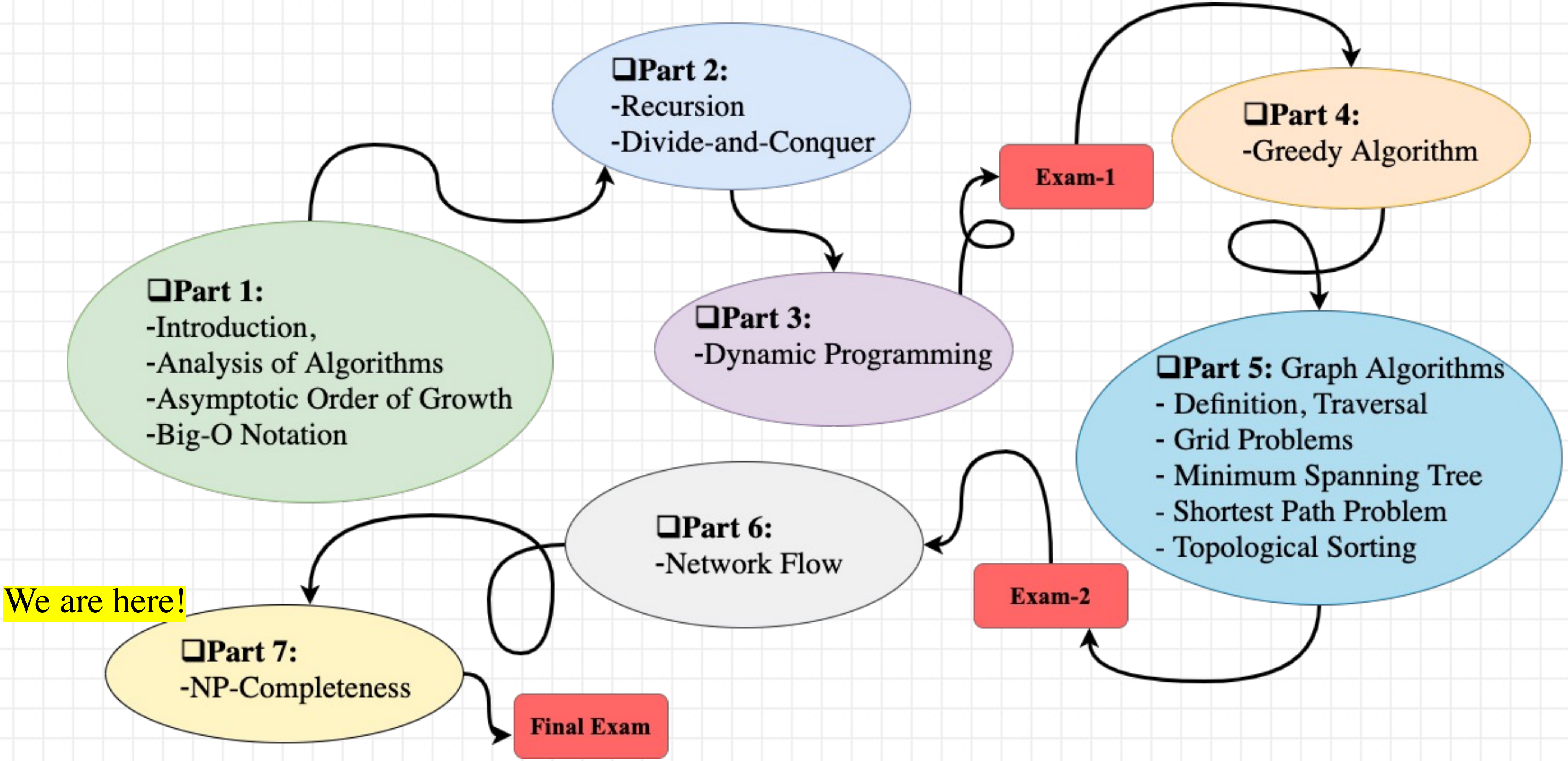
CS-3510: Design and Analysis of Algorithms

NP Completeness I

Instructor: Shahrokh Shahi

College of Computing
Georgia Institute of Technology
Summer 2022

Roadmap



❑Part 2:
-Recursion
-Divide-and-Conquer
Exam-1
❑Part 4:
-Greedy Algorithm
❑Part 1:
-Introduction,
-Analysis of Algorithms
-Asymptotic Order of Growth
-Big-O Notation
❑Part 3:
-Dynamic Programming
❑Part 5: Graph Algorithms
- Definition, Traversal
- Grid Problems
- Minimum Spanning Tree
- Shortest Path Problem
- Topological Sorting
❑Part 6:
-Network Flow
We are here!
Exam-2
❑Part 7:
-NP-Completeness
Final Exam

NP-Completeness

- So far, we have seen a lot of good news!
  - Problems can be solved quickly/efficiently (i.e., linear time, or at least a time that is some small polynomial function of the input size)

- NP-completeness is a form of bad news!
  - There exist many important problems that cannot be solved quickly.

- NP-complete problems really come up all the time!

NP-Completeness

- Why should we care?
  - Knowing that they are hard lets you stop beating your head against a wall trying to solve them!

  - **Restrict the problem:** find special restrictions/variants to the problem for which there is a polynomial time algorithm.
  - **Use a heuristic:** come up with a method for solving a reasonable fraction of the common cases.
  - **Solve approximately:** come up with a method that finds solutions provably close to the optimal.
  - **Use an exponential time solution:** if you really have to solve the problem exactly and stop worrying about finding a better solution.

Optimization vs. Decision Problems

- **Decision problems**
  - Given an input and a question regarding a problem, determine if the answer is yes or no

- **Optimization problems**
  - Find a solution with the “best” value

- Optimization problems can be cast as decision problems that are easier to study

Optimization vs. Decision Problems

- **Casting optimization problems as decision problems**
- Ex.
  - Problem: shortest path problem in unweighted graphs
  - Optimization problem: Find a path between u and v that uses the fewest edges
  - Decision problem: Does a path exist from u to v consisting of at most k edges?

Minimum / least / shortest /… → at most k
Maximum/ greatest / longest / …→ at least k

Class "P"

- **Class P** consists of [decision] problems that are solvable in polynomial time
- Recall from the first lecture:
  - [slide #36] <u>Polynomial time</u> → Running time is $O(n^k)$ for some constant $k > 0$.
  - Examples
    - Linear search O(n)
    - Dynamic programming solutions (O(n), $O(n^2)$, $O(n^3)$, …)
    - Sorting ($O(n^2)$, O(nlogn))
    - Divide-and-conquer solutions
    - Graph algorithms O(n+m), O(mlogn), …
  - Non-polynomial time → $O(2^n), O(a^n), O(n!), O(n^n), \ldots$

Class "P"

- **Class P** consists of [decision] problems that are solvable in polynomial time
- Recall from the first lecture:
  - [slide #36] Polynomial time → Running time is $O(n^k)$ for some constant $k > 0$.
  - Examples
    - Linear search O(n)
    - Dynamic programming solutions (O(n), O(n^2) , O(n^3), …)
    - Sorting (O(n^2), O(nlogn))
    - Divide-and-conquer solutions
    - Graph algorithms O(n+m), O(mlogn), …
  - Non-polynomial time → $O(2^n), O(a^n), O(n!), O(n^n), \ldots$

Problems in P are
Considered/called tractable

Problem not in P are
intractable

Class "P"

- **Class P** consists of [decision] problems that are solvable in polynomial time
  - Problems in P are Considered/called tractable
  - Problem not in P are intractable

- Note this does not mean that non-polynomial algorithms are always worst than polynomial algorithms!
  - $O(n^{100\ldots0000})$ technically tractable (polynomial time), but practically impossible.
  - $O(n^{\log\log\log n})$ technically intractable, but practiclly easy to solve.
- Recall the "asymptotic" meaning of running time.

