P versus NP, NP hard and NP complete

Shifting Focus

- Most of the class has been about how to efficiently solve problems
- Now we're going to shift to a higher-level question
    - What problems can a computer solve efficiently?
    - What problem can a computer not solve efficiently?

Efficiency: Polynomial time

- What problems can a computer solve in polynomial time?
- What problems can a computer (probably) **not** solve in polynomial time?

Technical Setup

- We will now focus on **decision problems** — problems with a yes or no answer
  - Does this directed graph have a topological order?
  - Is this graph bipartite?
  - Do these two strings have Edit Distance at most 10?
  - Does this flow network have a max flow of at least 20?

Technical Setup

- Most problems have a decision analog
  - Find the flow of this network -> "does this network have flow at least k?"
  - Find the optimal schedule of these intervals -> "can we schedule at least k intervals?"
- These are (essentially) the same—-after all, can always binary search for the optimal value

Technical Setup

- Decision problem means that every solution is “yes” or “no”
- Yes instances can represented as a set of inputs A
  - $x \in A$ means that the solution to x is “yes”
  - $x \notin A$ means that the solution to x is “no”
- So can have (for example): A is the set of all flow networks which permit flow at least k
- Or can have: A is the set of all pairs of strings (a, b) where the edit distance between a and b is at most k

Class P

- **P**: the class of decision problems that can be solved in polynomial time [in the size of the input]
  - Edit distance is in **P**
  - Max flow is in **P**
  - Bipartite matching is in **P**
  - Knapsack?
    - dynamic programming algorithm we saw is pseudo-polynomial! So we don't know yet

Class NP

Class NP—Intuition

- **NP** is the class of problems that can be *verified* in polynomial time
- If I give you helpful information, say a proposed solution, you can easily check that it is correct

Class NP—Intuition

8								
		3	6					
	7			9		2		
	5				7			
				4	5	7		
			1				3	
		1					6	8
		8	5				1	
	9					4		

AI14473 (c) Arto Inkala www.aisudoku.com

☆☆☆☆☆☆☆☆☆☆☆

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

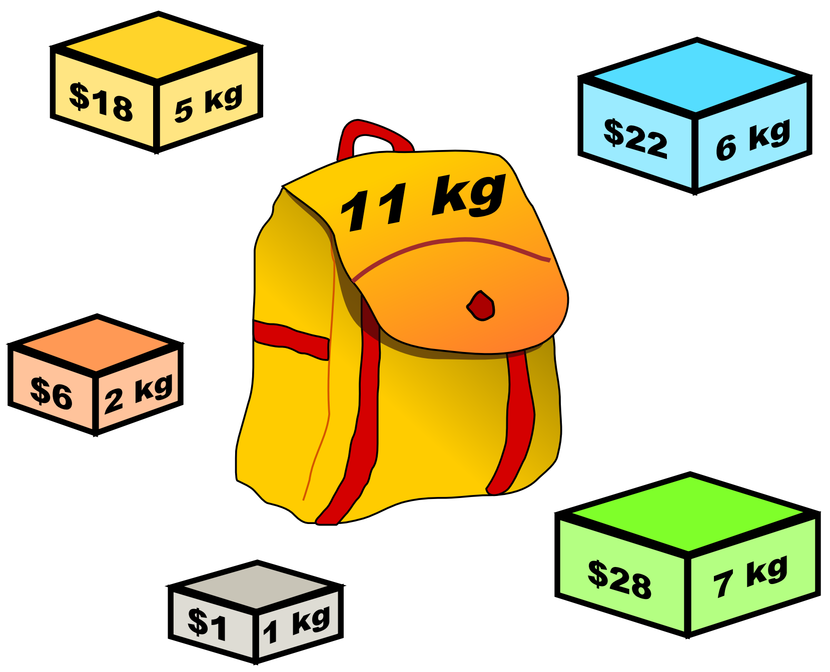
Sudoku is easy if I give you information (by giving you the solution). So sudoku is in **NP**

Class NP—Intuition

- Example (Knapsack capacity C = 11)
  - {3, 4} has value $40 (and weight 11)

i	v_i	w_i
1	$1	1 kg
2	$6	2 kg
3	$18	5 kg
4	$22	6 kg
5	$28	7 kg

knapsack instance
(weight limit W = 11)



Knapsack is easy if I give you information (by giving you the solution). So knapsack is in NP

Class NP: Formally

Definition. Algorithm $V(s, c)$ is a verifier for problem X if for every input s there exists a certificate, a string c, such that $V(s, c) =$ yes iff $s \in X$.

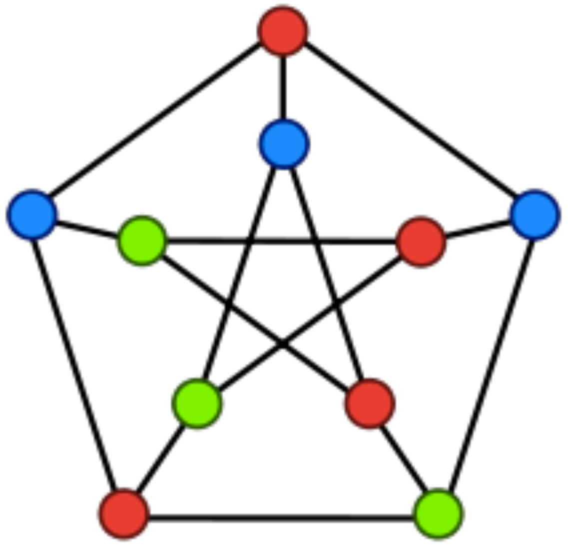
Definition. NP = set of decision problems for which there exists a polynomial-time verifier

- $V(s, c)$ is a polynomial time algorithm
- Certificate c is of polynomial size:
  - $|c| \leq p(|s|)$ for some polynomial $p(\,.\,)$
- A solution is often a good certificate! But any polynomial-size certificate is allowed

Graph-Coloring $\in$ NP

Graph-Coloring. Given a graph $G = (V, E)$, is it possible to color the vertices of G using only three colors, such that no edge has both end points colored with the same color.

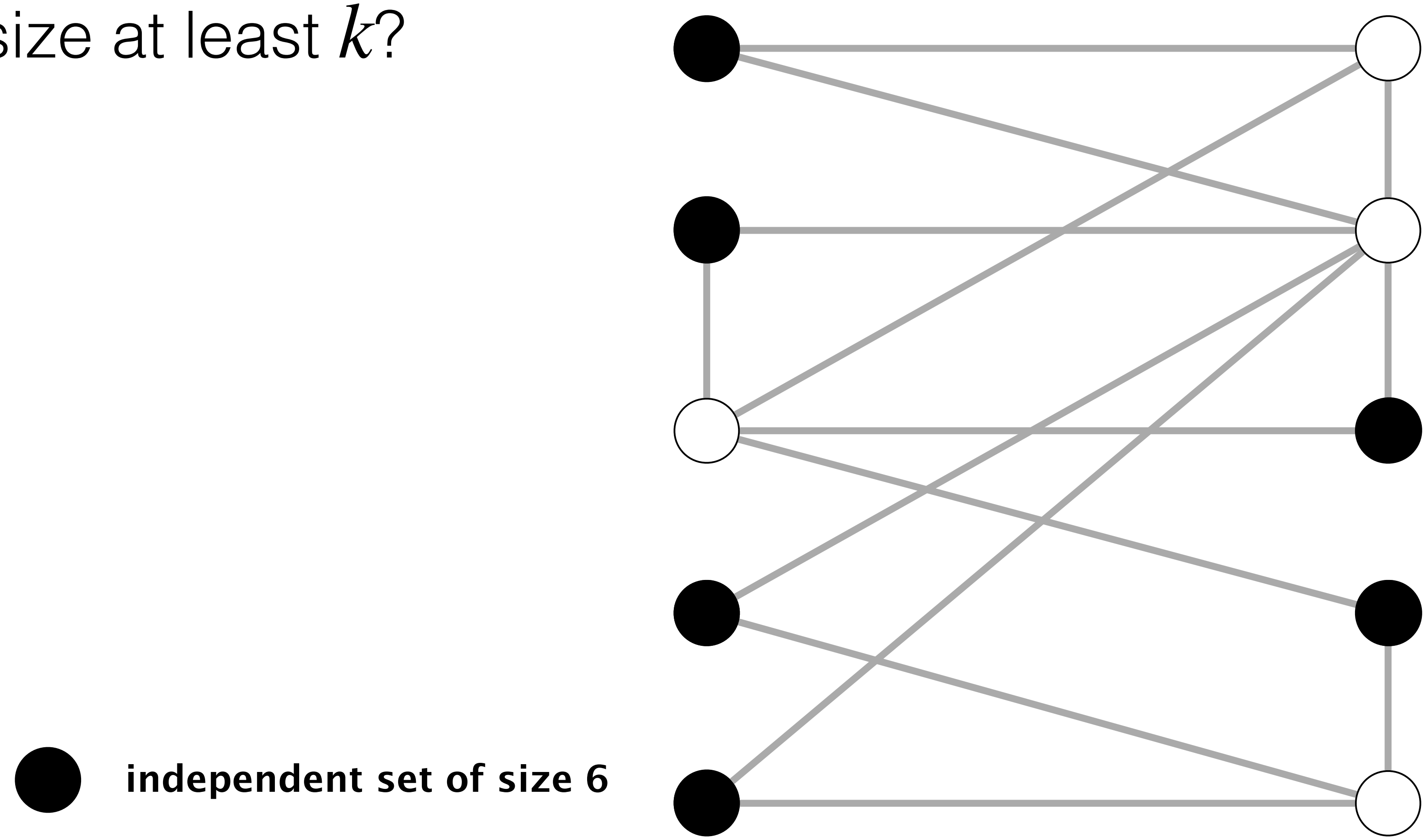
- Graph-Coloring $\in$ NP
  - **Certificate:** assignment of colors to vertices
  - **Poly-time verifier:** check if at most 3 colors used, check for each edge if ends points same color or not



