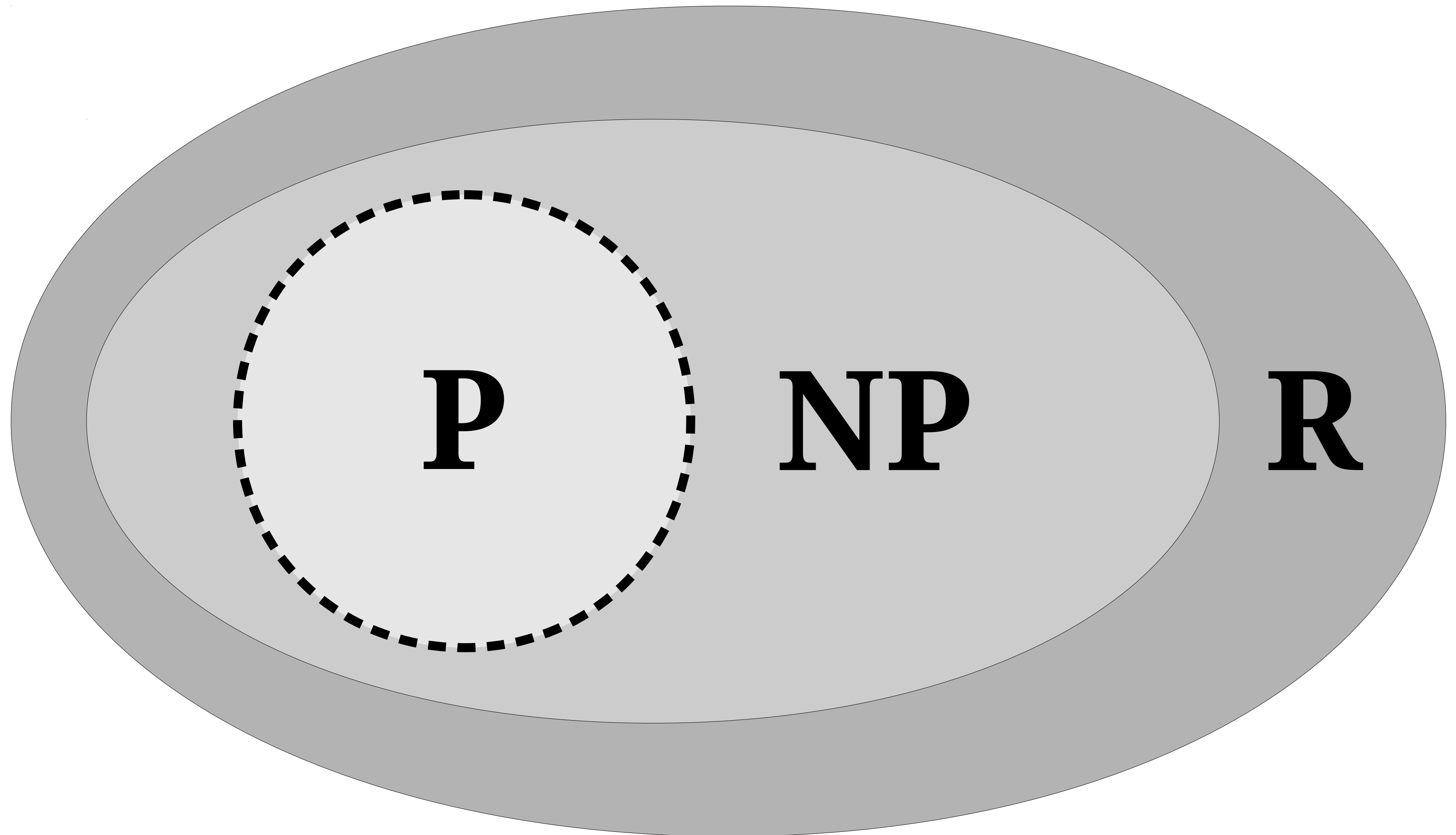
NP-Completeness

Part One

Recap from Last Time

The Limits of Efficient Computation



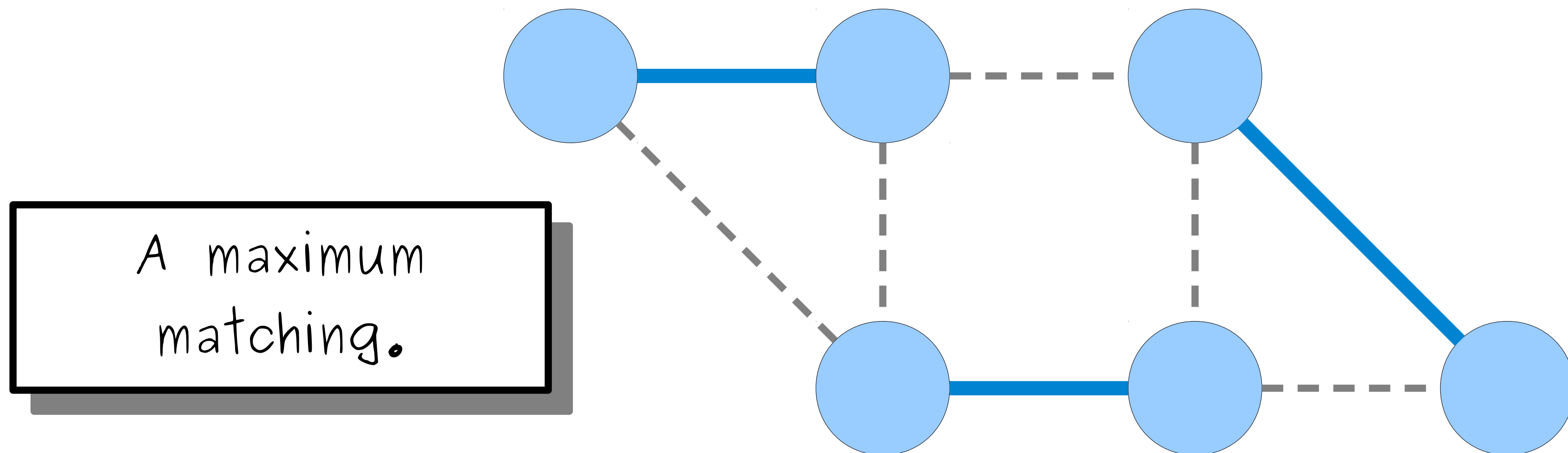
P and **NP** Refresher

- The class **P** consists of all problems solvable in deterministic *polynomial* time.
- The class **NP** consists of all problems solvable in *nondeterministic* polynomial time.
- Equivalently, **NP** consists of all problems for which there is a deterministic, polynomial-time verifier for the problem.

Reducibility

Maximum Matching

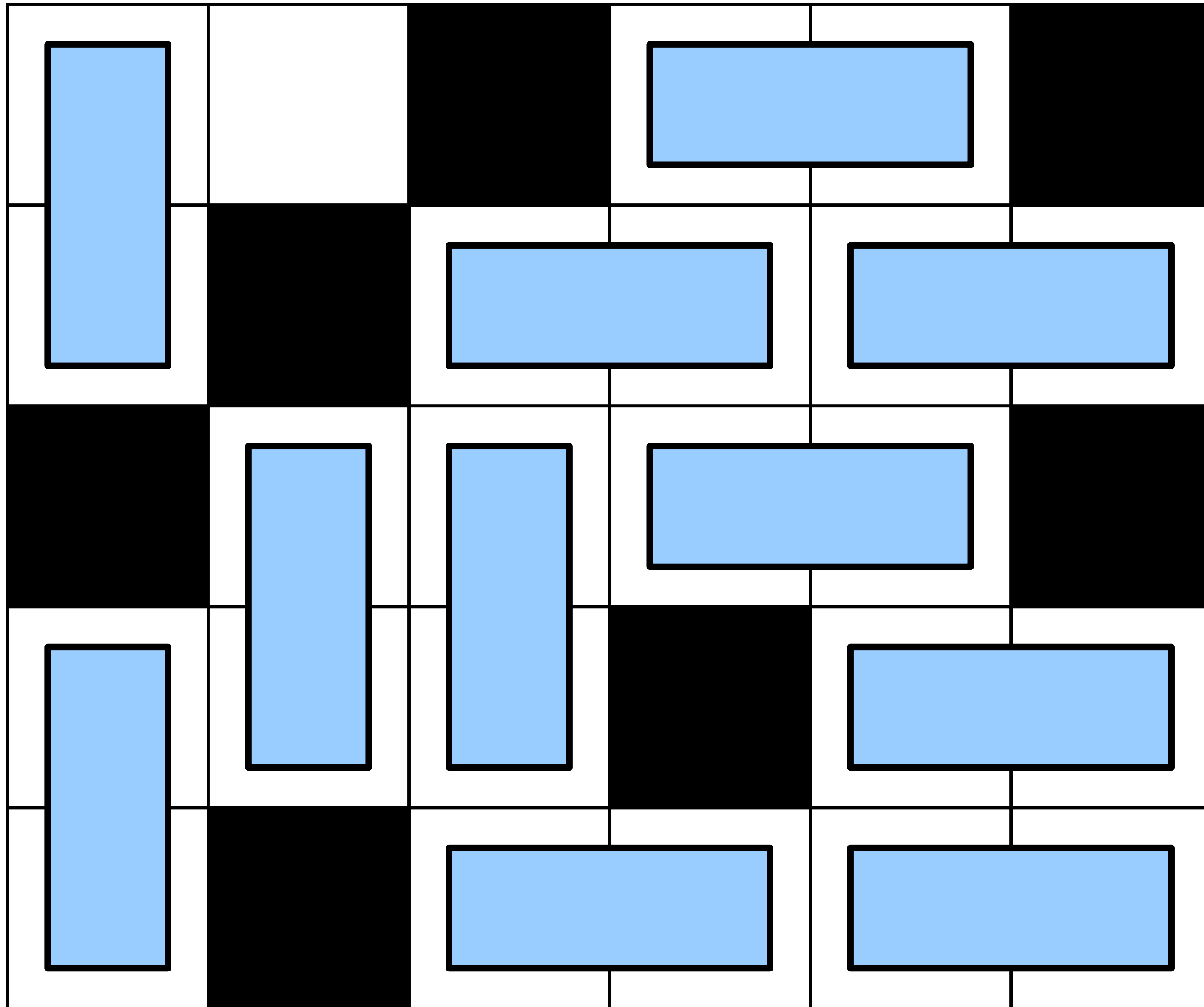
- Given an undirected graph G, a ***matching*** in G is a set of edges such that no two edges share an endpoint.
- A ***maximum matching*** is a matching with the largest number of edges.



