CSE 332: Data Structures & Parallelism

Lecture 26: Complexity Classes and Reductions

Ruth Anderson

Autumn 2025

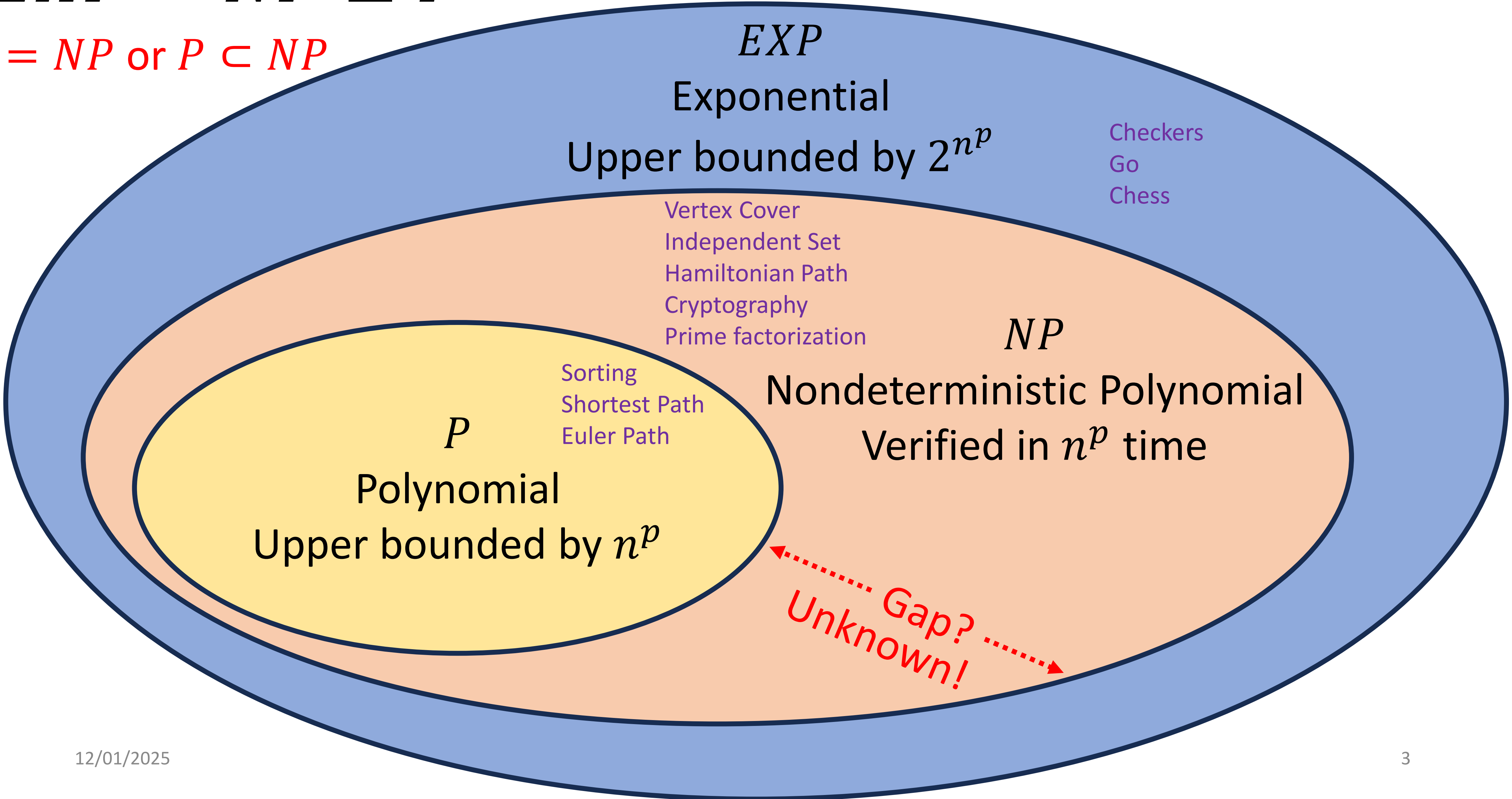
(Slides adapted from Nathan Brunelle)

Administrative

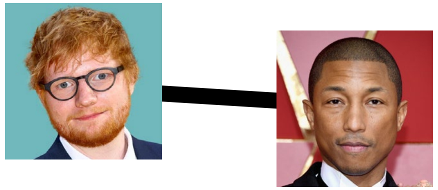
- EX11 – MSTs, programming, Due TONIGHT **Mon** Dec 1
- EX12 – P/NP, last exercise! Due Fri Dec 5 (last day of class)
  - Released later today. O.k. to use late days on EX12, will close Mon Dec 8
- Lecture on Fri Dec 5
  - Final Exam Review Session (similar to midterm review during lecture)
- Resources!
  - **Conceptual Office Hours**: 11:30 Tues (Connor) and 11:30 Wed (Samarth) both in CSE1 006. A space to ask about **course content and topics only** *as opposed to direct help with exercises*.
  - 1-on-1 Meeting Requests - Request a meeting with a staff member if you cannot make it to regularly scheduled office hours, or feel like you have an issue that requires a more in depth discussion.