A 3-colorable graph

Independent Set

- Given a graph $G = (V, E)$, an independent set is a subset of vertices $S \subseteq V$ such that no two of them are adjacent, that is, for any $x, y \in S$, $(x, y) \notin E$
- **IND-SET Problem.**
  Given a graph $G = (V, E)$ and an integer k, does G have an independent set of size at least k?



IND-SET $\in$ NP

- Given a graph $G = (V, E)$, an independent set is a subset of vertices $S \subseteq V$ such that no two of them are adjacent, that is, for any $x, y \in S$, $(x, y) \notin E$

- **IND-SET Problem.** Given a graph $G = (V, E)$ and an integer k, does G have an independent set of size at least k?
- **IND-SET** $\in$ NP.
  - **Certificate:** a subset of vertices (the independent set of size at least k)
  - **Poly-time verifier:** check if any two vertices are adjacent and check if size is at least k

Testing Your Intuition

Not all problems can be easily verified (not all problems are in NP)

- Is there an input that causes this computer program to run infinitely?
- You can give me an input and claim that the computer program runs infinitely, but I can't verify that in polynomial time

I mean **can't.** Not obvious: you'll explore in 361

Quick Question

- Is P ⊆ NP?
    - If a problem is in **P**, does that mean that it is in **NP**?
- Yes! If a problem can be solved in polynomial time, it can be verified in polynomial time.
- Just solve directly (Can just set $c =$ ””—we don’t need advice to solve this problem)

Satisfiability

- The next problem is the classic example of a problem in **NP**
  - (and, as we'll soon see, probably not in **P**)
- Many different small variations on the same problem (we'll see a couple)
- **Idea**: given a logical equation, can we assign "true" and "false" to the variables to satisfy the equation?

SAT, 3SAT $\in$ NP

- **SAT**. Given a CNF formula ϕ, does it have a satisfying truth assignment?
- **3SAT.** A SAT formula where each clause contains exactly 3 literals (corresponding to different variables)
- $\phi = (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_4)$
- Satisfying instance: $x_1 = 1,\ x_2 = 1,\ x_3 = 0,\ x_4 = 0$, where $1:$ true, $0:$ false
- SAT, 3-SAT $\in$ NP
    - Certificate: truth assignment to variables
    - Poly-time verifier: check if assignment evaluates to true

P versus NP

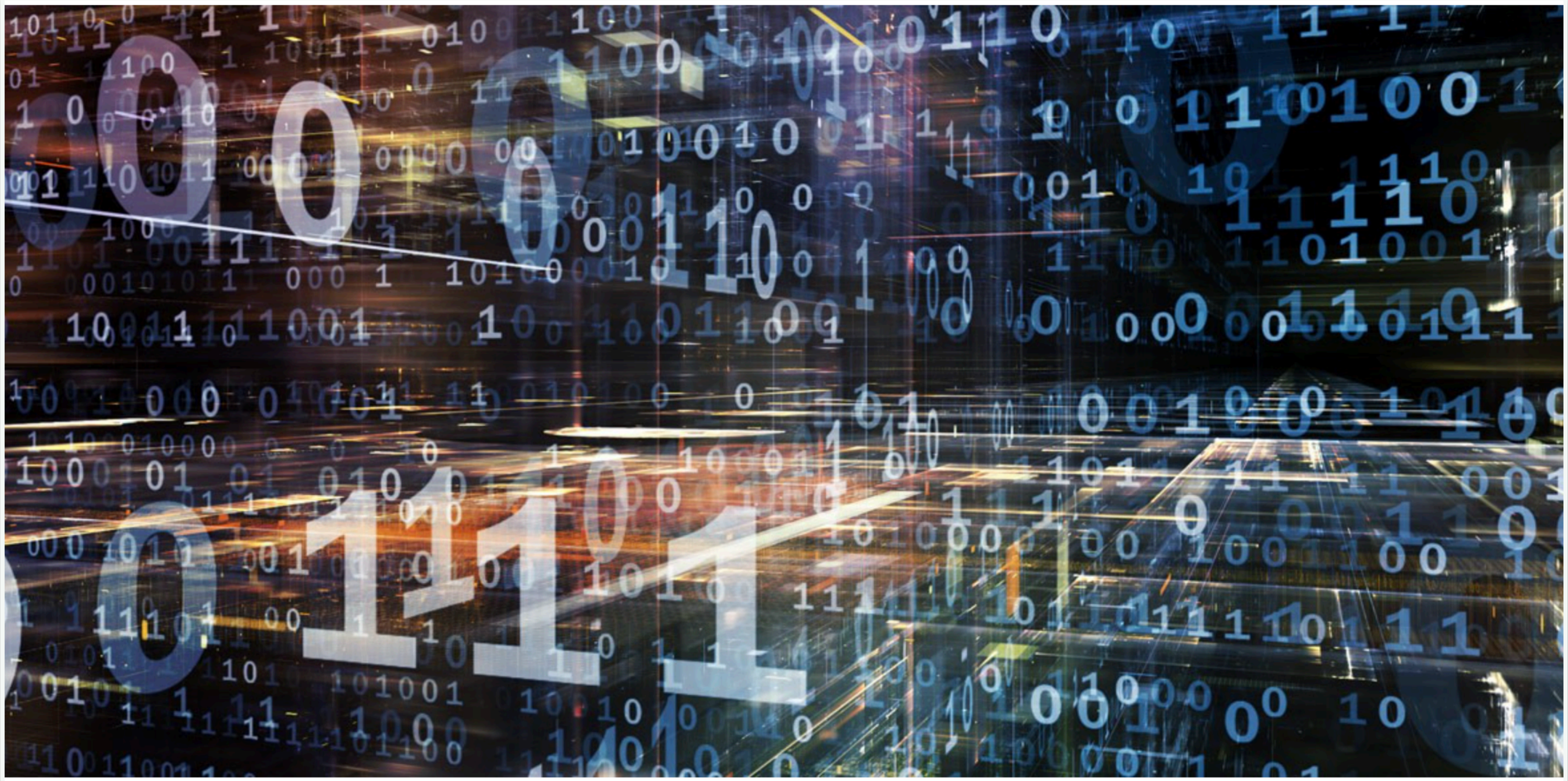
P vs NP

- We know that every problem in **P** is also in **NP**
- What about the reverse? That is to say:
    - If a problem can be efficiently *verified*, does that mean it can be efficiently solved in the first place?
    - Or, do there exist problems that can be verified quickly that are *impossible* to solve quickly?

Why Do We Care?

- If $\mathsf{P} = \mathsf{NP}$, the consequences:
  - Lots of important problems can be solved quickly!
  - Can build things better, faster, more efficiently
  - (Public key) cryptography does not exist
- If $\mathsf{P} \neq \mathsf{NP}$:
  - Many problems can't be solved quickly
  - Can stop trying to solve them
  - Most researchers think this is more likely to be the case

Million Dollar Question:
P vs NP



P vs NP and the $1M Millennium Prize Problems

What's the most difficult way to earn $1M US Dollars?

https://medium.com/@mpreziuso/

Million Dollar Question:
P vs NP

- The biggest open problem in computer science
- One of the biggest in math as well
- We are not even close to solving it!

