CS 4102: Algorithms

Lecture 25: P vs. NP

David Wu

Fall 2019

Today’s Keywords

Reductions

NP-Completeness

P vs. NP

CLRS Readings: Chapter 34

Homework

HW9, **HW10C** due Thursday, December 5, 11pm

- Graphs, Reductions
- Written (LaTeX)

Final Exam

Monday, December 9, 7pm in Olsson 120

- Practice exam coming soon
- Review session likely the weekend before
- SDAC: Please sign-up for a time on December 9

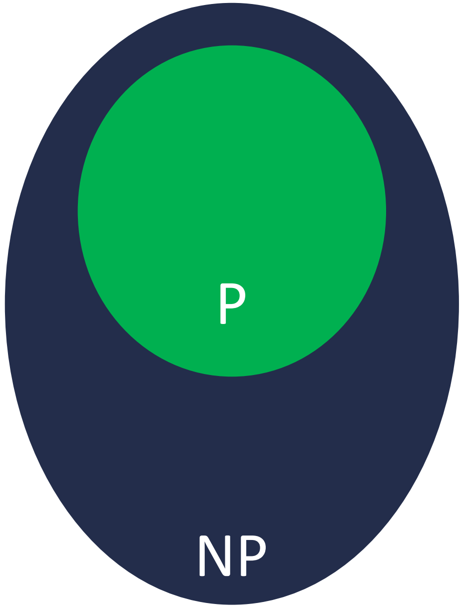
P vs. NP

A language $\mathcal{L} \in \mathrm{NP}$ if there exists a deterministic polynomial-time "verifier" $\mathcal{R}$ such that

$$x \in \mathcal{L} \Leftrightarrow \exists w \in \{0,1\}^{\mathrm{poly}(|x|)}: \mathcal{R}(x, w) = 1$$

A language $\mathcal{L} \in \mathrm{P}$ if there exists a deterministic polynomial-time "solver" $\mathcal{R}$ such that

$$x \in \mathcal{L} \Leftrightarrow \mathcal{R}(x) = 1$$



If we can decide a problem in polynomial time, we can verify a solution to the problem in polynomial time:

$$\mathrm{P} \subseteq \mathrm{NP}$$

Biggest open problem in computer science: is this containment strict?

$\mathrm{P} = \mathrm{NP}$ or $\mathrm{P} \neq \mathrm{NP}$

Understanding the Landscape of NP

Question: What are the hard problems in NP?

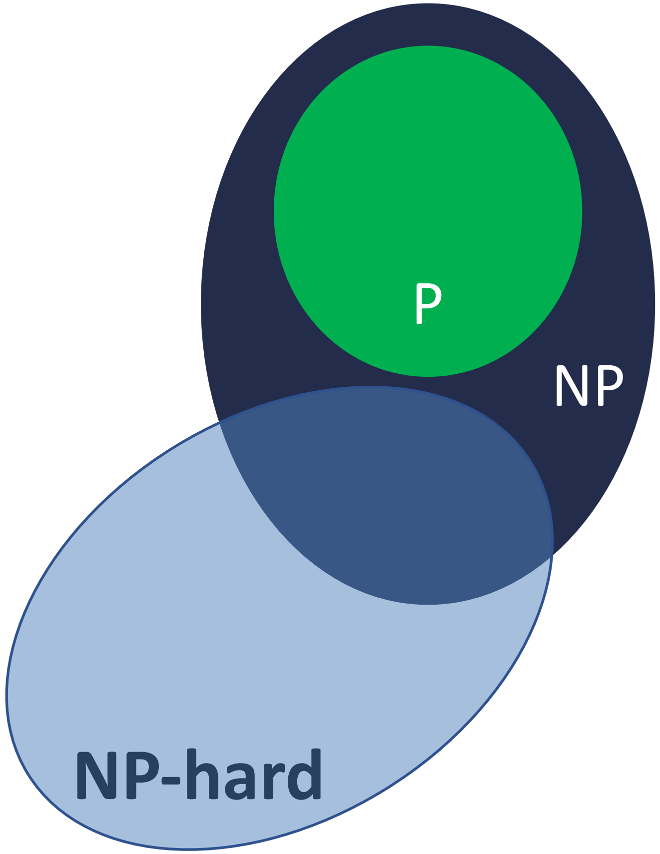
- Can we systematically characterize these?
- Can we use insights from one problem to help solve another problem?

Strategy: Identify problems at least as "hard" as NP

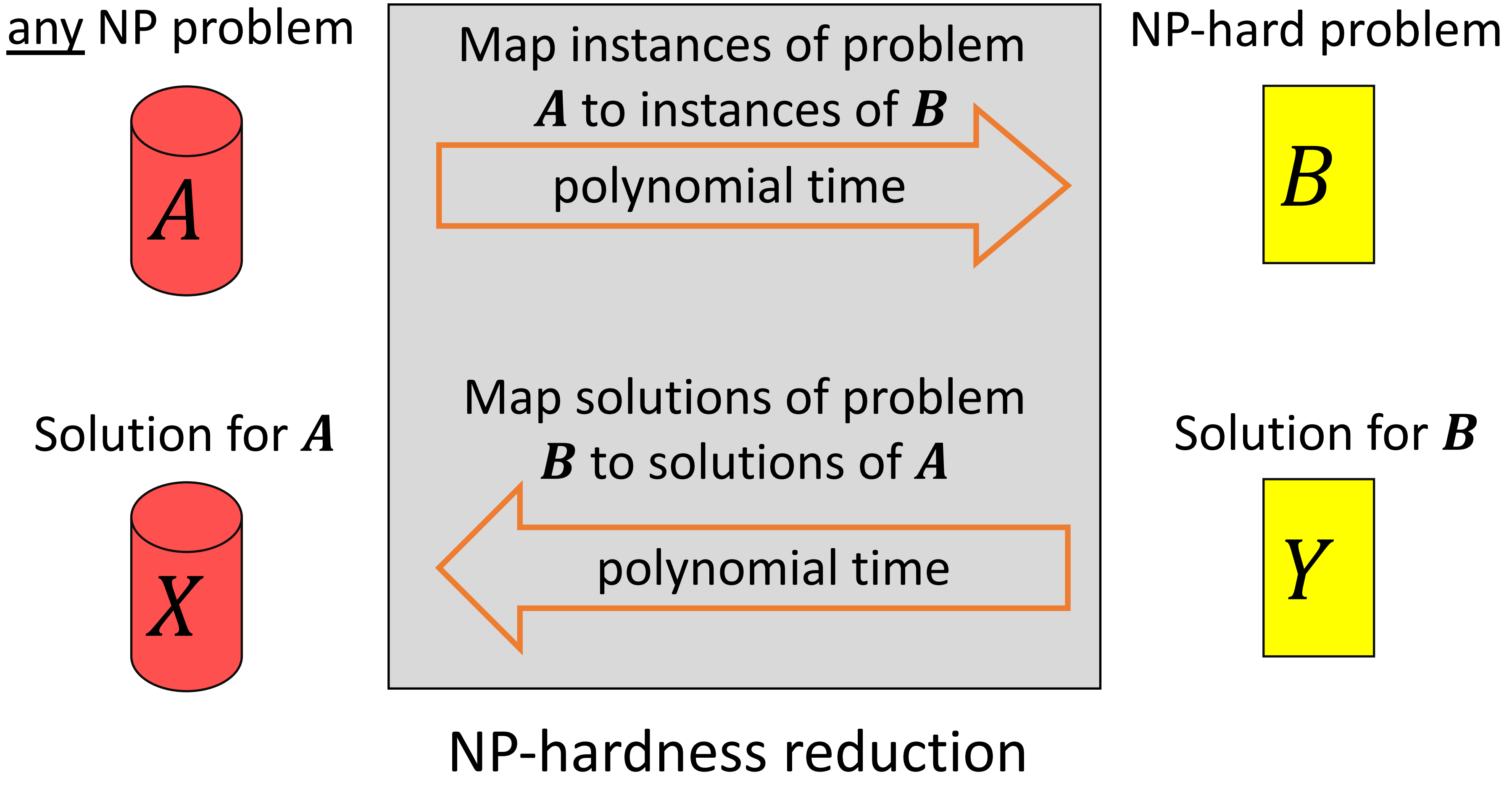
- If any of these "hard" problems can be solved in polynomial time, then all NP problems can be solved in polynomial time

A problem (or language) B is **NP-hard**

- $\forall A \in \text{NP}, A \leq_p B$
- $A \leq_p B$ means A reduces to B in polynomial time

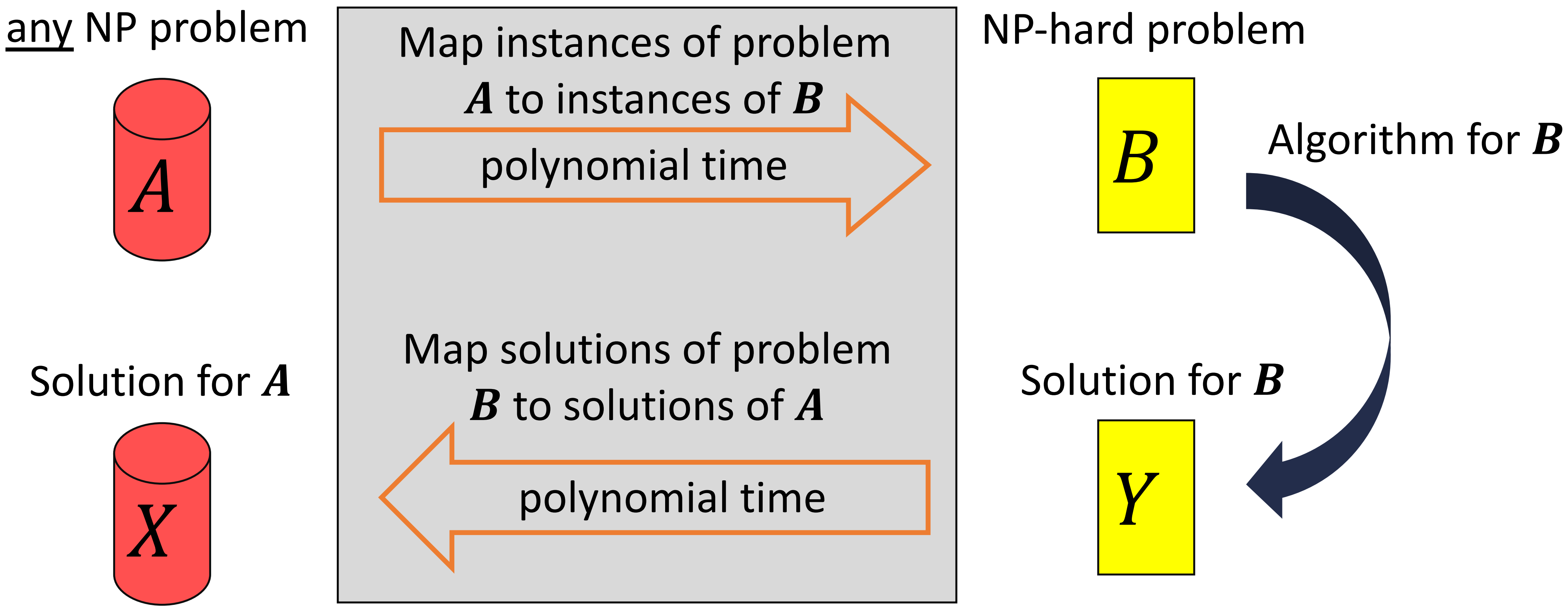


NP-Hardness



$\boldsymbol{A} \leq_p \boldsymbol{B}$: there is a polynomial-time reduction from $\boldsymbol{A}$ to $\boldsymbol{B}$

NP-Hardness



Very powerful: if we can solve even one NP-hard problem in polynomial time, we can solve all of them!

Understanding the Landscape of NP

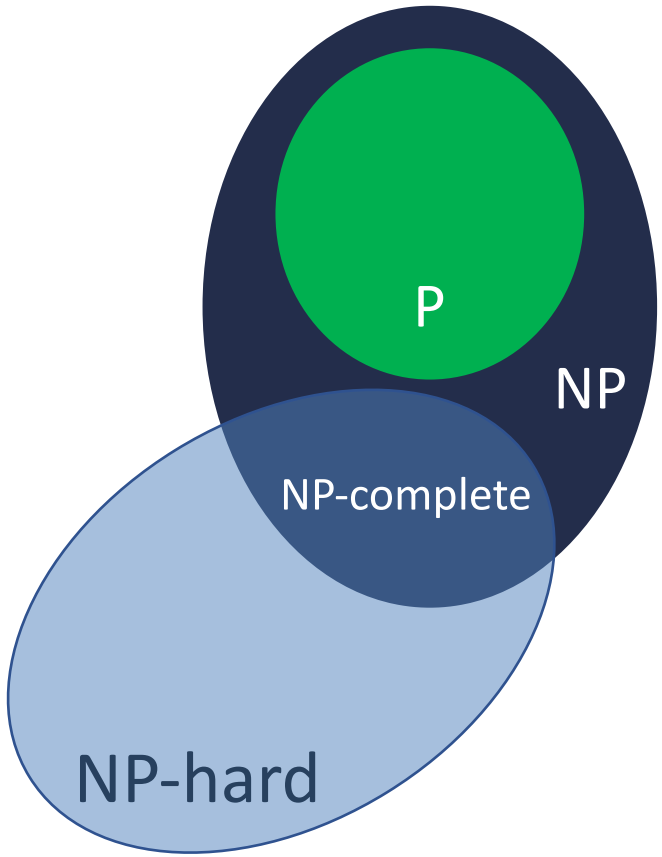
Question: What are the hardest problems in NP?

- By definition, an efficient algorithm for an NP-hard problem implies an efficient algorithm for every NP problem
- **Answer:** the ones that are NP-hard (if there are any)

NP-complete = NP ∩ NP-hard

"Complete" for NP in the sense that a solution to one implies a solution to all

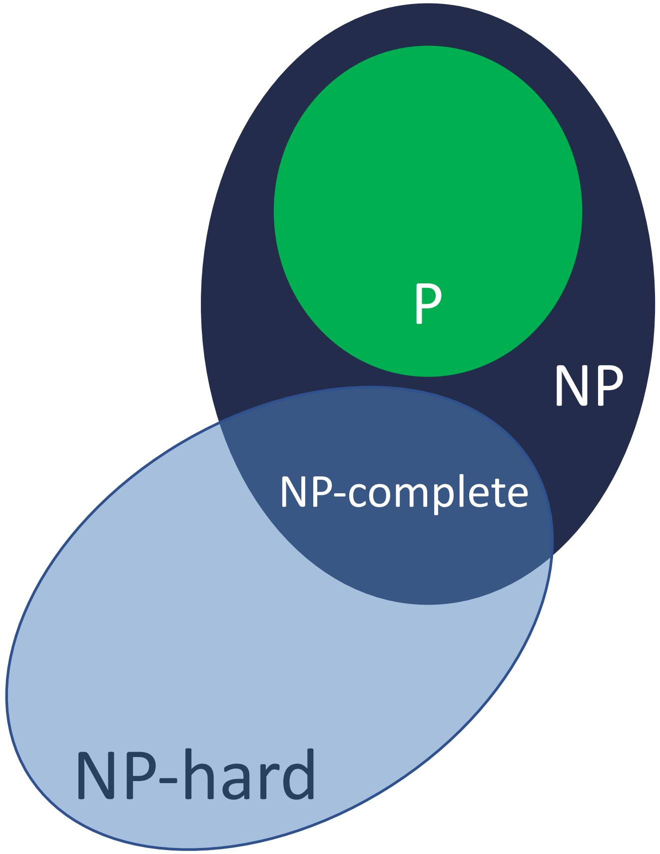
- To show $\mathrm{P} = \mathrm{NP}$, just need a single polynomial-time algorithm for a single NP-complete (or NP-hard) problem
- To show $\mathrm{P} \neq \mathrm{NP}$, just need a single lower-bound that some NP problem cannot be solved in polynomial time



Understanding the Landscape of NP

Question: What are the <u>hardest</u> problems in NP?