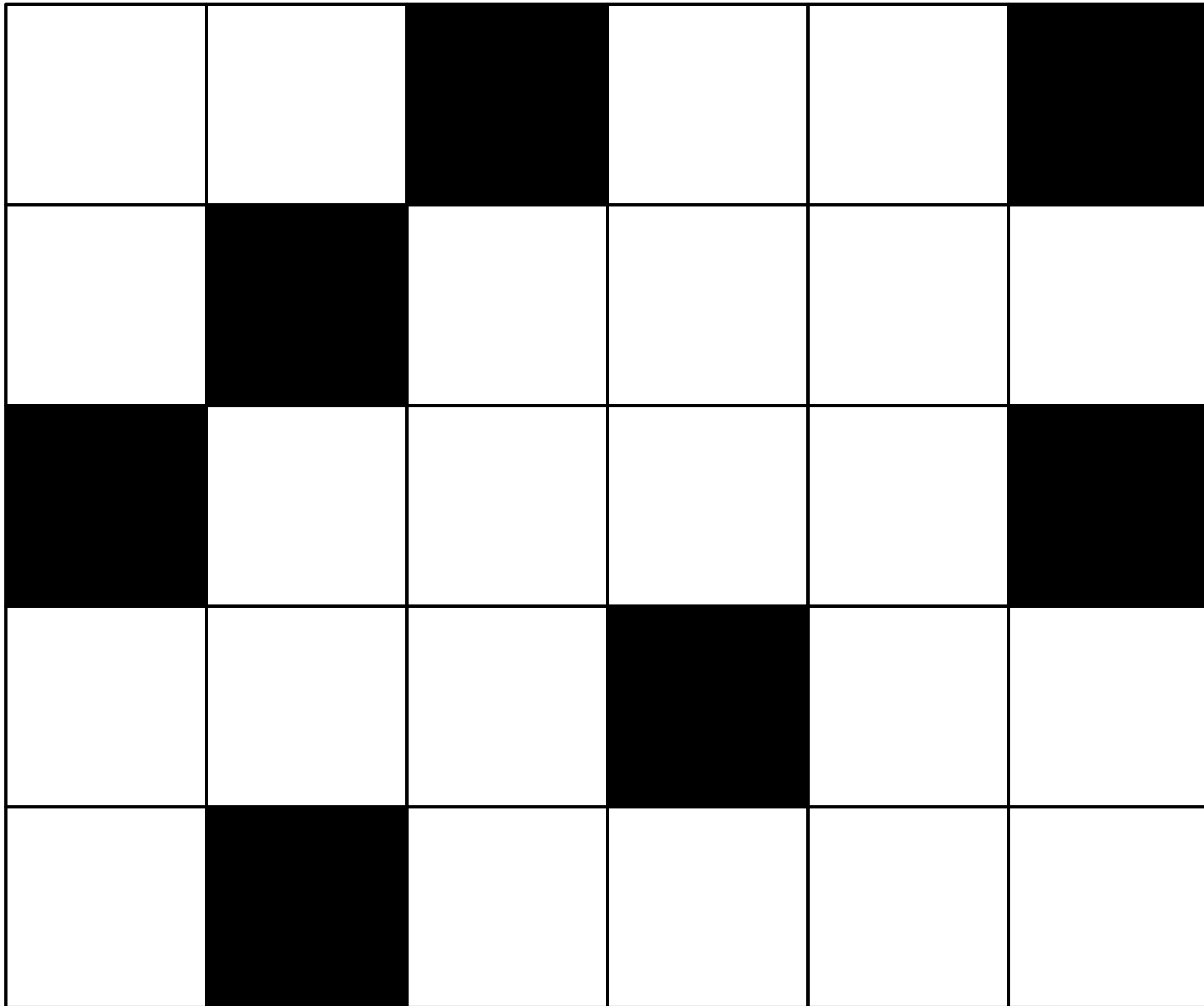
Maximum Matching

- Jack Edmonds' paper "Paths, Trees, and Flowers" gives a polynomial-time algorithm for finding maximum matchings.
  - (This is the same Edmonds as in "Cobham-Edmonds Thesis.)
- Using this fact, what other problems can we solve?