Class "NP"

- First of all: **NP does NOT stand for not-P!**

NP = Nondeterministic Polynomial

- NP is the class of problems for which a candidate solution can be verified in polynomial time.

- P is a subset of NP ($P \subseteq NP$)

Class "NP"

- First of all: **NP does NOT stand for not-P!**

NP = Nondeterministic Polynomial

- NP is the class of problems for which a candidate solution can be verified in polynomial time.

- P is a subset of NP ($P \subseteq NP$)

Class "NP"

- Nondeterministic algorithms entail a two-stage procedure:

1. Nondeterministic "guessing" stage
   - Generate randomly an arbitrary candidate solution (≡ "certificate")

2. Deterministic "verifying" stage
   - Take the certificate and the instance to the problem and returns YES if the certificate represents a solution (verifying in polynomial time)

Class "NP"

- Nondeterministic algorithms entail a two-stage procedure:

Note in NP algorithms the verification step is polynomial

1. Nondeterministic "guessing" stage
   - Generate randomly an arbitrary candidate solution (≡ "certificate")

2. Deterministic "verifying" stage
   - Take the certificate and the instance to the problem and returns YES if the certificate represents a solution (verifying in polynomial time)

Class "NP"

- Nondeterministic algorithms entail a two-stage procedure:

But what does it mean "verifying" a candidate solution?

1. Nondeterministic "guessing" stage
   - Generate randomly an arbitrary candidate solution (≡ "certificate")
2. Deterministic "verifying" stage
   - Take the certificate and the instance to the problem and returns YES if the certificate represents a solution (verifying in polynomial time)

Class "NP"

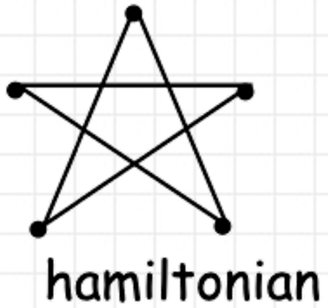
- Difference between solving a problem and verifying a candidate solution:

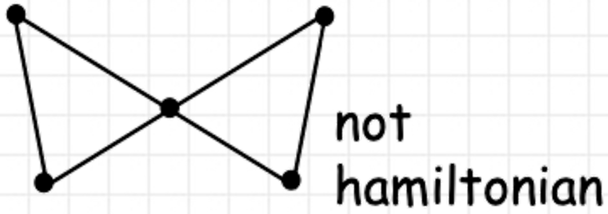
  - Solving a problem: is there a path in graph G from vertex u to vertex v with at most k edges?

  - Verifying a candidate solution: is $v_0, v_1, \ldots, v_m$ a path in graph G from vertex u to vertex v with at most k edges?

Class "NP" solving vs verifying

- Difference between solving a problem and verifying a candidate solution:
- Example:
  - A Hamiltonian cycle in an undirected graph is a cycle that visits every vertex exactly once.



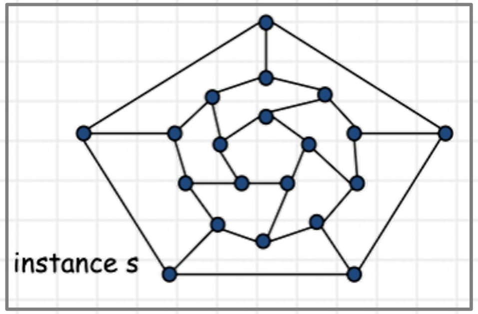


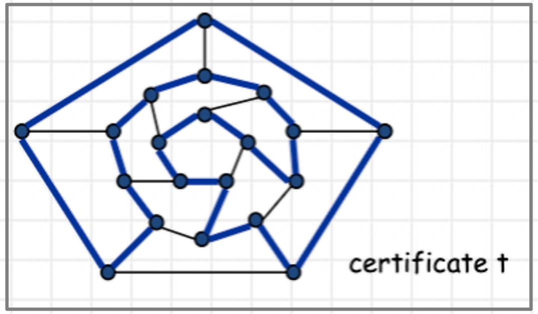
  - Solving a problem: is there a Hamiltonian cycle in graph G?
  - Verifying a candidate solution: is $v_0,\ldots, v_m$ a Hamiltonian cycle on graph G?

Class “NP” solving vs verifying

- Example:
  - A Hamiltonian cycle in an undirected graph is a cycle that visits every vertex exactly once.
  - Solving a problem: is there a Hamiltonian cycle in graph G?
  - Verifying a candidate solution: is $v_0, \ldots, v_m$ a Hamiltonian cycle on graph G?

- Certificate: A list of n nodes.
- Certifier: Check that the list contains each node in V exactly once, and that there is an edge between each pair of adjacent nodes in the permutation.
- Conclusion: HAM-CYCLE is in NP.





Class “NP” solving vs verifying

- Intuitively, solving a problem from scratch seems much harder (and more time consuming) in comparison to just verifying whether a candidate solution can solve the problem or not.

- Note if there are many candidate solutions to check, then even if each individual one is quick to check, overall, it can take a long time.

P vs. NP

- Is P = NP?
- Mentioned earlier that any problem in P is also in NP. So, P is a subset of NP ($P \subseteq NP$)
- But the big (and open) question is whether $NP \subseteq P$, and so P=NP.

P vs. NP

- Is P = NP?
- Mentioned earlier that any problem in P is also in NP. So, P is a subset of NP ($P \subseteq NP$)
- But the big (and open) question is whether $NP \subseteq P$, and so P=NP.

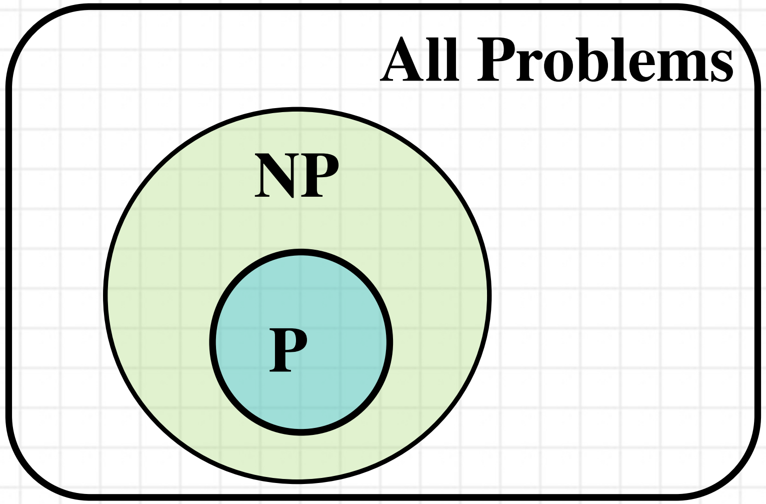
What does it mean?