- By definition, an efficient algorithm for an NP-hard problem implies an efficient algorithm for <u>every</u> NP problem
- **Answer:** the ones that are NP-hard (if there are any)



NP-complete = NP ∩ NP-hard

"Complete" for NP
solution to <u>all</u>

- To show $\mathrm{P} = \mathrm{N}$
  single NP-comp
- To show $\mathrm{P} \neq \mathrm{NP}$, just need a <u>single</u> lower-bound that some NP problem cannot be solved in polynomial time

Not only do our existing techniques for proving lower bounds not work here, we are able to <u>prove</u> that most of our techniques will <u>always</u> fail…

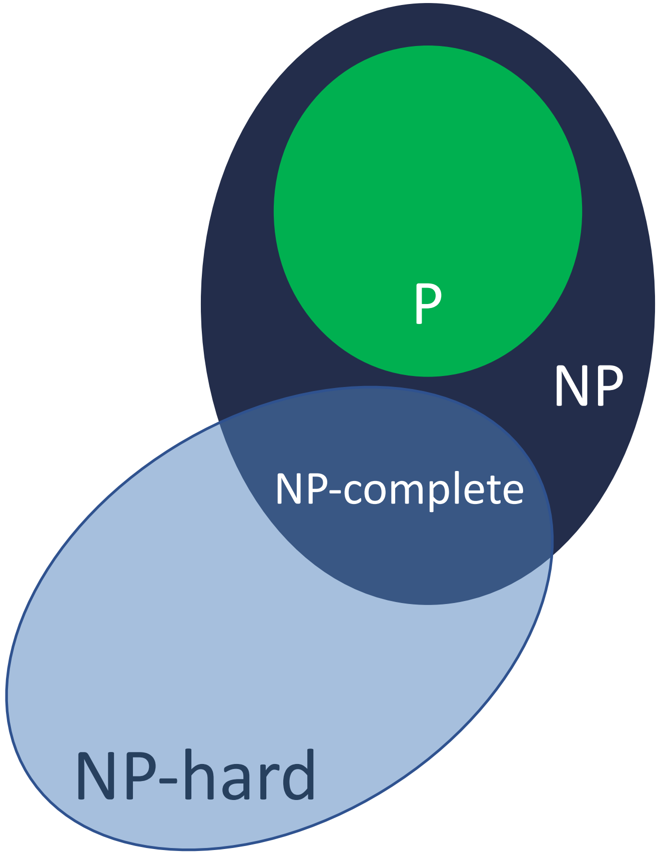
NP-Completeness

NP-complete = NP ∩ NP-hard

To prove that a problem (or language) is NP-complete:

- Show it is in NP (i.e., construct a polynomial-time verifier)
- Show it is NP-hard (i.e., show <u>every</u> problem in NP reduces to it)

But there are a <u>lot</u> of problems in NP…



NP-Completeness

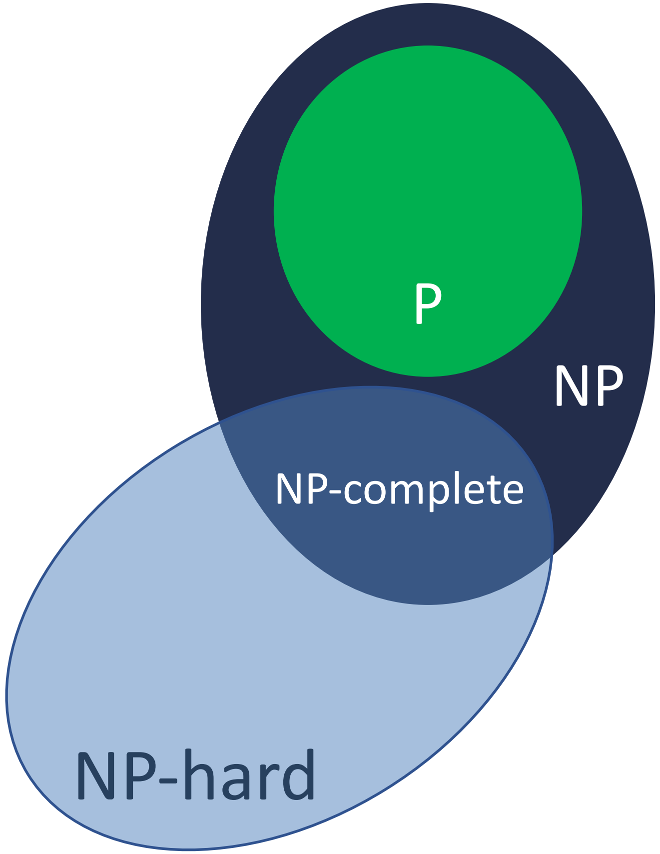
NP-complete = NP ∩ NP-hard

To prove that a problem (or language) is NP-complete:

- Show it is in NP (i.e., construct a polynomial-time verifier)
- Show it is NP-hard (i.e., show every problem in NP reduces to it)
  - Sufficient to show that another NP-hard problem reduces to it
  - Suppose C is NP-hard and $C \leq_p B$; then for all $A \in \mathrm{NP}$

    $$A \leq_p C \leq_p B \Rightarrow A \leq_p C$$

  - **Challenge:** coming up with a first NP-hard problem



3-SAT (Satisfiability)

Shown to be NP-hard by Cook and Levin (independently)

Given a 3-CNF formula (logical AND of clauses, each an OR of 3 variables), is there an assignment of true/false to each variable to make the formula true (i.e., satisfy the formula)?

$$(x \vee y \vee z) \wedge (x \vee \bar{y} \vee y) \wedge (u \vee y \vee \bar{z}) \wedge (z \vee \bar{x} \vee u) \wedge (\bar{x} \vee \bar{y} \vee \bar{z})$$

Clause

Variables

$x = \text{true}$
$y = \text{false}$
$z = \text{false}$
$u = \text{true}$

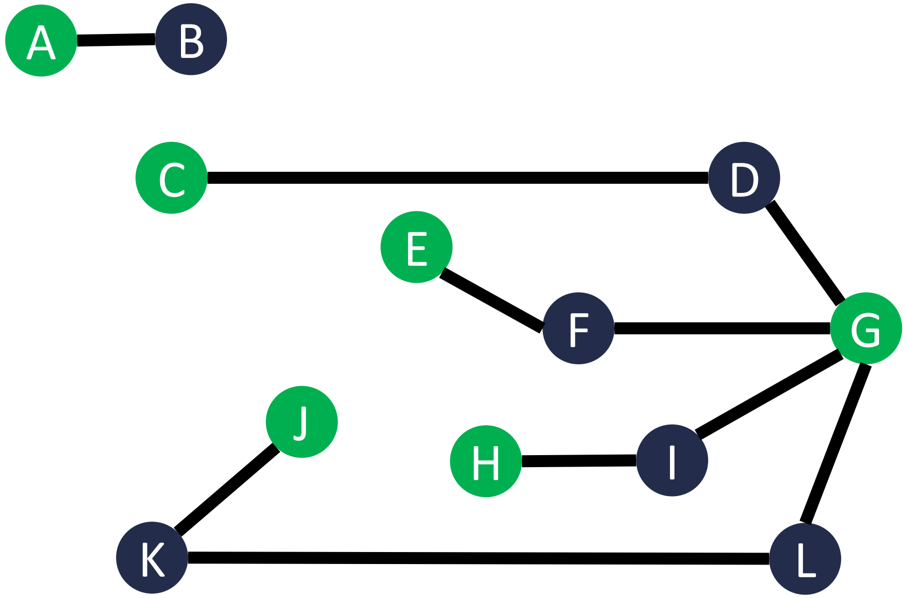
k-Independent Set is NP-Complete

1. Show that it belongs to NP
2. Show it is NP-Hard
   - Show 3-SAT $\leq_p$ k-Independent Set

k-Independent Set is in NP

Show: For any graph G:

- There is a short witness (i.e., proof) that G has a k-independent set
- The proof can be checked efficiently (in polynomial time)



Graph G

Witness for G: $S = \{A, C, E, G, H, J\}$
(nodes in the k-independent set)

Checking the witness:

- Check that $|S| = k$ $O(k) = O(|V|)$
- Check that every edge is incident on at most one node in S $O(|V| + |E|)$

Total time: $O(|E| + |V|) = \text{poly}(|V| + |E|)$

k-Independent Set is NP-Complete

1. Show that it belongs to NP ✅
2. Show it is NP-Hard
   - Show 3-SAT $\leq_p$ k-Independent Set

3-SAT $\leq_p$ k-Independent Set

3-SAT

$(x \vee y \vee z) \wedge (x \vee \bar{y} \vee y) \wedge (u \vee y \vee \bar{z})$

$x = \text{true}$
$y = \text{false}$
$z = \text{false}$
$u = \text{true}$

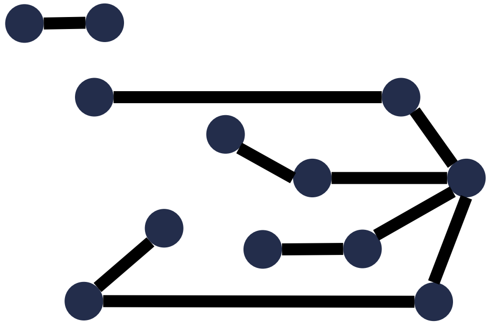
Map instances of problem $\boldsymbol{A}$ to instances of $\boldsymbol{B}$

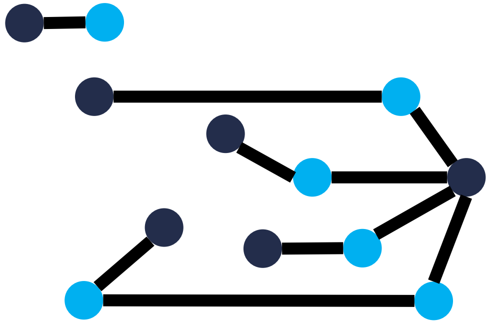
polynomial time

Map solutions of problem $\boldsymbol{B}$ to solutions of $\boldsymbol{A}$

polynomial time

k-independent set

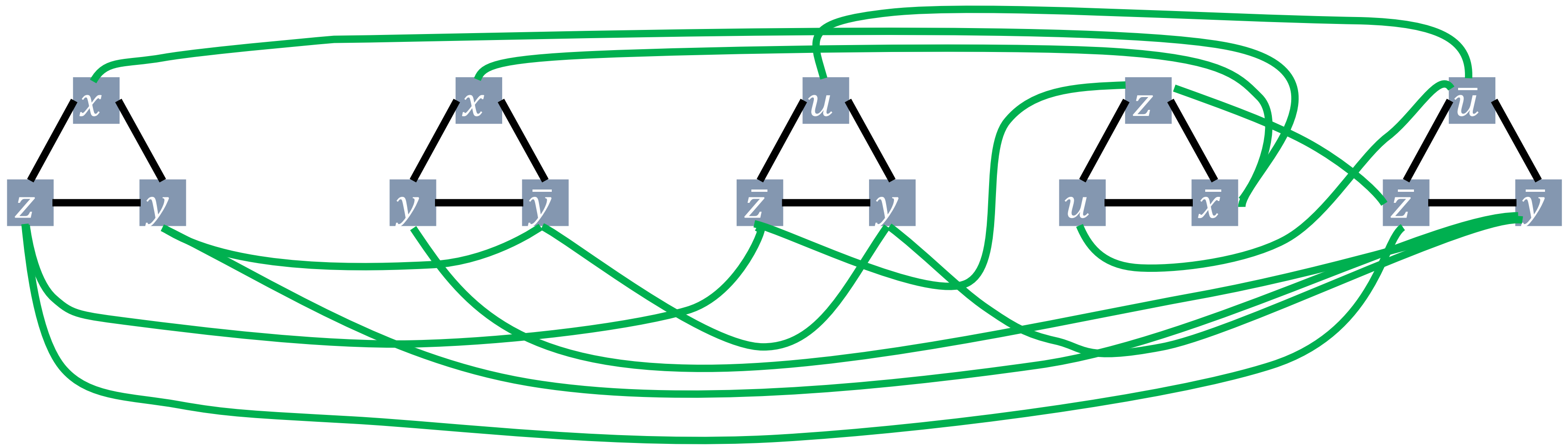




polynomial-time reduction

3-SAT $\leq_p$ k-Independent Set

$(x \vee y \vee z) \wedge (x \vee \bar{y} \vee y) \wedge (u \vee y \vee \bar{z}) \wedge (z \vee \bar{x} \vee u) \wedge (\bar{x} \vee \bar{y} \vee \bar{z})$



For each clause, construct a <u>triangle graph</u> with its three variables as nodes

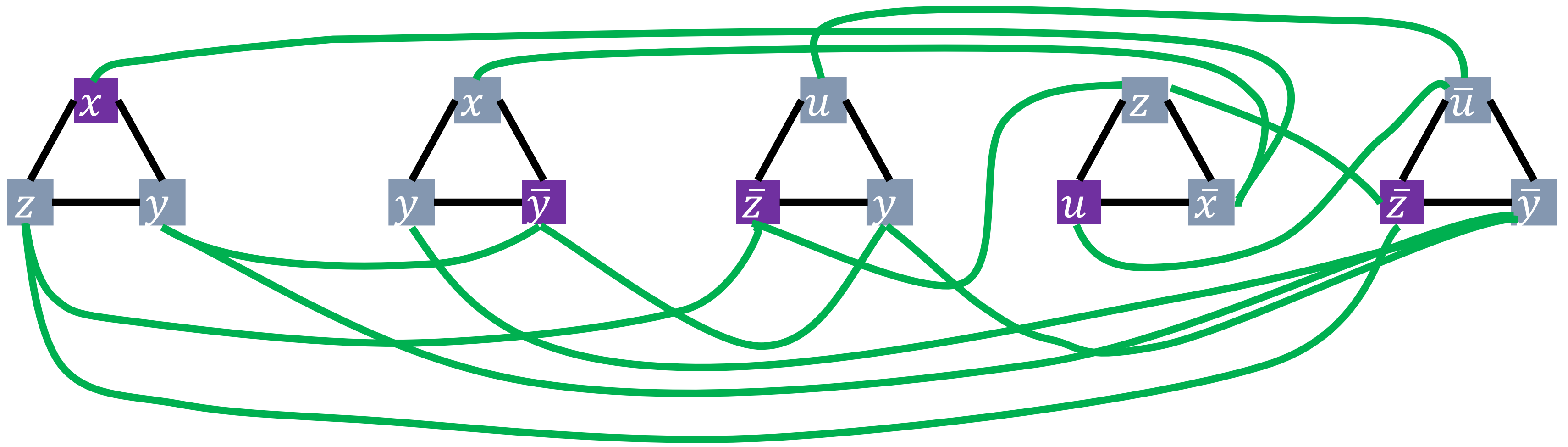
Add an edge between each node and its negation

Let $k =$ number of clauses

Claim. There is a k-independent set in this graph if and only if there is a satisfying assignment

3-SAT $\le_p$ k-Independent Set

$$(x \vee y \vee z) \wedge (x \vee \bar{y} \vee y) \wedge (u \vee y \vee \bar{z}) \wedge (z \vee \bar{x} \vee u) \wedge (\bar{x} \vee \bar{y} \vee \bar{z})$$



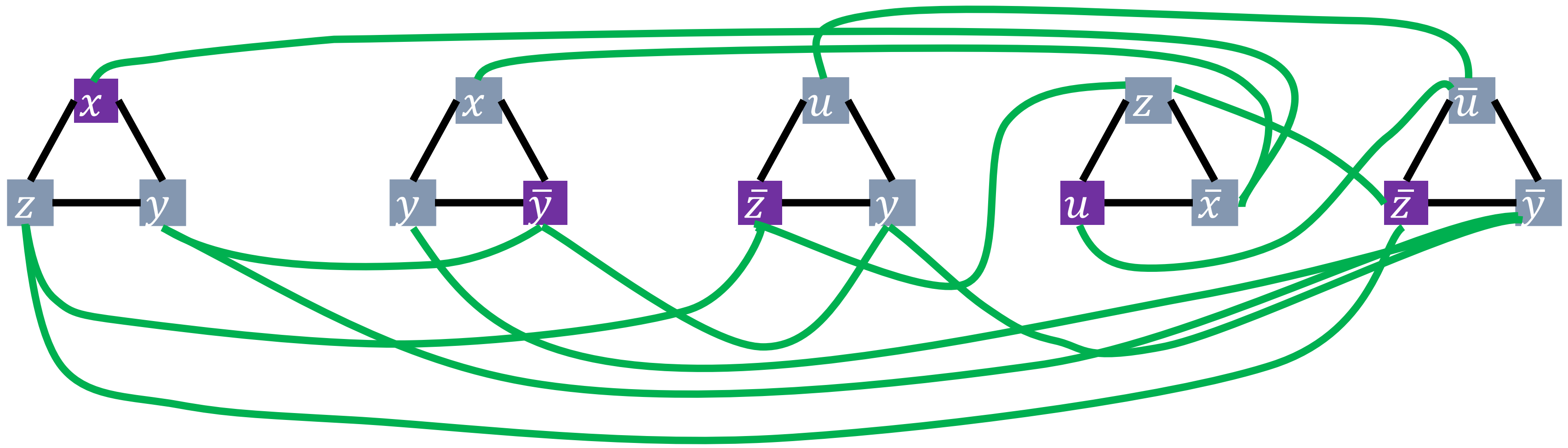
$x = \text{true}$
$y = \text{false}$
$z = \text{false}$
$u = \text{true}$

Suppose there is a k-independent set S in this graph G

- By construction of G, at most one node from each triangle is in S
- Since $|S| = k$ and there are k triangles, each triangle contributes one node
- If a variable x is selected in one triangle, then $\bar{x}$ is never selected in another triangle (since each variable is connected to its negation)
- There are no contradicting assignments, so can set variable chosen in each triangle to "true"; satisfying assignment by construction

3-SAT $\leq_p$ k-Independent Set

$$(x \lor y \lor z) \land (x \lor \bar{y} \lor y) \land (u \lor y \lor \bar{z}) \land (z \lor \bar{x} \lor u) \land (\bar{x} \lor \bar{y} \lor \bar{z})$$



$x = \text{true}$
$y = \text{false}$
$z = \text{false}$
$u = \text{true}$

Suppose there is a satisfying assignment to the formula

- At least one variable in each clause must be true
- Add the node to that variable to the set S
- There are k clauses, so set S has exactly k nodes
- If we use x in any clause, we will never use $\bar{x}$, so there are no edges among the nodes in S

3-SAT $\leq_p$ k-Independent Set

3-SAT

$(x \vee y \vee z) \wedge (x \vee \bar{y} \vee y) \wedge (u \vee y \vee \bar{z})$

