify possible edge (u, v) not covered (and modify C) is $O(m + n)$
- # of recursive calls $\leq 2^k$
- Total runtime $O(2^k(m + n))$
- Somewhat of a "trick" of asymptotic analysis, but could be practical for small k, and you've "baked in" your feasible input size for other developers.
  - Make it so the algorithm only works when k is small enough for your application

Other ways to approach Vertex Cover

- So Far:
  - Make assumptions about the graph
  - Decide you don’t need to be perfect
  - Fix some parameter to be constant so the running time is polynomial
- Another Option:
  - Decide you’re ok with exponential time, but make your exponential time algorithm really fast
    - Many out-of-the-box algorithms for solving SAT, that are usually very fast (but occasionally slow or incorrect)

What to do if the problem you want to solve is NP-hard

Try to make an exponential-time solution as efficient as possible.

e.g. Try to search the space of possible hints/certificates in a more efficient way and hope that it is quick enough.

Backtracking search (choose, explore, unchoose)

e.g., for **SAT**, search through the 2^n possible truth assignments...

...but set the truth values one-by-one so we can able to figure out whole parts of the space to avoid,

e.g. Given $F = (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge (x_4 \vee \neg x_3) \wedge (x_1 \vee x_4)$

after setting $x_1 = 1$ and $x_2 = 0$ we don't even need to set x_3 or x_4 to know that it won't satisfy F.

Today: More clever backtracking search for **SAT** solutions

SAT Solving

SAT is an extremely flexible problem:

- The fact that **SAT** is an **NP**-complete problem says that we can re-express a huge range of problems as **SAT** problems

This means that good algorithms for **SAT** solving would be useful for a huge range of tasks.

Since roughly 2001, there has been a massive improvement in our ability to solve **SAT** on a wide range of practical instances

- These algorithms aren't perfect. They fail on many worst-case instances.

CNF Satisfiability

SAT: satisfiability problem for CNF formulas with any clause size

Write CNFs with the ∧ between clauses implicit:

$$F = (x_1 \vee \overline{x_2} \vee x_4)(\overline{x_1} \vee x_3)(\overline{x_3} \vee x_2)(\overline{x_4} \vee \overline{x_3})$$

Write assignment as literals assigned true: $x_1, x_2, x_3, \overline{x_4}$

Defn: Given partial assignment x_3 where

$$F = (x_1 \vee \overline{x_2} \vee x_4)(\overline{x_1} \vee x_3)(\overline{x_3} \vee x_2)(\overline{x_4} \vee \overline{x_3})$$

define **simplify**(F, x_3) by

simplify$(F, x_3) = (x_1 \vee \overline{x_2} \vee x_4)$ $\quad\quad\quad\quad$ x_2 $\quad$ $\overline{x_4}$

That is: remove satisfied clauses and remove unsatisfied literals from clauses.

F is satisfiable iff all clauses disappear under some assignment. It is not satisfiable under a partial assignment if we ever have 2 clauses containing only contradictory variables (e.g. $\overline{x_4}$ and x_4)

Satisfiability Algorithms

Local search: Solve **SAT** as a special case of **MaxSAT**

(incomplete, may fail to find satisfying assignment)

GSAT – random local search [Selman,Levesque,Mitchell 92]

Walksat – Metropolis [Kautz,Selman 96]

Backtracking search (complete)

- DPLL [Davis,Putnam 60], [Davis,Logeman,Loveland 62]
- CDCL: Adds clause learning and restarts

  GRASP, SATO, zchaff, MiniSAT, Glucose, etc.

Summary: If you need to solve an NP-Hard Problem

- Look for assumptions that simplify the problem
- Made design decisions so that you can make simplifying assumptions
- Fix some parameter to a reasonable size, then write an algorithm that is exponential only in that parameter (and therefore polynomial when fixed)
- Give up on trying to find the best solution, and instead approximate
  - When a minimization/maximization problem
  - May be helpful to find/create assumptions for better approximation
- Give up on trying to write a polynomial time algorithm at all, and instead use a fast exponential time algorithm
  - For example, reduce your problem to SAT (if it's NP-complete), then use an out-of-the-box SAT solver

Other Approches you might hear about

Genetic algorithms:

- View each solution as a **string** (analogy with DNA)
- Maintain a **population of good solutions**
- Allow **random mutations** of single characters of individual solutions
- **Combine two solutions** by taking part of one and part of another (analogy with crossover in sexual reproduction)
- Get rid of solutions that have the worst values and make multiple copies of solutions that have the best values (analogy with natural selection -- survival of the fittest).

*Usually very slow. In the rare cases when they produce answers with better objective function values than other methods they tend to produce very **brittle** solutions – that are very bad with respect to small changes to the requirements.*

Deep Neural Nets and NP-hardness?

- **Artificial neural networks**
  - based on very elementary model of human neurons
  - **Set up a circuit of artificial neurons**
    - each artificial neuron is an analog circuit gate whose computation depends on a set of **connection strengths**
  - **Train the circuit**
    - Adjust the connection strengths of the neurons by giving many positive & negative training examples and seeing if it behaves correctly
  - **The network is now ready to use**

Despite their wide array of applications, they have not been shown to be useful for NP-hard problems.

Quantum Computing and NP-hardness?

Use physical processes at the quantum level to implement "weird" kinds of circuit gates based on unitary transformations

- Quantum objects can be in a "superposition" of many pure states at once
  - Can have n objects together in a superposition of 2^n states
- Each quantum circuit gate operates on the whole superposition of states at once
  - Inherent parallelism but classical randomized algorithms have a similar parallelism: *not enough on its own*
  - Advantage over classical: copies interfere with each other.
- Exciting direction - theoretically able to factor efficiently.
  Major practical problems wrt errors, decoherence to be overcome.
- *Small brute force improvement but unlikely to produce exponential advantage for NP.*