P vs. NP

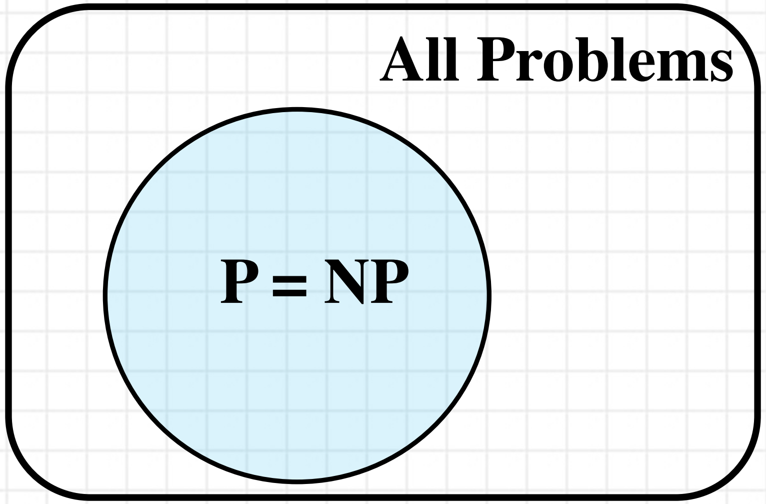
- Is P = NP?
- Mentioned earlier that any problem in P is also in NP. So, P is a subset of NP ($P \subseteq NP$)
- But the big (and open) question is whether $NP \subseteq P$, and so P=NP.
  - It means if it is always easy to check a candidate solution, should it also be easy to find a solution?
  - Answer? Most computer scientists believe that this is false, but we do not have a proof

P vs. NP

- Is P = NP?
- Answer? Most computer scientists believe that this is false, but we do not have a proof
- Therefore, there are two possibilities/beliefs:



P≠NP



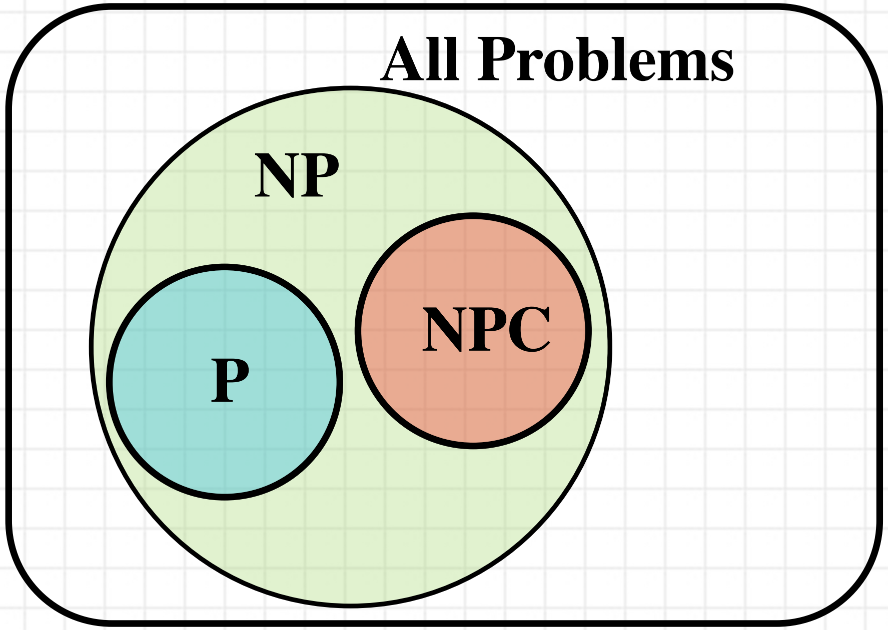
P=NP

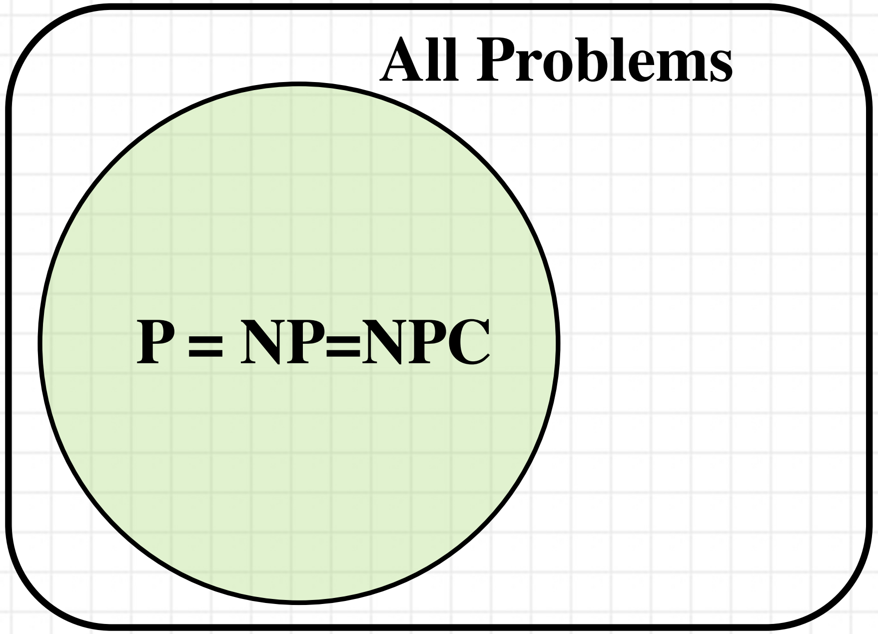
NP-Complete (NPC)

- NP-complete problems are a class of "hardest" problems in NP.
- If you can solve an NP-complete problem, then you can solve all NP problems (show later).
- Hence, if any NP-complete problem can be solved in polynomial time, then all problems in NP can be, and thus P = NP.
- Precise/formal definition coming later...

Possible Worlds

- Therefore, there are two possibilities:





P≠NP

P=NP

Reductions

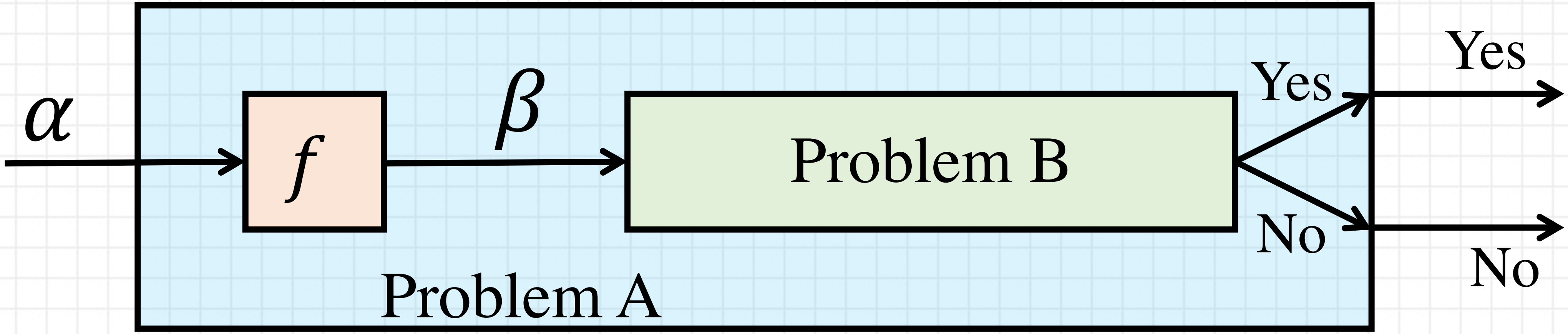
- Reduction from A to B is showing that we can solve A using the algorithm that solves B

- We say that problem A is easier than problem B, and
- We write $A \leq B$

Reductions

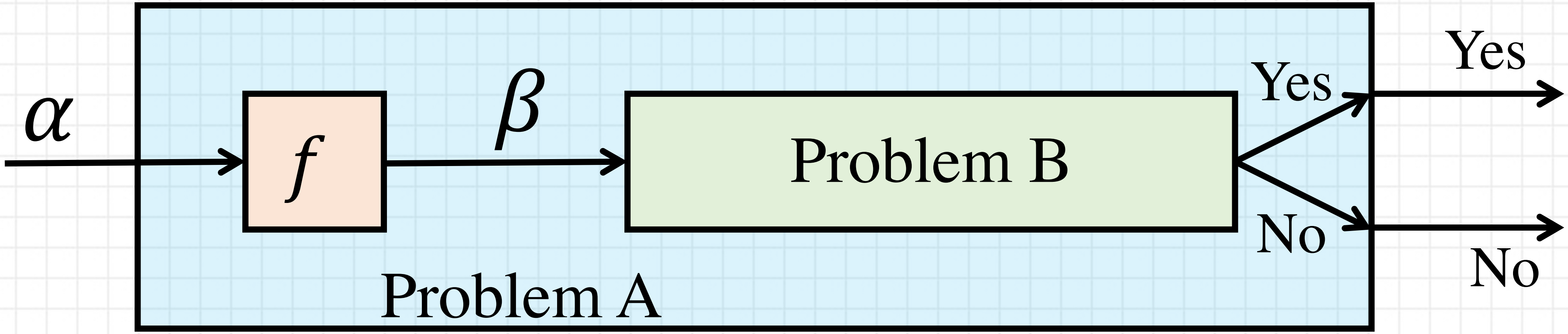
- Reduction from A to B is showing that we can solve A using the algorithm that solves B