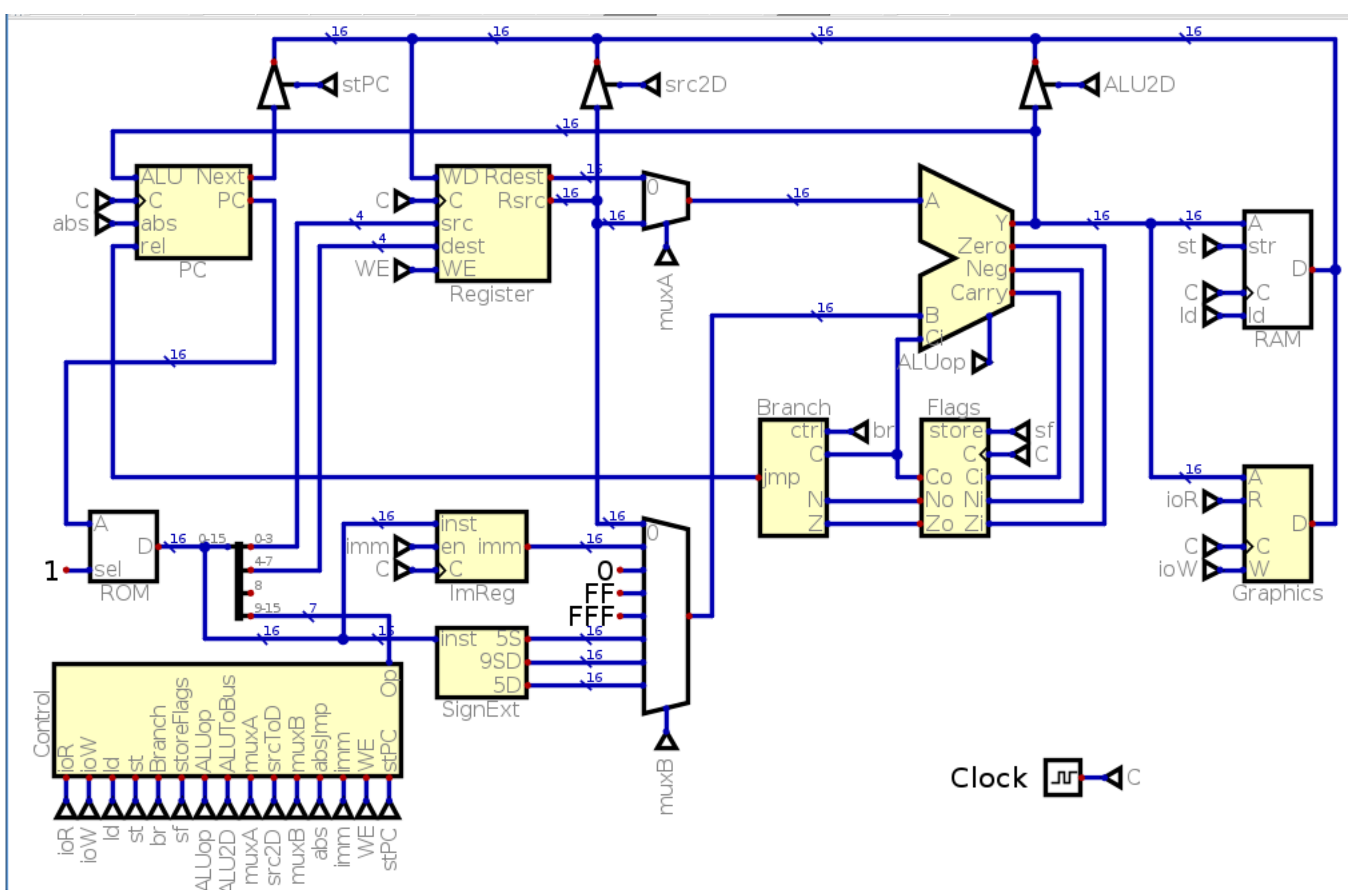
NP-hard and NP-Complete Problems

Cook-Levin Theorem

- If **SAT** can be solved in polynomial time, then *any* problem in **NP** can be solved in polynomial time
- So if **SAT** can be solved in polynomial time, then **P = NP**
- How is this possible?

Cook-Levin Theorem

- Idea: any computer program can be represented by a circuit.
- Solve **SAT** in poly time -> can figure out the answer given by the circuit for NP problem in poly time



You'll see the proof in CS 361

NP-Hard Problems

- A problem X is **NP-hard** if:
  - If X can be solved in polynomial time, then any problem in **NP** can be solved in polynomial time
  - That is, if X can be solved in polynomial time, then **P = NP**

What Does This Mean?

- We think that, probably, $\mathsf{P} \neq \mathsf{NP}$
- So if a problem is **NP**-hard, then you probably cannot obtain a polynomial-time algorithm for it

Classifying Problems as Hard

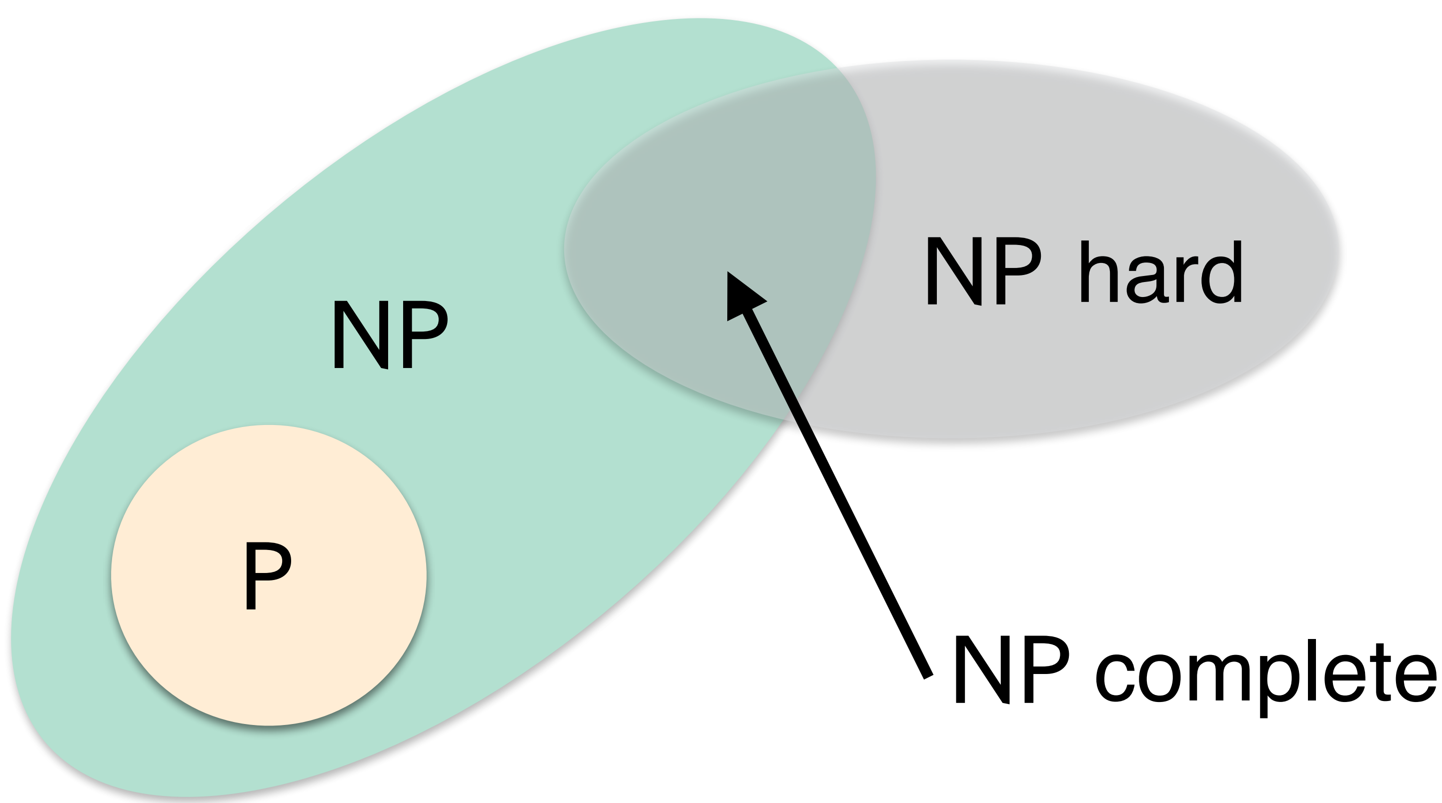
- We are frustratingly unable to prove a lot of problems are **impossible** to solve efficiently
- Instead, we say problem X is likely very hard to solve by saying, *if a polynomial-time algorithm was found for* X*, then something we all believe is impossible will happen*
- Instead we say X is NP-hard: if $X \in \mathsf{P}$, then $\mathsf{P} = \mathsf{NP}$