$EXP \supset NP \supseteq P$

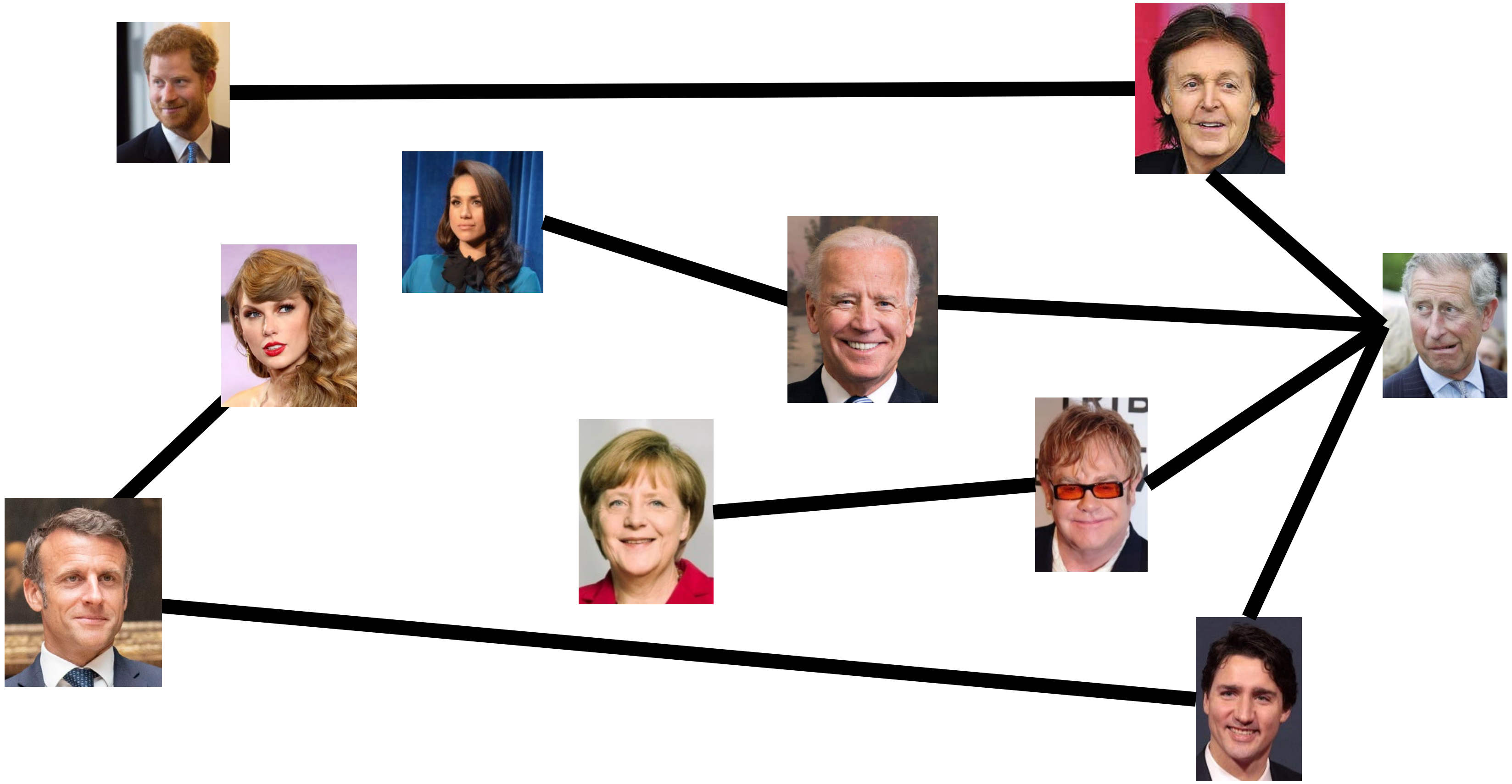
$P = NP$ or $P \subset NP$



Party Problem



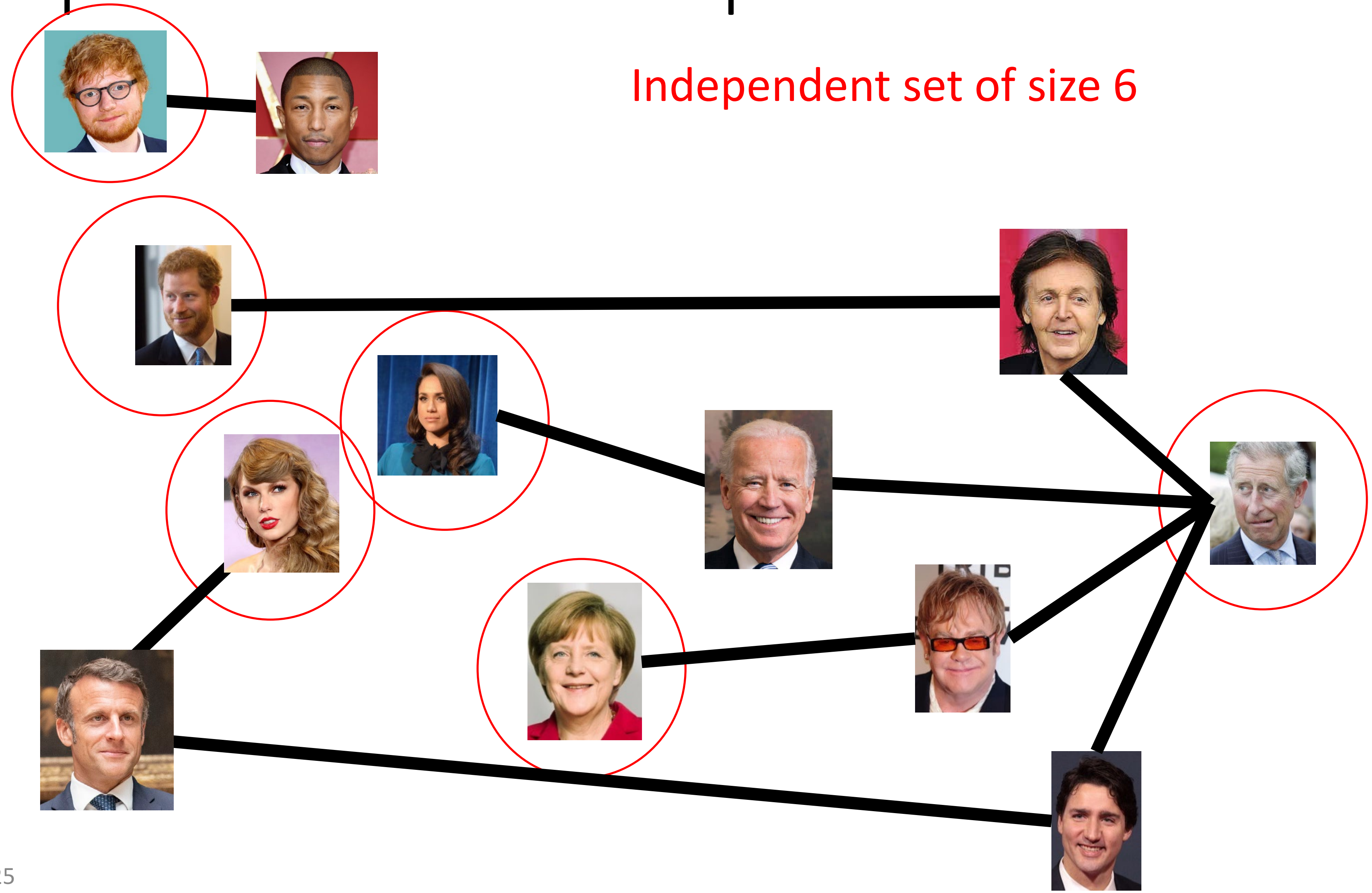
Draw Edges between people who don’t get along
How many people can I invite to a party if everyone must get along?



Independent Set

- Independent set:
  - $S \subseteq V$ is an independent set if no two nodes in S share an edge
- Independent Set Problem:
  - Given a graph $G = (V, E)$ and a number k, determine whether there is an independent set S of size k