- We say that problem A is easier than problem B, and
- We write $A \leq B$



Reductions

- $A \leq B$: Reduction from A to B is showing that we can solve A using the algorithm that solves B
- If we have an oracle for solving B, then we can solve A by making polynomial number of computations and polynomial number of calls to the oracle for B
- We can transform the inputs of A to inputs of B



Reductions

- Before discussing further regarding reduction in NPC, note that we can also do reductions on polynomial time (poly-time) algorithms.

- Examples:
  - Transforming a given problem to a graph, and solving the problem using graph algorithms (for example, SCC)

  - Solving all-pairs shortest path problem using multiple (polynomial number of) calls to Dijkstra's algorithm

Polynomial Reductions

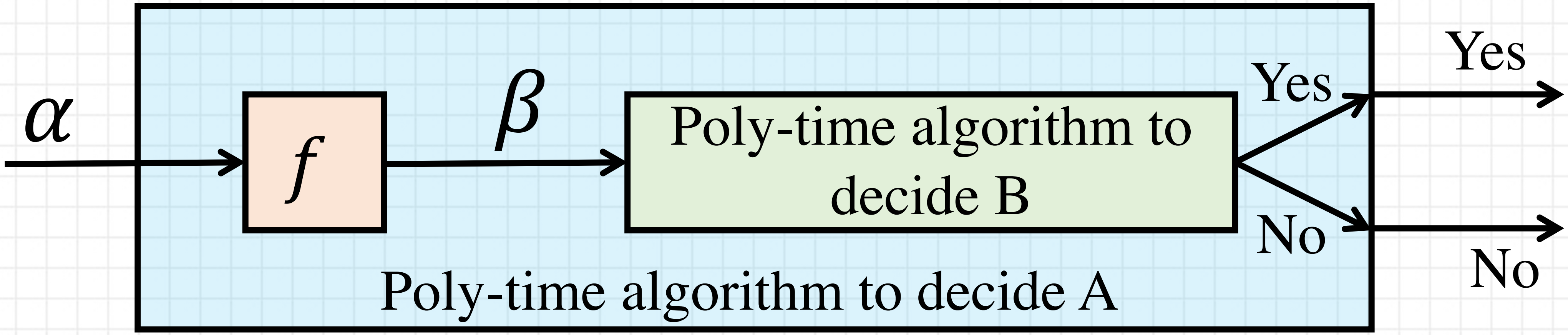
- Given two problems, A and B, we say that A is polynomially reducible to B, and write it as $A \leq_p B$ if:

1. There exists a function f that converts the input of A **to** inputs of B in polynomial time

2. $A(i) = \text{YES} \Leftrightarrow B\big(f(i)\big) = \text{YES}$

Proving Polynomial Time

1. Use a polynomial time reduction algorithm to transform A into B.
2. Run a known polynomial time algorithm for B.
3. Use the answer for B as the answer for A.



Implications of Polynomial-Time Reductions

- **Purpose**. Classify problems according to relative difficulty.

- **Design algorithms**. If $X \leq_p Y$ and Y can be solved in polynomial-time, then X can also be solved in polynomial time.

- **Establish intractability**. If $X \leq_p Y$ and X cannot be solved in polynomial-time, then Y cannot be solved in polynomial time.

- **Establish equivalence**. If $X \leq_p Y$ and $Y \leq_p X$, we use notation $X \equiv_p Y$.

- **Transitivity**. If $X \leq_p Y$ and $Y \leq_p Z$,then $X \leq_p Z$.

Reductions Strategies

- Given two problems, A and B, we say that A is polynomially reducible to B, and write it as $A \leq_p B$ if:
  1. There exists a function f that converts the input of A **to** inputs of B in polynomial time
  2. $A(i) = \text{YES} \iff B\big(f(i)\big) = \text{YES}$

- Reductions Strategies
  - Reduction by simple equivalence.
  - Reduction from a special case to a general case.
  - Reduction by encoding with gadgets.

Example

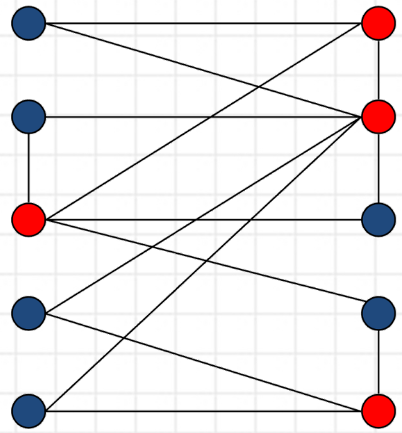
- We want to show the problem VERTEX-COVER is polynomially reducible to the SET-COVER problem, i.e.,

$$\texttt{VERTEX-COVER} \leq_P \texttt{SET-COVER}$$

- `VERTEX-COVER` problem?
- `SET-COVER` problem?
- Reduction process?

Example

- Vertex Cover
  - MINIMUM VERTEX COVER: Given a graph G = (V, E), **find the smallest** subset of vertices $S \subseteq V$, such that for each edge at least one of its endpoints is in S?
  - VERTEX COVER: Given a graph G = (V, E) and an integer k, **is there a subset** of vertices $S \subseteq V$ such that $|S| \leq k$, and for each edge, at least one of its endpoints is in S?



  - Ex. Is there a vertex cover of size ≤ 4? Yes.
  - Ex. Is there a vertex cover of size ≤ 3? No.

Example

- Vertex Cover
  - SET COVER: Given a set U of elements, a collection $S_1, S_2, \ldots, S_m$ of subsets of U, and an integer k, does there exist a collection of k of these sets whose union is equal to U?
- Ex. U = { 1, 2, 3, 4, 5, 6, 7 } k=2
  - $S_1=\{3,7\}$
  - $S_2=\{3,4,5,6\}$
  - $S_3=\{1\}$
  - $S_4=\{2,4\}$
  - $S_5=\{5\}$
  - $S_6= \{1,2,6,7\}$

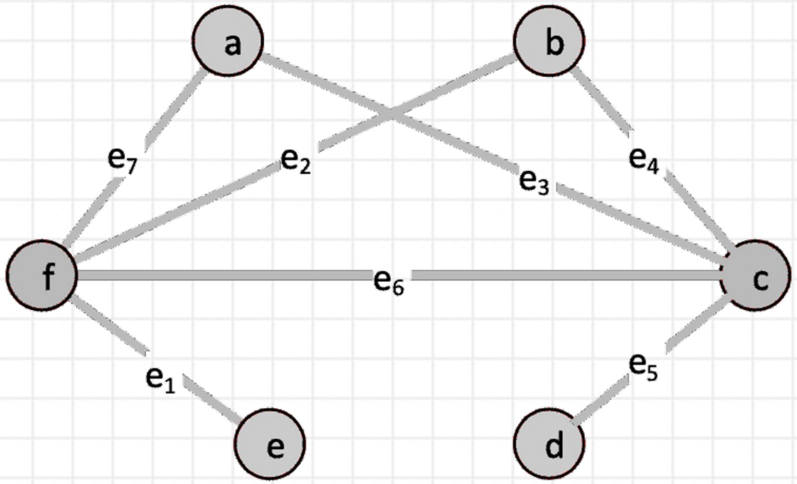
Example

- **We want to show the problem VERTEX-COVER is polynomially reducible to the SET-COVER problem.**
- **Theorem**. `VERTEX-COVER` $\leq_P$ `SET-COVER`
- **Proof**. Given a `VERTEX-COVER` instance $G = (V, E)$ and k, we construct a `SET-COVER` instance (U, S, k) that has a set cover of size k iff G has a vertex cover of size k.
- **Construction**.
  - Universe $U = E$.
  - Include one subset for each node $v \in V$: $S_v = \{e \in E: e \text{ incident to } v\}$.
  - The transformation takes linear time on the size of the VC instance.