Classifying Problems as Hard

- Instead, we say problem X is likely very hard to solve by saying, *if a polynomial-time algorithm was found for* X*, then something we all believe is impossible will happen*
- Instead we say X is NP-hard: if $X \in \mathsf{P}$, then $\mathsf{P} = \mathsf{NP}$
- (Erickson) Calling a problem **NP** hard is like saying, *"If I own a dog, then it can speak fluent English"*
  - You probably don't know whether or not I own a dog, but you are definitely sure I don't own *a talking dog*
  - Corollary: No one should believe that I own a dog
- If a problem is **NP** hard, no one should believe it can be solved in polynomial time

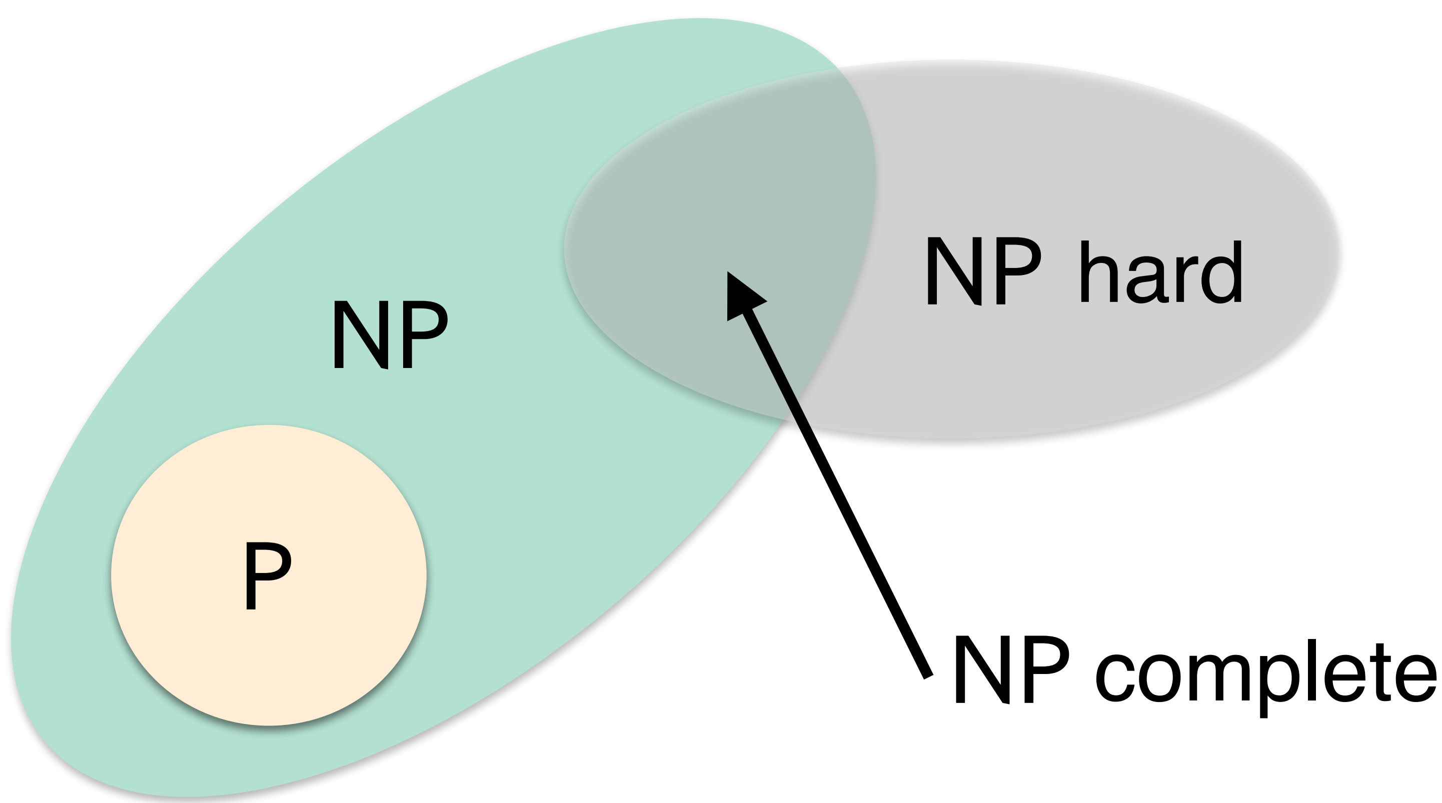
NP Completeness

- **Definition.** A problem X is NP complete if X is NP hard and $X \in$ NP
- SAT is **NP** complete
  - SAT $\in$ **NP**: given an assignment to input gates (certificate), can verify whether output is one or zero in poly-time
  - SAT is **NP** hard (Cook-Levin Theorem); probably not in **P**



Summary

- X is **NP**-hard **NP**-hard $\Leftrightarrow$ if $X \in$ **P**, then **P** = **NP**
- A problem X is NP complete if X is NP hard and $X \in$ **NP**
- Alternate definition of NP hard:
  - X is NP hard if all languages in NP reduce it to in polynomial time
- Thus, NP-complete problems are the hardest problems in NP



NP Hardness Reductions

Relative Hardness

- How do we compare the relative hardness of problems?
- Recurring idea in this class: **reductions!**
- Informally, we say a problem X reduces to a problem Y, if can use an algorithm for Y to solve X
    - Bipartite matching reduces to max flow
    - Finding opportunity cycles reduces to finding negative cycles

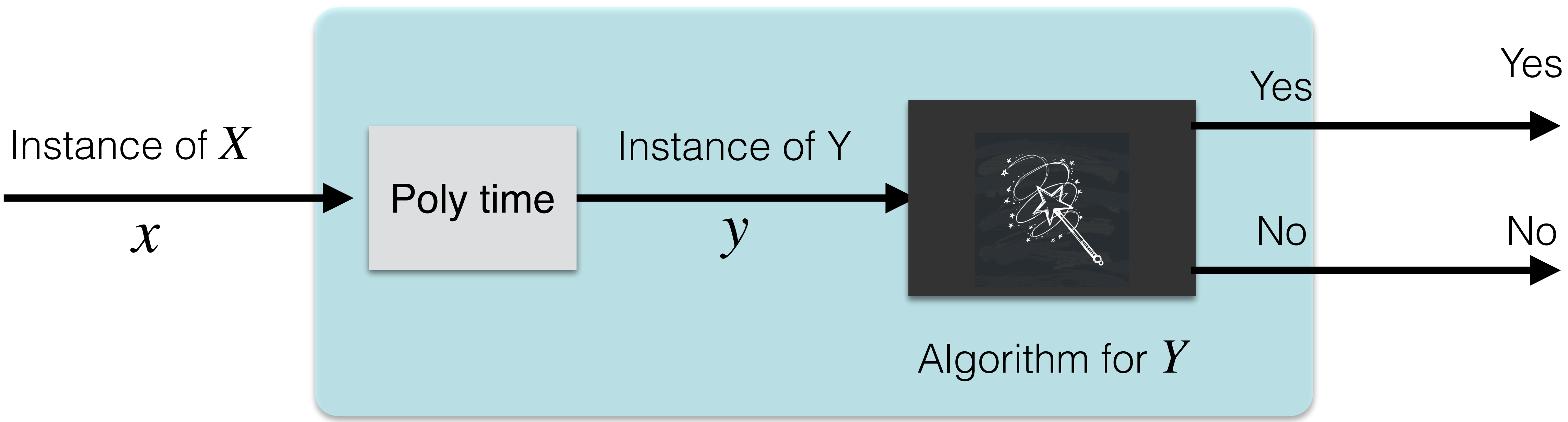
Intuitively, if problem X reduces to problem Y, then solving X is no harder than solving Y

[Karp] Reductions

Definition. Decision problem X polynomial-time (Karp) reduces to decision problem Y if given any instance x of X, we can construct an instance y of Y in polynomial time s.t $x \in X$ if and only if $y \in Y$.