$x = \text{true}$
$y = \text{false}$
$z = \text{false}$
$u = \text{true}$

Map instances of problem $\boldsymbol{A}$ to instances of $\boldsymbol{B}$

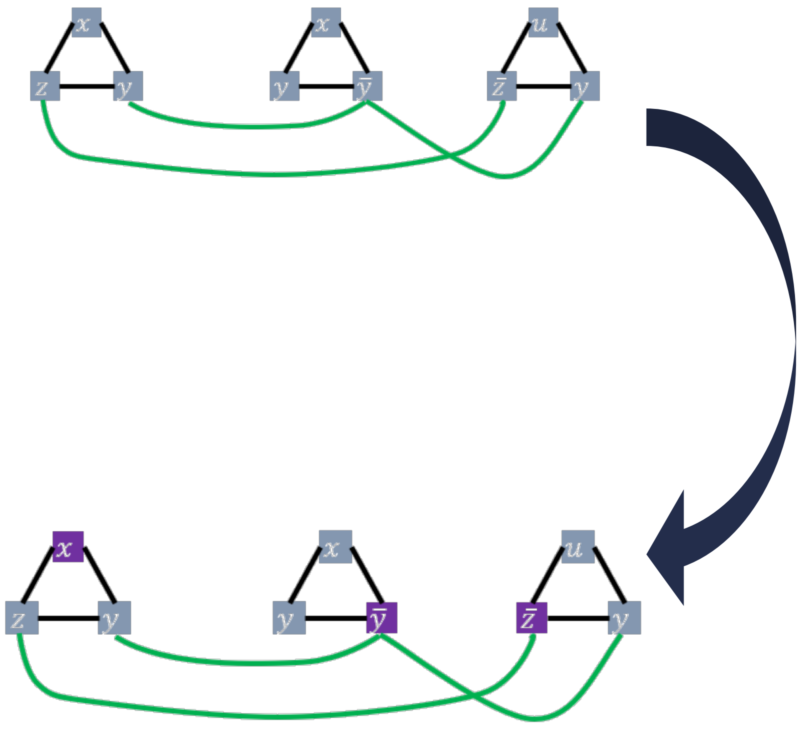
polynomial time

Map solutions of problem $\boldsymbol{B}$ to solutions of $\boldsymbol{A}$

polynomial time

polynomial-time reduction

k-independent set

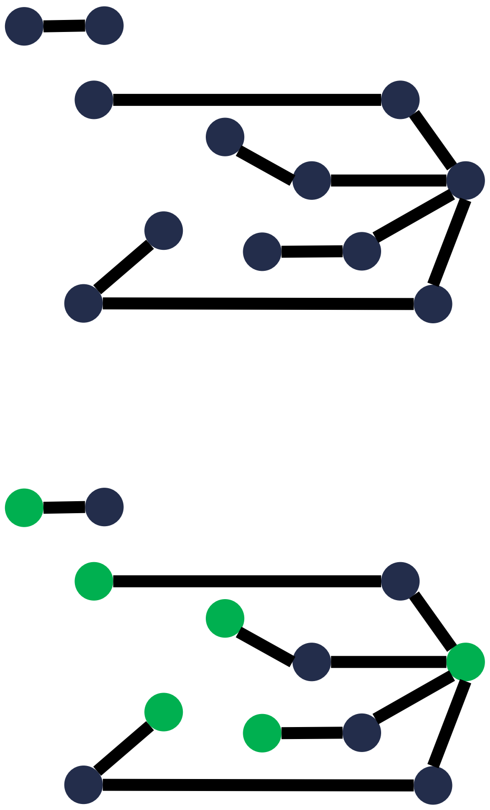


k-Independent Set is NP-Complete

1. Show that it belongs to NP ✅
2. Show it is NP-Hard ✅
   - Show 3-SAT $\leq_p$ k-independent set

Max Independent Set $\leq_p$ Min Vertex Cover

k-independent set



Map instances of problem $\boldsymbol{A}$ to instances of $\boldsymbol{B}$

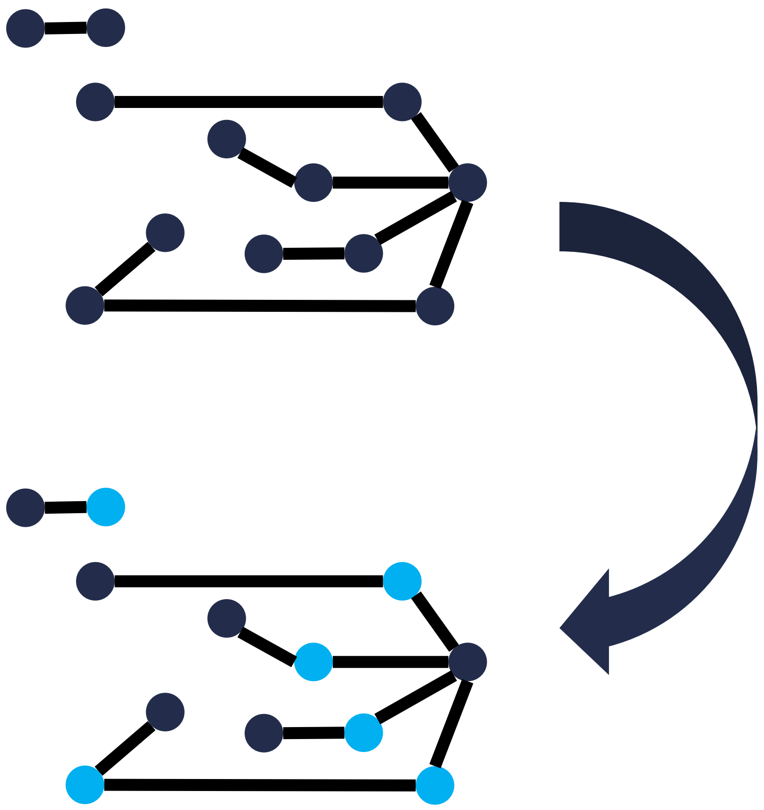
$O(1)$ time

Map solutions of problem $\boldsymbol{B}$ to solutions of $\boldsymbol{A}$

$O(|V|)$ time

Reduction

k-vertex cover



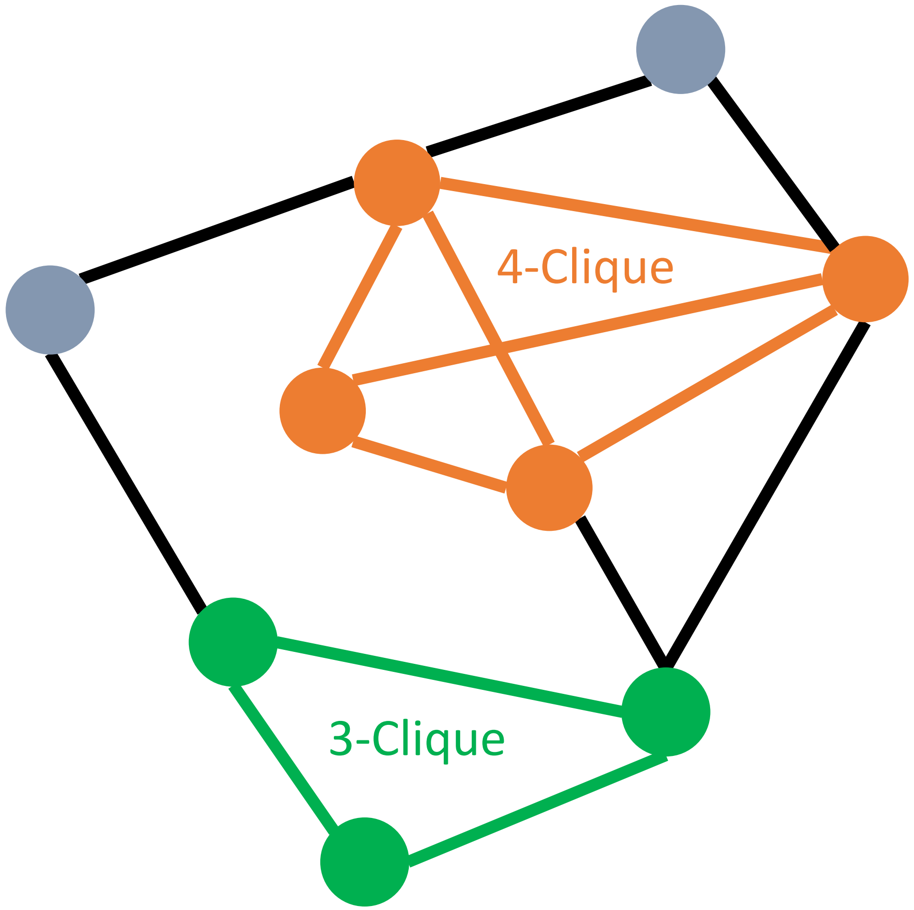
k-Vertex Cover is NP-Complete

1. Show that it belongs to NP
   - Given a candidate cover, check that every edge is covered
2. Show it is NP-Hard
   - Show k-independent set $\leq_p$ k-vertex cover

k-Clique Problem

Clique: A complete subgraph

k-Clique problem: given a graph G and a number k, is there a clique of size k?



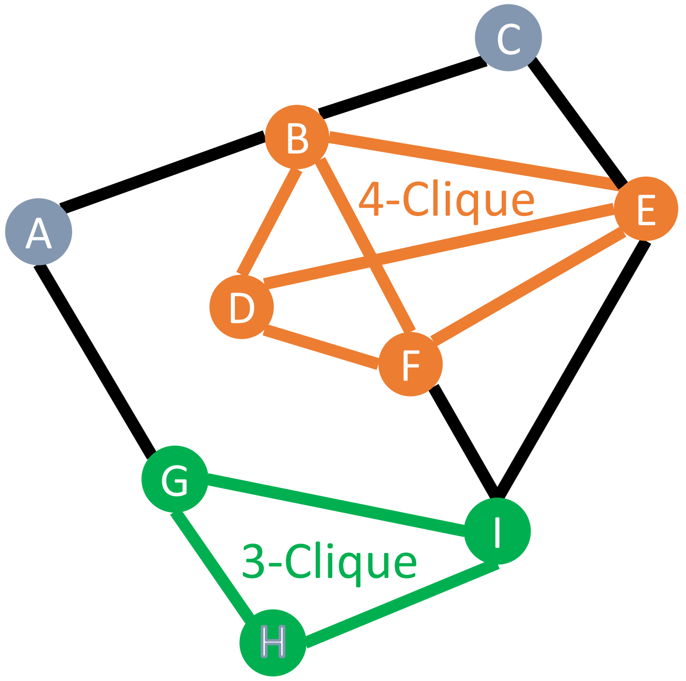
k-Clique is NP-Complete

1. Show that it belongs to NP
   - Give a polynomial time verifier
2. Show it is NP-Hard
   - Give a reduction from a known NP-Hard problem
   - We will show 3-SAT $\leq_p$ k-clique

k-Clique is in NP

Show: For any graph G:

- There is a short witness (i.e., proof) that G has a k-clique
- The proof can be checked efficiently (in polynomial time)



Graph G

Suppose $k = 4$

Witness for G: $S = \{B, D, E, F\}$
(nodes in the k-clique)

Checking the witness:

- Check that $|S| = k$ $\quad O(k) = O(|V|)$
- Check that every pair of nodes in S share an edge $\quad O(k^2) = O(|V|^2)$

Total time: $O(|V|^2) = \text{poly}(|V| + |E|)$

k-Clique is NP-Complete

1. Show that it belongs to NP
    - Give a polynomial time verifier
2. Show it is NP-Hard
    - Give a reduction from a known NP-Hard problem
    - We will show 3-SAT $\leq_p$ k-clique

3-SAT $\leq_p$ k-Clique

3-SAT

$(x \vee y \vee z) \wedge (x \vee \bar{y} \vee y) \wedge (u \vee y \vee \bar{z})$

$x = \text{true}$
$y = \text{false}$
$z = \text{false}$
$u = \text{true}$

Map instances of problem $\boldsymbol{A}$ to instances of $\boldsymbol{B}$

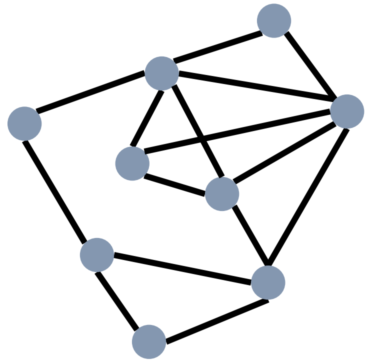
polynomial time

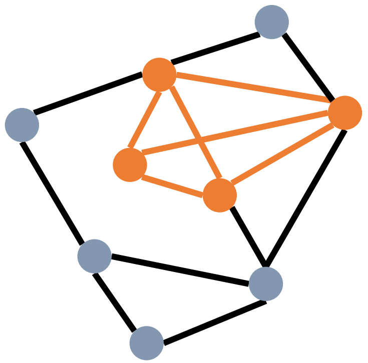
Map solutions of problem $\boldsymbol{B}$ to solutions of $\boldsymbol{A}$

polynomial time

polynomial-time reduction

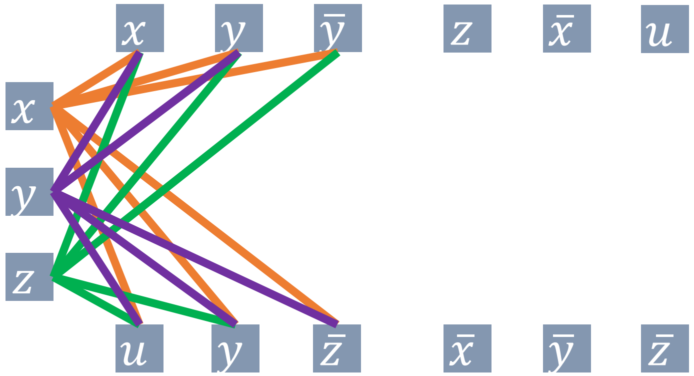
k-clique





3-SAT $\leq_p$ k-Clique

$$(x \lor y \lor z) \land (x \lor \bar{y} \lor y) \land (u \lor y \lor \bar{z}) \land (z \lor \bar{x} \lor u) \land (\bar{x} \lor \bar{y} \lor \bar{z})$$



(also do this for the other clauses, omitted due to clutter)

For each clause, introduce a node for each of its three variables

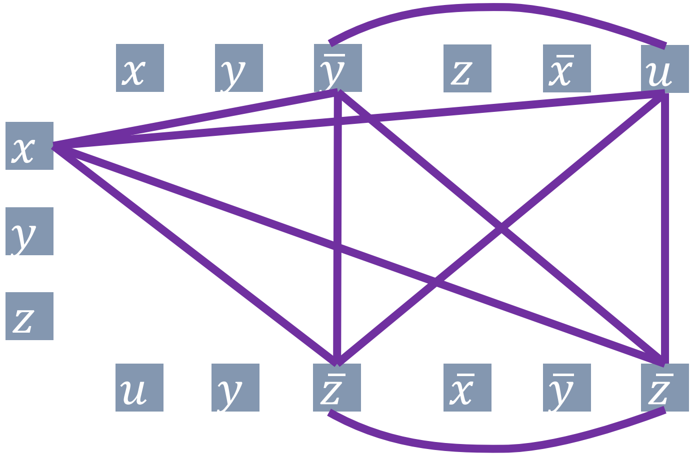
Add an edge from each node to all non-contradictory nodes in the other clauses (i.e., to all nodes that is not the negation of its own variable)

Let k = number of clauses

Claim. There is a k-clique in this graph if and only if there is a satisfying assignment

3-SAT $\leq_p$ k-Clique

$$(x \lor y \lor z) \land (x \lor \bar{y} \lor y) \land (u \lor y \lor \bar{z}) \land (z \lor \bar{x} \lor u) \land (\bar{x} \lor \bar{y} \lor \bar{z})$$

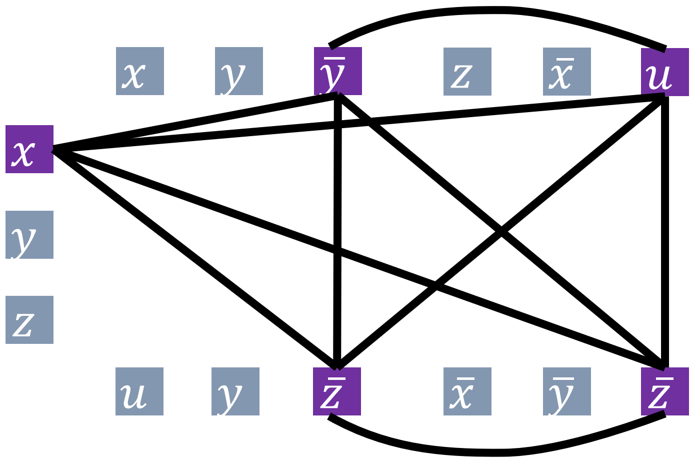


Suppose there is a k-clique in this graph

- There are no edges between nodes for variables in the same clause, so k-clique must contain one node from each clause
- Nodes in clique cannot contain variable and its negation
- Nodes in clique must then correspond to a satisfying assignment

3-SAT $\leq_p$ k-Clique

$$(x \vee y \vee z) \wedge (x \vee \bar{y} \vee y) \wedge (u \vee y \vee \bar{z}) \wedge (z \vee \bar{x} \vee u) \wedge (\bar{x} \vee \bar{y} \vee \bar{z})$$



Suppose there is a satisfying assignment to the formula

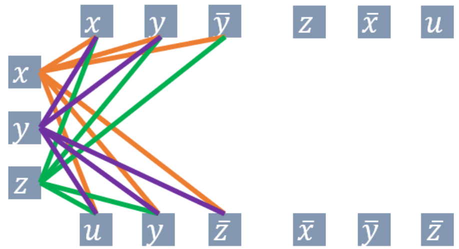
- For each clause, choose one node whose value is true
- There are k clauses, so this yields a collection of k nodes
- Since the assignment is consistent, there is an edge between every pair of nodes, so this constitutes a k-clique