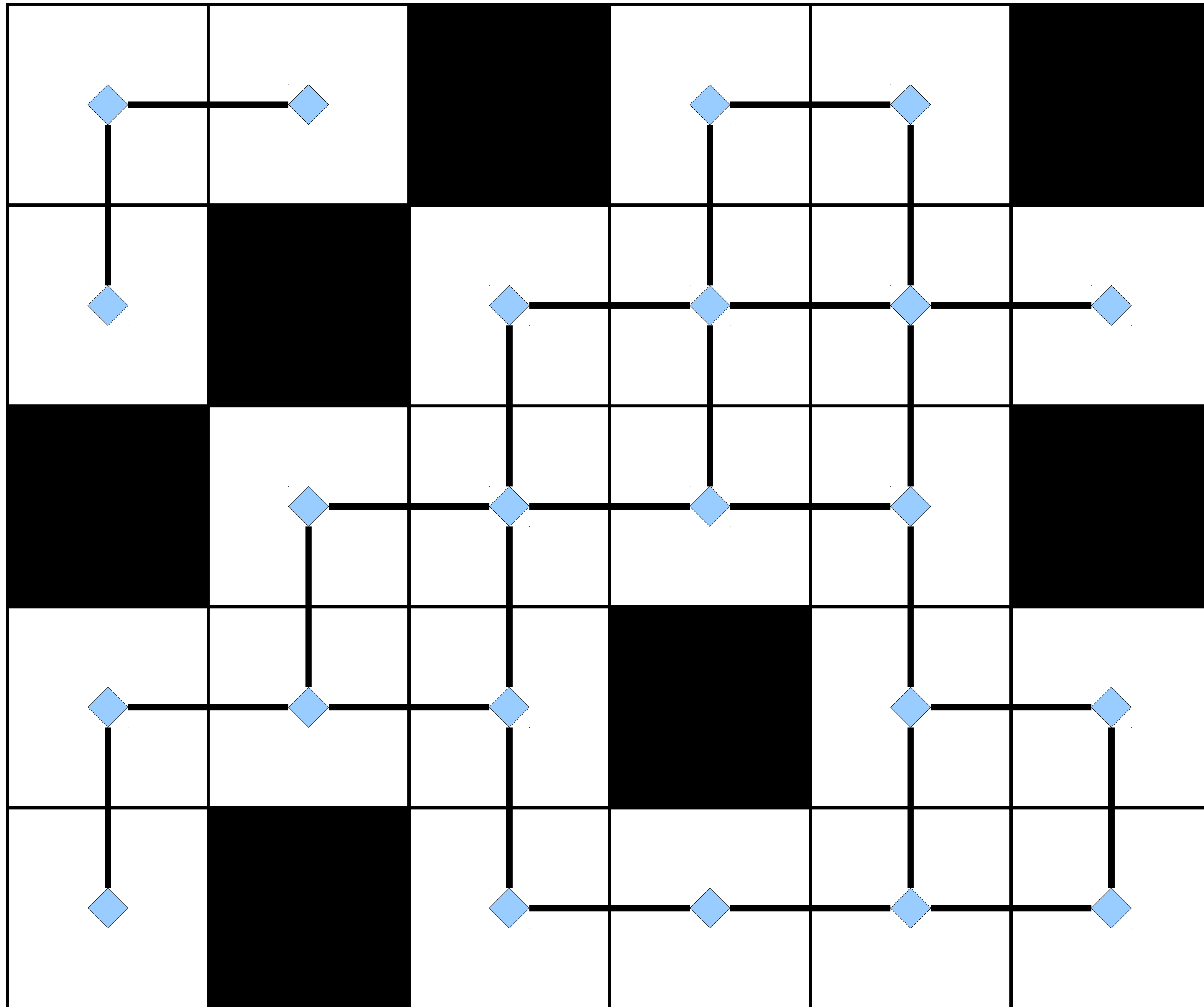
Domino Tiling



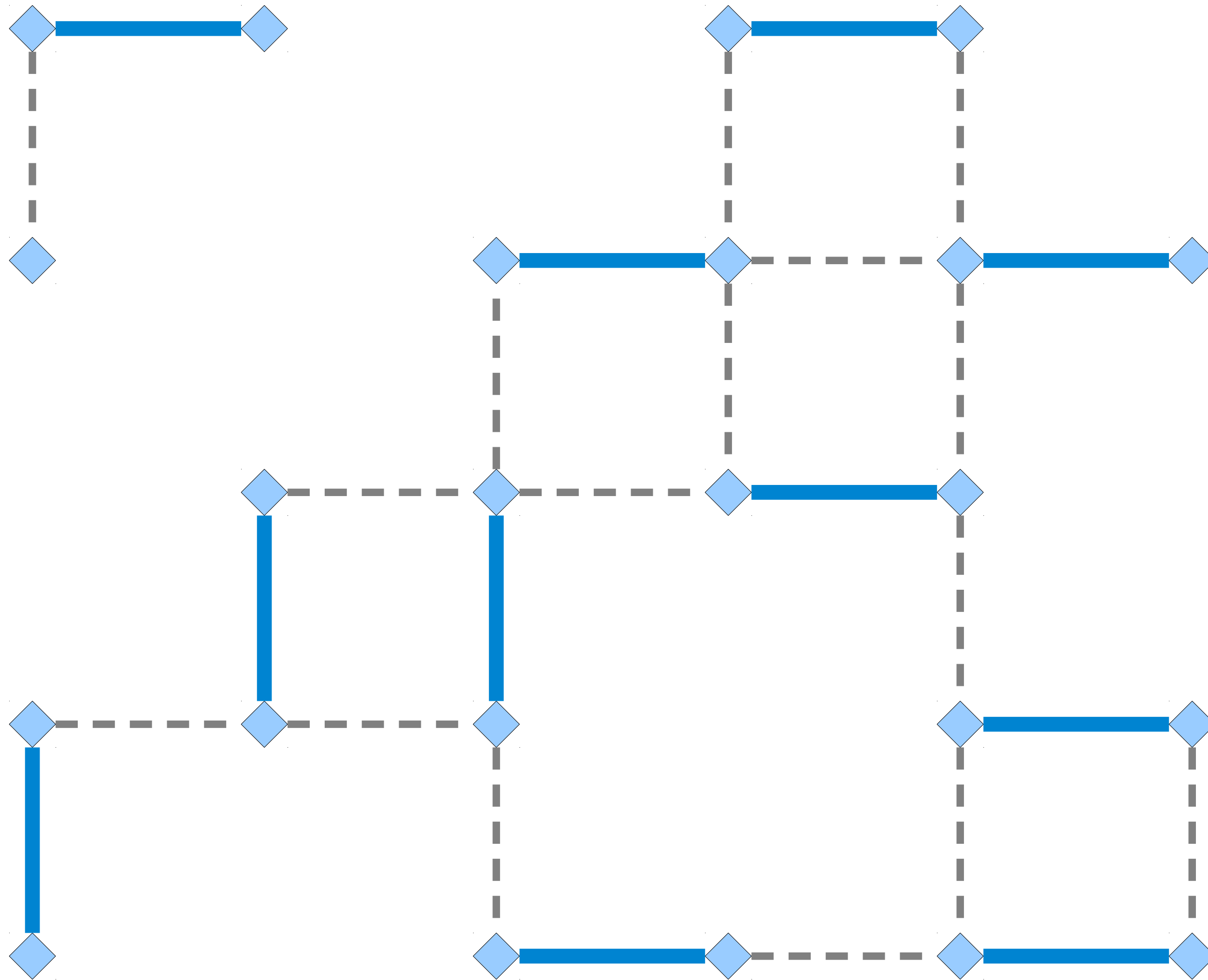
Solving Domino Tiling



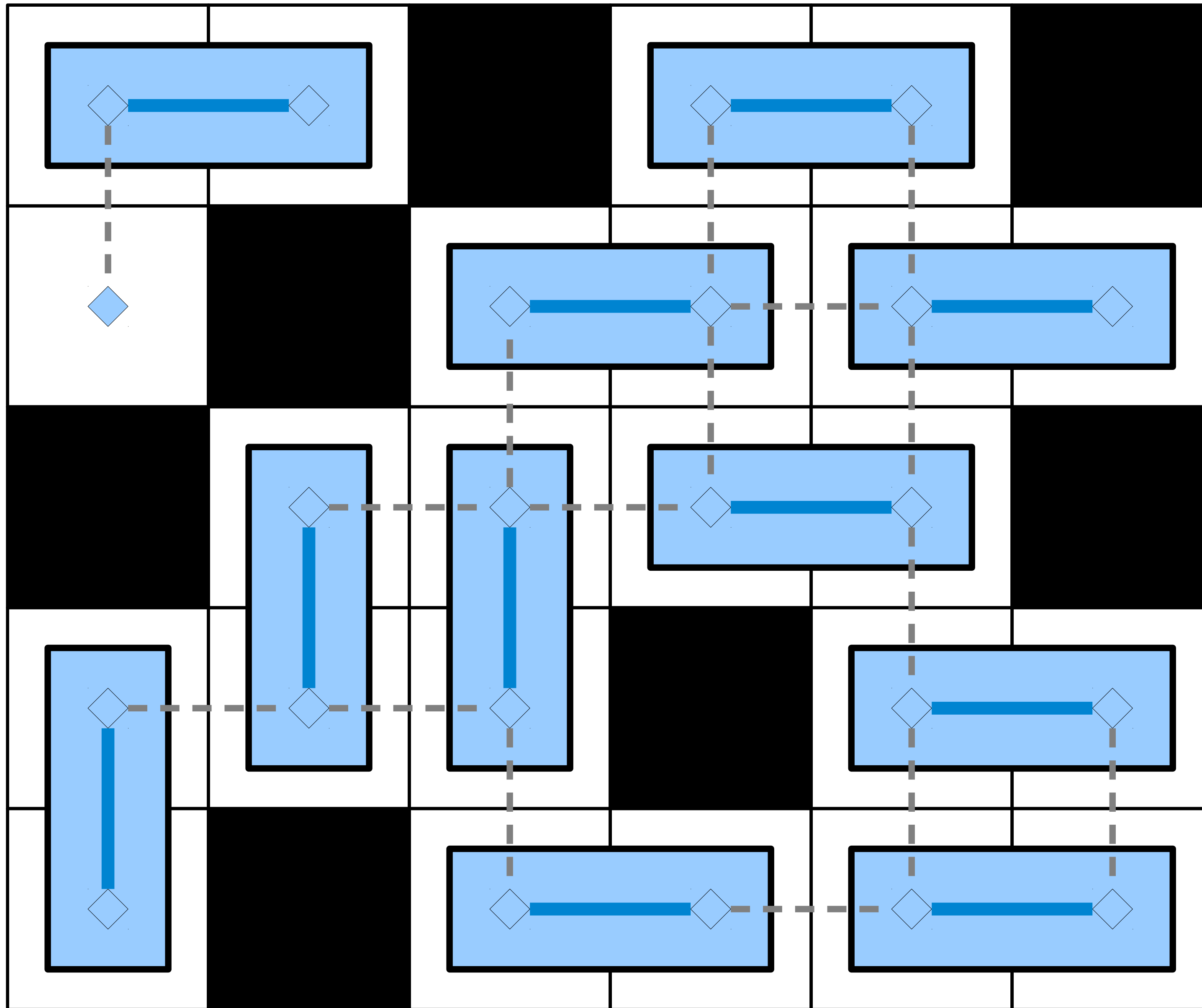
Solving Domino Tiling



Solving Domino Tiling



Solving Domino Tiling



The Setup

- To determine whether you can place at least k dominoes on a crossword grid, do the following:
  - Convert the grid into a graph: each empty cell is a node, and any two adjacent empty cells have an edge between them.
  - Ask whether that graph has a matching of size k or greater.
  - Return whatever answer you get.
- **Claim:** This runs in polynomial time.