Notation. $X \leq_p Y$

- Solving X is no harder than solving Y: if we have an algorithm for Y, we can use it + poly time reduction to solve X

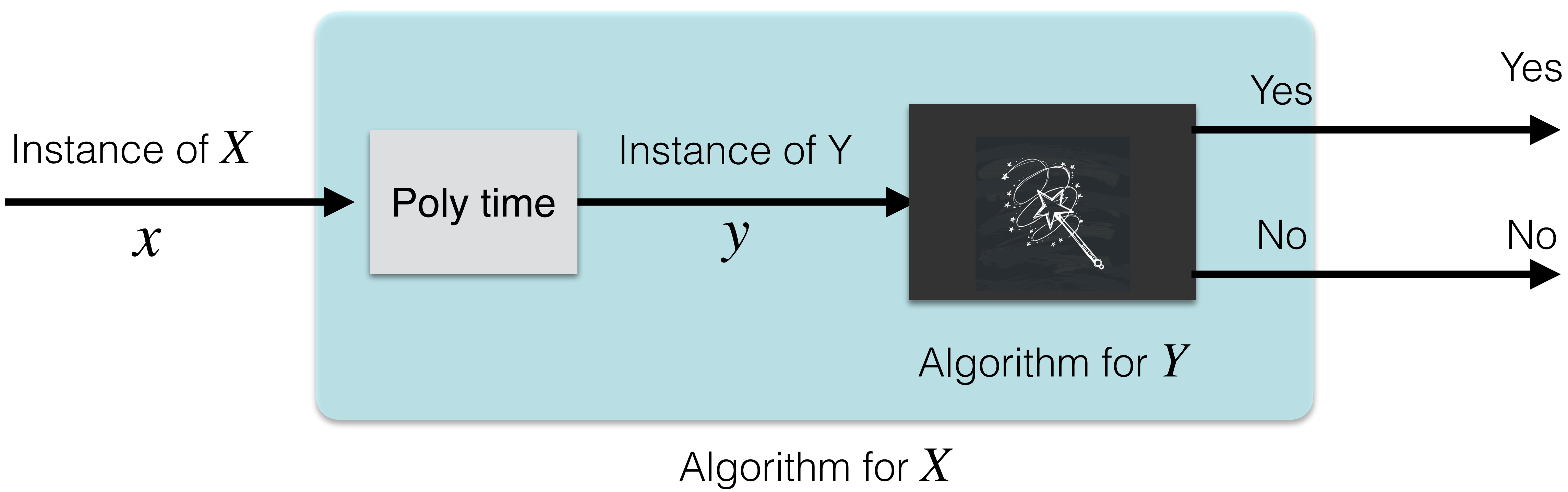


Algorithm for X

Reductions Quiz

Say $X \leq_p Y$. Which of the following can we infer?

- If X can be solved in polynomial time, then so can Y.
- X can be solved in poly time iff Y can be solved in poly time.
- If X cannot be solved in polynomial time, then neither can Y.
- If Y cannot be solved in polynomial time, then neither can X.



Digging Deeper

- **Graph 2-Color** reduces to **Graph 3-color**
  - Let's do this on the board
- **Graph 2-Color** can be solved in polynomial time
  - How?
  - Can decide if a graph is bipartite in $O(n+m)$ time using BFS
- **Graph 3-color** (we'll show) is NP hard and unlikely to have a polynomial-time solution

Intuitively, if problem X reduces to problem Y, then solving X is no harder than solving Y

Use of Reductions: $X \leq_p Y$

Design algorithms:

- If Y can be solved in polynomial time, we know X can also be solved in polynomial time

Establish intractability:

- If we know that X is known to be impossible/hard to solve in polynomial-time, then we can conclude the same about problem Y

Establish Equivalence:

- If $X \leq_p Y$ and $Y \leq_p X$ then X can be solved in poly-time iff Y can be solved in poly time and we use the notation $X \equiv_p Y$

NP hard: Operational Definition

- **New definition of NP hard using reductions.**
  - A problem Y is NP hard, if for any problem $X \in$ NP, $X \leq_p Y$
- Recall we said Y is NP hard if $Y \in$ P, then P $=$ NP.
- Lets show that both definitions are equivalent
  - $(\Rightarrow)$ every problem in **NP** reduces to Y, and if $Y \in$ P, then P $=$ NP
  - $(\Leftarrow)$ Suppose $Y \in$ P, then P $=$ NP: which means every problem in NP$(=$ P$)$ reduces to Y

Proving NP Hardness

- To prove problem Y is NP-hard
  - Difficult to prove every problem in NP reduces to Y
  - Instead, we use a known-NP-hard problem Z
  - We know every problem X in NP, $X \leq_p Z$
  - Notice that $\leq_p$ is transitive
  - Thus, enough to prove $Z \leq_p Y$

To prove that a problem Y is NP hard, reduce a known NP hard problem Z to Y

Known NP Hard Problems?

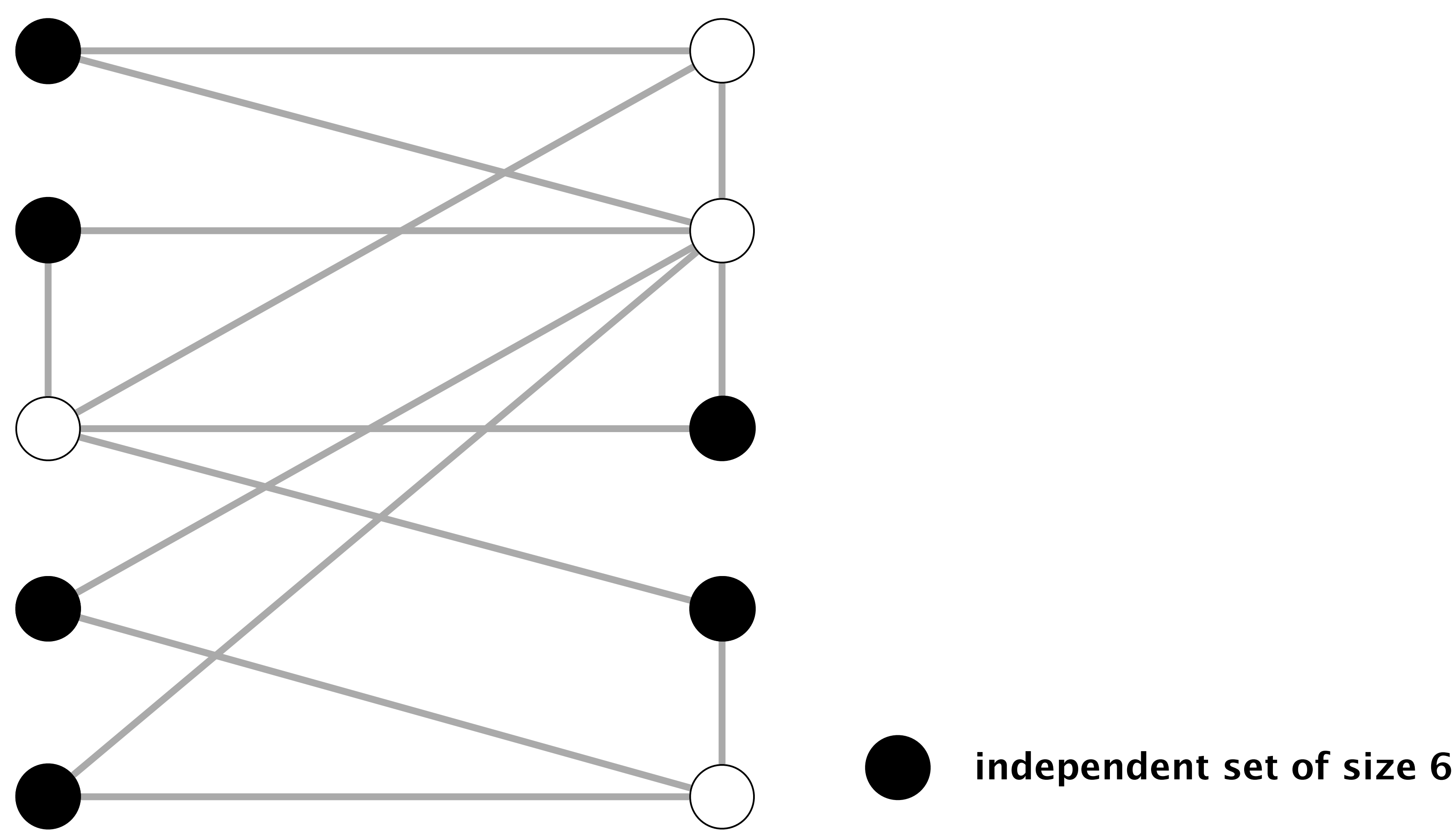
- For now: **3SAT** and **SAT** (Cook-Levin Theorem)
- We will prove a whole repertoire of NP hard and NP complete problems by using reductions
- Before reducing **3SAT** to other problems to prove them NP hard, let us practice some easier reductions first