Independent Set Example



Solving and Verifying Independent Set

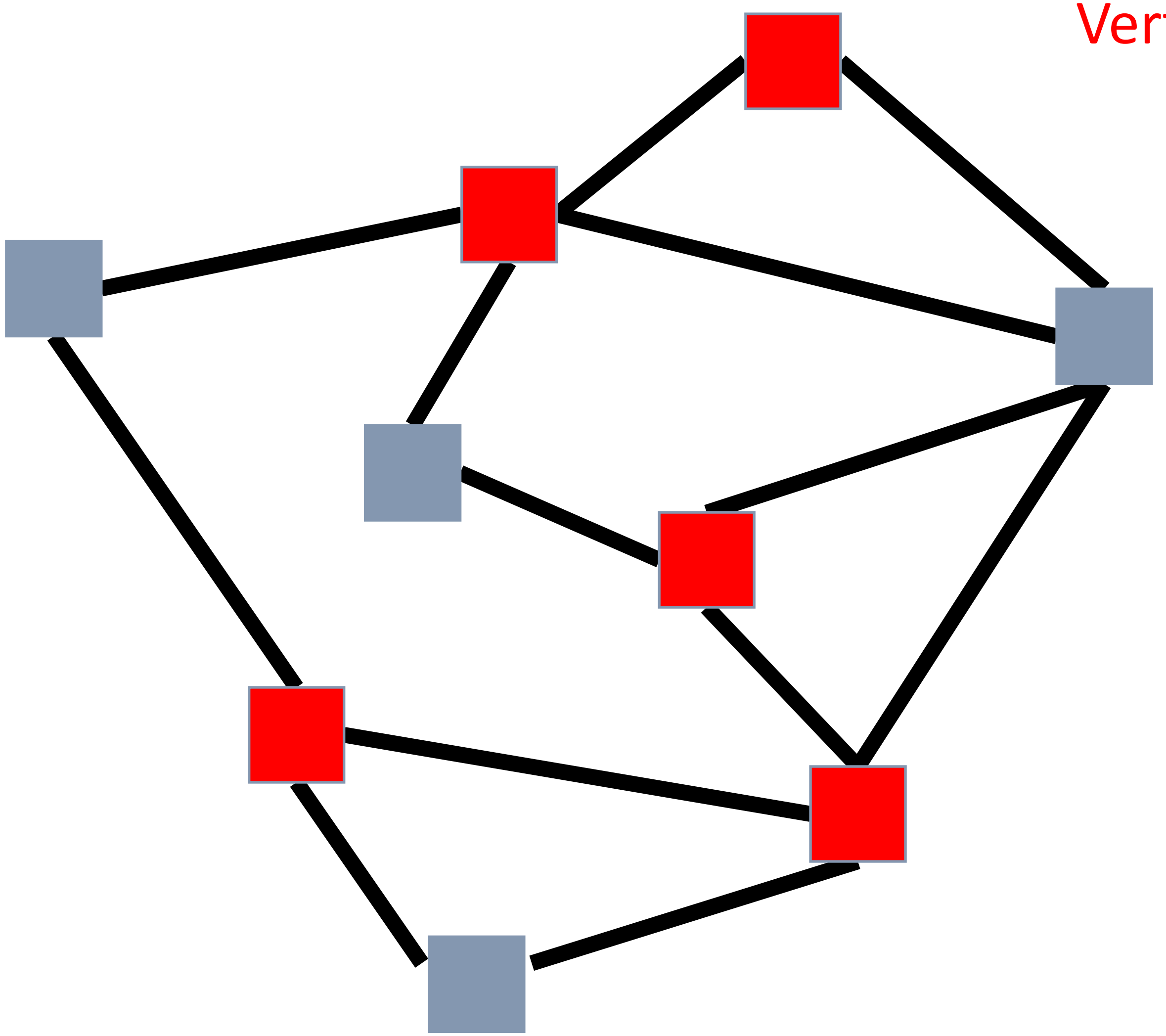
- Give an algorithm to **solve** independent set
  - Input: $G = (V, E)$ and a number k
  - Output: True if G has an independent set of size k
- Give an algorithm to **verify** independent set
  - Input: $G = (V, E)$,a number k, **and a set** $\boldsymbol{S \subseteq V}$
  - Output: True if S is an independent set of size k

Vertex Cover

- Vertex Cover:
  - $C \subseteq V$ is a vertex cover if every edge in E has one of its endpoints in C
- Vertex Cover Problem:
  - Given a graph $G = (V, E)$ and a number k, determine if there is a vertex cover C of size k

Vertex Cover Example

Vertex cover of size 5

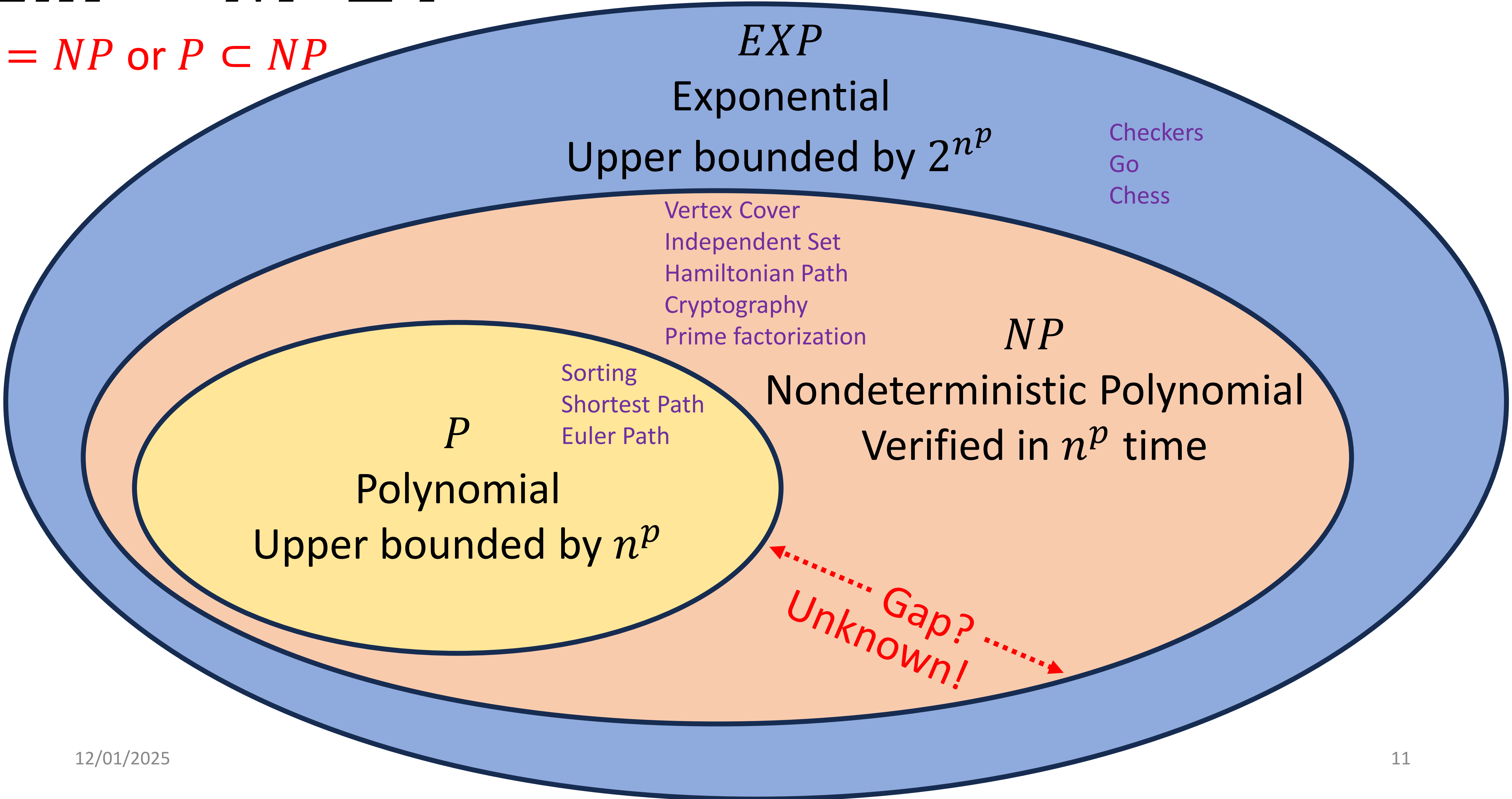


Solving and Verifying Vertex Cover

- Give an algorithm to **solve** vertex cover
  - Input: $G = (V, E)$ and a number k
  - Output: True if G has a vertex cover of size k
- Give an algorithm to **verify** vertex cover
  - Input: $G = (V, E)$,a number k, **and a set $\boldsymbol{S} \subseteq \boldsymbol{V}$**
  - Output: True if S is a vertex cover of size k