3-SAT $\leq_p$ k-Clique

3-SAT

$(x \vee y \vee z) \wedge (x \vee \bar{y} \vee y) \wedge (u \vee y \vee \bar{z})$

k-clique



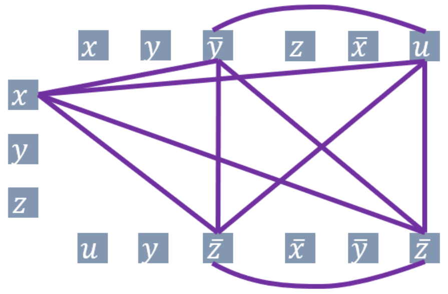
Map instances of problem
$\boldsymbol{A}$ to instances of $\boldsymbol{B}$

polynomial time

Map solutions of problem
$\boldsymbol{B}$ to solutions of $\boldsymbol{A}$

polynomial time

$x = \text{true}$
$y = \text{false}$
$z = \text{false}$
$u = \text{true}$



polynomial-time reduction

k-Clique is NP-Complete

1. Show that it belongs to NP ✅
    - Give a polynomial time verifier
2. Show it is NP-Hard ✅
    - Give a reduction from a known NP-Hard problem
    - We will show 3-SAT $\leq_p$ k-clique

Bonus Material: Coping with NP-Hardness

Material from subsequent slides will not be on the exam

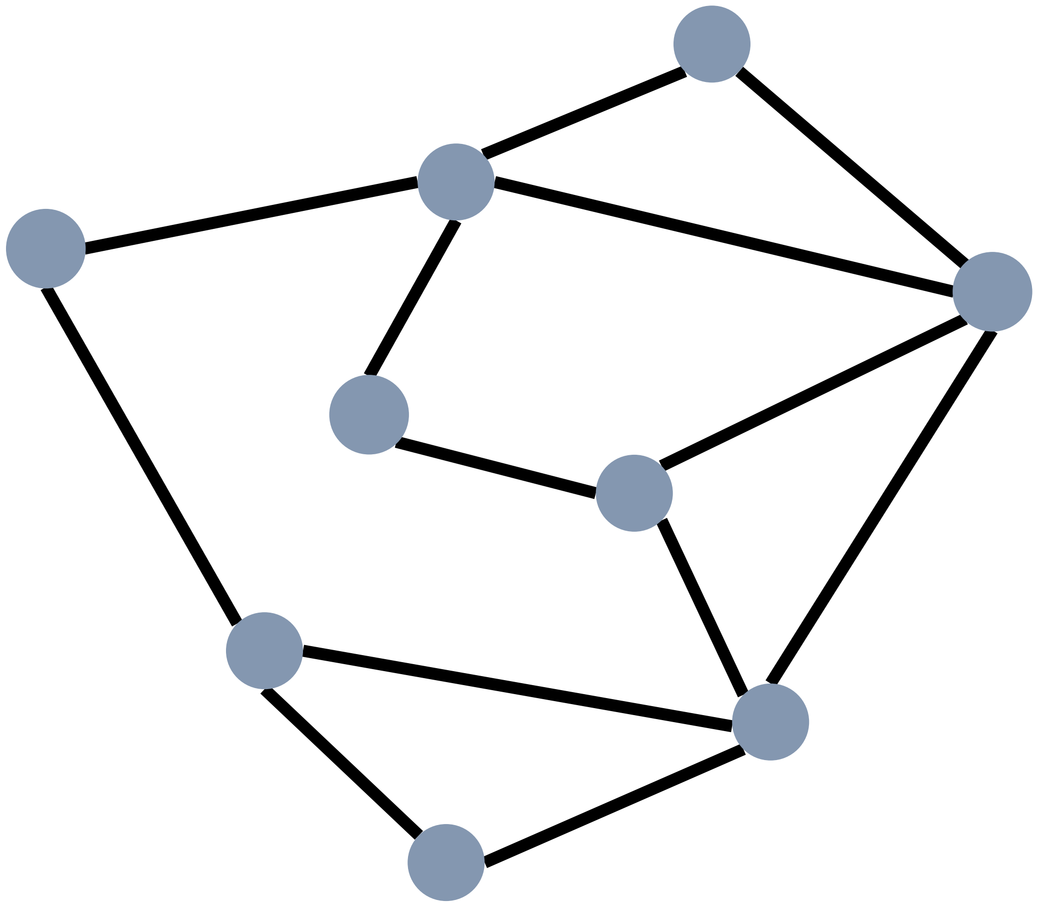
Coping with NP-Hardness

Many optimization problems that come up in practice are NP-complete

What do we do?

Approach 1: Find an algorithm that gives nearly-optimal solutions

Greedy Vertex Cover

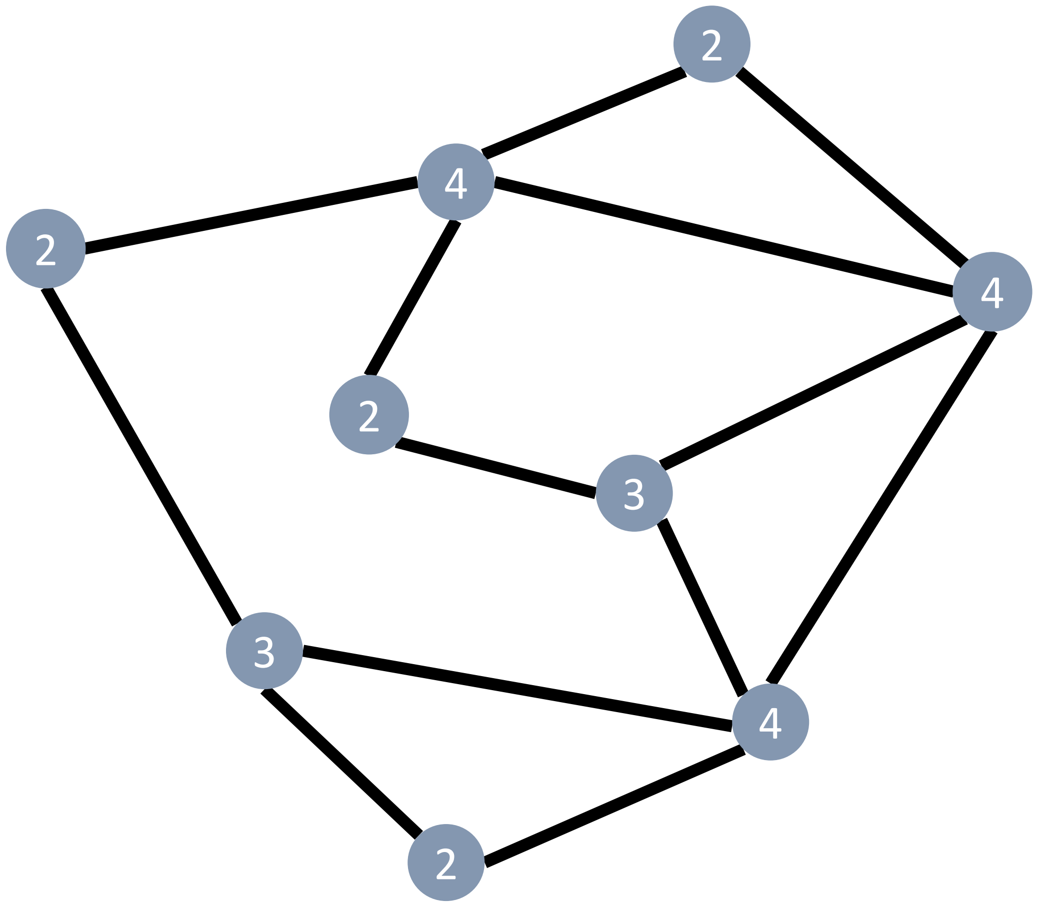


Goal: Find a set of nodes such that every edge is incident on one of the nodes

Greedy approach?

Greedy choice: Node with highest degree (e.g., node that covers the most edges)

Greedy Vertex Cover

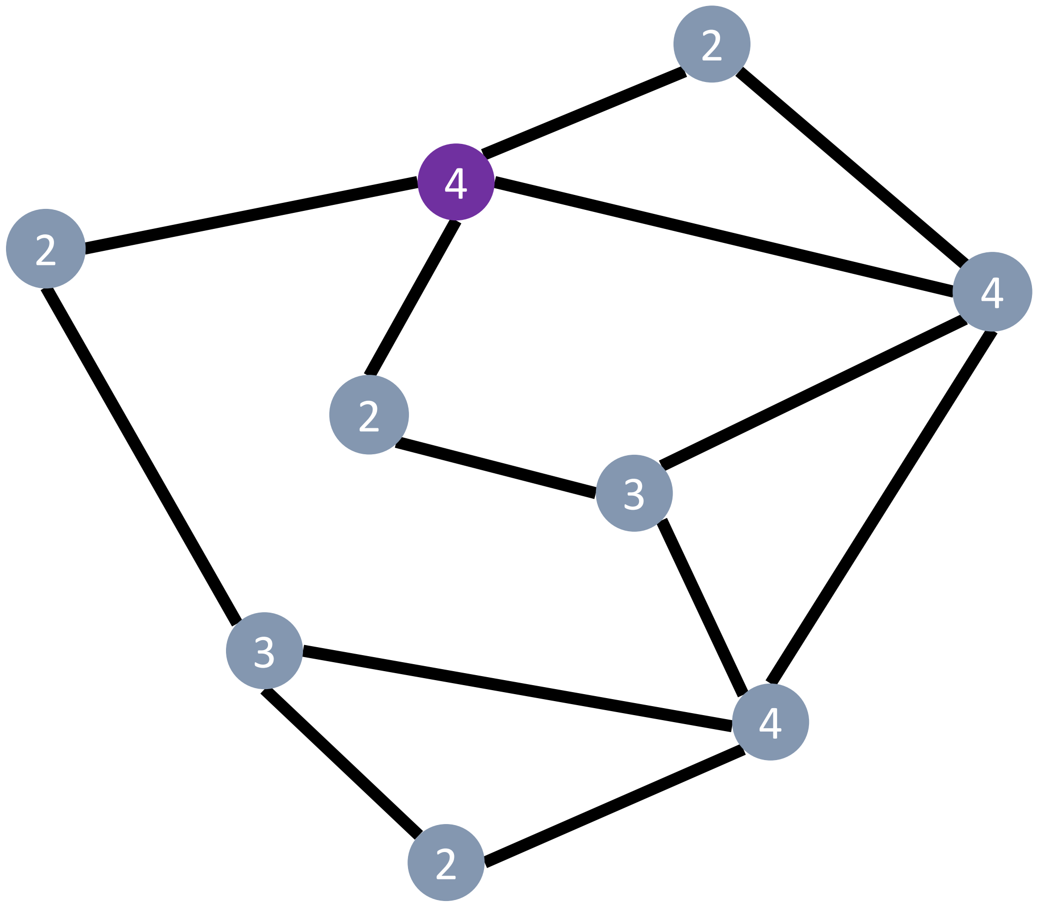


Goal: Find a set of nodes such that every edge is incident on one of the nodes

Greedy approach?

Greedy choice: Node with highest degree (e.g., node that covers the most edges)

Greedy Vertex Cover

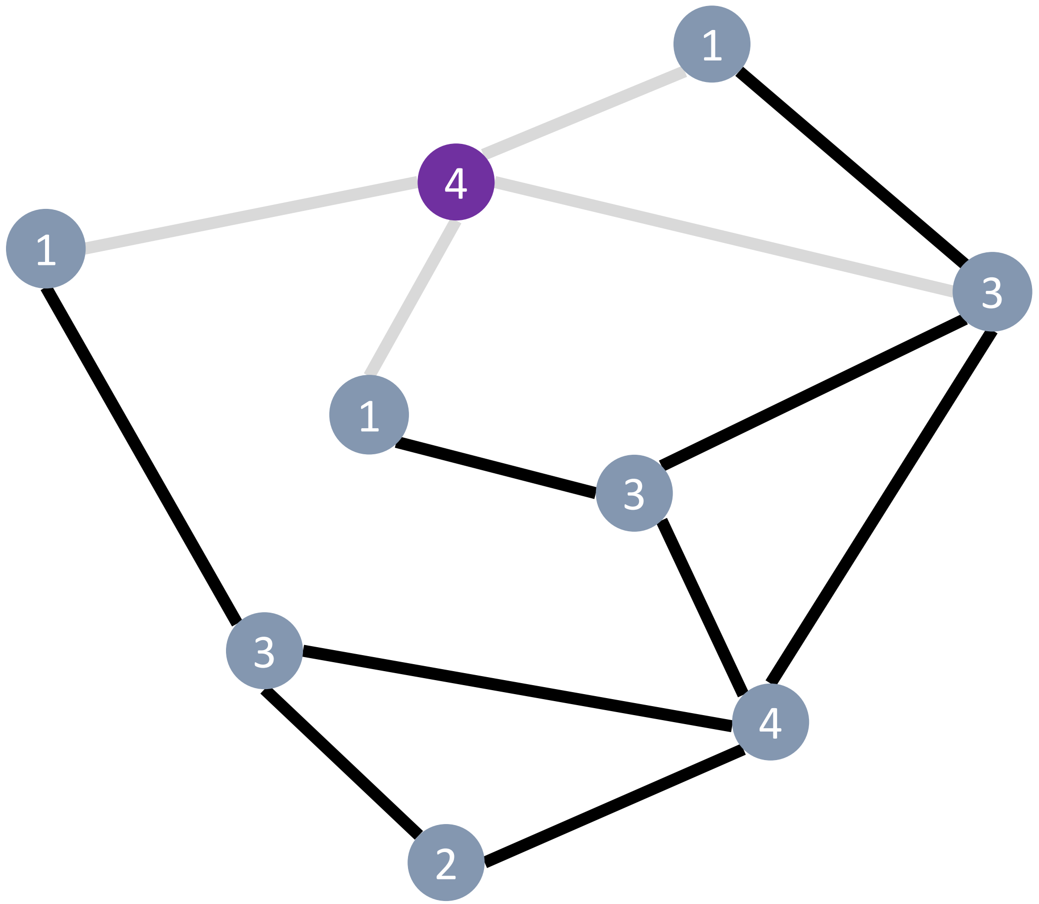


Goal: Find a set of nodes such that every edge is incident on one of the nodes

Greedy approach?

Greedy choice: Node with highest degree (e.g., node that covers the most edges)

Greedy Vertex Cover

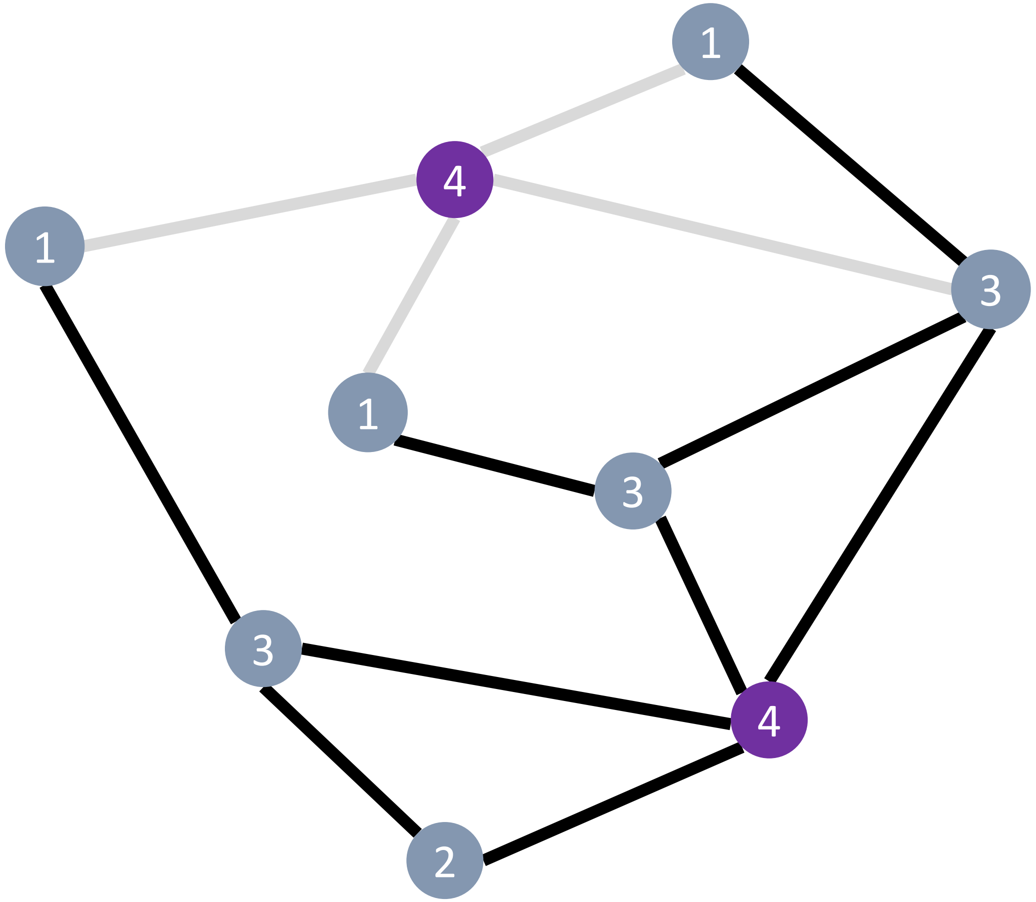


Goal: Find a set of nodes such that every edge is incident on one of the nodes

Greedy approach?