In Pseudocode

```
boolean canPlaceKDominos(Grid G, int k) {
  return hasMatching(gridToGraph(G), k);
}
```

Another Example

Reachability

- Consider the following problem:

  Given an directed graph *G* and nodes *s* and *t* in *G*, is there a path from *s* to *t*?

- As a formal language:

  ***REACHABILITY* =**
  { ⟨*G*, *s*, *t*⟩ | *G* is a directed graph, *s* and *t* are nodes in *G*, and there's a path from *s* to *t* }

- ***Theorem:*** *REACHABILITY* ∈ **P**.
- Given that we can solve the reachability problem in polynomial time, what other problems can we solve in polynomial time?

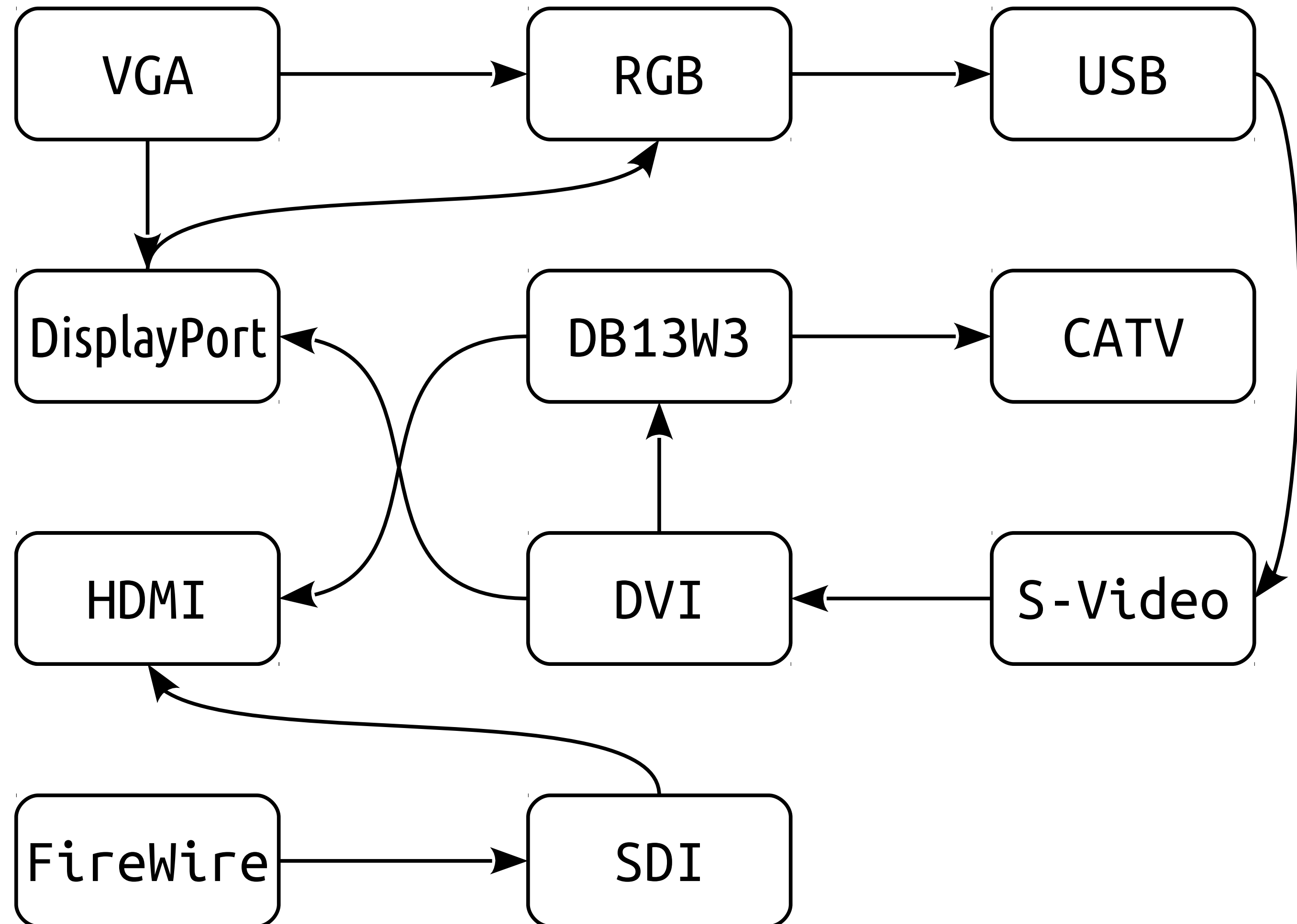
Converter Conundrums

- Suppose that you want to plug your laptop into a projector.
- Your laptop only has a VGA output, but the projector needs HDMI input.
- You have a box of connectors that convert various types of input into various types of output (for example, VGA to DVI, DVI to DisplayPort, etc.)
- **Question:** Can you plug your laptop into the projector?

Converter Conundrums

Connectors

RGB to USB
VGA to DisplayPort
DB13W3 to CATV
DisplayPort to RGB
DB13W3 to HDMI
DVI to DB13W3
S-Video to DVI
FireWire to SDI
VGA to RGB
DVI to DisplayPort
USB to S-Video
SDI to HDMI



Converter Conundrums

Converter Conundrums

- Given a a list of plug converters, here's a algorithm for determining whether you can plug your computer into the projector:
  - Create a graph with one node per connector type and an edge from one type of connector to another if there's a converter from the first type to the second.
  - Use the reachability algorithm to see whether you can get from VGA to HDMI.
  - Return whatever the result of that algorithm is.
- **Claim:** This runs in polynomial time.

In Pseudocode

```
boolean canPlugIn(List<Plug> plugs) {
  return isReachable(plugsToGraph(plugs),
                     VGA, HDMI);
}
```

A Commonality

- Both of the solutions to our previous problems had the following form:

```
boolean solveProblemA(input) {
   return solveProblemB(transform(input));
}
```

- **Important observation:** Assuming that we already have a solver for problem B, the only work done here is transforming the input to problem A into an input to problem B.
- All the "hard" work is done by the solver for B; we just turn one input into another.

Mathematically Modeling this Idea

Polynomial-Time Reductions

- Let A and B be languages.
- A ***polynomial-time reduction*** from A to B is a function $f : \Sigma^* \to \Sigma^*$ such that
  - For any $w \in \Sigma^*$, $w \in A$ iff $f(w) \in B$.
  - The function f can be computed in polynomial time.
- What does this mean?

Polynomial-Time Reductions

- If f is a polynomial-time reduction from A to B, then

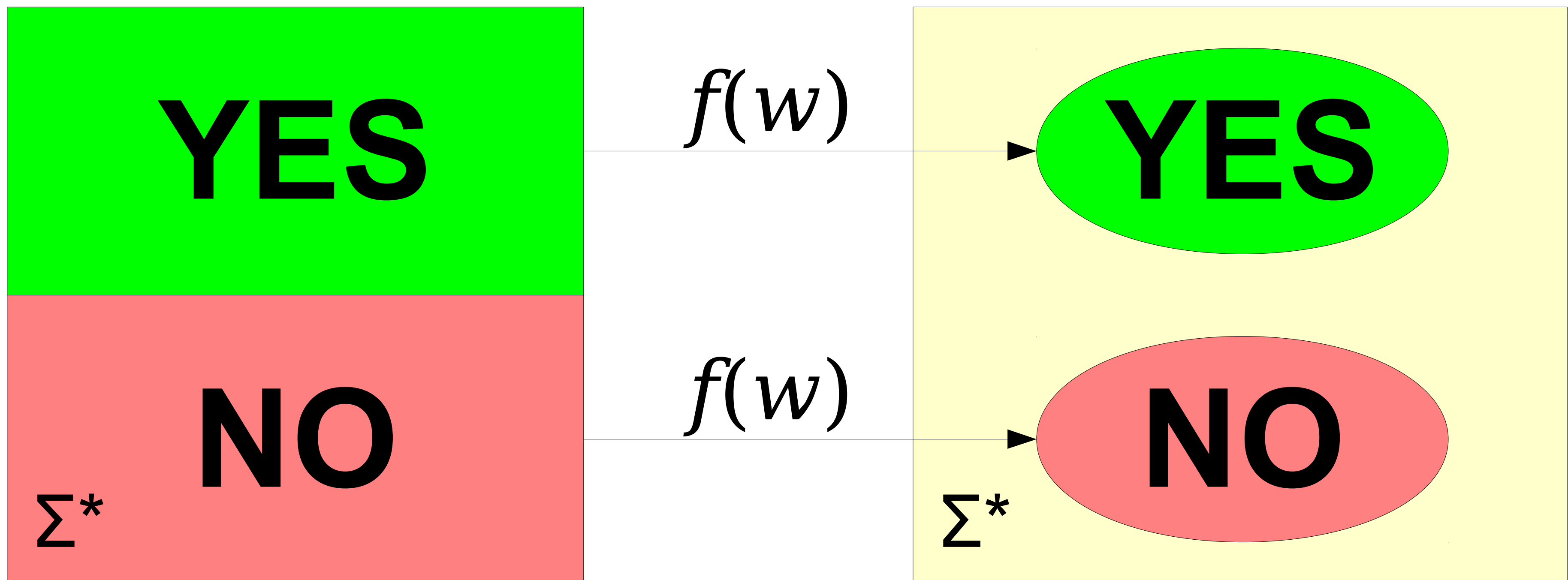
$$\forall w \in \Sigma^*. \ (w \in A \leftrightarrow f(w) \in B)$$