$EXP \supset NP \supseteq P$

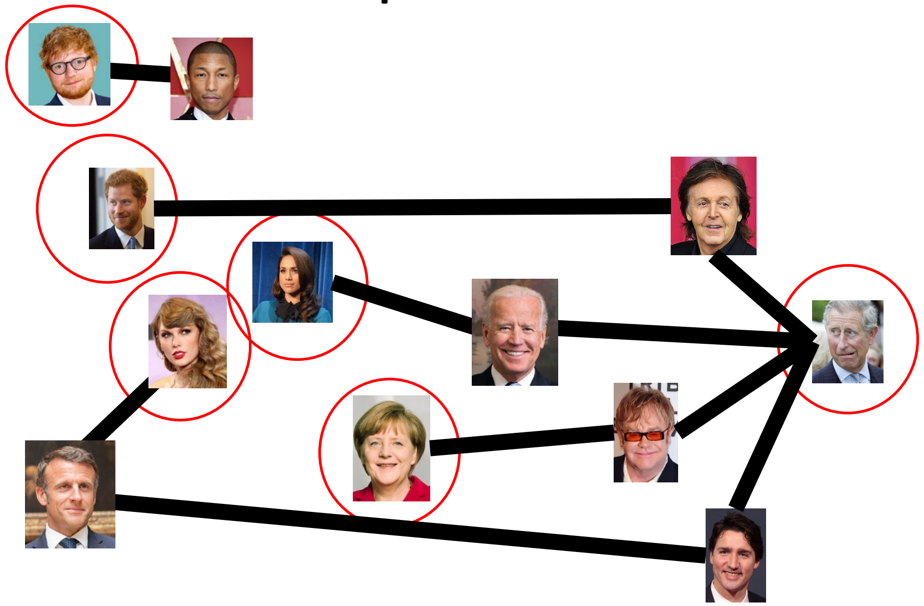
$P = NP$ or $P \subset NP$



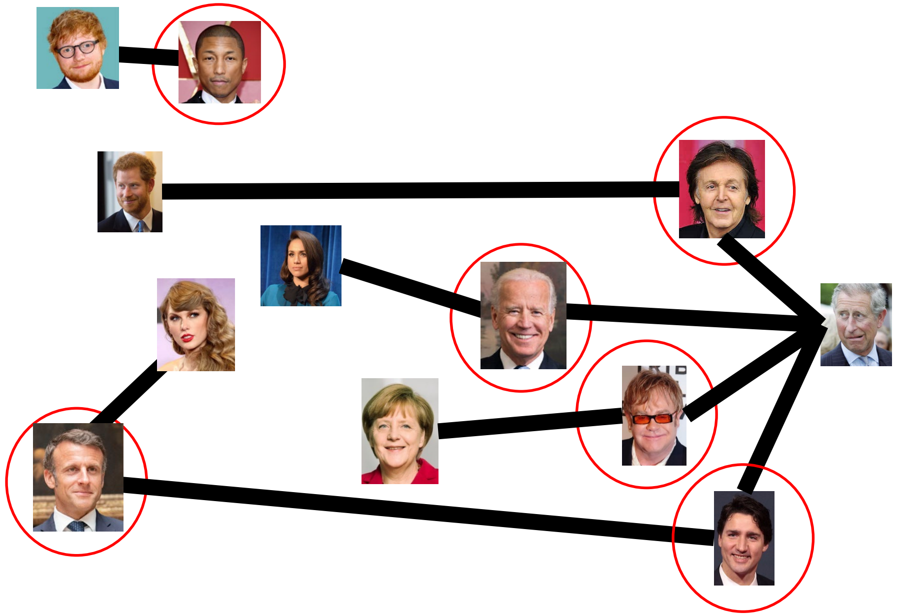
It's easy to convert an **Independent Set** into a **Vertex Cover**!

S is an **independent set** of G iff $V - S$ is a **vertex cover** of G

Independent Set



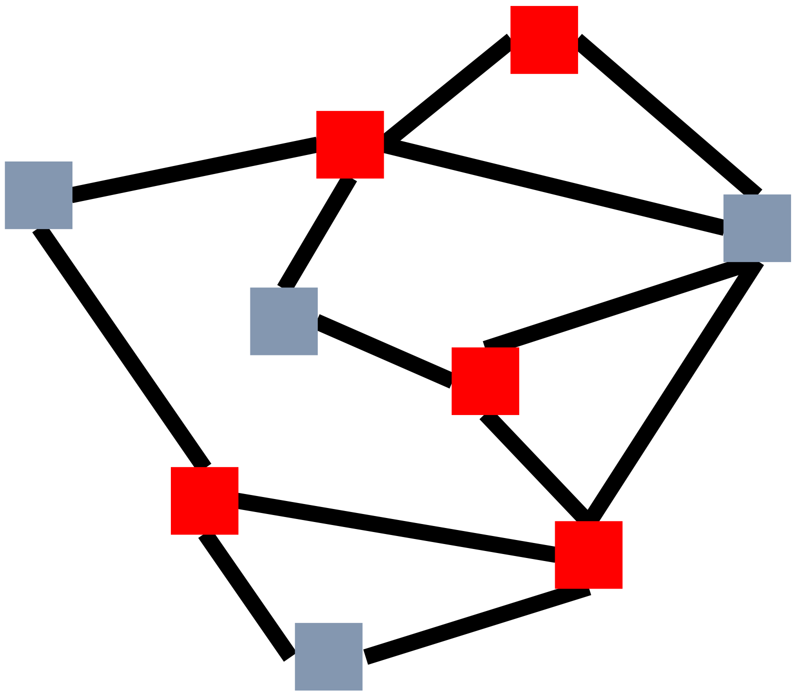
Vertex Cover



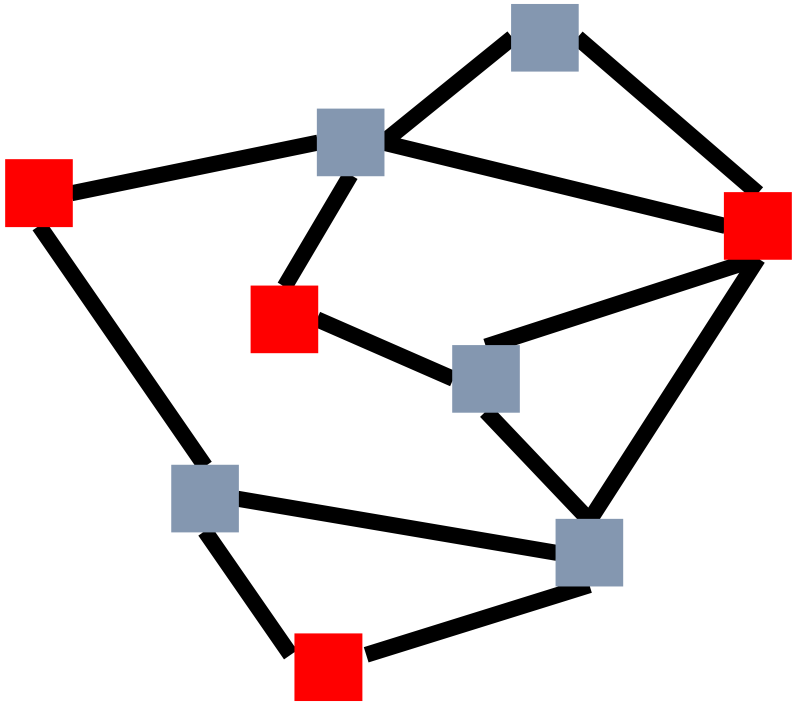
It's easy to convert a **Vertex Cover** into an **Independent Set**!