TO PROVE THAT A PROBLEM Y IS NP HARD, REDUCE A KNOWN NP HARD PROBLEM Z TO Y

VERTEX-COVER $\equiv_p$ IND-SET

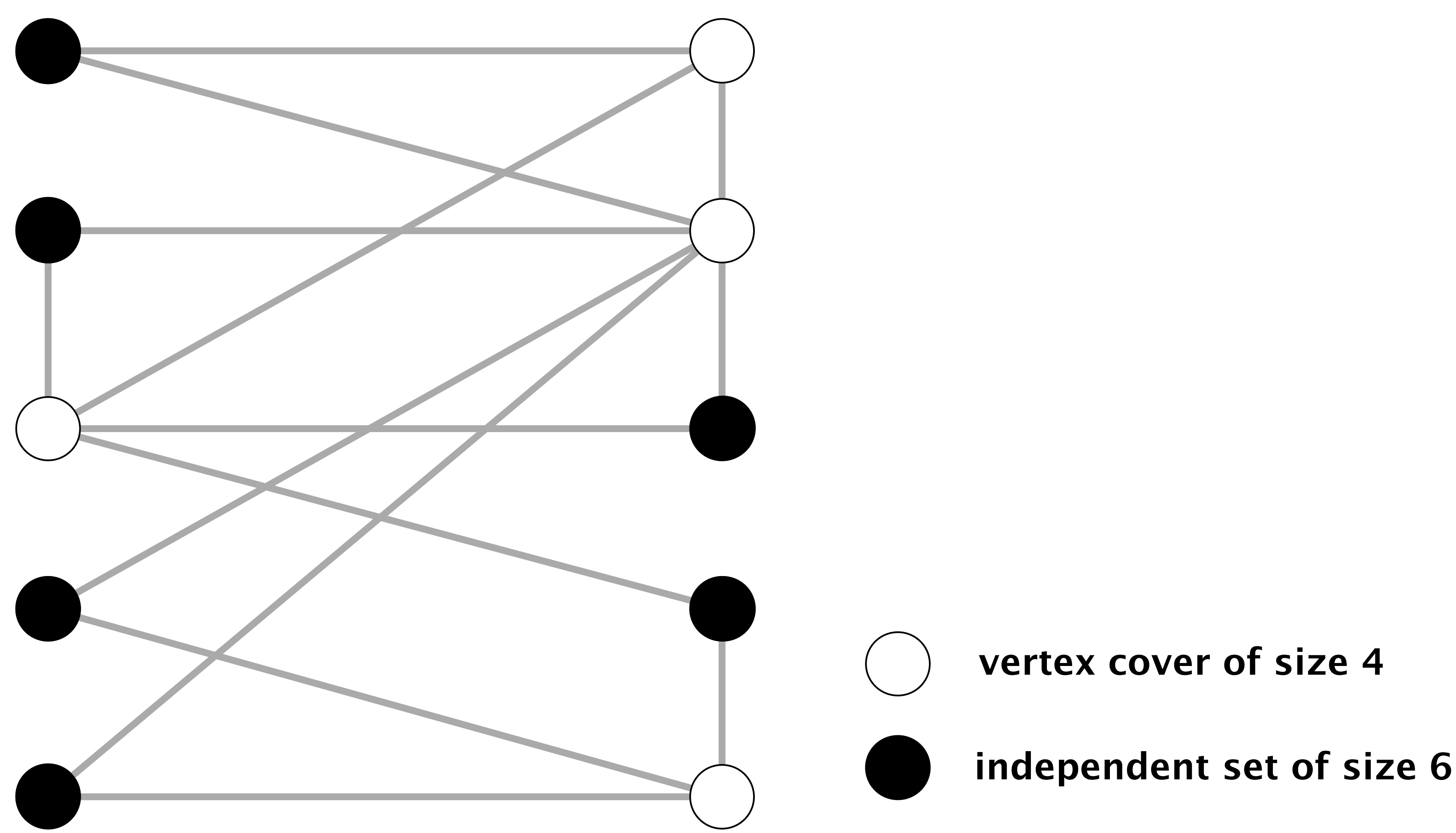
IND-SET

- Given a graph $G = (V, E)$, an independent set is a subset of vertices $S \subseteq V$ such that no two of them are adjacent, that is, for any $x, y \in S,\ (x, y) \notin E$
- **IND-SET Problem.** Given a graph $G = (V, E)$ and an integer k, does G have an independent set of size at least k?



Vertex-Cover

- Given a graph $G = (V, E)$, a vertex cover is a subset of vertices $T \subseteq V$ such that for every edge $e = (u, v) \in E$, either $u \in T$ or $v \in T$.
- **VERTEX-COVER Problem.** Given a graph $G = (V, E)$ and an integer k, does G have a vertex cover of size at most k?



Our First Reduction

- **VERTEX-COVER** $\leq_p$ **IND-SET**
  - Suppose we know how to solve independent set, can we use it to solve vertex cover?
- **Claim.** S is an independent set of size k iff $V - S$ is a vertex cover of size $n - k$.
- **Proof.** (⇒) Consider an edge $e = (u, v) \in E$
  - S is independent: u, v both cannot be in S
  - At least one of $u, v \in V - S$
  - $V - S$ covers e
  - ∎

Our First Reduction

- **VERTEX-COVER** $\leq_p$ **IND-SET**
  - Suppose we know how to solve independent set, can we use it to solve vertex cover?
- **Claim.** S is an independent set of size k iff $V - S$ is a vertex cover of size $n - k$.
- **Proof.** ($\Leftarrow$) Consider an edge $e = (u, v) \in E$
  - $V - S$ is a vertex cover: at least one of u, v must be in $V - S$
  - Both u, v cannot be in S
  - Thus, S is an independent set. $\blacksquare$

Vertex Cover $\equiv_p$ IND Set

- **VERTEX-COVER** $\leq_p$ **IND-SET**
- **Reduction.** Let $G' = G,\ k' = n - k$.
  - $(\Rightarrow)$ If G has a vertex cover of size at most k then G' has an independent set of size at least k'
  - $(\Leftarrow)$ If G' has an independent set of size at least k' then G has a vertex cover of size at most k
- **IND-SET** $\leq_p$ **VERTEX-COVER**
  - Same reduction works: $G' = G$, $k' = n - k$
- **VERTEX-COVER** $\equiv_p$ **IND-SET**

VERTEX-COVER $\leq_p$ SET-COVER

Set Cover

- **Set-Cover.** Given a set U of elements, a collection $\mathcal{S}$ of subsets of U and an integer k, are there **at most** k subsets $S_1, \ldots, S_k$ whose union covers U, that is, $U \subseteq \cup_{i=1}^{k} S_i$