Greedy choice: Node with highest degree (e.g., node that covers the most edges)

Greedy Vertex Cover

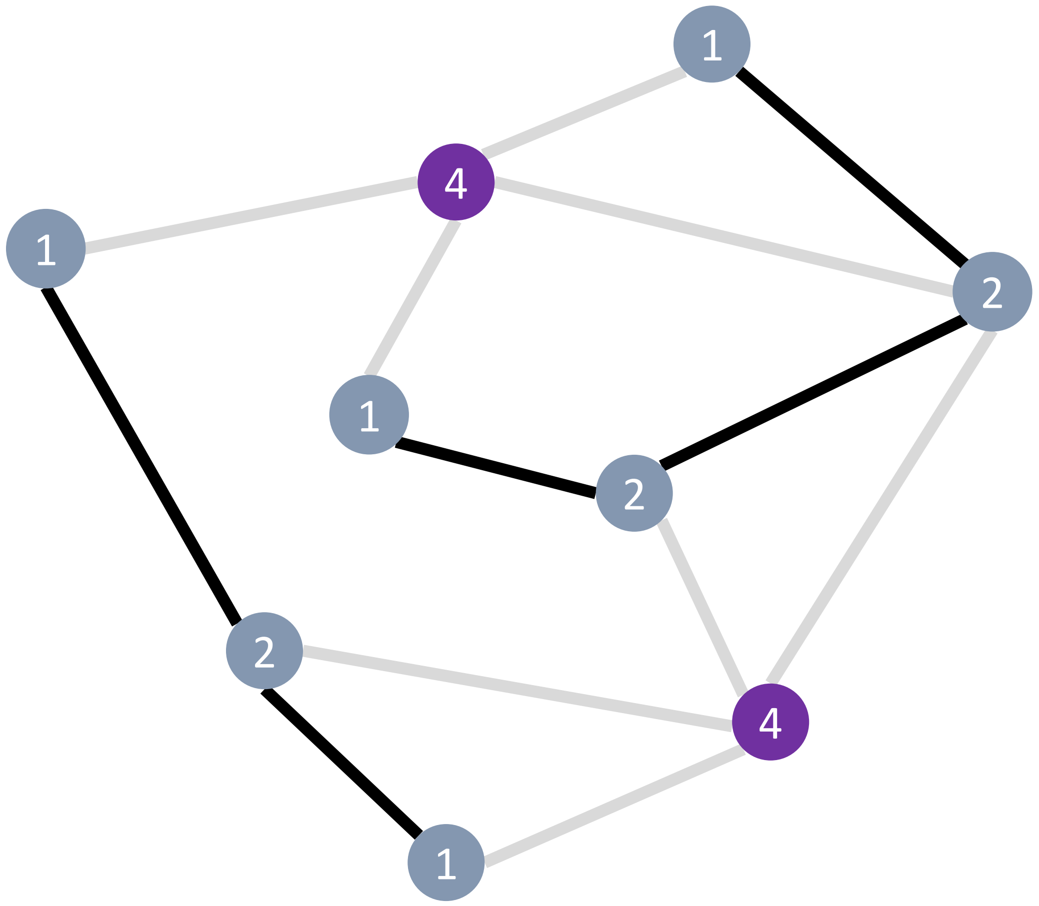


Goal: Find a set of nodes such that every edge is incident on one of the nodes

Greedy approach?

Greedy choice: Node with highest degree (e.g., node that covers the most edges)

Greedy Vertex Cover

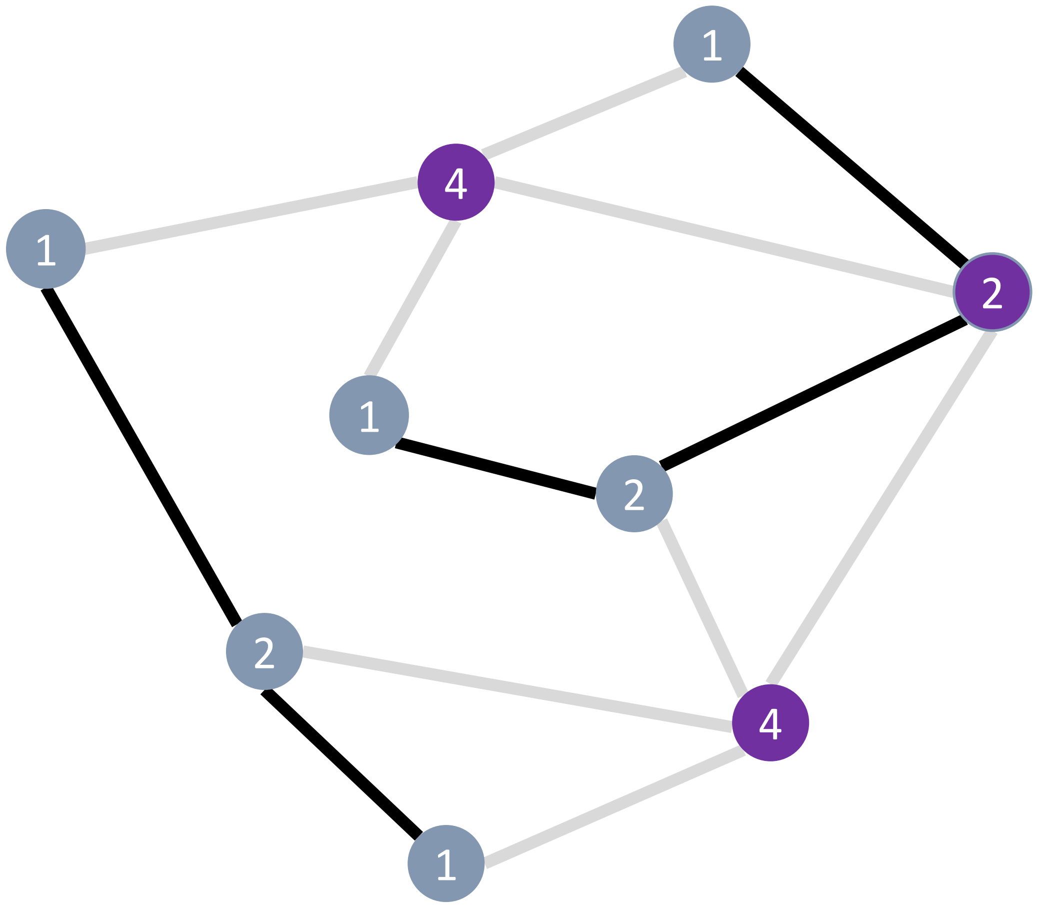


Goal: Find a set of nodes such that every edge is incident on one of the nodes

Greedy approach?

Greedy choice: Node with highest degree (e.g., node that covers the most edges)

Greedy Vertex Cover

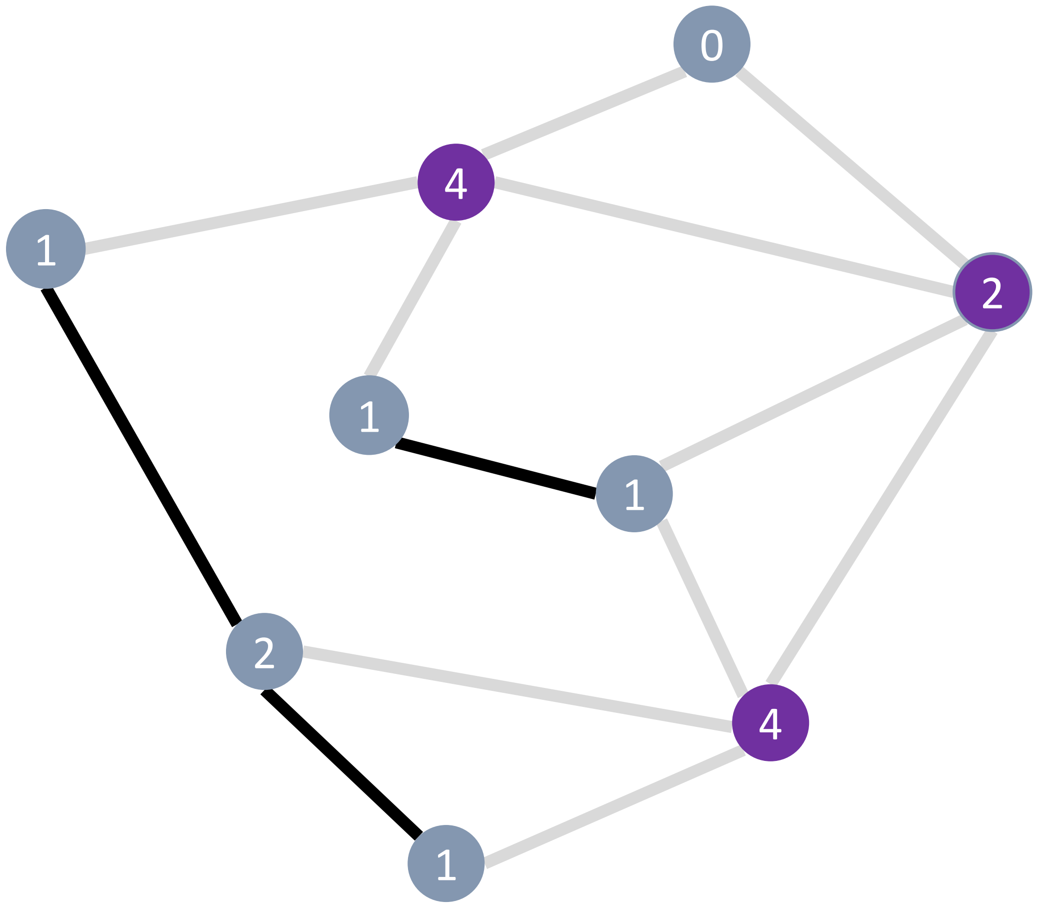


Goal: Find a set of nodes such that every edge is incident on one of the nodes

Greedy approach?

Greedy choice: Node with highest degree (e.g., node that covers the most edges)

Greedy Vertex Cover

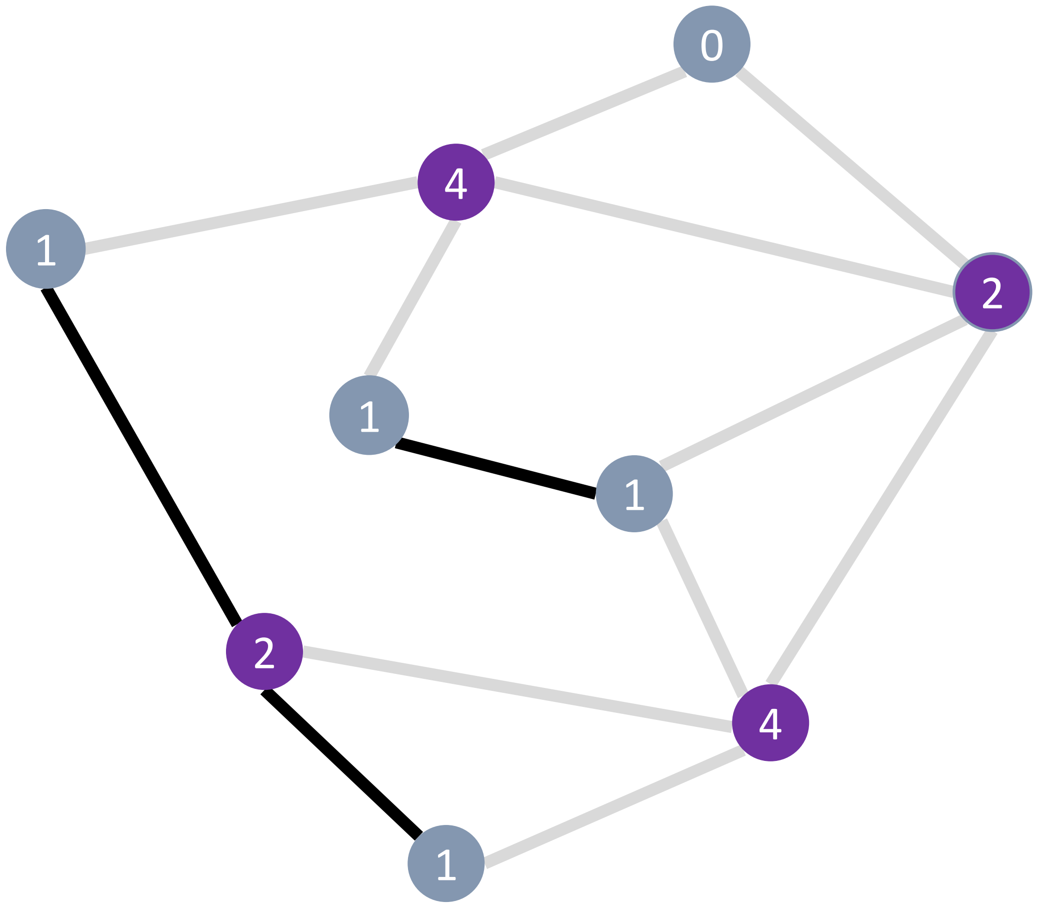


Goal: Find a set of nodes such that every edge is incident on one of the nodes

Greedy approach?

Greedy choice: Node with highest degree (e.g., node that covers the most edges)

Greedy Vertex Cover

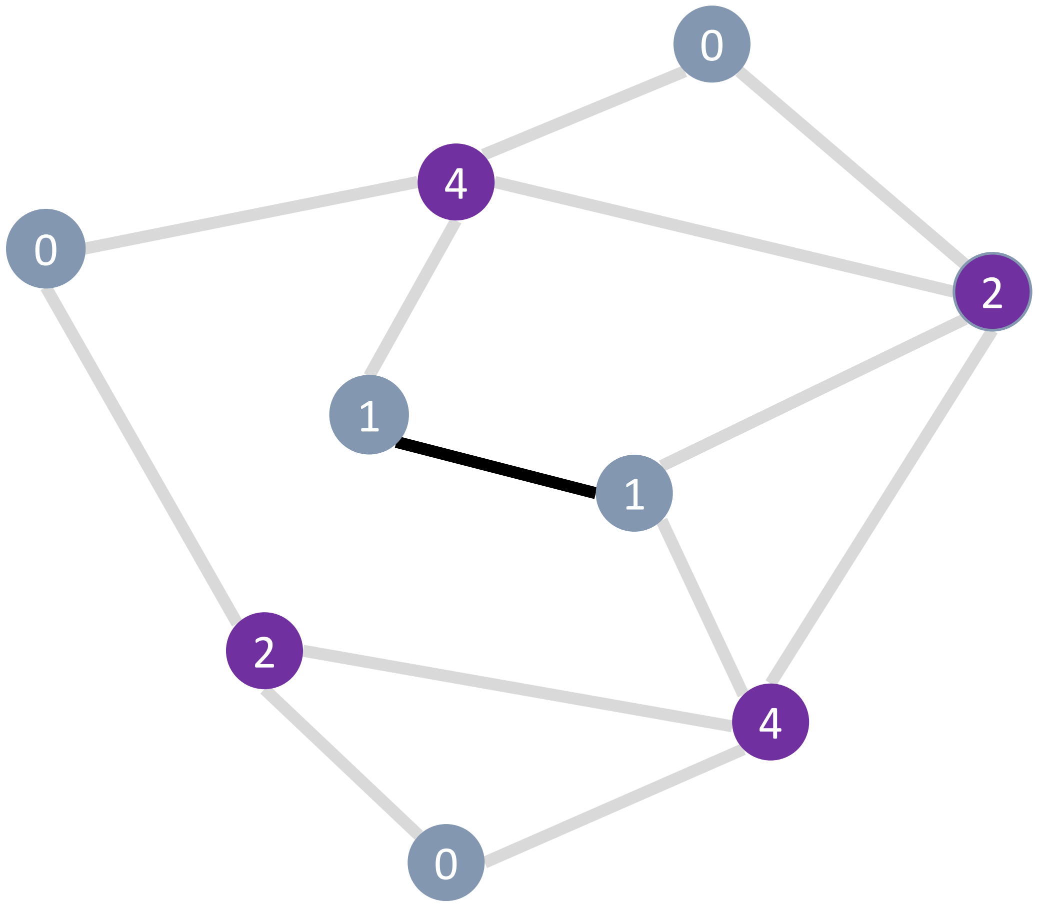


Goal: Find a set of nodes such that every edge is incident on one of the nodes

Greedy approach?