Example

- **Construction.**
  - Universe $U = E$.
  - Include one subset for each node $v \in V$: $S_v = \{e \in E: e \text{ incident to } v\}$.
  - The transformation takes linear time on the size of the VC instance.

k = 2



vertex cover instance (k = 2)

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

$S_a = \{ 3, 7 \}$	$S_b = \{ 2, 4 \}$
$S_c = \{ 3, 4, 5, 6 \}$	$S_d = \{ 5 \}$
$S_e = \{ 1 \}$	$S_f = \{ 1, 2, 6, 7 \}$

set cover instance (k = 2)

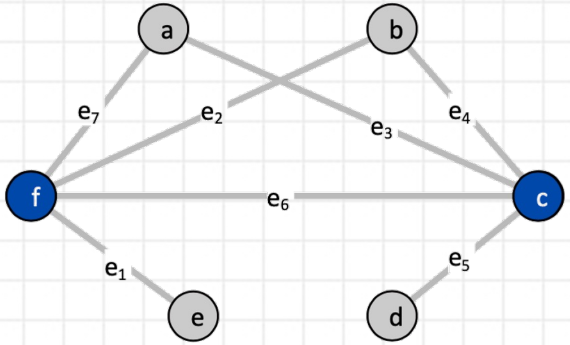
Example

- **Lemma**. $G = (V, E)$ contains a vertex cover of size k iff (U, S, k) contains a set cover of size k.

That is, VC(i) = yes $\Leftrightarrow$ SC(f(i)) = yes

- **Proof**. ($\Rightarrow$)
  - Let $X \subseteq V$ be a vertex cover of size k in G.
  - Then, $Y = \{S_v : v \in X\}$ is a set cover of size k.

k = 2



vertex cover instance (k = 2)

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

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

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

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

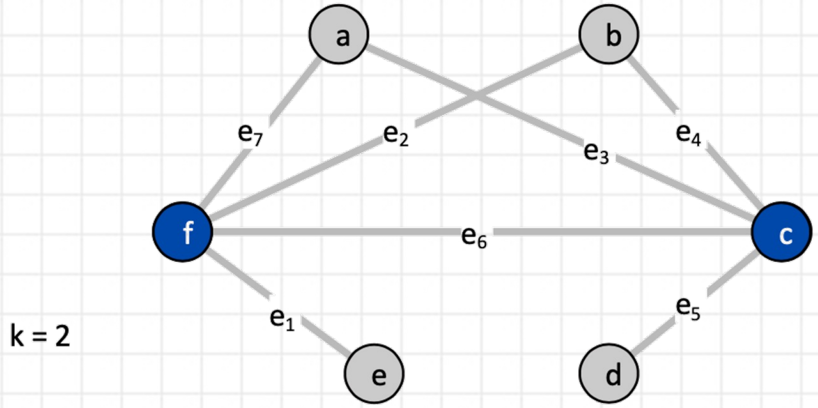
set cover instance (k = 2)

Example

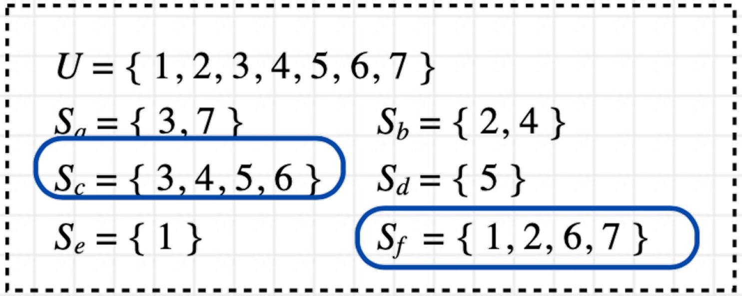
- **Lemma**. VC(i) = yes ⟺ SC(f(i)) = yes
- **Proof**. (⇒) VC(i) = yes ⇒ SC(f(i)) = yes
  - VC(i) is a yes instance ⟹ it has a solution; let V' ⊆ V be such a solution |V'| ≤ k, every edge has at least one end point in V'
  - Consider V' = $\{i_1, i_2, i_3, \ldots, i_l\}$, $l \leq k$, and therefore, A = $\{S_{i_1}, S_{i_2}, \ldots, S_{i_l}\}$.
  - For the sake of contradiction assume A is not a solution to SC(f(i)):
    - The number of sets in A is $l \leq k$. Thus, it must be the case that $S_{i_1} \cup S_{i_2} \cup \ldots \cup S_{i_l} \neq U$
    - This means there is at least an edge $e \in U$ that is not in $S_{i_1} \cup S_{i_2} \cup \ldots \cup S_{i_l}$.
    - This e is also corresponds to an edge in VC(i), $e = (u, v)$, so S_u and S_v are not in A, i.e., $S_u, S_v \notin A \Rightarrow$ u, v $\notin V'$ (by construction of A)
    - This means $e = (u, v)$ would not have been covered by V'
    - So, V' is not solution to VC, which is a contradiction.

Example

- **Lemma**. VC(i) = yes ⇔ SC(f(i)) = yes
- **Proof**. (⇐) (VC(i) = yes ⇐ SC(f(i)) = yes) Or (SC(f(i)) = yes ⇒ VC(i) = yes)
  - Let $Y \subseteq S$ be a set cover of size k in (U, S, k)
  - Then, $X = \{v\text{: } S_v \in Y\}$ is a vertex cover of size k in G.



vertex cover instance (k = 2)



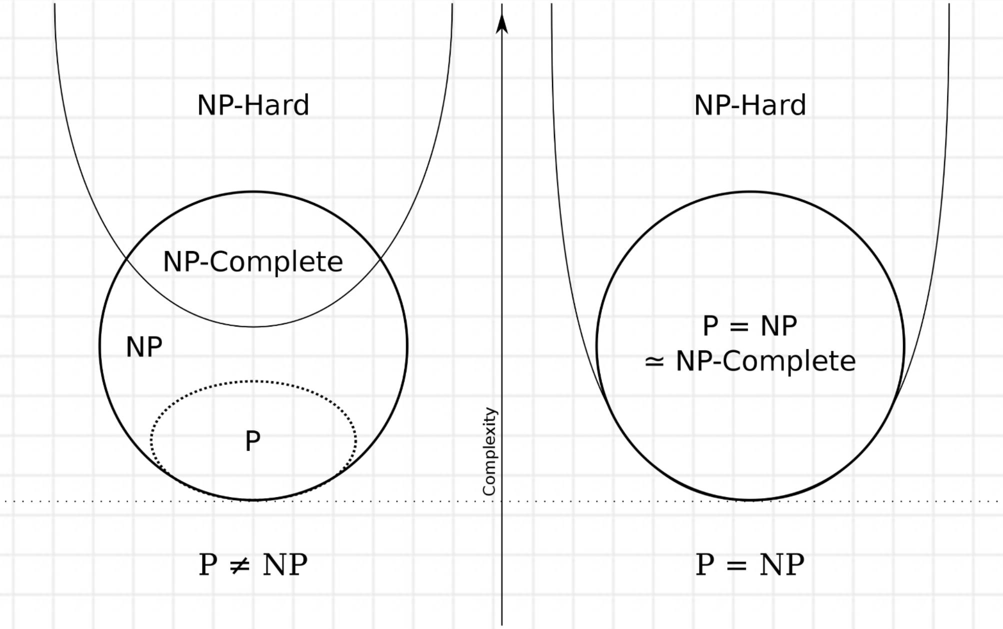
set cover instance (k = 2)