$U = \{\ 1, 2, 3, 4, 5, 6, 7\ \}$

$S_a = \{\ 3, 7\ \}$ $\qquad$ $S_b = \{\ 2, 4\ \}$

$S_c = \{\ 3, 4, 5, 6\ \}$ $\qquad$ $S_d = \{\ 5\ \}$

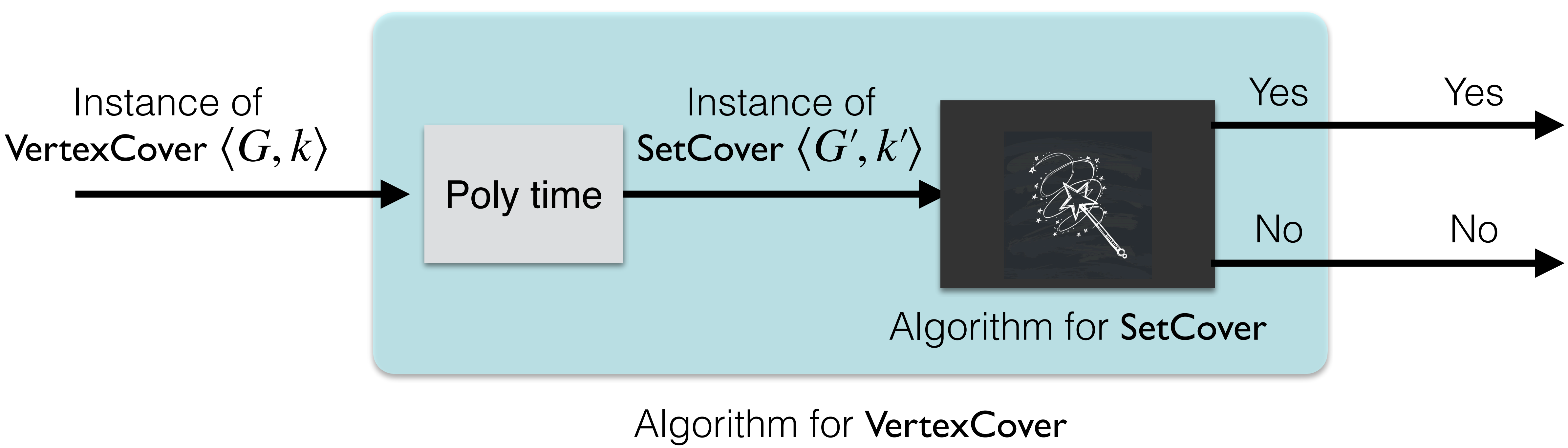
$S_e = \{\ 1\ \}$ $\qquad$ $S_f = \{\ 1, 2, 6, 7\ \}$

$k = 2$

a set cover instance

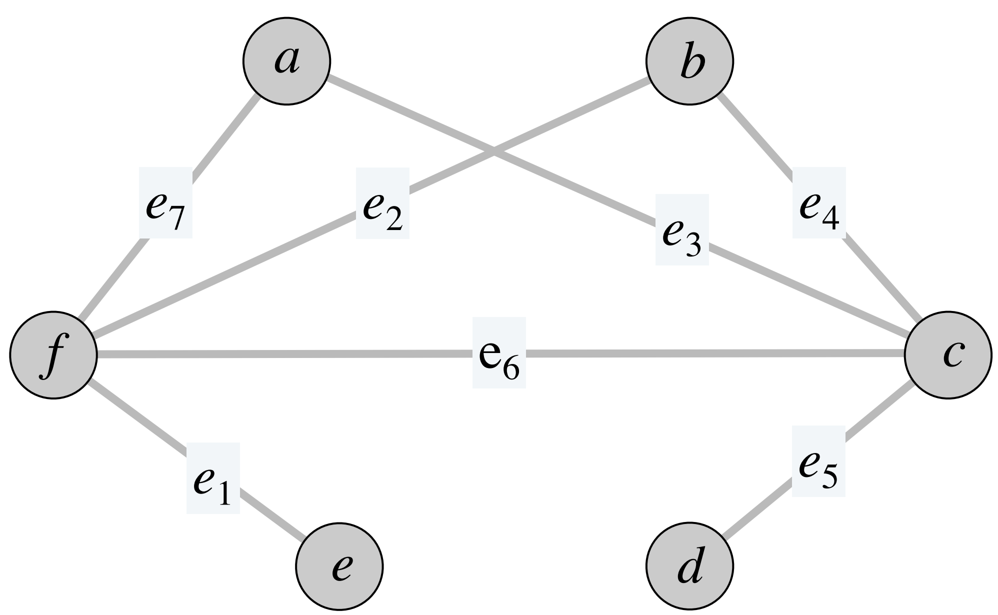
Vertex Cover $\leq_p$ Set Cover

- **Theorem.** VERTEX-COVER $\leq_p$ SET-COVER
- **Proof.** Given instance $\langle G, k \rangle$ of vertex cover, construct an instance $\langle U, \mathcal{S}, k' \rangle$ of set cover problem such that
- G has a vertex cover of size at most k if and only if $\langle U, \mathcal{S}, k' \rangle$ has a set cover of size at most k.



Vertex Cover $\leq_p$ Set Cover

- **Theorem.** VERTEX-COVER $\leq_p$ SET-COVER
- **Proof.** Given instance $\langle G, k \rangle$ of vertex cover, construct an instance $\langle U, \mathcal{S}, k \rangle$ of set cover problem that has a set cover of size k iff G has a vertex cover of size k.
- **Reduction.** $U = E$, for each node $v \in V$, let $S_v = \{e \in E \mid e \text{ incident to } v\}$



vertex cover instance
(k = 2)

$U = \{ e_1, e_2, \ldots, e_7 \}$

$S_a = \{ e_3, e_7 \}$ $\quad S_b = \{ e_2, e_4 \}$

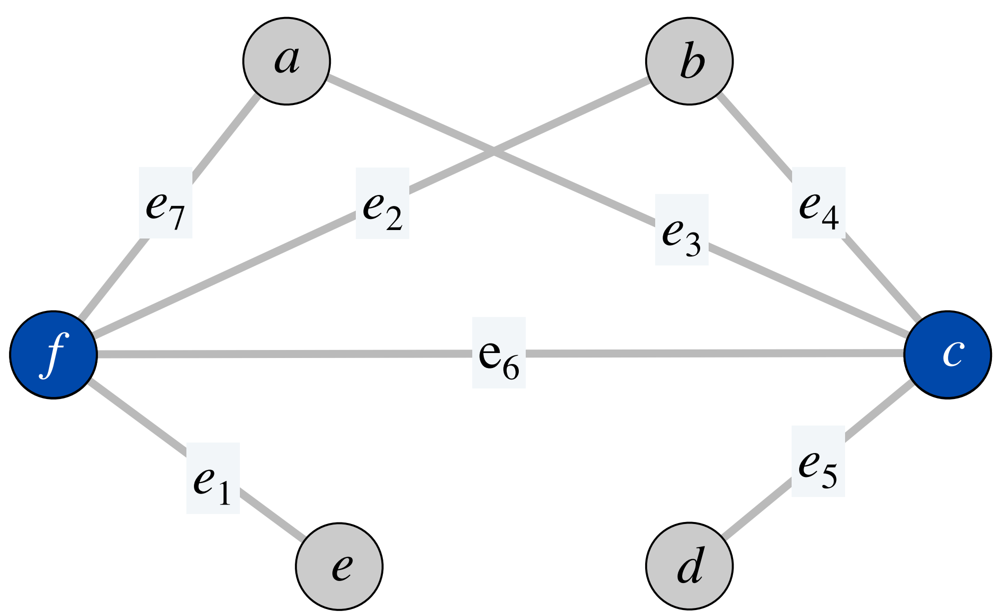
$S_c = \{ e_3, e_4, e_5, e_6 \}$ $\quad S_d = \{ e_5 \}$

$S_e = \{ e_1 \}$ $\quad S_f = \{ e_1, e_2, e_6, e_7 \}$

set cover instance
(k = 2)

Correctness

- **Claim.** $(\Rightarrow)$ If G has a vertex cover of size at most k, then U can be covered using at most k subsets.
- **Proof.** Let $X \subseteq V$ be a vertex cover in G
  - Then, $Y = \{S_v \mid v \in X\}$ is a set cover of U of the same size



vertex cover instance
(k = 2)

$U = \{ e_1, e_2, \ldots, e_7 \}$

$S_a = \{ e_3, e_7 \}$ $\quad$ $S_b = \{ e_2, e_4 \}$

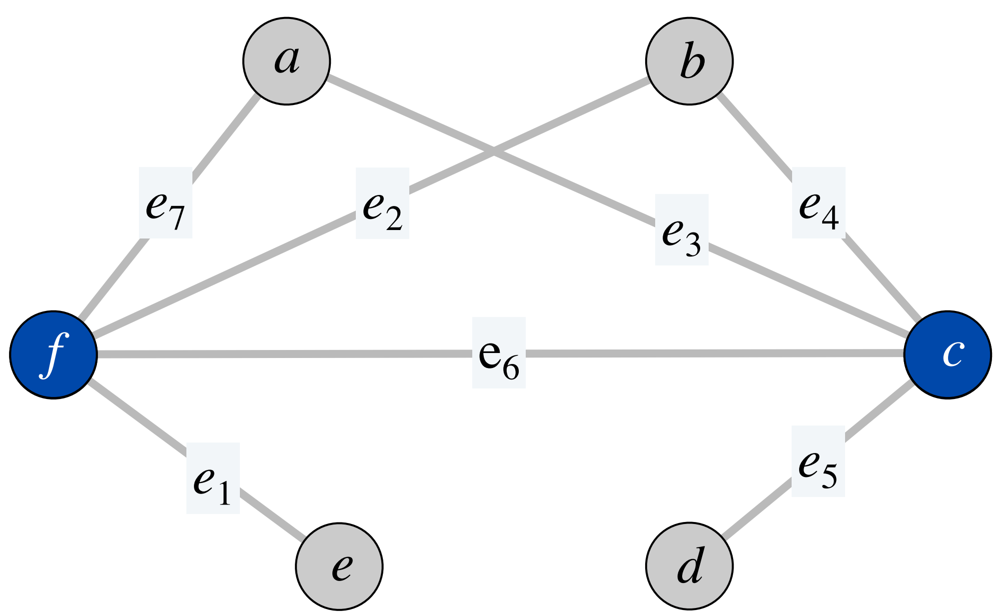
$S_c = \{ e_3, e_4, e_5, e_6 \}$ $\quad$ $S_d = \{ e_5 \}$