- ***If you want to know whether $w \in A$, you can instead ask whether $f(w) \in B$.***
  - Every $w \in A$ maps to some $f(w) \in B$.
  - Every $w \notin A$ maps to some $f(w) \notin B$.

Polynomial-Time Reductions

- If f is a polynomial-time reduction from A to B, then

$$\forall w \in \Sigma^*. \ (w \in A \leftrightarrow f(w) \in B)$$



Reductions, Programmatically

- Suppose we have a solver for problem B that's defined in terms of problem A in this specific way:

```
boolean solveProblemA(input) {
    return solveProblemB(transform(input));
}
```

- The reduction from A to B is the function `transform` in the above setup: it maps "yes" answers to A to "yes" answers to B and "no" answers to A to "no" answers to B.

Reducibility among Problems

- Suppose that *A* and *B* are languages where there's a polynomial-time reduction from *A* to *B*.
- We'll denote this by writing

$$A \leq_p B$$

- You can read this aloud as "*A* polynomial-time reduces to *B*" or "*A* poly-time reduces to *B*."

Reductions and **P**

- ***Theorem:*** If $A \leq_p B$ and $B \in \mathbf{P}$, then $A \in \mathbf{P}$.
- ***Proof sketch:*** Show that this pseudocode runs in polynomial time:

```
boolean solveProblemA(input) {
   return solveProblemB(transform(input));
}
```

  Calling `transform` only takes polynomial time. Since it runs in polynomial time, the `transform` function can only produce an output that's polynomially larger than its input. Feeding that into a polynomial-time function `solveProblemB` then only takes polynomial time. Overall, this is a polynomial-time algorithm for A, so $A \in \mathbf{P}$. ■

Time-Out for Announcements!

Please evaluate this course in Axess.

Your feedback does make a difference.

Your Questions!

"What's the best way to become a stronger coder? The summer seems like a great time to take advantage of before the next school year. Seems like classes like CS 107 reward experience which is hard to compete with when you haven't coded for years!"

Find something fun that you want to work on, then go work on it. You'll have a lot more fun if you actually enjoy what you're doing.

And don't worry about CS107. People freak out way too much about that course. You don't need to prepare for months in advance before taking it. There aren't that many students in the course that have a huge headstart over everyone. Just be prepared to put in a decent amount of time and be strategic.

“I'm sticking around campus for summer – any suggestions on what to put on my summer bucket list?”

Rent a car (you get a discount for being a Stanford student) and drive up Skyline and walk around the open space preserves. It's beautiful there.

Then go to Pescadero and visit a world-class goat dairy farm while eating artichoke-garlic bread.

Go to the Mission in SF. Do something totally hipsterish (Urban Putt, Mission Cheese, Dandelion Chocolate) and something more local (get pupusas).

“After your teaser during Wednesday's lecture and 2 days of waiting, we're even more excited for you to speak on something that fascinates you. We'd also like to thank you for answering these questions with such enthusiasm throughout the quarter. Go!”

Back to CS103!

Reductions and **NP**

```
boolean solveProblemA(input) {
  return solveProblemB(transform(input));
}
```

What happens if `transform` runs in ***deterministic*** polynomial time, but `solveProblemB` runs in ***nondeterministic*** polynomial time?

Reductions and **NP**

- We can reduce problems in **NP** to one another using polynomial-time reductions.
- The reduction itself must be computable in ***deterministic*** polynomial time.
- The output of that reduction is then fed in as input to a ***nondeterministic, polynomial-time*** algorithm.
- Remember – *the goal of the reduction is to transform the problem, not solve it!*

A Sample Reduction

Satisfiability

- A propositional logic formula φ is called ***satisfiable*** if there is some assignment to its variables that makes it evaluate to true.
  - $p \wedge q$ is satisfiable.
  - $p \wedge \neg p$ is unsatisfiable.
  - $p \rightarrow (q \wedge \neg q)$ is satisfiable.
- An assignment of true and false to the variables of φ that makes it evaluate to true is called a ***satisfying assignment***.

SAT

- The ***boolean satisfiability problem*** (***SAT***) is the following:

Given a propositional logic formula φ, is φ satisfiable?

- Formally:

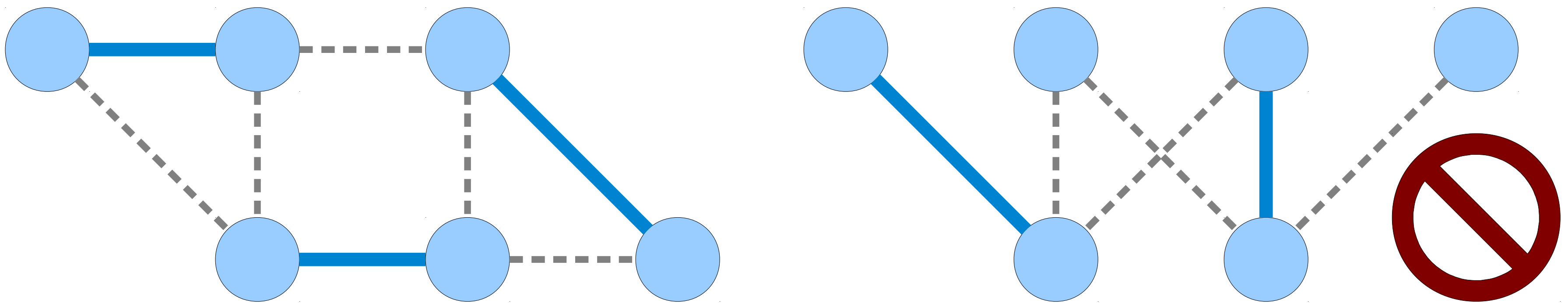
SAT **= { ⟨φ⟩ | φ is a satisfiable PL formula }**

- ***Claim:*** *SAT* ∈ **NP**.
  - (Do you see why?)

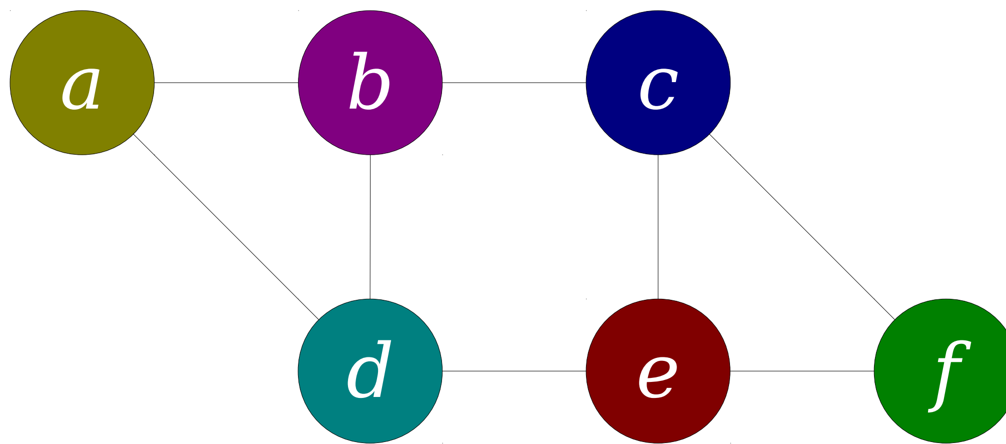
What other problems can we solve with a solver for SAT?

Perfect Matchings

- A ***perfect matching*** in a graph G is a matching where every node is adjacent to some edge in the matching.