S is an **independent set** of G iff $V - S$ is a **vertex cover** of G

Vertex Cover



Independent Set



Solving Vertex Cover and Independent Set

- Algorithm to solve **vertex cover**
  - Input: $G = (V, E)$ and a number k
  - Output: True if G has a **vertex cover** of size k
    - Check if there is an **Independent Set** of G of size $|V| - k$
- Algorithm to solve **independent set**
  - Input: $G = (V, E)$ and a number k
  - Output: True if G has an **independent set** of size k
    - Check if there is a **Vertex Cover** of G of size $|V| - k$

Either both problems belong to P, or else neither does!

Reduction

A strategy for creating algorithms to solve problems by:

- taking solutions to one problem and
  using them to solve another problem.

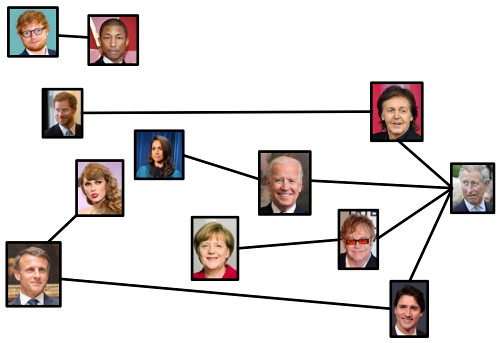
To solve **your** problem:

1. Convert it into a different problem, then
2. Use an algorithm to solve that other problem
3. Convert the result of the other problem back into
   the result for your problem

Independent Set Reduces To **Vertex Cover**

Independent Set **Input**

k



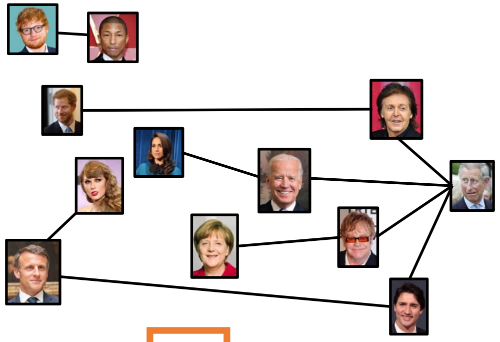
O(V) Time

Relate Independent Set input to Vertex Cover input

1) Use same graph!
2) Set $k' = |V| - k$

Vertex Cover **Input**

k'



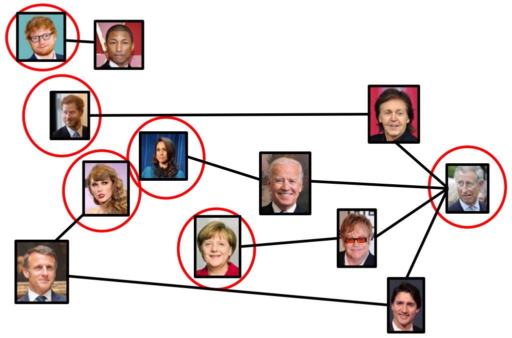
Any Algorithm for Vertex Cover

Relate Vertex Cover output to Independent Set output

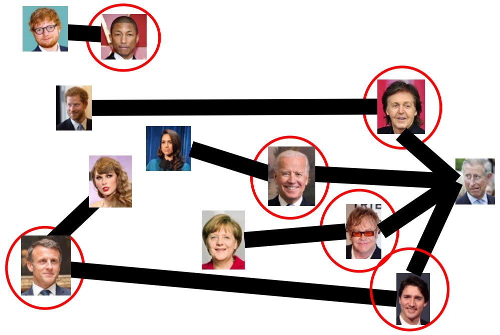
Return S = V – S’

Reduction

Independent Set **Output** (S)



Vertex Cover **Output** (S’)



Polynomial Time Reducible

We say A reduces to B in polynomial time, if there is an algorithm that, using a (hypothetical) polynomial-time algorithm for B, solves problem A in polynomial-time.

- I could solve problem A efficiently, if you give me a library that solves problem B efficiently
- If A reduces to B then A should be "easier" than B. (for us as algorithm designers)
  - If we can solve B, we can definitely solve A.
- Usually denoted A $\leq_P$ B.

The Direction Matters!

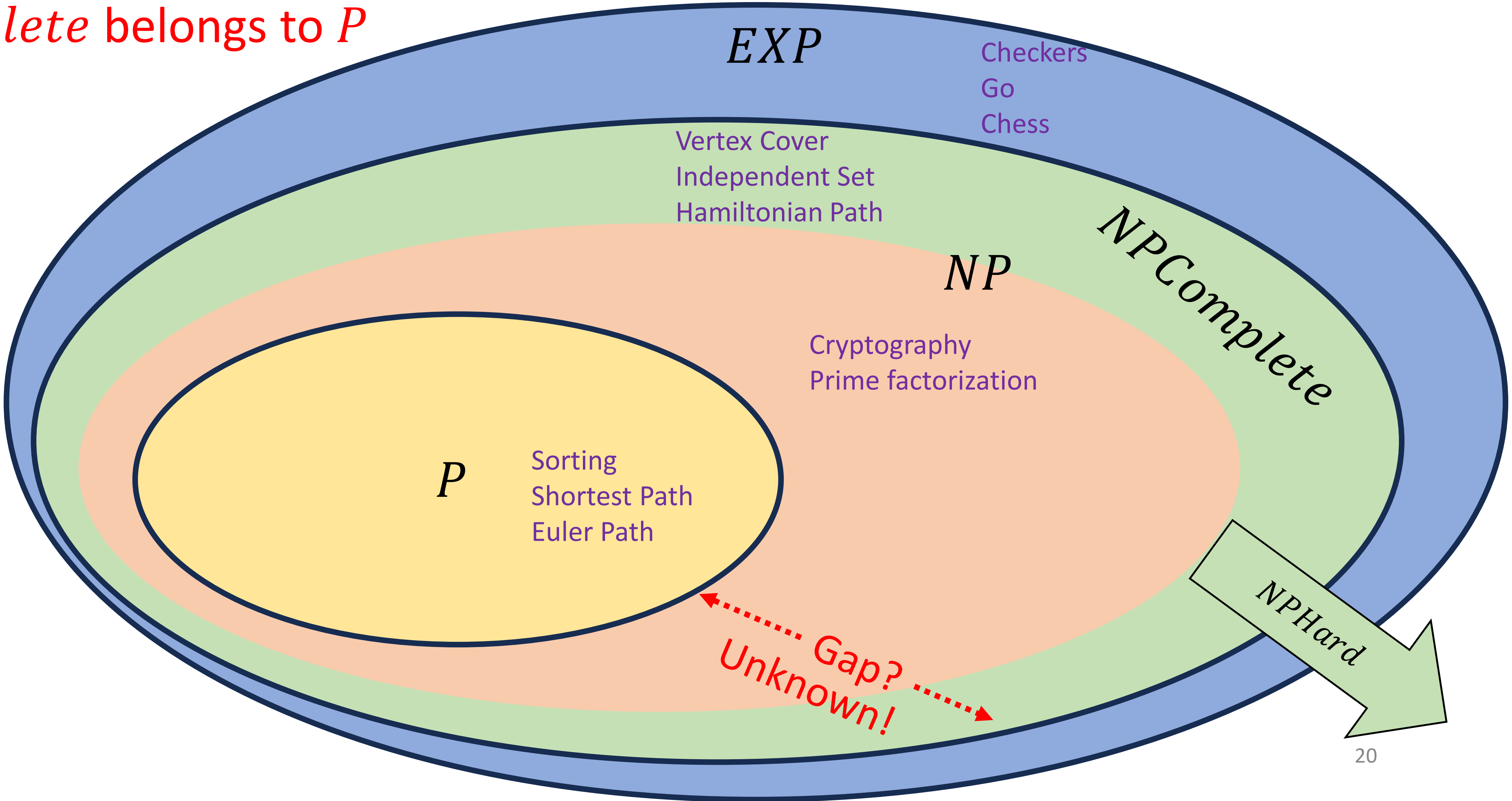
- Direction matters, and is often confusing:
  $A \leq B$ “A reduces to B”
- I wrote an algorithm to solve problem A using a library designed to solve problem B
- “A is no harder than B” (solving B guarantees you can solve A, but maybe there’s a different way to solve A)