$S_e = \{ e_1 \}$ $\quad$ $S_f = \{ e_1, e_2, e_6, e_7 \}$

set cover instance
(k = 2)

Correctness

- **Claim.** $(\Leftarrow)$ If U can be covered using at most k subsets then G has a vertex cover of size at most k.
- **Proof.** Let $Y \subseteq \mathcal{S}$ be a set cover of size k
  - Then, $X = \{v \mid S_v \in Y\}$ is a vertex cover of size k



vertex cover instance
(k = 2)

$U = \{ e_1, e_2, \ldots, e_7 \}$

$S_a = \{ e_3, e_7 \}$ $\quad S_b = \{ e_2, e_4 \}$

$S_c = \{ e_3, e_4, e_5, e_6 \}$ $\quad S_d = \{ e_5 \}$

$S_e = \{ e_1 \}$ $\quad S_f = \{ e_1, e_2, e_6, e_7 \}$

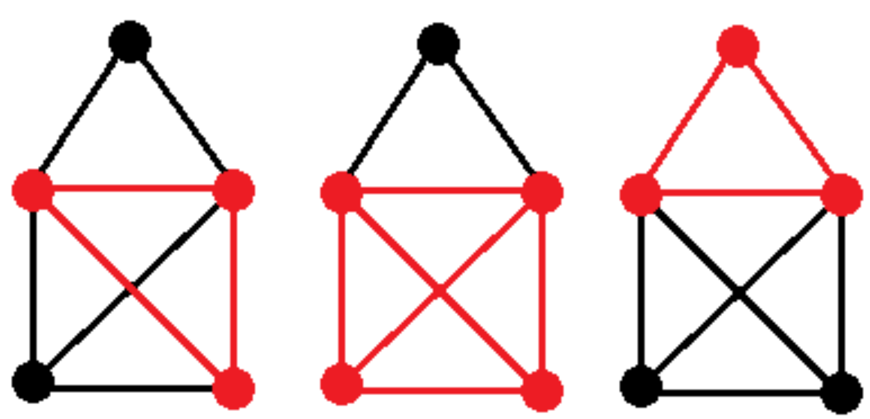
set cover instance
(k = 2)

Class Exercise

IND-SET $\leq_p$ Clique

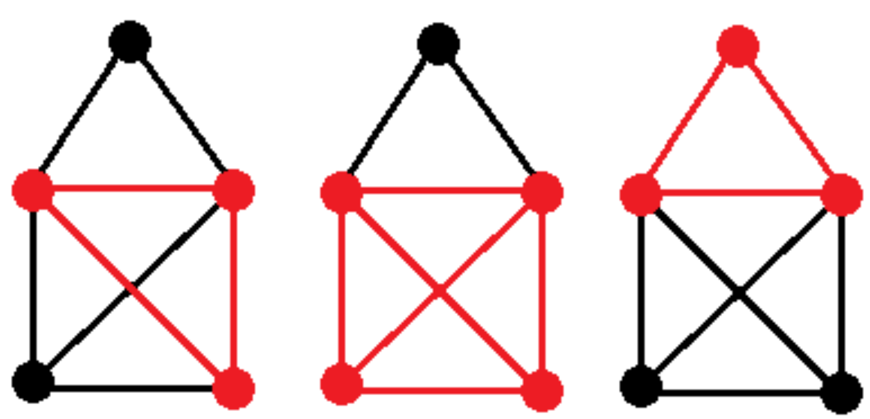
Clique

- A **clique** in an undirected graph is a subset of nodes such that every two nodes are connected by an edge. A k-clique is a clique that contains k nodes.
- **CLIQUE.** Given a graph G and a number k, does G contain a k -clique?



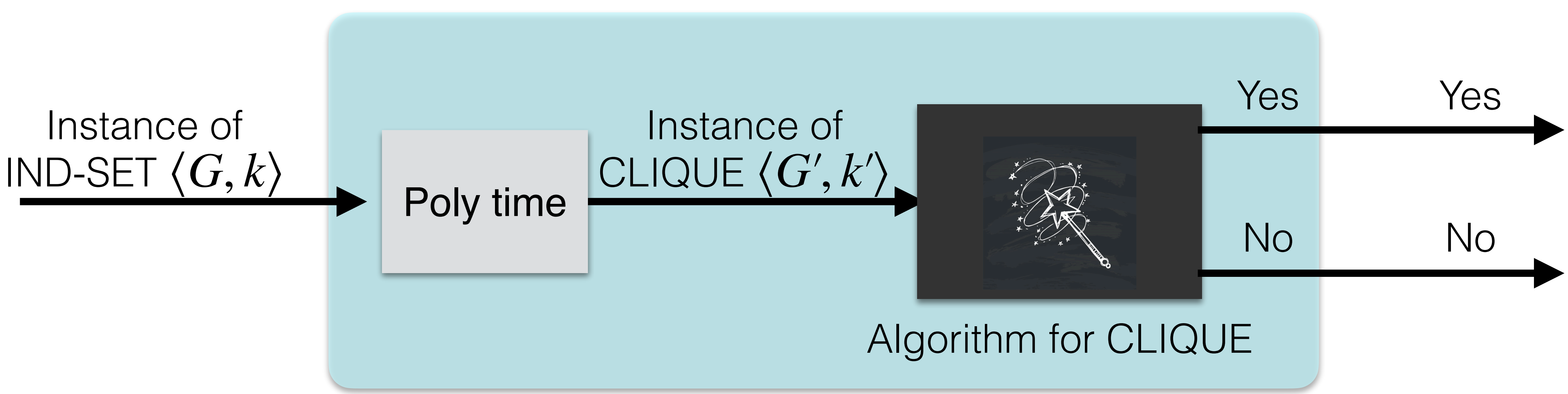
Clique

- A **clique** in an undirected graph is a subset of nodes such that every two nodes are connected by an edge. A k-clique is a clique that contains k nodes.
- **CLIQUE.** Given a graph G and a number k, does G contain a k -clique?
- **CLIQUE** $\in$ NP
  - Certificate: a subset of vertices
  - Poly-time verifier: check is each pair of vertices have an edge between them and if size of subset is k



IND-SET to CLIQUE

- **Theorem.** **IND-SET $\leq_p$ CLIQUE.**
- **In class exercise.** Reduce IND-SET to Clique. Given instance $\langle G, k \rangle$ of independent set, construct an instance $\langle G', k' \rangle$ of clique such that
  - G has independent set of size k iff G' has clique of size k'.



Algorithm for IND-SET

IND-SET to CLIQUE

- **Theorem.** IND-SET $\leq_p$ CLIQUE.
- Proof. Given instance $\langle G, k \rangle$ of independent set, we construct an instance $\langle G', k' \rangle$ of clique such that G has independent set of size k iff G' has clique of size k'
- **Reduction**.
  - Let $G' = (V, \overline{E})$, where $e = (u, v) \in \overline{E}$ iff $e \notin E$ and $k' = k$
  - $(\Rightarrow)$ G has an independent set S of size k, then S is a clique in G'
  - $(\Leftarrow)$ G' has a clique Q of size k, then Q is an independent set in G

Reductions: General Pattern

- Describe a polynomial-time algorithm to transform an arbitrary instance x of Problem X into a special instance y of Problem Y
- Prove that:
  - If x is a “yes” instance of X, then y is a “yes” instance of Y
  - If y is a “yes” instance of Y, then x is a “yes” instance of X

    $\iff$ if x is a "no" instance of X, then y is a "no" instance of Y