Greedy choice: Node with highest degree (e.g., node that covers the most edges)

Greedy Vertex Cover

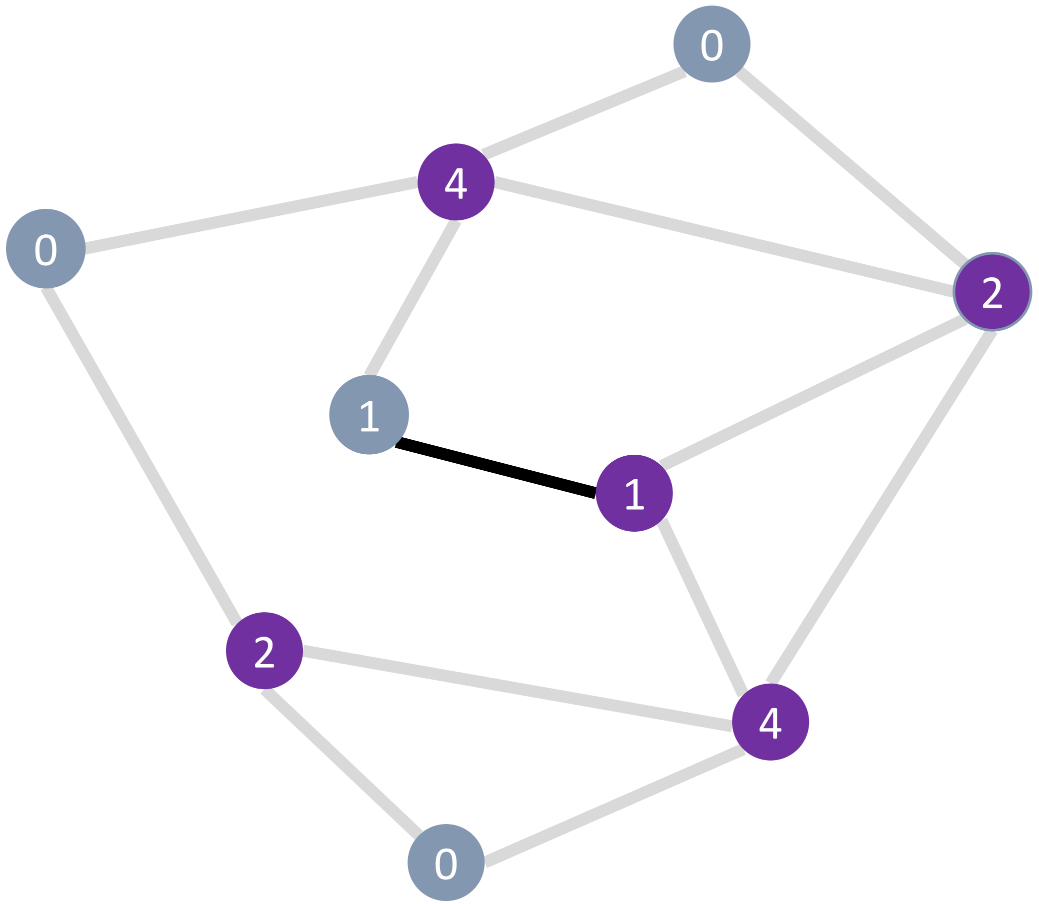


Goal: Find a set of nodes such that every edge is incident on one of the nodes

Greedy approach?

Greedy choice: Node with highest degree (e.g., node that covers the most edges)

Greedy Vertex Cover

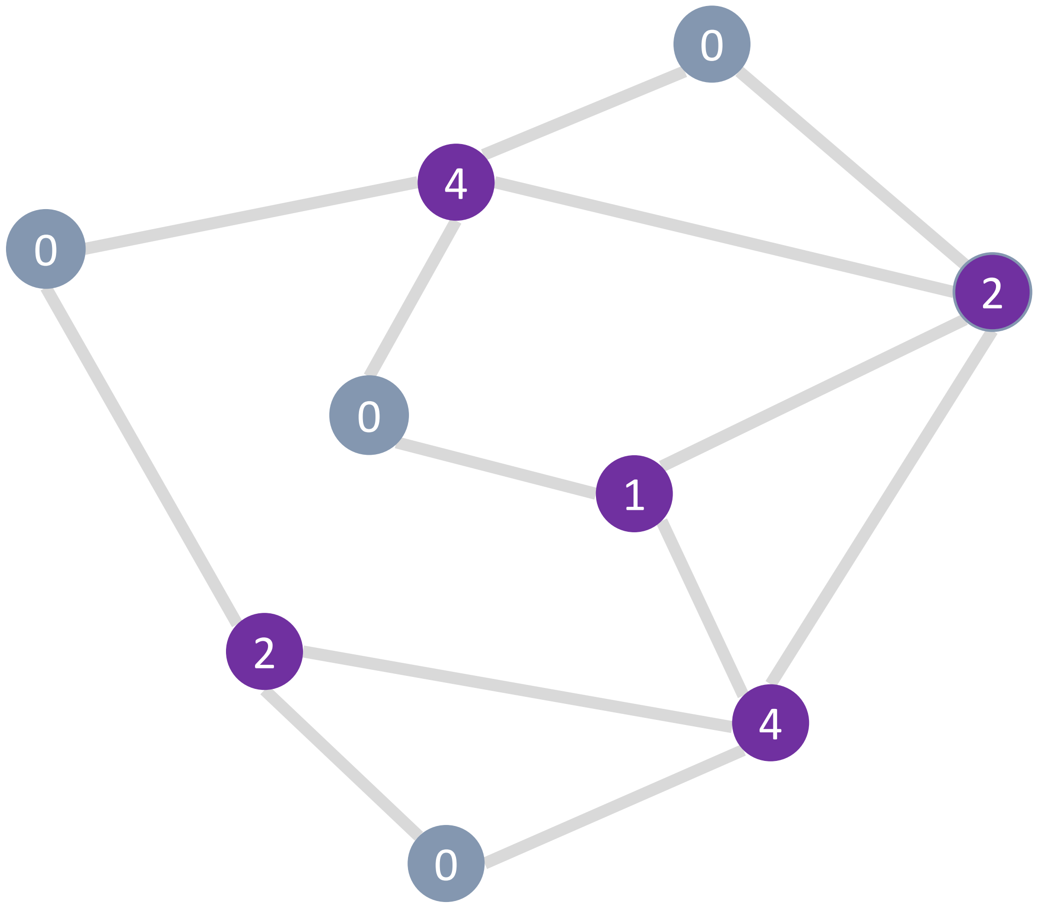


Goal: Find a set of nodes such that every edge is incident on one of the nodes

Greedy approach?

Greedy choice: Node with highest degree (e.g., node that covers the most edges)

Greedy Vertex Cover



Goal: Find a set of nodes such that every edge is incident on one of the nodes

Greedy approach?

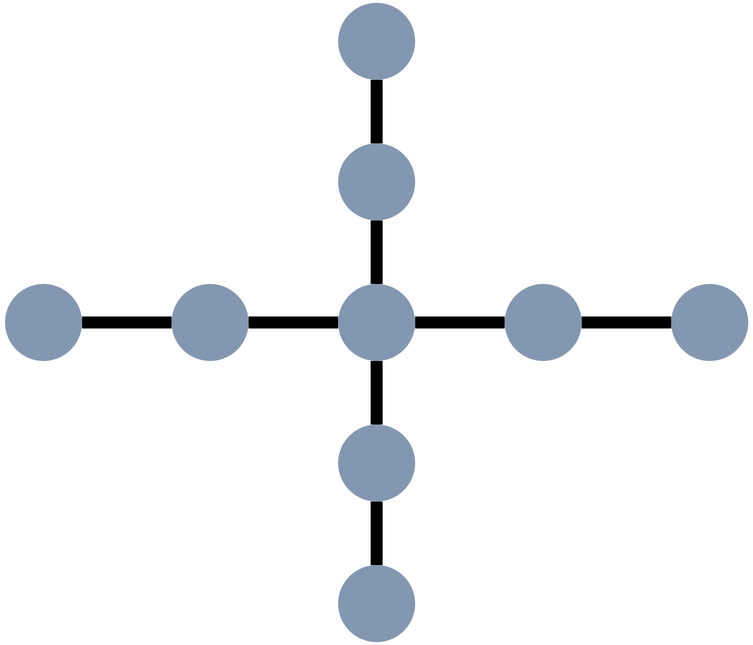
Greedy choice: Node with highest degree (e.g., node that covers the most edges)

Size of vertex cover: 5

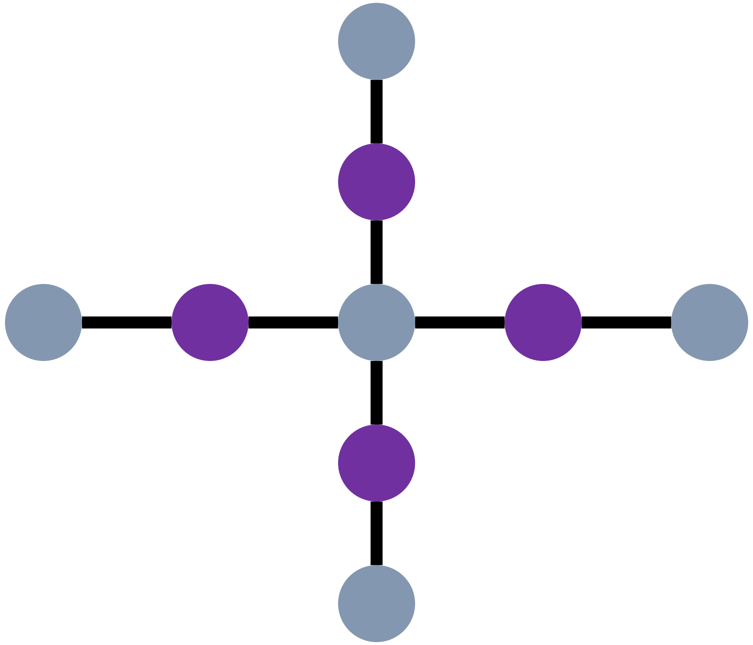
In this case, actually optimal!

Greedy Vertex Cover

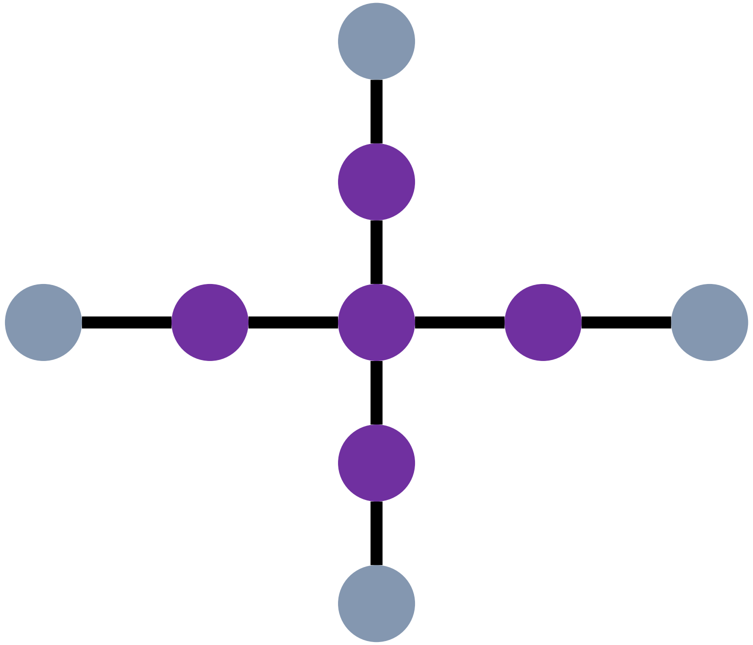
But not always optimal…



Graph G



Optimal



Greedy

Greedy Vertex Cover

But is it “good enough?”

How do we measure good enough?

Let $\mathrm{OPT}(G)$ denote the size of the minimum vertex cover in G and $|A(G)|$ be the size of the cover output by algorithm A

Define the approximation factor of A to be

$$\mathrm{ApproxFactor}(A) = \frac{|A(G)|}{\mathrm{OPT}}$$

The larger this value is, the worse the quality of the approximation (**Goal:** as close to 1 as possible)

Greedy Vertex Cover

But is it “good enough?”

How do we measure good enough?

Let $\mathrm{OPT}(G)$ denote the size of the minimum vertex cover in G and $|A(G)|$ be the size of the cover output by algorithm A

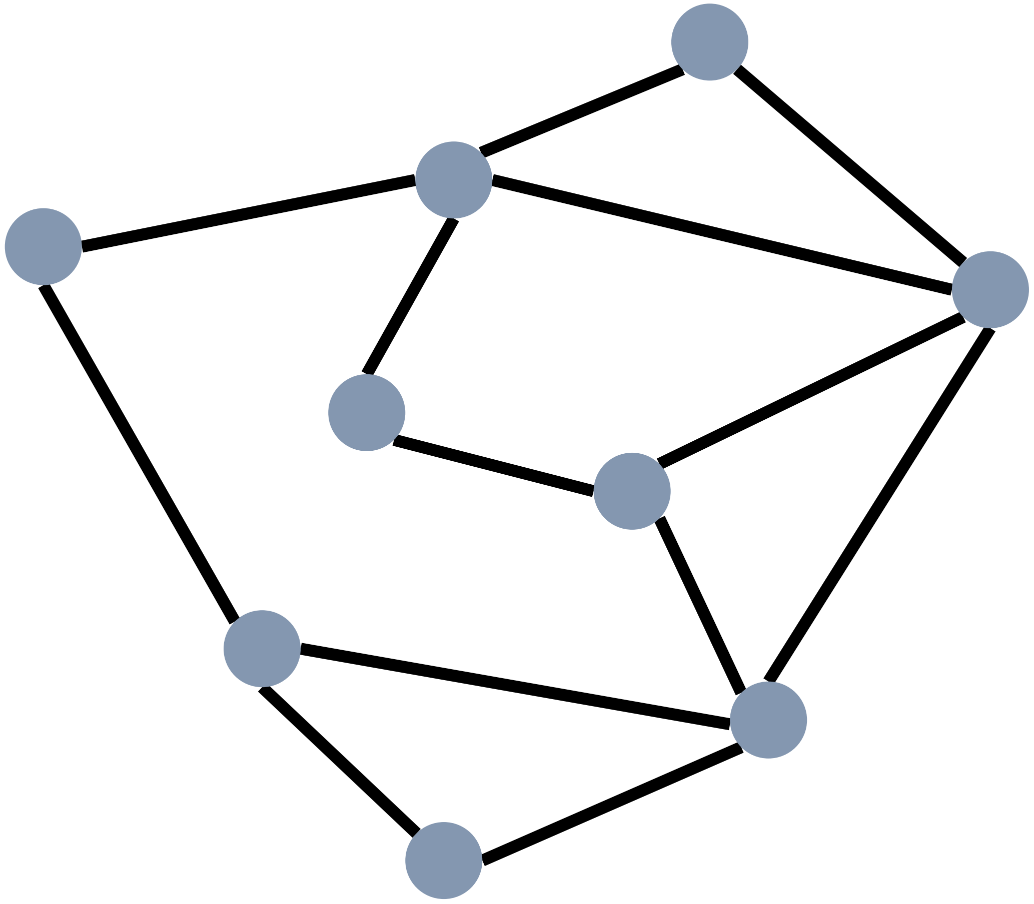
Define the approximation factor of A to be

$$\mathrm{ApproxFactor}(A) = \frac{|A(G)|}{\mathrm{OPT}}$$

Theorem. The greedy algorithm for vertex cover achieves an approximation factor of $\Omega(\log|V|)$

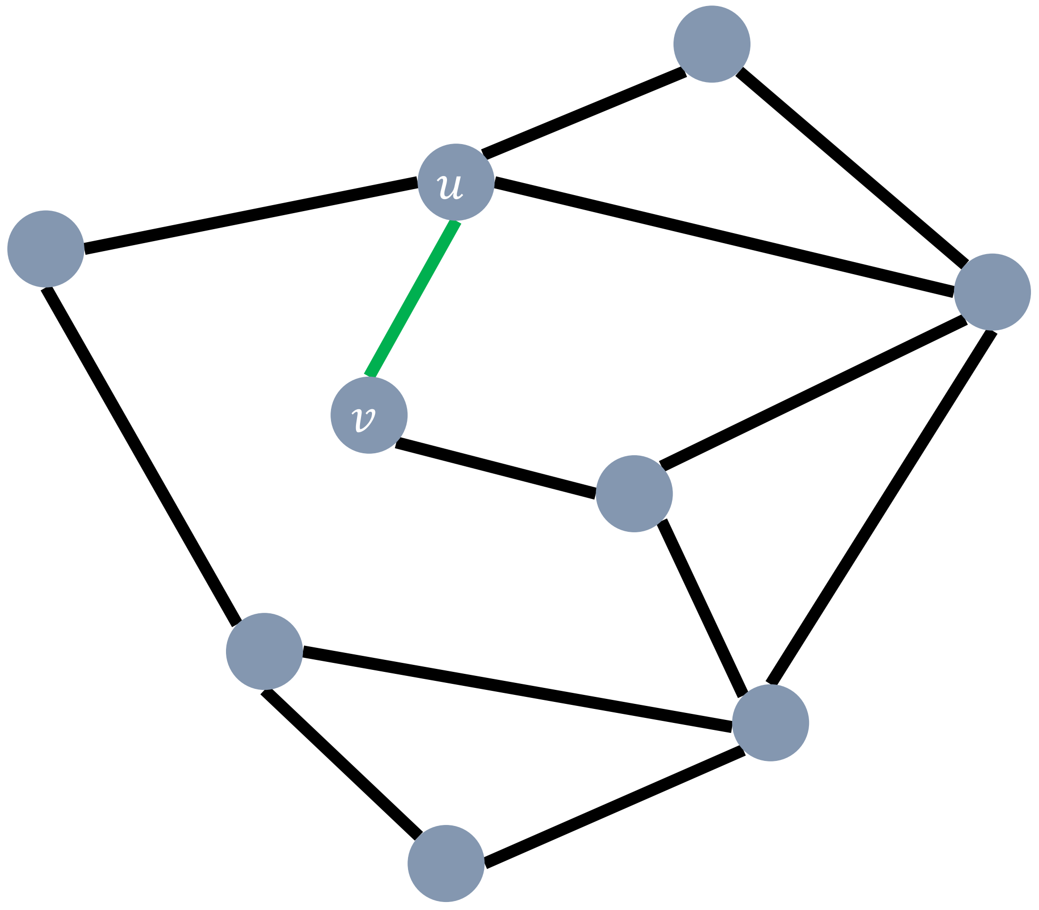
Not that great… quality of solution is worse for large instances