NP-Complete

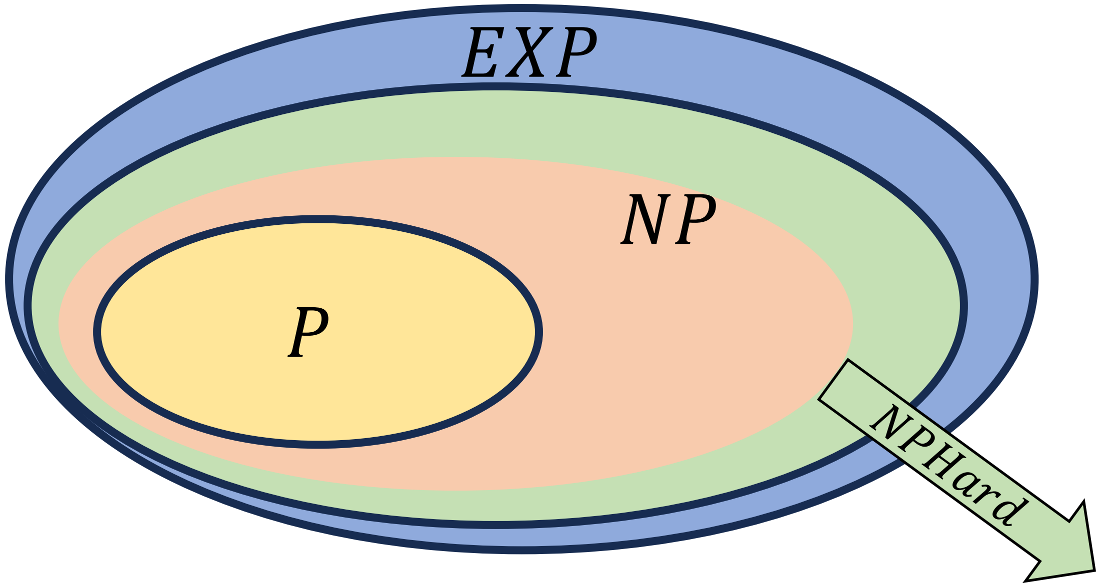
- A set of “together they stand, together they fall” problems
- The problems in this set either all belong to P, or none of them do
- Intuitively, the “hardest” problems in NP
- Collection of problems from NP that can all be “transformed” into each other in polynomial time
  - Like we could transform independent set to vertex cover, and vice-versa
  - We can also transform vertex cover into Hamiltonian path, and Hamiltonian path into independent set, and …
- A problem B is NP-complete if:
  - B is in NP and
  - for all problems A in NP, A reduces to B in polynomial time.