- ***Claim:*** We can reduce the problem of determining whether a graph has a perfect matching to the boolean satisfiability problem.



For each edge $\{u, v\}$, we'll introduce a propositional variable p_{uv} meaning "edge $\{u, v\}$ is included in the perfect matching."

We need to enforce the following:

- Every node is adjacent to at least one edge. ("perfect")
- Every node is adjacent to at most one edge. ("matching")

$$
\begin{gathered}
(p_{ab} \lor p_{ad}) \land (p_{ab} \lor p_{bc} \lor p_{bd}) \land (p_{bc} \lor p_{ce} \lor p_{cf}) \land \\
(p_{ad} \lor p_{bd} \lor p_{de}) \land (p_{ce} \lor p_{de} \lor p_{ef}) \land (p_{cf} \lor p_{ef}) \land \\
(\neg p_{ab} \lor \neg p_{ad}) \land \\
(\neg p_{ab} \lor \neg p_{bc}) \land (\neg p_{ab} \lor \neg p_{bd}) \land (\neg p_{bc} \lor \neg p_{bd}) \land \\
(\neg p_{bc} \lor \neg p_{ce}) \land (\neg p_{bc} \lor \neg p_{cf}) \land (\neg p_{ce} \lor \neg p_{cf}) \land \\
(\neg p_{ad} \lor \neg p_{bd}) \land (\neg p_{ad} \lor \neg p_{de}) \land (\neg p_{bd} \lor \neg p_{de}) \land \\
(\neg p_{ce} \lor \neg p_{de}) \land (\neg p_{ce} \lor \neg p_{ef}) \land (\neg p_{de} \lor \neg p_{ef}) \land \\
(\neg p_{cf} \lor \neg p_{ef})
\end{gathered}
$$

- ☐ p_{ab}
- ☐ p_{ad}
- ☐ p_{bc}
- ☐ p_{bd}
- ☐ p_{ce}
- ☐ p_{cf}
- ☐ p_{de}
- ☐ p_{ef}

Notice that this formula is the many-way AND of lots of smaller formulas of the form

$$(l_1 \lor l_2 \lor \ldots \lor l_n)$$

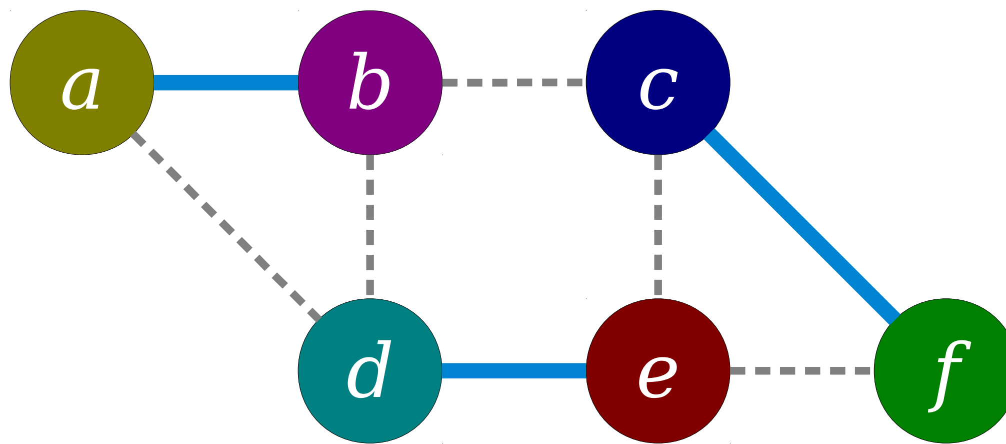
where each l_k is either a variable or its negation. Each l_k is called a ***literal***, and the multi-way OR of literals is called a ***clause***.

To satisfy this formula, we need to choose truth values so that every clause has at least one true literal.

· Every node is adjacent to at most one edge. ("matching")

$$(p_{ab} \lor p_{ad}) \land (p_{ab} \lor p_{bc} \lor p_{bd}) \land (p_{bc} \lor p_{ce} \lor p_{cf}) \land$$
$$(p_{ad} \lor p_{bd} \lor p_{de}) \land (p_{ce} \lor p_{de} \lor p_{ef}) \land (p_{cf} \lor p_{ef}) \land$$
$$(\neg p_{ab} \lor \neg p_{ad}) \land$$
$$(\neg p_{ab} \lor \neg p_{bc}) \land (\neg p_{ab} \lor \neg p_{bd}) \land (\neg p_{bc} \lor \neg p_{bd}) \land$$
$$(\neg p_{bc} \lor \neg p_{ce}) \land (\neg p_{bc} \lor \neg p_{cf}) \land (\neg p_{ce} \lor \neg p_{cf}) \land$$
$$(\neg p_{ad} \lor \neg p_{bd}) \land (\neg p_{ad} \lor \neg p_{de}) \land (\neg p_{bd} \lor \neg p_{de}) \land$$
$$(\neg p_{ce} \lor \neg p_{de}) \land (\neg p_{ce} \lor \neg p_{ef}) \land (\neg p_{de} \lor \neg p_{ef}) \land$$
$$(\neg p_{cf} \lor \neg p_{ef})$$

- ☐ p_{ab}
- ☐ p_{ad}
- ☐ p_{bc}
- ☐ p_{bd}
- ☐ p_{ce}
- ☐ p_{cf}
- ☐ p_{de}
- ☐ p_{ef}



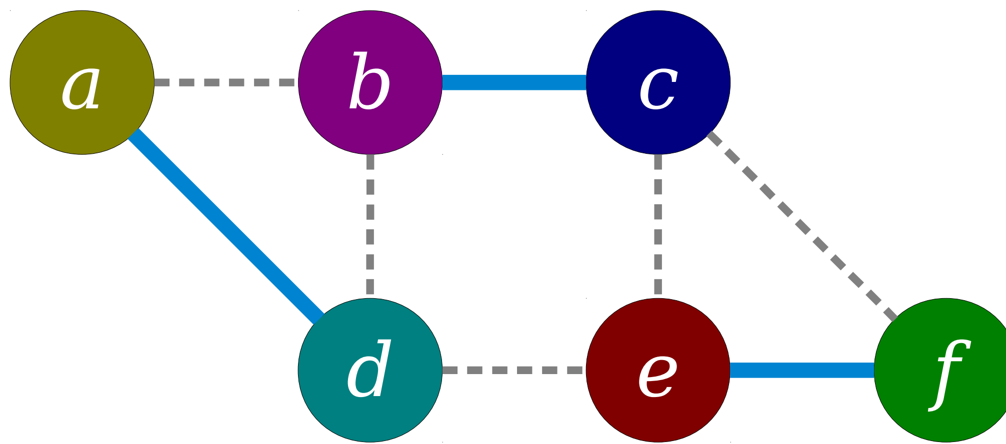
For each edge $\{u, v\}$, we'll introduce a propositional variable p_{uv} meaning "edge $\{u, v\}$ is included in the perfect matching."

We need to enforce the following:

- Every node is adjacent to at least one edge. ("perfect")
- Every node is adjacent to at most one edge. ("matching")

$$(p_{ab} \lor p_{ad}) \land (p_{ab} \lor p_{bc} \lor p_{bd}) \land (p_{bc} \lor p_{ce} \lor p_{cf}) \land$$
$$(p_{ad} \lor p_{bd} \lor p_{de}) \land (p_{ce} \lor p_{de} \lor p_{ef}) \land (p_{cf} \lor p_{ef}) \land$$
$$(\neg p_{ab} \lor \neg p_{ad}) \land$$
$$(\neg p_{ab} \lor \neg p_{bc}) \land (\neg p_{ab} \lor \neg p_{bd}) \land (\neg p_{bc} \lor \neg p_{bd}) \land$$
$$(\neg p_{bc} \lor \neg p_{ce}) \land (\neg p_{bc} \lor \neg p_{cf}) \land (\neg p_{ce} \lor \neg p_{cf}) \land$$
$$(\neg p_{ad} \lor \neg p_{bd}) \land (\neg p_{ad} \lor \neg p_{de}) \land (\neg p_{bd} \land \neg p_{de}) \land$$
$$(\neg p_{ce} \lor \neg p_{de}) \land (\neg p_{ce} \lor \neg p_{ef}) \land (\neg p_{de} \lor \neg p_{ef}) \land$$
$$(\neg p_{cf} \lor \neg p_{ef})$$