Approximate Vertex Cover



Goal: Obtain a <u>2-approximation</u> (i.e., vertex cover that is at most twice as large as the optimal)

Approximate Vertex Cover

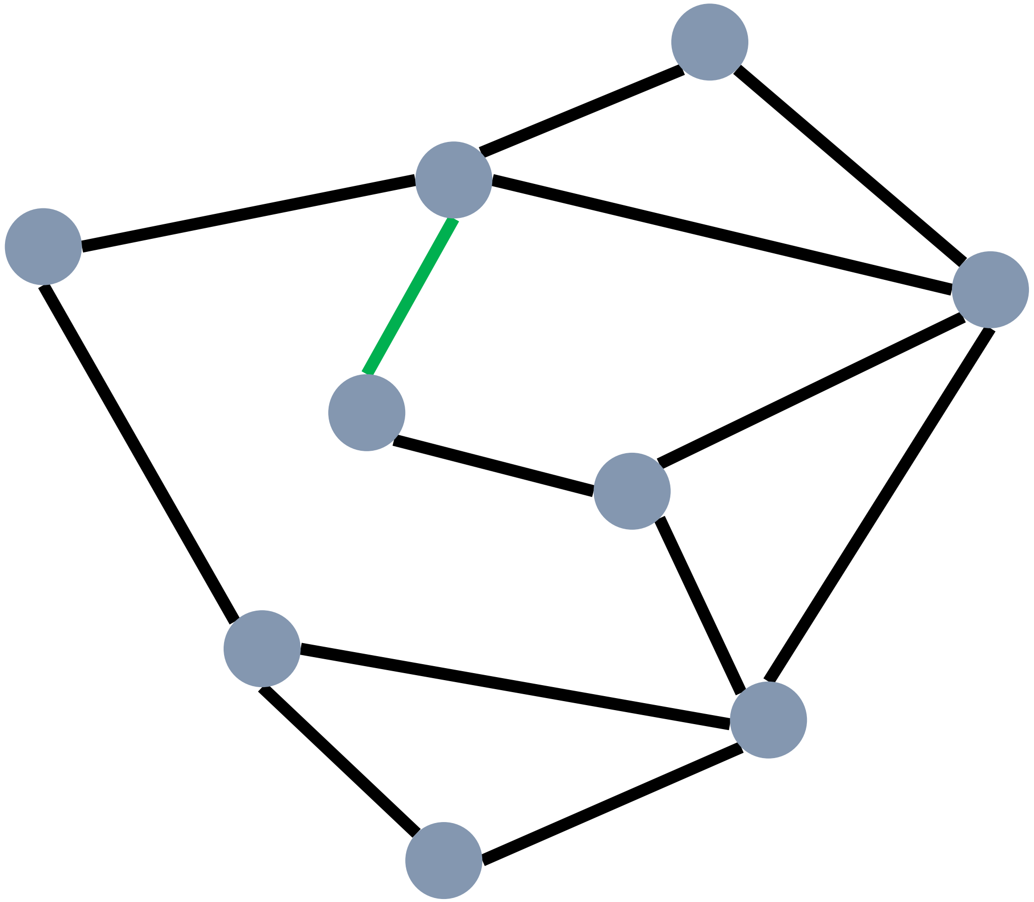


Goal: Obtain a <u>2-approximation</u> (i.e., vertex cover that is at most twice as large as the optimal)

Consider an edge $e = (u, v) \in E$

- Optimal vertex covering must contain either u or v
- **Our approach:** take <u>both</u> of them!
  - Add u, v to cover
  - Remove all edges incident on u and v
  - Repeat until no edges remain

Approximate Vertex Cover

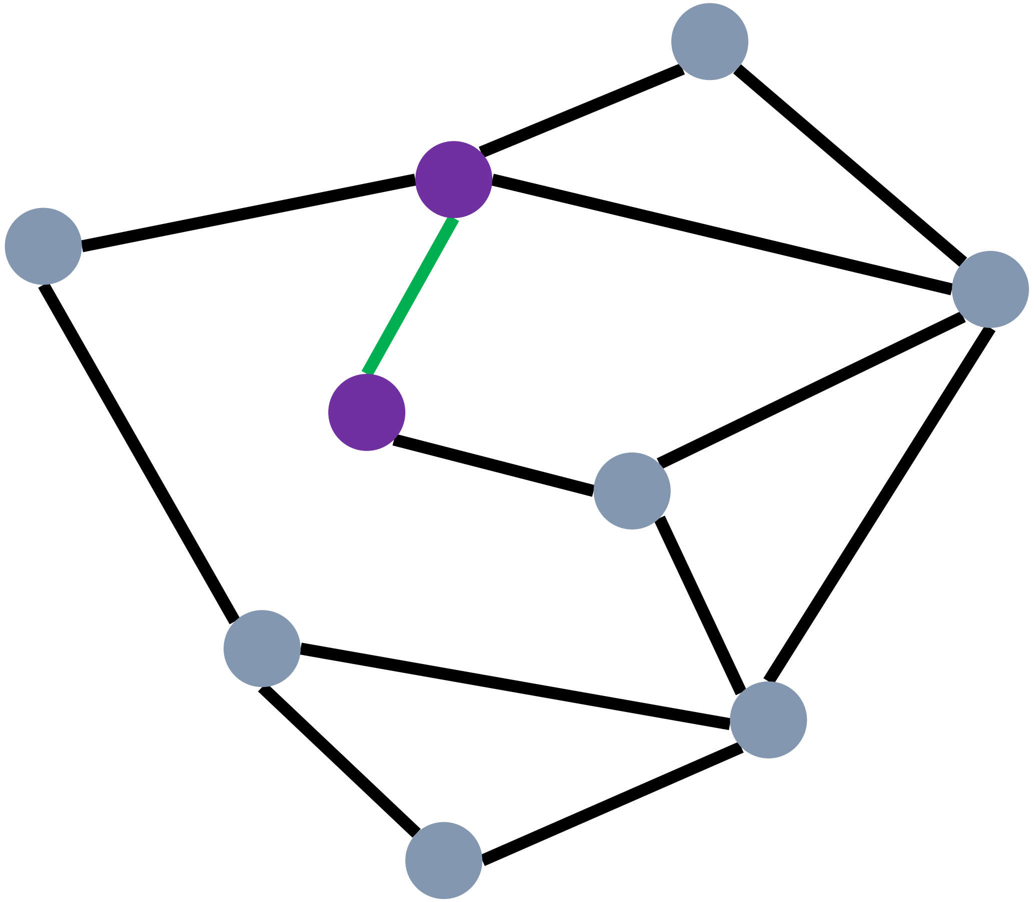


Goal: Obtain a 2-approximation (i.e., vertex cover that is at most twice as large as the optimal)

Consider an edge $e = (u, v) \in E$

- Optimal vertex covering must contain either u or v
- **Our approach:** take both of them!
  - Add u, v to cover
  - Remove all edges incident on u and v
  - Repeat until no edges remain

Approximate Vertex Cover

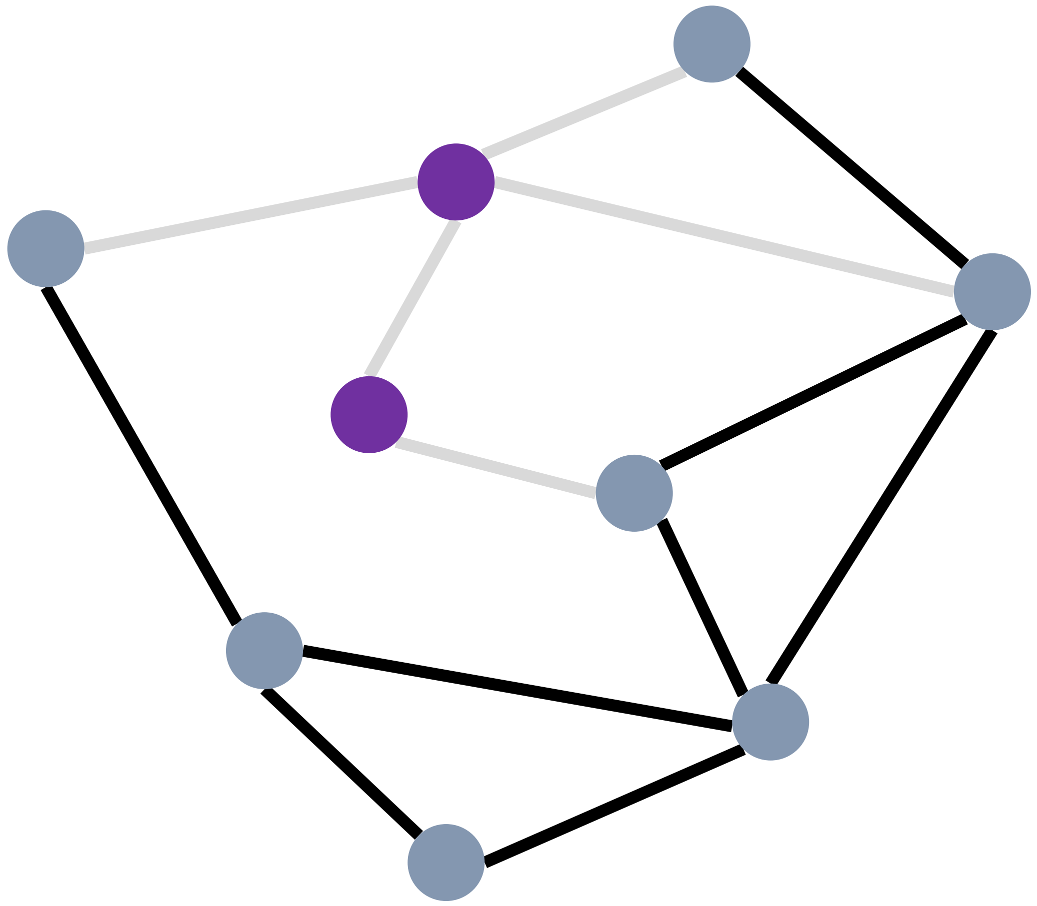


Goal: Obtain a 2-approximation (i.e., vertex cover that is at most twice as large as the optimal)

Consider an edge $e = (u, v) \in E$

- Optimal vertex covering must contain either u or v
- **Our approach:** take both of them!
  - Add u, v to cover
  - Remove all edges incident on u and v
  - Repeat until no edges remain

Approximate Vertex Cover

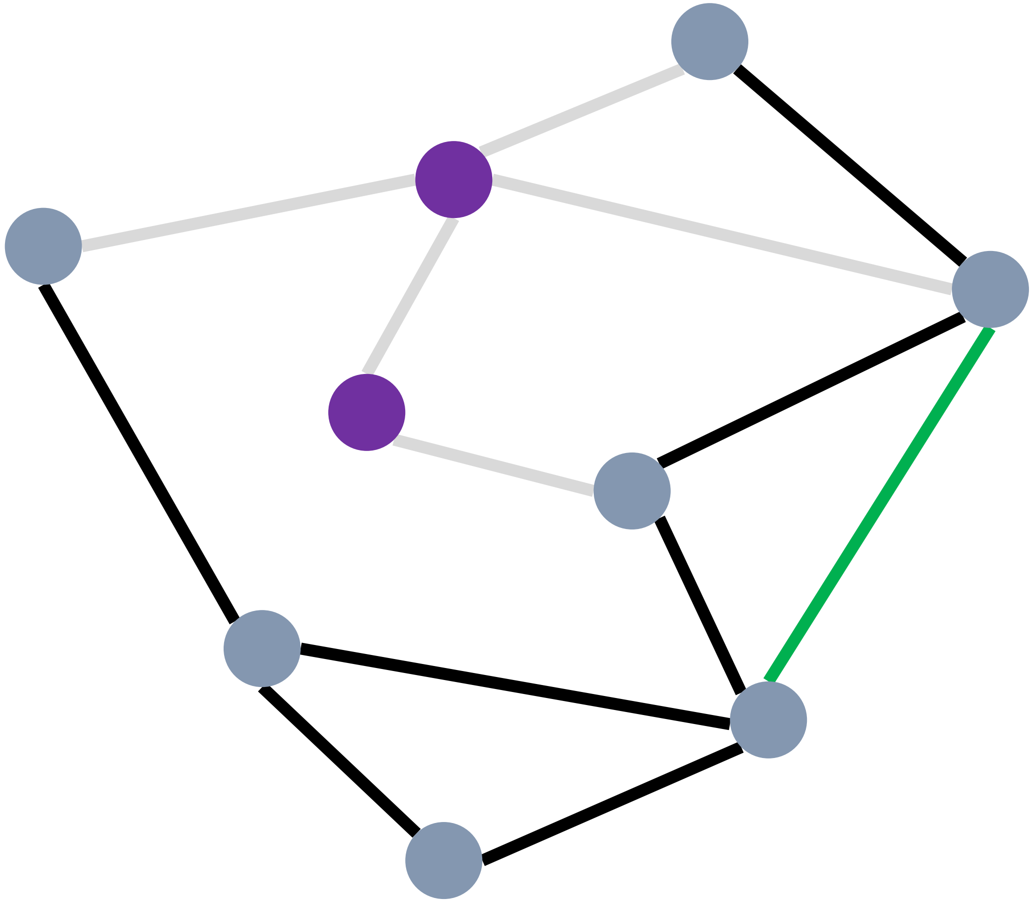


Goal: Obtain a 2-approximation (i.e., vertex cover that is at most twice as large as the optimal)

Consider an edge $e = (u, v) \in E$

- Optimal vertex covering must contain either u or v
- **Our approach:** take both of them!
  - Add u, v to cover
  - Remove all edges incident on u and v
  - Repeat until no edges remain

Approximate Vertex Cover

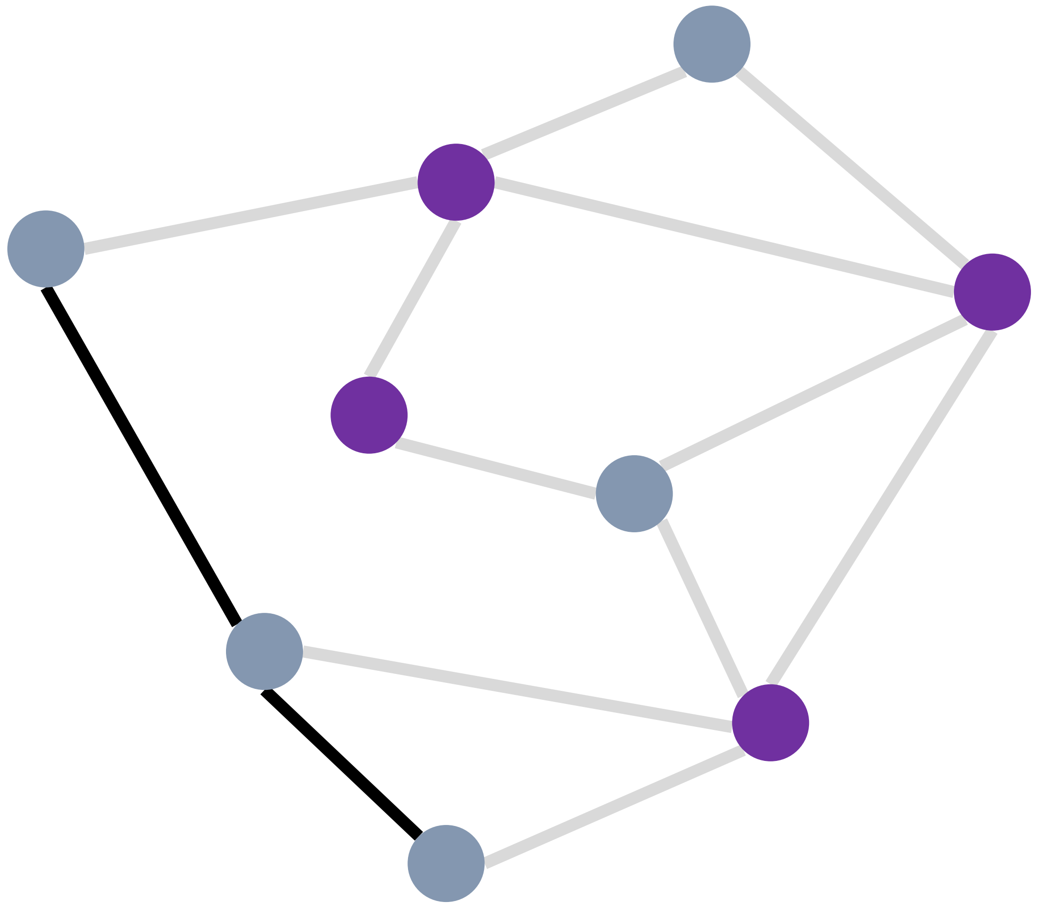


Goal: Obtain a <u>2-approximation</u> (i.e., vertex cover that is at most twice as large as the optimal)

Consider an edge $e = (u, v) \in E$

- Optimal vertex covering must contain either u or v
- **Our approach:** take <u>both</u> of them!
  - Add u, v to cover
  - Remove all edges incident on u and v
  - Repeat until no edges remain