$EXP \supset NP \supseteq P$

$P = NP$ iff some problem from $NPComplete$ belongs to P

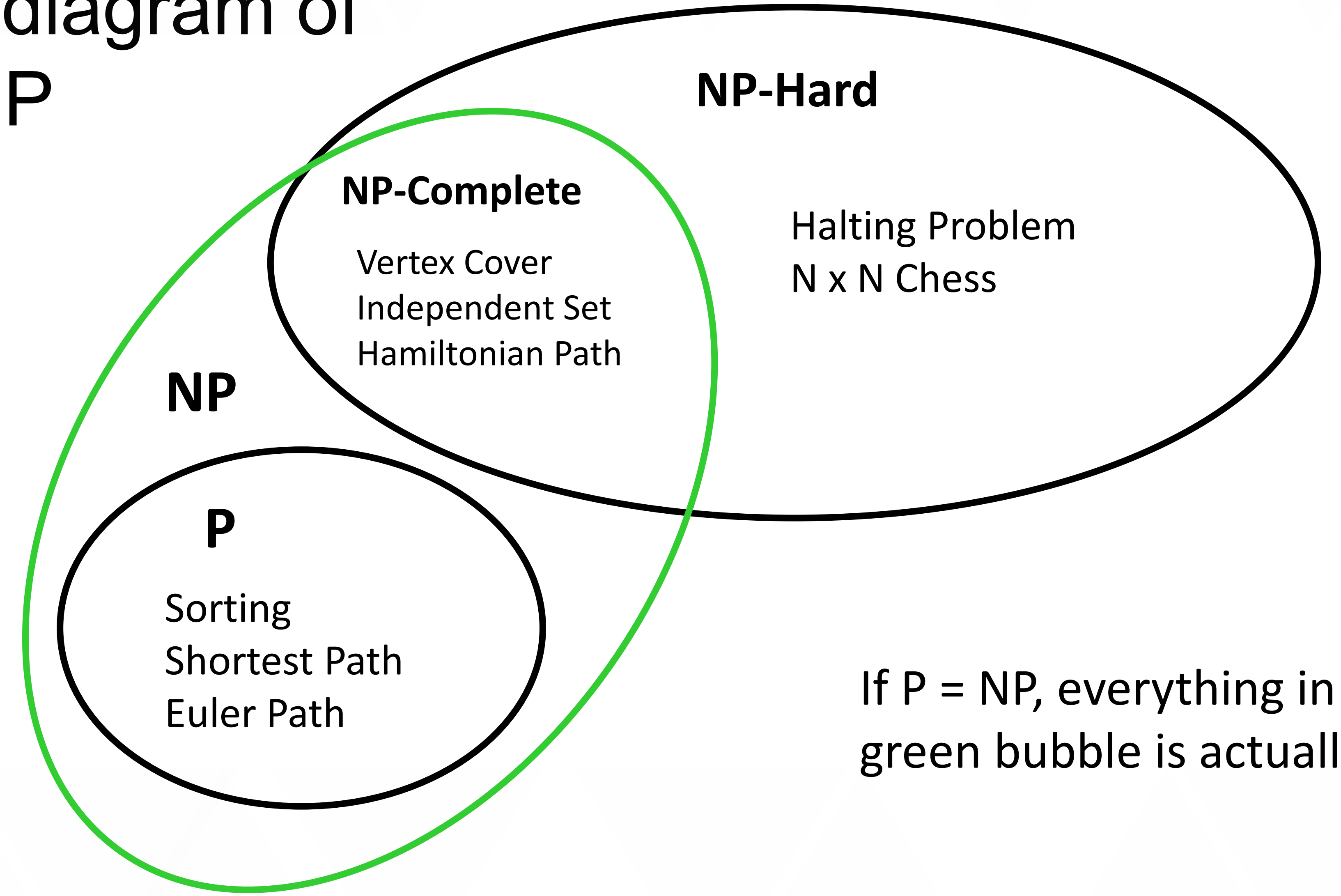


NP-Hard



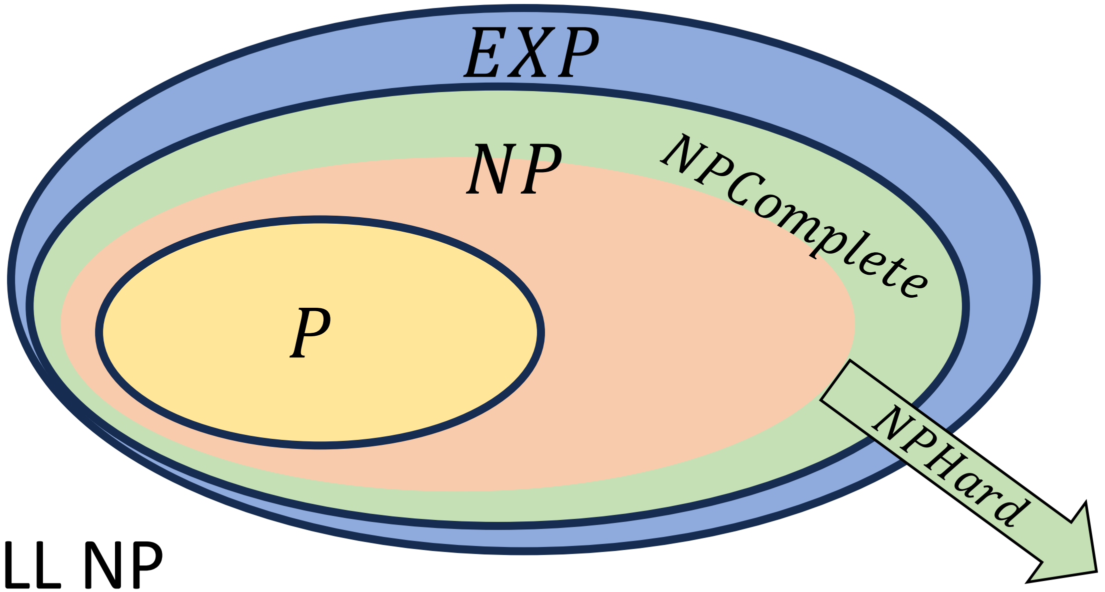
- How can we try to figure out if P=NP?
- Identify problems at least as “hard” as NP
- NP-Hard: problems at least as hard as any of the problems in NP
  - If any of these “hard” problems can be solved in polynomial time, then **<u>all NP problems</u>** can be solved in polynomial time.
- Definition: NP-Hard:
  - Problem B is NP-Hard provided EVERY problem within NP reduces to B in polynomial time

An alternate diagram of P/NP



If P = NP, everything in the green bubble is actually P

NP-Complete

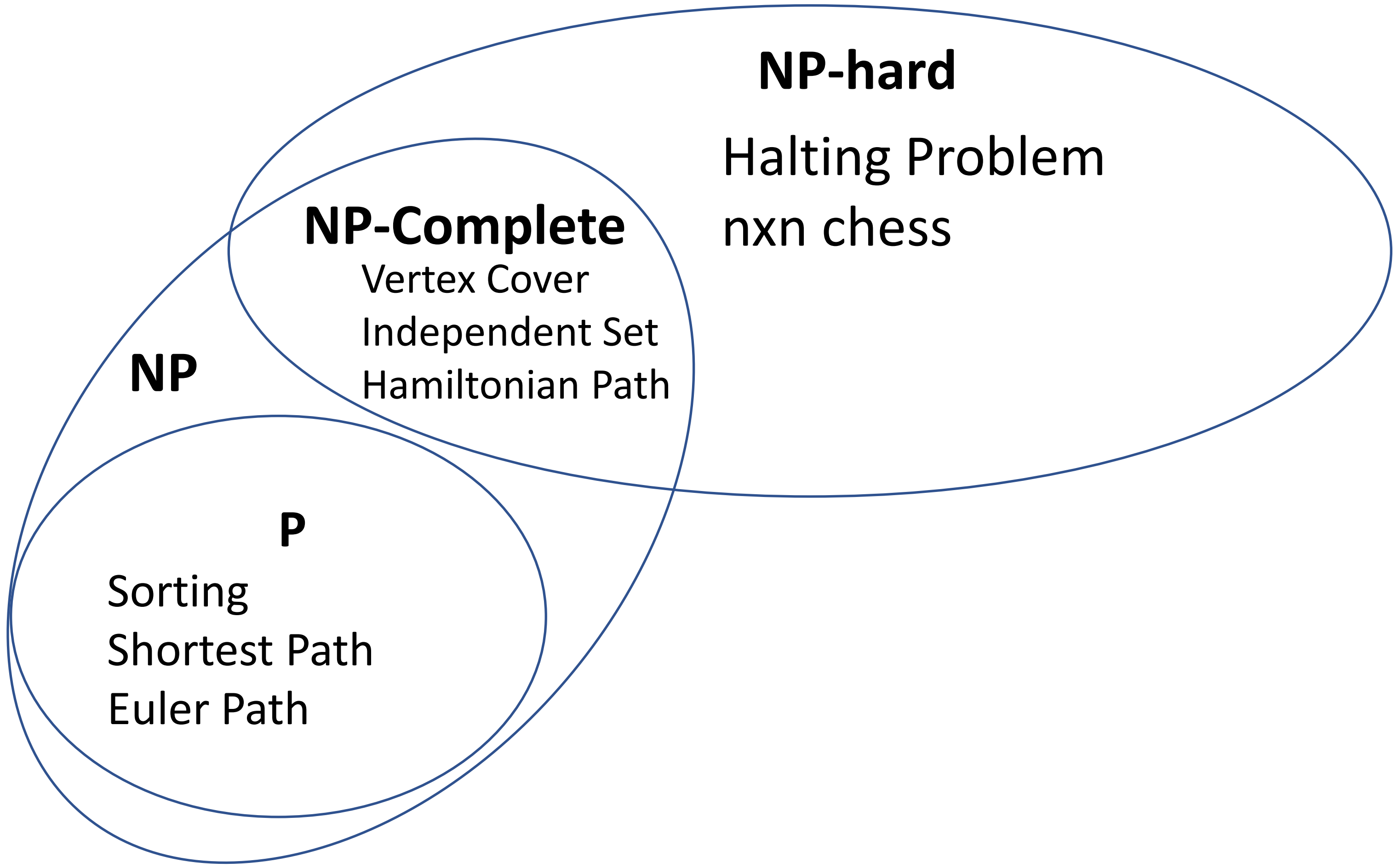


- “Together they stand, together they fall”
- Problems solvable in polynomial time iff ALL NP problems are.
- NP-Complete = NP ∩ NP-Hard
- **How to show a problem is NP-Complete?**
  - Show it belongs to NP
    - Give a polynomial time verifier
  - Show it is NP-Hard
    - Give a reduction from another NP-Hard problem