Example

- **Lemma**. VC(i) = yes ⇔ SC(f(i)) = yes
- **Proof**. (⇐) (VC(i) = yes ⇐ SC(f(i)) = yes) Or (SC(f(i)) = yes ⇒ VC(i) = yes)
  - SC(f(i)) is a yes instance ⟹ it has a solution; let A = $\{S_{i_1}, S_{i_2}, \ldots, S_{i_l}\}$ be such a solution.
  - ⟹ l ≤ k and $S_{i_1} \cup S_{i_2} \cup \ldots \cup S_{i_l} = U$ (by definition of SC)
  - Consider the vertex set V' = $\{i_1, i_2, i_3, \ldots, i_l\}$
  - For the sake of contradiction assume V' is not a solution to VC(i):
    - The number of vertices in V' is l ≤ k. Thus, it must be breaking the edge covering requirement of VC.
    - Therefore, there must be at least an edge $e \in E$ such that u $\notin V'$, v $\notin V'$
    - This implies S_u and S_v were not included in solution A.
    - By construction of f(i), $e = (u, v) \in U$, and S_u, S_v were the only sets containing e.
    - Thus, $e \notin S_{i_1} \cup S_{i_2} \cup \ldots \cup S_{i_l}$, i.e., e is not covered by the solution set A. So, A is not a solution (Contradiction)
    - ⟹ V' is a solution to VC(i)

NP-Completeness (Formal Definition)

- A problem Y is **NP-hard** if $X \leq_p Y$ for all $X \in$ **NP**
  - A problem is NP-hard if and only if a polynomial-time algorithm for it implies a polynomial-time algorithm for every problem in NP
  - NP-hard problems are at least as hard as any NP problem
- A problem Y is **NP-complete** if:
  1. $Y \in$ **NP**
  2. Y is **NP-hard**



https://en.wikipedia.org/wiki/P_versus_NP_problem

Establishing NP-Completeness

- Establishing NP-completeness → using "reduction"
- Once we establish the first "natural" NP-complete problem, others fall like dominoes!

- Recipe to establish NP-completeness of problem Y.
  - Step1. Show that Y is in NP. ($Y \in \mathbf{NP}$)
  - Step 2. Choose an NP-complete problem X.
  - Step 3. Prove that $X \leq_p Y$ (poly-time reduction).

Establishing NP-Completeness

- Establishing NP-completeness → using "reduction"
- Once we establish the first "natural" NP-complete problem, others fall like dominoes!

- Recipe to establish NP-completeness of problem Y.
  - Step1. Show that Y is in NP. ($Y \in \mathbf{NP}$)
  - Step 2. Choose an NP-complete problem X.
  - Step 3. Prove that $X \leq_p Y$ (poly-time reduction).

Why does it work?

Establishing NP-Completeness

- Recipe to establish NP-completeness of problem Y.
  - Step1. Show that Y is in NP. ($Y \in \mathbf{NP}$)
  - Step 2. Choose an NP-complete problem X.
  - Step 3. Prove that $X \leq_p Y$ (poly-time reduction).

- Justification: If X is an NP-complete problem, and Y is a problem in NP with the property that $X \leq_P Y$ then Y is NP-complete.
- Proof.
  - Let W be any problem in NP. Then, $W \leq_P X \leq_P Y$
  - By transitivity, $W \leq_P Y$.
  - Hence, Y is NP-complete.

by definition of NPC

Establishing NP-Completeness

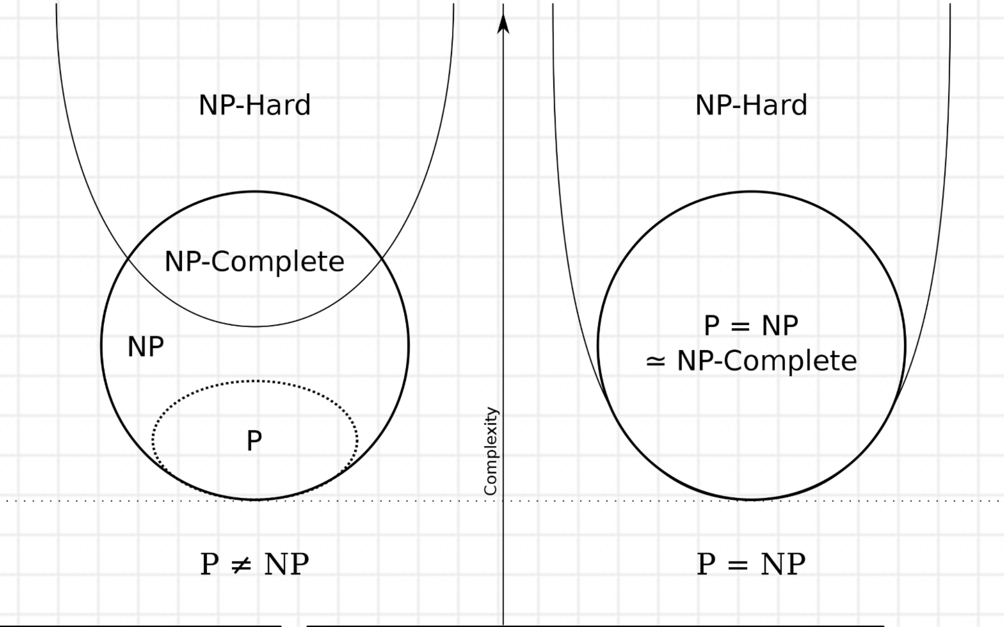
- Recipe to establish NP-completeness of problem Y.
  - Step1. Show that Y is in NP. ($Y \in \mathbf{NP}$)
    - Describe how a potential solution will be represented
    - Describe a procedure to check whether the potential solution is a correct solution to the problem instance, and argue that this procedure takes polynomial time
  - Step 2. Choose an NP-complete problem X.
  - Step 3. Prove that $X \leq_p Y$ (X is **poly-time reducible** to Y).
    - Describe a procedure f that converts the inputs i of X to inputs of Y in polynomial time
    - Show that the reduction is correct by showing that $\boxed{X(i) = \text{YES} \Leftrightarrow Y\big(f(i)\big) = \text{YES}}$
      Note this is an "if and only if" condition, so proofs are needed for both directions.

Establishing NP-Completeness

- Important note about step 2 and 3:
  - To establish NP-completeness of problem Y, we show that some other NP-complete problem X is polynomially reducable to this algorithm.
  - Note the reduction is <u>from algorithm X to Y</u> ($X \le_P Y$), not the reverse direction!

Revisit "Is P = NP?"