Approximate Vertex Cover

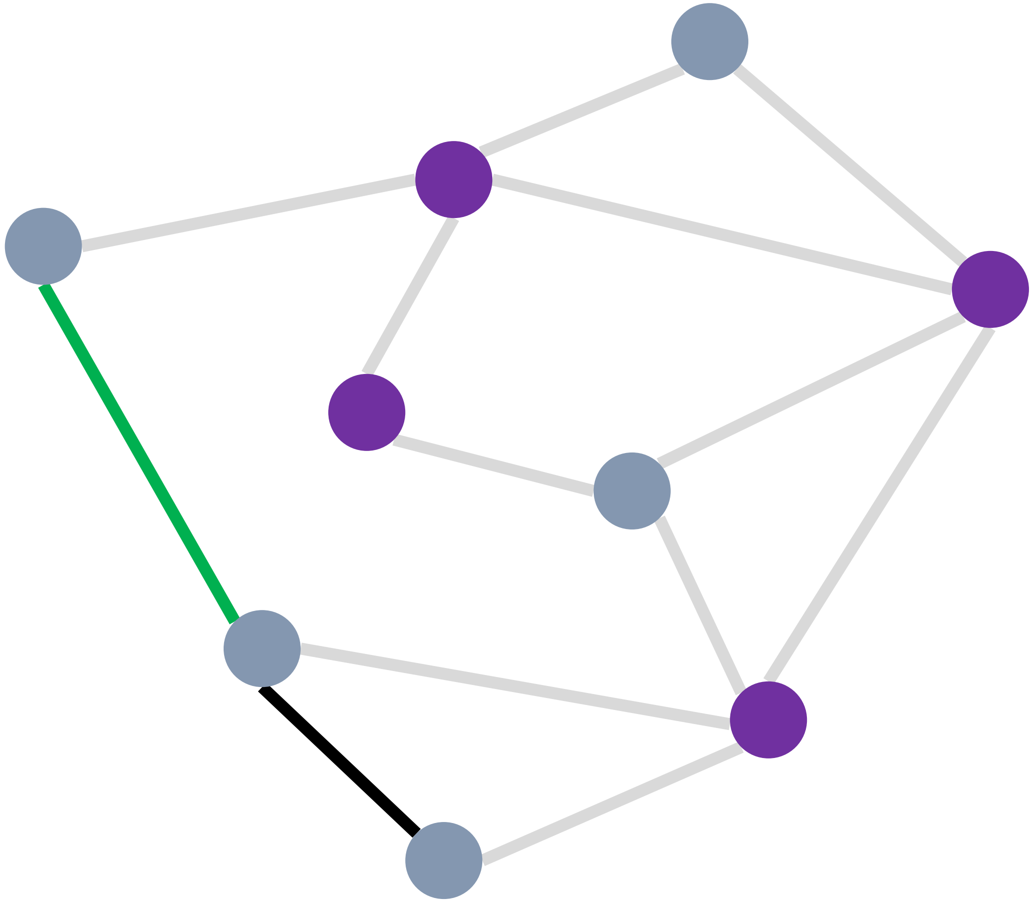


Goal: Obtain a 2-approximation (i.e., vertex cover that is at most twice as large as the optimal)

Consider an edge $e = (u, v) \in E$

- Optimal vertex covering must contain either u or v
- **Our approach:** take both of them!
  - Add u, v to cover
  - Remove all edges incident on u and v
  - Repeat until no edges remain

Approximate Vertex Cover

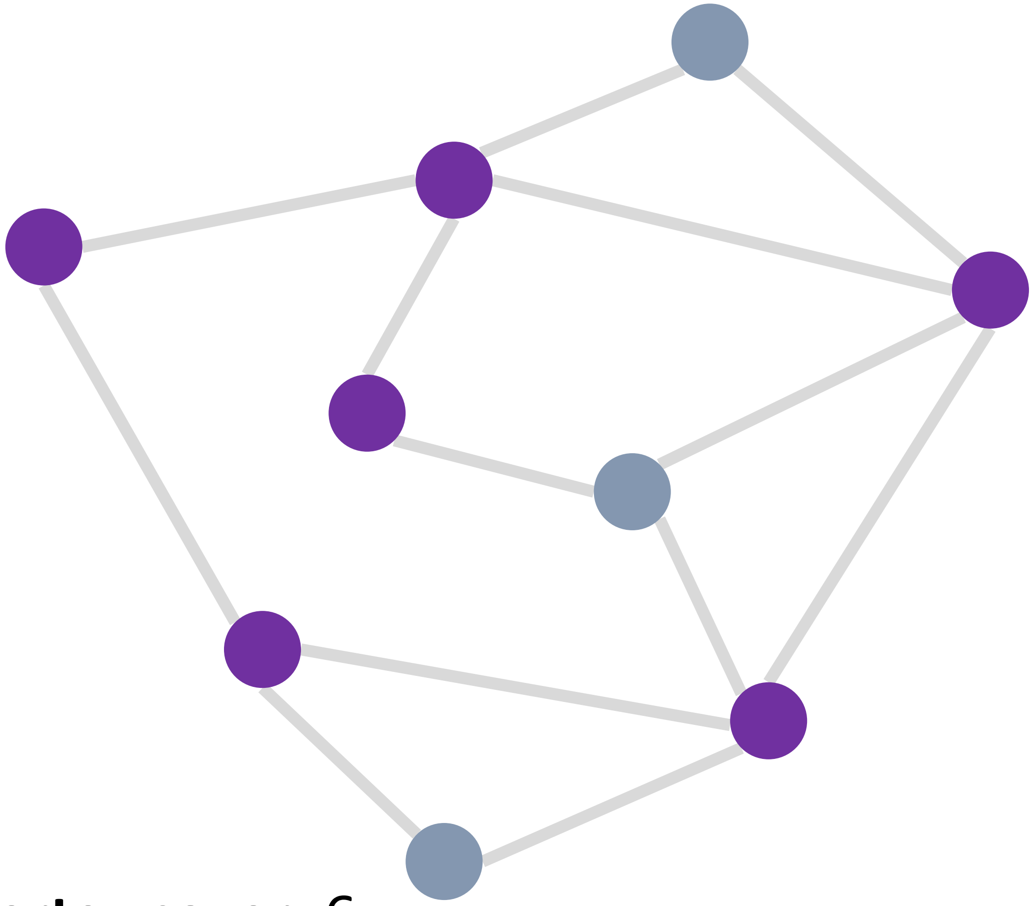


Goal: Obtain a 2-approximation (i.e., vertex cover that is at most twice as large as the optimal)

Consider an edge $e = (u, v) \in E$

- Optimal vertex covering must contain either u or v
- **Our approach:** take both of them!
  - Add u, v to cover
  - Remove all edges incident on u and v
  - Repeat until no edges remain

Approximate Vertex Cover



Size of vertex cover: 6

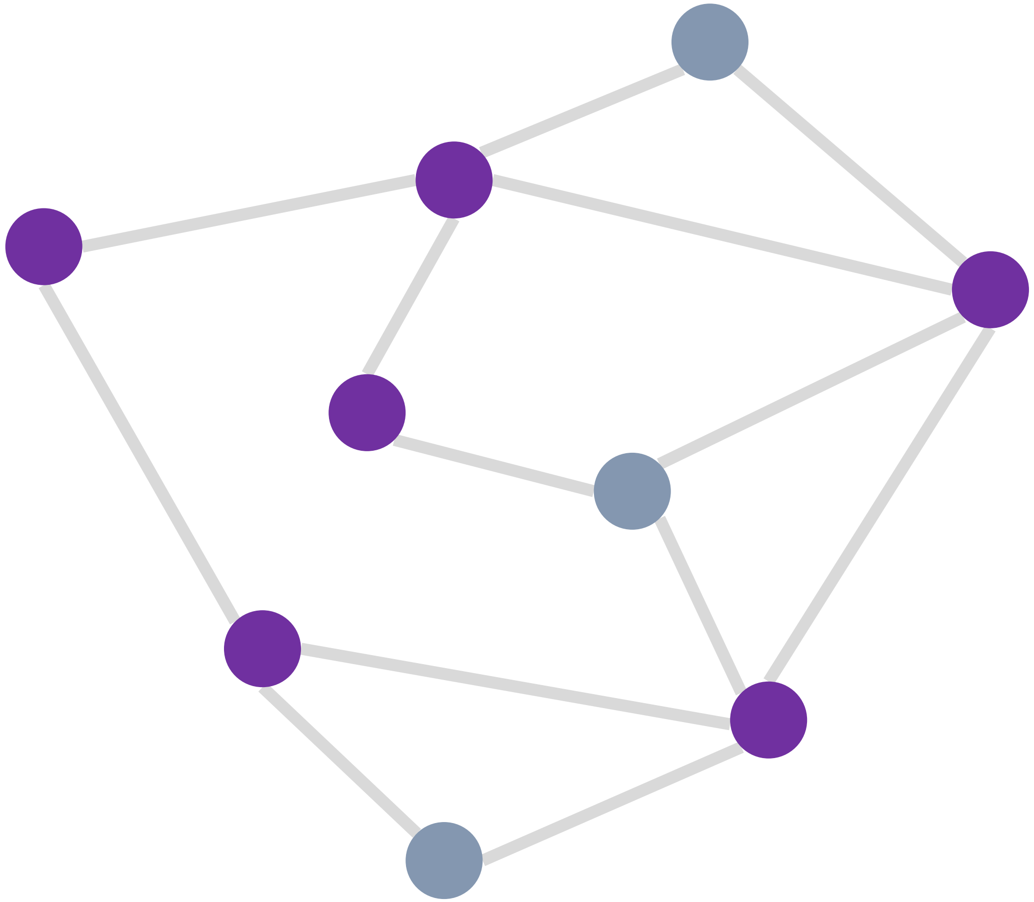
Size of optimal vertex cover: 5

Goal: Obtain a 2-approximation (i.e., vertex cover that is at most twice as large as the optimal)

Consider an edge $e = (u, v) \in E$

- Optimal vertex covering must contain either u or v
- **Our approach:** take both of them!
  - Add u, v to cover
  - Remove all edges incident on u and v
  - Repeat until no edges remain

Approximate Vertex Cover



Theorem. The approximate algorithm for vertex cover achieves an approximation factor of 2

Consider an edge $e = (u, v) \in E$

- Optimal vertex covering must contain either u or v
- **Our approach:** take both of them!
  - Add u, v to cover
  - Remove all edges incident on u and v
  - Repeat until no edges remain

Coping with NP-Hardness

Many optimization problems that come up in practice are NP-complete

What do we do?

Approach 1: Find an algorithm that gives <u>nearly-optimal</u> solutions

Question: Can we do better than a 2-approximation?

Slightly… there is an algorithm that achieves a $\left(2 - O\left(1/\sqrt{\log|V|}\right)\right)$ approximation

Open Problem: Obtain a $(2 - \varepsilon)$-approximation for constant $\varepsilon > 0$

Question: What's the best we could hope for? Can we have a 1.00001-approximation?

Unlikely, computing a $\sqrt{2} \approx 1.41$ approximation is NP-hard (Khot-Minzer-Safra, 2018)

Earlier lower bounds: $7/6 \approx 1.17$ (Håstad, 1997), $10\sqrt{5} - 21 \approx 1.36$ (Dinur-Safra, 2005)

Coping with NP-Hardness

Many optimization problems that come up in practice are NP-complete

What do we do?

Approach 1: Find an algorithm that gives nearly-optimal solutions

Question: Can we do better than a 2-approximation?

$\sqrt{\log|V|}\Big)\Big)$ approximation

Open Problem: Obta

> **Hardness of approximation:** many NP-hard problems are hard not only to solve exactly, but even hard to approximate (beautiful theory – see also PCP theorem)

Question: What's the ... ation?

Unlikely, computing a $\sqrt{2} \approx 1.41$ approximation is NP-hard (Khot-Minzer-Safra, 2018)

Earlier lower bounds: $7/6 \approx 1.17$ (Håstad, 1997), $10\sqrt{5} - 21 \approx 1.36$ (Dinur-Safra, 2005)

Coping with NP-Hardness

Many optimization problems that come up in practice are NP-complete

What do we do?

Approach 1: Find an algorithm that gives nearly-optimal solutions

Approach 2: For small instances, solve using brute force or dynamic programming
Can also improve (expected) run-time using heuristics

Coping with NP-Hardness

Many optimization problems that come up in practice are NP-complete

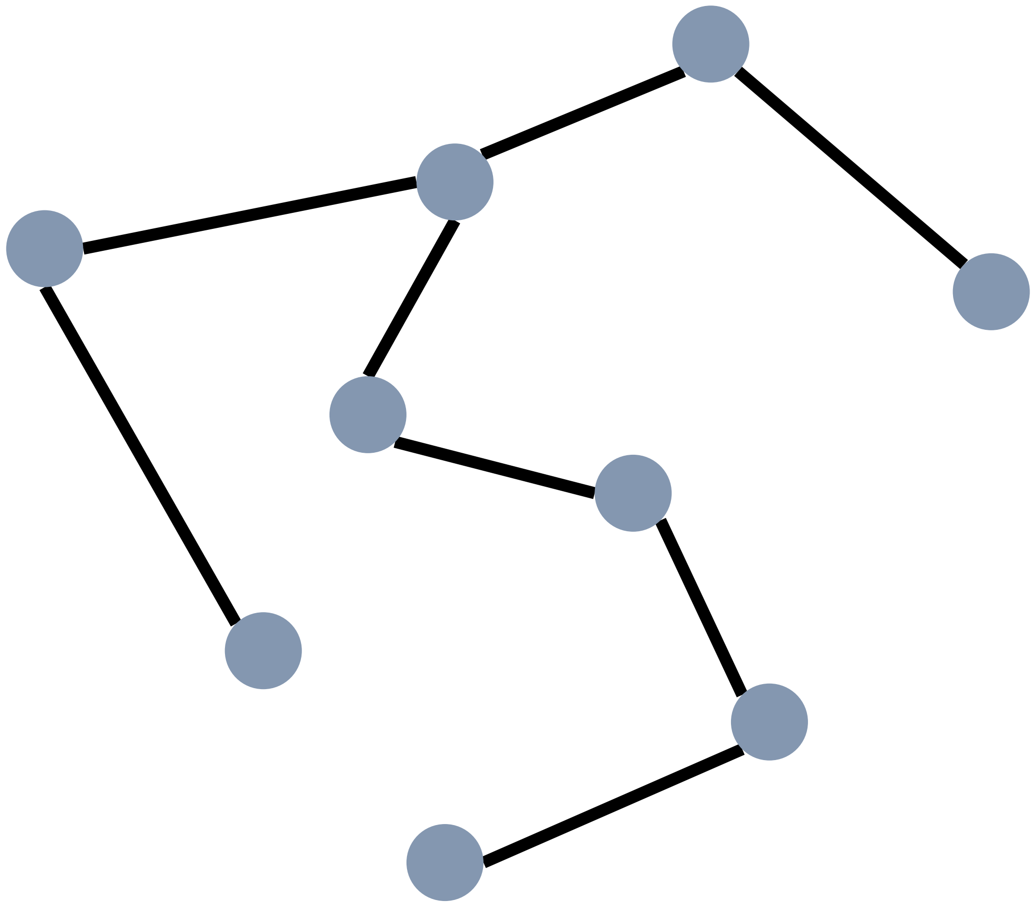
What do we do?

Approach 1: Find an algorithm that gives <u>nearly-optimal</u> solutions

Approach 2: For small instances, solve using brute force or dynamic programming
Can also improve (expected) run-time using heuristics

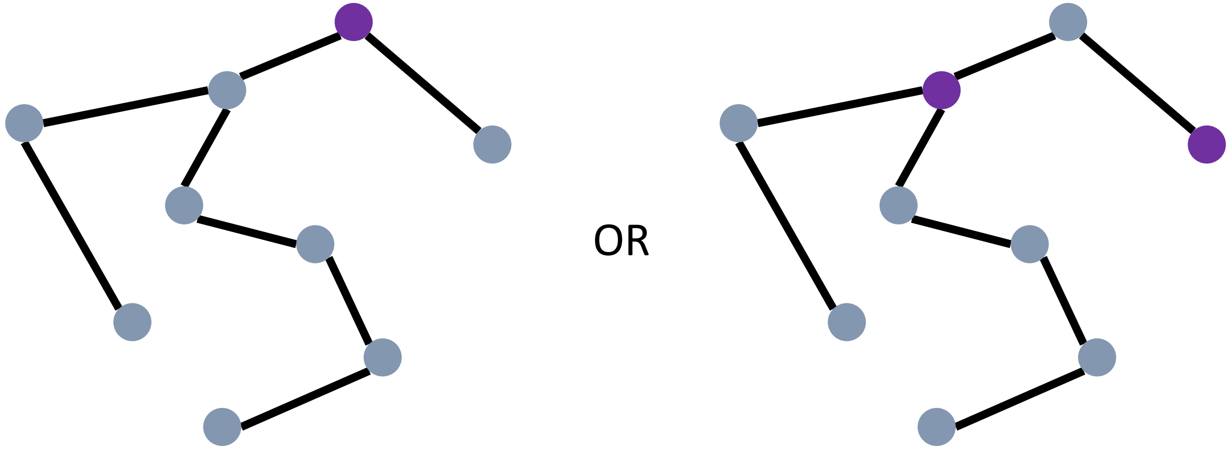
Approach 3: Special cases of the problems can be tractable

Vertex Cover on a Tree



When the graph is a tree, vertex cover can be solved using dynamic programming:

- Consider the root node
- Either it is part of the cover or all of its children are part of the cover



Solve vertex cover on subtrees and take the minimum

Coping with NP-Hardness

Many optimization problems that come up in practice are NP-complete

What do we do?

Approach 1: Find an algorithm that gives nearly-optimal solutions

Approach 2: For small instances, solve using brute force or dynamic programming

Can also improve (expected) run-time using heuristics

Approach 3: Special cases of the problems can be tractable

(see also **parameterized complexity**)