T	p_{ab}
F	p_{ad}
F	p_{bc}
F	p_{bd}
F	p_{ce}
T	p_{cf}
T	p_{de}
F	p_{ef}



$$(p_{ab} \lor p_{ad}) \land (p_{ab} \lor p_{bc} \lor p_{bd}) \land (p_{bc} \lor p_{ce} \lor p_{cf}) \land$$
$$(p_{ad} \lor p_{bd} \lor p_{de}) \land (p_{ce} \lor p_{de} \lor p_{ef}) \land (p_{cf} \lor p_{ef}) \land$$
$$(\neg p_{ab} \lor \neg p_{ad}) \land$$
$$(\neg p_{ab} \lor \neg p_{bc}) \land (\neg p_{ab} \lor \neg p_{bd}) \land (\neg p_{bc} \lor \neg p_{bd}) \land$$
$$(\neg p_{bc} \lor \neg p_{ce}) \land (\neg p_{bc} \lor \neg p_{cf}) \land (\neg p_{ce} \lor \neg p_{cf}) \land$$
$$(\neg p_{ad} \lor \neg p_{bd}) \land (\neg p_{ad} \lor \neg p_{de}) \land (\neg p_{bd} \land \neg p_{de}) \land$$
$$(\neg p_{ce} \lor \neg p_{de}) \land (\neg p_{ce} \lor \neg p_{ef}) \land (\neg p_{de} \lor \neg p_{ef}) \land$$
$$(\neg p_{cf} \lor \neg p_{ef})$$

F	p_{ab}
T	p_{ad}
T	p_{bc}
F	p_{bd}
F	p_{ce}
F	p_{cf}
F	p_{de}
T	p_{ef}

The Reduction

- Suppose we have a graph with n nodes.
- For each of the n nodes, we introduce a clause of at most n literals to force each node to be adjacent to at least one chosen edge.
  - Total size: $O(n^2)$
- For each of the n nodes, we introduce $O(n^2)$ clauses with two literals each to force each node to be adjacent to at most one chosen edge.
  - Total size: $O(n^3)$
- Total size of the generated formula: at most $O(n^3)$.
- ***Unsubstantiated but true claim:*** This can be computed in polynomial time.

The Reduction

```
boolean hasPerfectMatching(Graph G) {
  return isSatisfiable(graphToFormula(G));
}
```

Reductions and **NP**

- ***Theorem:*** If $A \leq_p B$ and $B \in \mathbf{NP}$, then $A \in \mathbf{NP}$.
- ***Proof sketch:*** Show that this pseudocode runs in ***nondeterministic*** polynomial time:

```
boolean solveProblemA(input) {
    return solveProblemB(transform(input));
}
```

Calling `transform` only takes polynomial time. The `transform` function can only produce an output that's polynomially larger than its input. Feeding that into a polynomial-time function `solveProblemB` then only takes ***nondeterministic*** polynomial time. Overall, this is a ***nondeterministic*** polynomial-time algorithm for A, so $A \in \mathbf{NP}$. ■

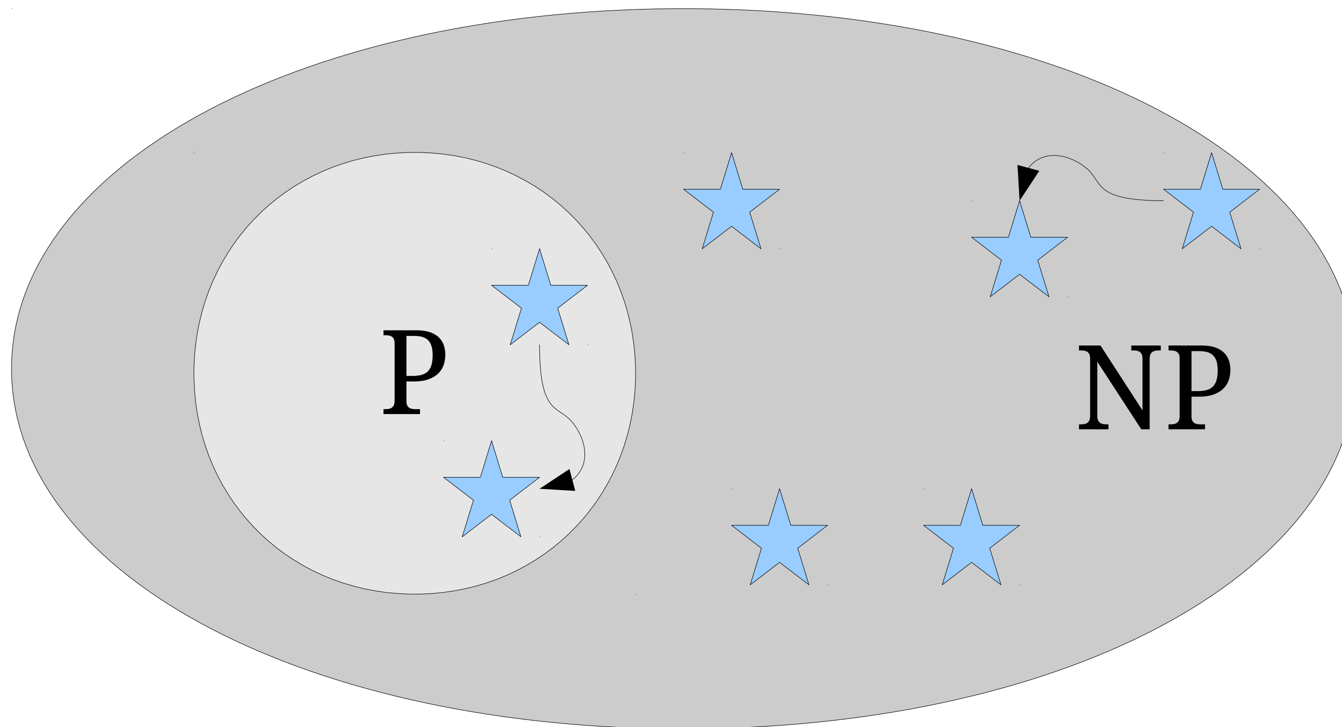
Reducibility, Summarized

- A polynomial-time reduction is a way of transforming inputs to problem A into inputs to problem B that preserves the correct answer.
- If there is a polynomial-time reduction from A to B, we denote this by writing $A \leq_p B$.
- Two major theorems:
  - ***Theorem:*** If $B \in \mathbf{P}$ and $A \leq_p B$, then $A \in \mathbf{P}$.
  - ***Theorem:*** If $B \in \mathbf{NP}$ and $A \leq_p B$, then $A \in \mathbf{NP}$.

NP-Hardness and **NP**-Completeness

Polynomial-Time Reductions

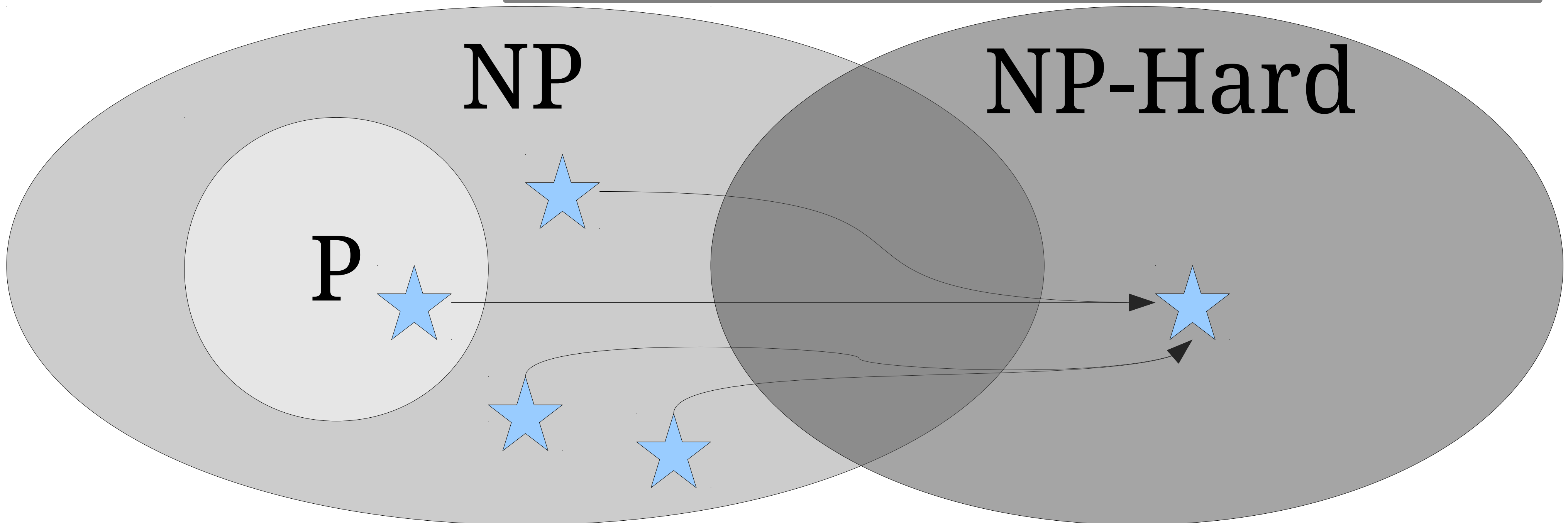
- If $L_1 \leq_P L_2$ and $L_2 \in \mathbf{P}$, then $L_1 \in \mathbf{P}$.
- If $L_1 \leq_P L_2$ and $L_2 \in \mathbf{NP}$, then $L_1 \in \mathbf{NP}$.