Overview

- Problems not belonging to P are considered intractable
- The problems within NP have some properties that make them seem like they might be tractable, but we've been unsuccessful with finding polynomial time algorithms for many
- The class $NPComplete$ contains problems with the properties:
  - All members are also members of NP
  - All members of NP can be transformed into every member of $NPComplete$
    - Because $NPComplete$ problems are both in NP and $NPHard$
  - If any one member of $NPComplete$ belongs to P, then $P = NP$
  - If any one member of $NPComplete$ is outside of P, then $P \neq NP$

What The World Looks Like (We Think)



What The World Looks Like (If P=NP)

P

Sorting
Shortest Path
Euler Path
Vertex Cover
Independent Set
Hamiltonian Path

Still hard:
nxn chess

Still impossible:
Halting Problem

Why should YOU care?

- If you can find a polynomial time algorithm for <u>any</u> $NPComplete$ problem then:
  - You will win \$1million
  - You will win a Turing Award
  - You will be world famous
- What if you are asked to write an algorithm for a problem that is known to be $NPComplete$?
  - You can tell that person everything above to set expectations
- What if the problem sounds like it is $NPComplete$ but you are not sure?

Use a Reduction to show your problem is hard

In complexity theory (where we're trying to show algorithms don't exist) we reduce well-studied problem A to new problem B.

To show problem B is NP-hard

- Reduce from A (a known NP-hard problem), to B.
- From the known-hard problem <u>to</u> your new problem—must be that direction!

Goal is a proof by contradiction.

1. Suppose (for sake of contradiction) new problem B has a nice algorithm.
2. But then we can use that for an algorithm for well-studied problem A.
3. But, uh, no one knows an algorithm for well-studied problem A.
4. "contradiction"

Travelling Salesman Problem (TSP)

- Given <u>complete</u> weighted graph G, integer k.
- Is there a cycle that visits all vertices with cost <= k?
- One of the canonical problems.

- Note difference from Hamiltonian cycle:
  - graph is complete
  - we care about weight.

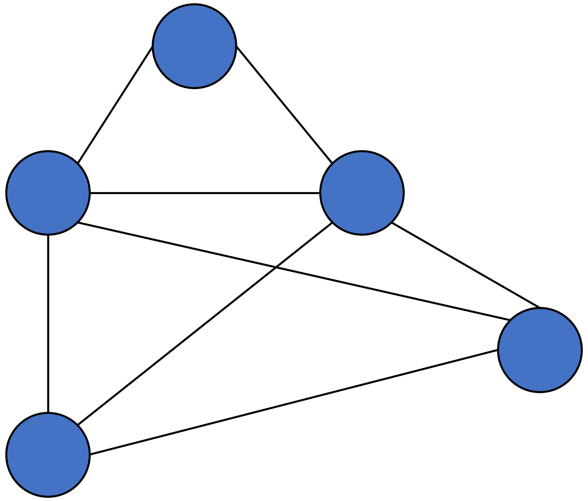
Transforming Hamiltonian Cycle to TSP

- We can “reduce” Hamiltonian Cycle to TSP.
- Given graph G=(V, E):
  - Assign weight of 1 to each edge
  - Augment the graph with edges until it is a complete graph G’=(V, E’)
  - Assign weights of 2 to the new edges
  - Let k = |V|.

Notes:

- The transformation must take polynomial time
- You reduce the known NP-complete problem into your problem (not the other way around)
- In this case we are assuming Hamiltonian Cycle is our known NP-complete problem (in reality, both are known NP-complete)

Known NP-Complete Problem: Hamiltonian Cycle

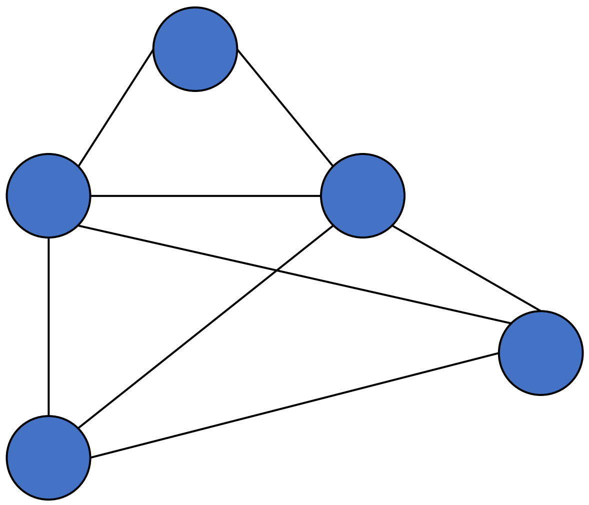


G

Input to Hamiltonian Cycle Problem

(Known to be NP-Complete)

Reduce Hamiltonian Cycle to TSP

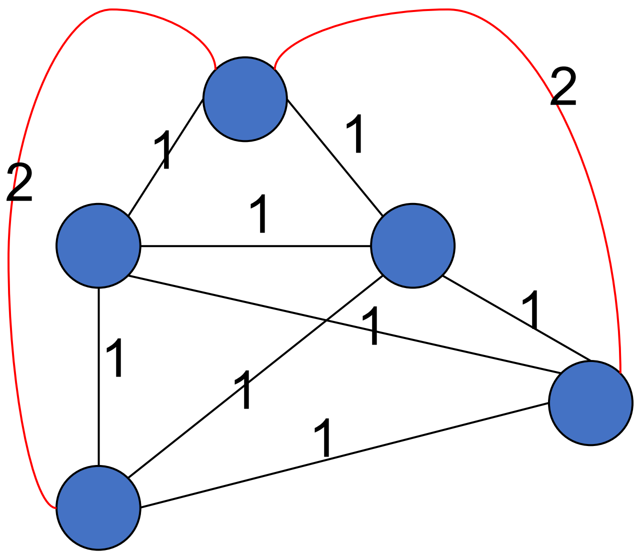


G

Input to Hamiltonian
Cycle Problem

(Known to be NP-Complete)

Polynomial time
transformation



k = 5

G’

Input to Traveling
Salesman Problem

(Your problem)

Polynomial-time transformation

- G’ has a TSP tour of weight |V| iff
  G has a Hamiltonian Cycle.
- What was the cost of transforming HC into TSP?

- In the end, because there is a polynomial time transformation from HC to TSP, we say *TSP is “at least as hard as” Hamiltonian cycle.*

What if still have to solve this problem?!?

- **Approximate the solution**:
  - Instead of finding a path that visits every node, find a path that visits at least 75% of the nodes
- **Add Assumptions**:
  - The problem might be tractable if we can assume the graph is acyclic, a tree
- **Use Heuristics**:
  - Write an algorithm that’s “good enough” for small inputs, ignore edge cases