- **<u>Theorem</u>**. Suppose Y is an NP-complete problem. Y is solvable in poly-time if and only if P = NP.
- <u>Proof (⇐)</u> If P=NP then Y is in P. Hence, Y can be solved in poly-time.
- <u>Proof (⇒)</u> Suppose Y can be solved in poly-time.
  - Let X be any problem in NP.
    Then, we know that X $\leq_P$ Y by definition of NP-complete and Y being NP-complete problem. Then we can solve X in poly-time by solving Y in poly-time. This implies any problem X in NP is also in P, i.e., NP ⊆ P.
  - We already know P ⊆ NP. Thus, P=NP.



https://en.wikipedia.org/wiki/P_versus_NP_problem

Examples of NPC problems

- **Shortest simple path**
  - Given a graph G = (V, E) find a **shortest** path from a source to all other vertices
  - Polynomial solution: Bellman-Ford O(VE)   (complexity class P)
- **Longest simple path**
  - Given a graph G = (V, E) find a **longest** path from a source to all other vertices
  - NP-complete

Examples of NPC problems

- **Euler tour**
  - G = (V, E) a connected, directed graph find a cycle that traverses **each edge** of G exactly once (may visit a vertex multiple times)
  - Polynomial solution O(E)

- **Hamiltonian cycle**
  - G = (V, E) a connected, directed graph find a cycle that visits **each vertex** of G exactly once
  - NP-complete

The First NPC Problem

- The satisfiability (SAT) problem was the first problem shown to be NP-complete **(Cook–Levin theorem)**

- **Satisfiability problem:** given a logical expression Φ, find an assignment of True/False values to binary variables x_i that causes Φ to evaluate to T.
- Ex. $\Phi = x_1 \lor \neg x_2 \land x_3 \lor \neg x_4$

Quick Review

- Boolean variables: take on values T (or 1) or F (or 0)
- Literal: variable or negation of a variable, e.g., $x_2, \neg x_2$

- Notation: $\neg x_2 = \overline{x_2}$ = not x_2

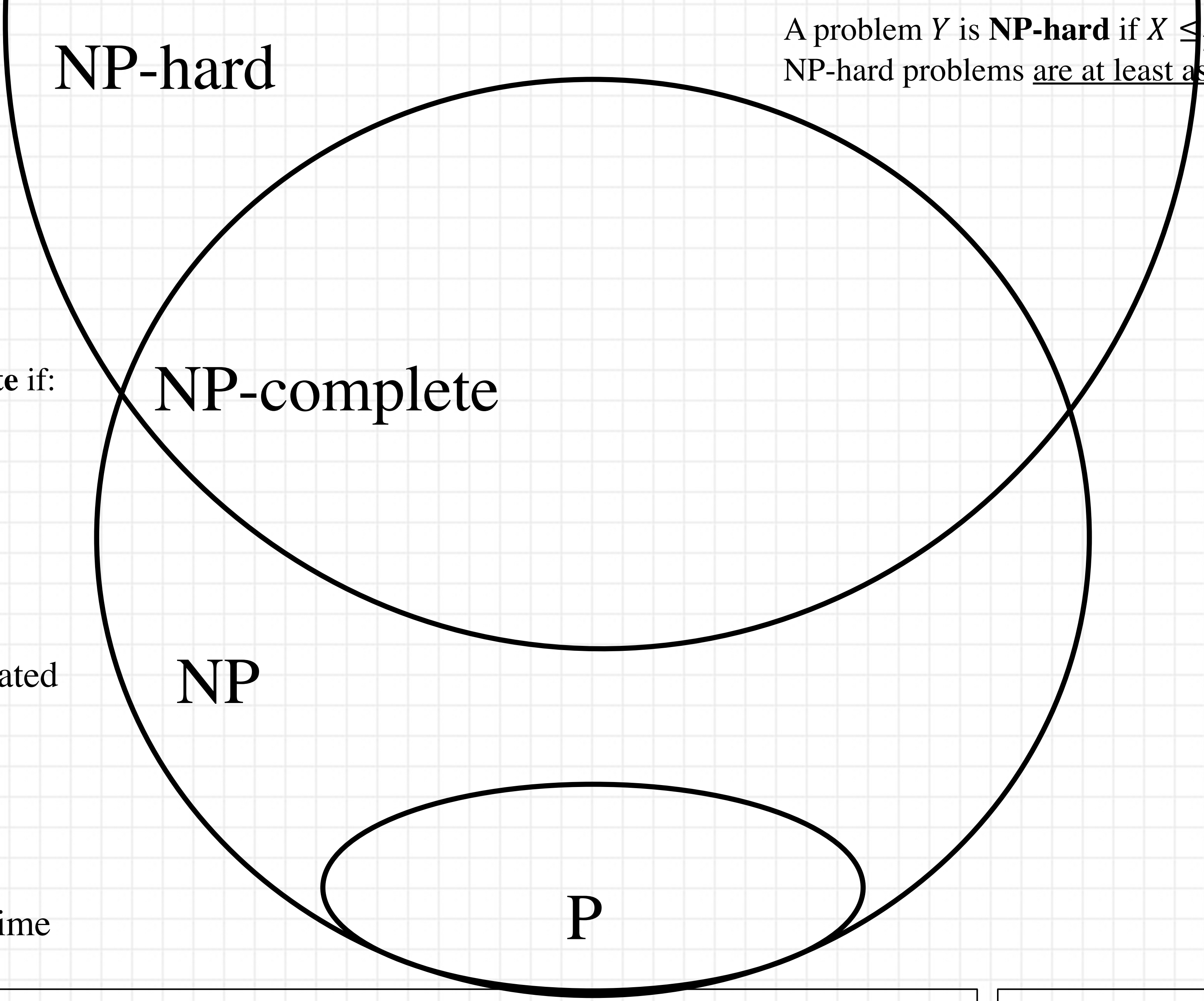
$\boldsymbol{x_1}$	$\boldsymbol{x_2}$	$\boldsymbol{x_1 \wedge x_2}$ **(AND)**	$\boldsymbol{x_1 \vee x_2}$ **(OR)**
T	T	T	T
T	F	F	T
0	T	F	T
0	F	F	F

The First NPC Problem

- **Satisfiability problem:** given a logical expression Φ, find an assignment of True/False values to binary variables x_i that causes Φ to evaluate to T.
- **SAT is in NP**: given a value assignment, check the Boolean logic of Φ evaluates to True (linear time)
- The satisfiability (SAT) problem was the first problem shown to be NP-complete **(Cook–Levin theorem)**

A problem Y is **NP-hard** if $X \leq_p Y$ for all $X \in$ **NP**

NP-hard problems are at least as hard as any NP problem



A problem Y is **NP-complete** if:

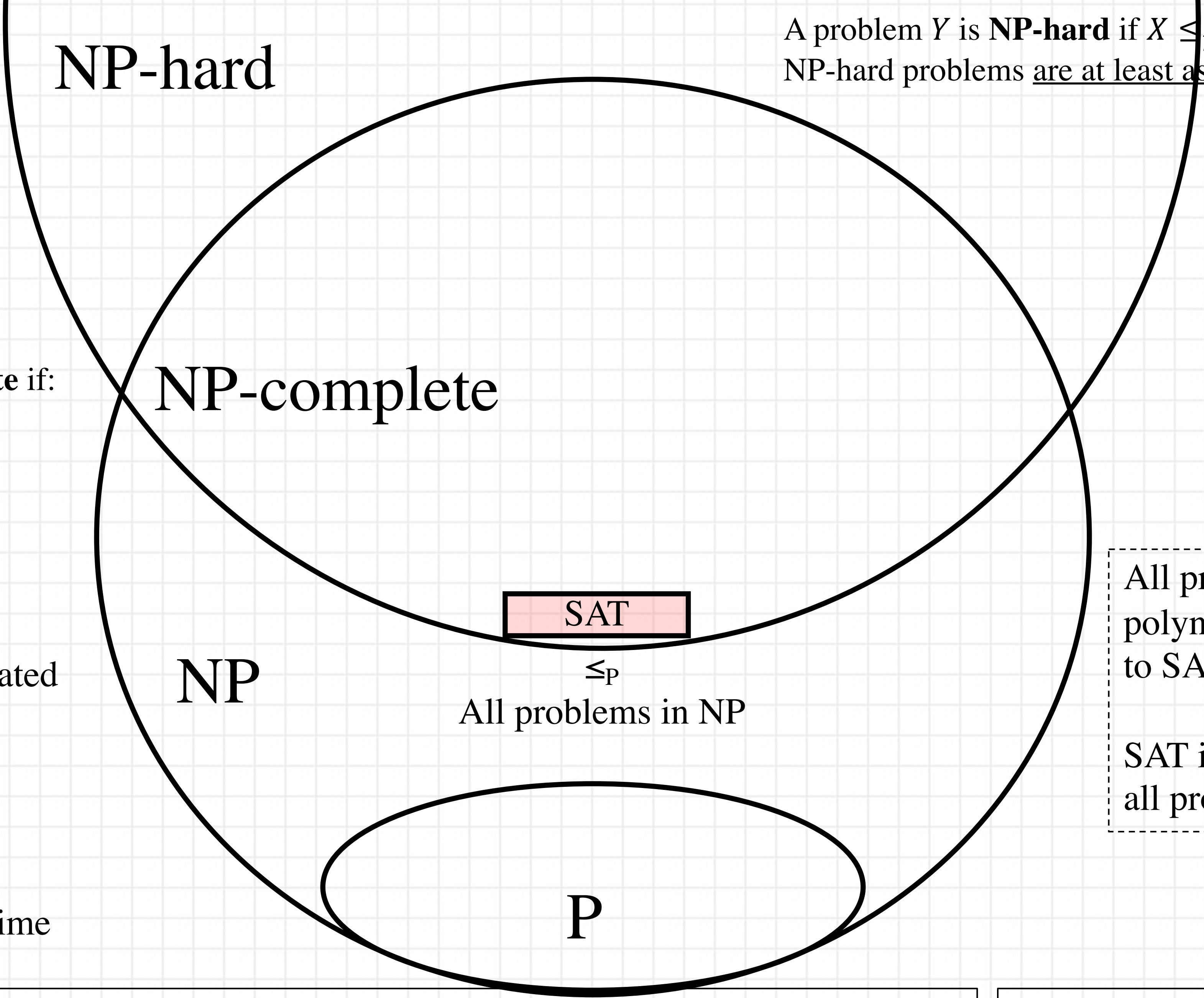
1. Y ∈ **NP**
2. Y is **NP-hard**

Candidate can be evaluated in poly-time

Solvable in poly-time

A problem Y is **NP-hard** if $X \leq_p Y$ for all $X \in$ **NP**
NP-hard problems <u>are at least as hard as any NP problem</u>

A problem Y is **NP-complete** if:
1. Y $\in$ **NP**
2. Y is **NP-hard**



Candidate can be evaluated in poly-time

Solvable in poly-time

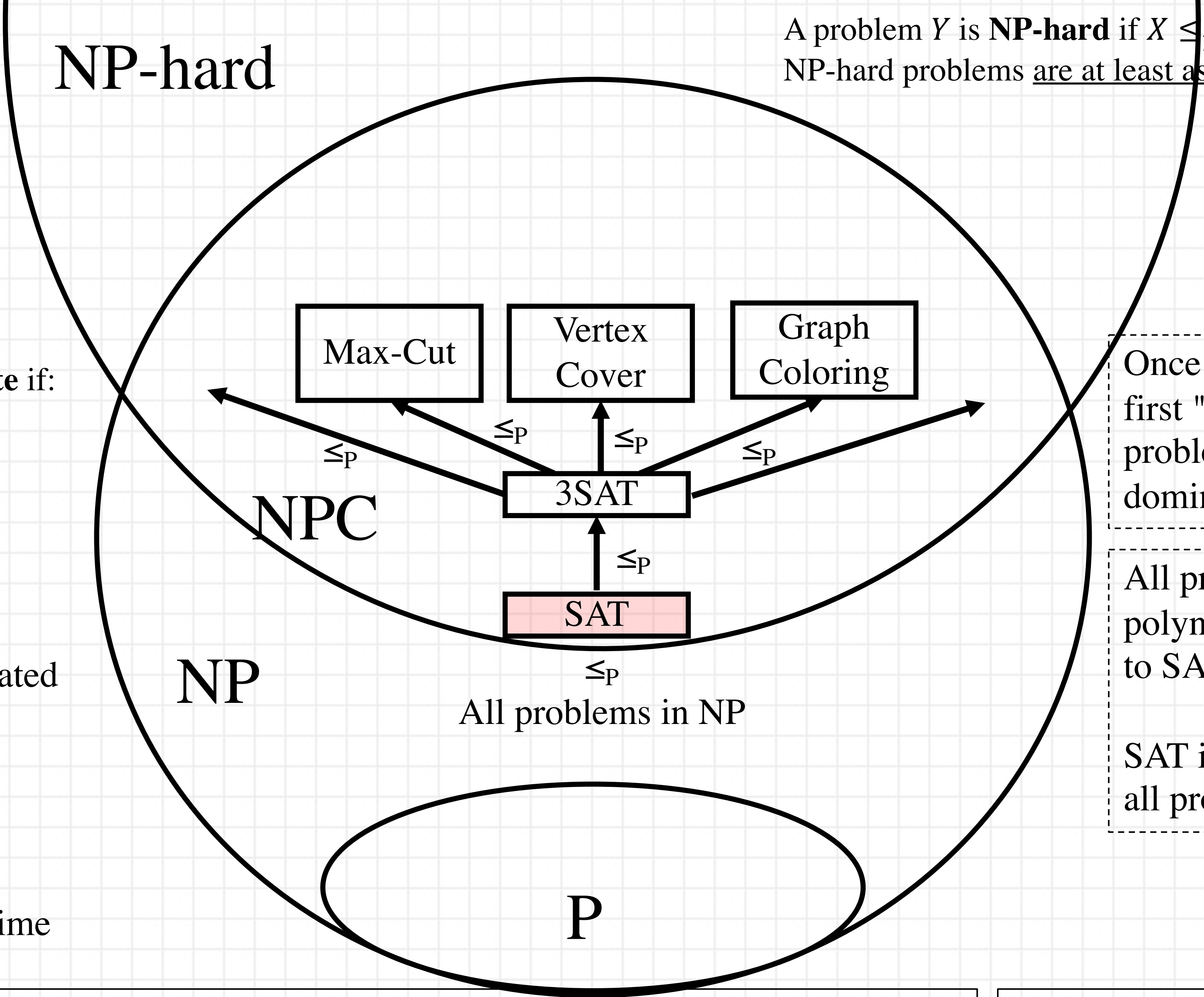
All problems in NP can polynomially be reduced to SAT

SAT is at least as hard as all problems in NP

A problem Y is **NP-hard** if $X \leq_p Y$ for all $X \in \mathbf{NP}$
NP-hard problems <u>are at least as hard as any NP problem</u>

A problem Y is **NP-complete** if:

1. Y $\in$ **NP**
2. Y is **NP-hard**



Candidate can be evaluated in poly-time

Solvable in poly-time

Once we establish the first "natural" NPC problem, others fall like dominoes!

All problems in NP can polynomially be reduced to SAT

SAT is at least as hard as all problems in NP

References

- The lecture slides are heavily based on the suggested textbooks and the corresponding published lecture notes:

  - Slides by Umit Catalyurek, Georgia Institute of Technology.
    (Based on slides by Bistra Dilkina, Anne Benoit, Jennifer Welch, George Bebis, and Kevin Wayne)
  - CLRS: Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. Introduction to Algorithms, Third Edition, MIT Press, 2009.
  - KT: Kleinberg, J., & Tardos, E. Algorithm design. Pearson/Addison-Wesley, 2006.