NP-Hardness

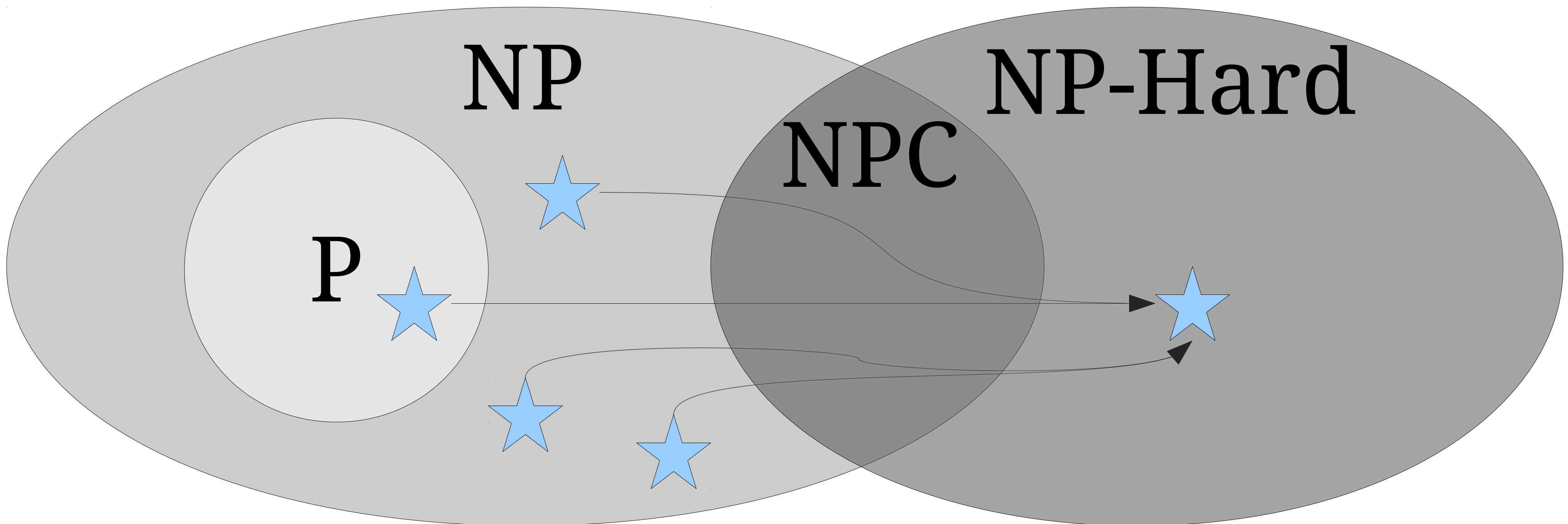
- A language L is called **NP-hard** if for *every* $L' \in$ **NP**, we have $L' \leq_P L$.

Intuitively: L has to be at least as hard as every problem in **NP**, since an algorithm for L can be used to decide all problems in **NP**.



NP-Hardness

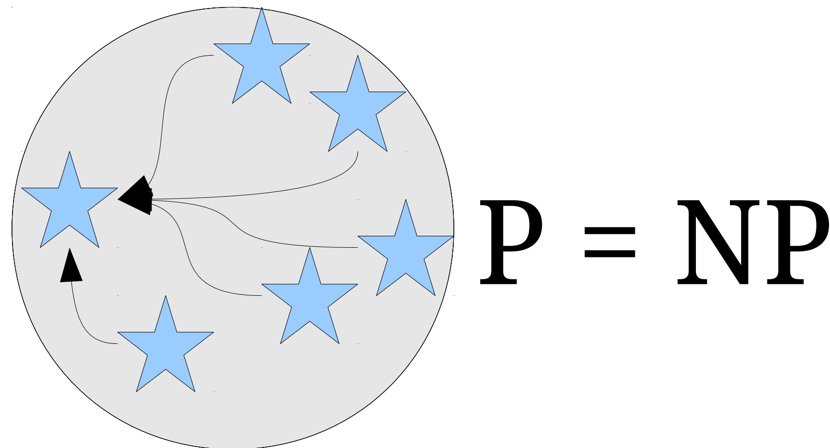
- A language L is called **NP-hard** if for *every* $L' \in$ **NP**, we have $L' \leq_P L$.
- A language in L is called **NP-complete** if L is **NP**-hard and $L \in$ **NP**.
- The class **NPC** is the set of **NP**-complete problems.



The Tantalizing Truth

Theorem: If *any* **NP**-complete language is in **P**, then **P** = **NP**.

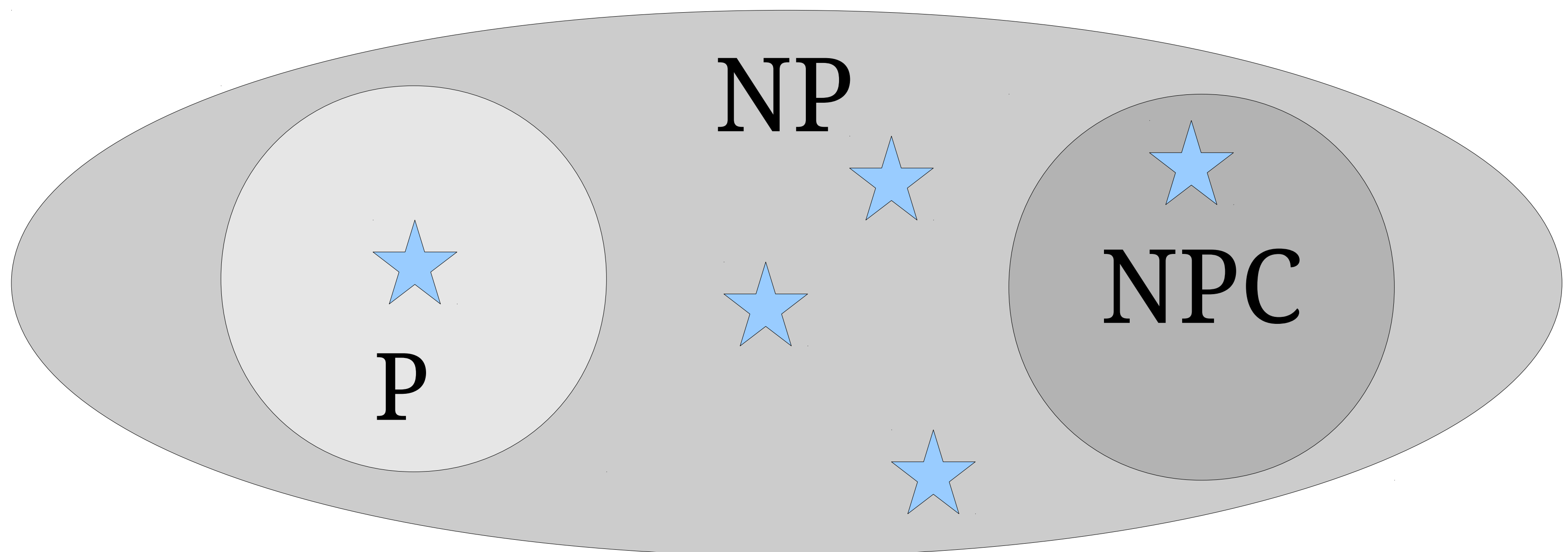
Proof: Suppose that L is **NP**-complete and $L \in$ **P**. Now consider any arbitrary **NP** problem X. Since L is **NP**-complete, we know that $X \leq_p L$. Since $L \in$ **P** and $X \leq_p L$, we see that $X \in$ **P**. Since our choice of X was arbitrary, this means that **NP** ⊆ **P**, so **P** = **NP**. ■



The Tantalizing Truth

Theorem: If *any* **NP**-complete language is not in **P**, then **P** ≠ **NP**.

Proof: Suppose that L is an **NP**-complete language not in **P**. Since L is **NP**-complete, we know that $L \in$ **NP**. Therefore, we know that $L \in$ **NP** and $L \notin$ **P**, so **P** ≠ **NP**. ■



A Feel for **NP**-Completeness

- If a problem is **NP**-complete, then under the assumption that $\mathbf{P} \neq \mathbf{NP}$, there cannot be an efficient algorithm for it.
- In a sense, **NP**-complete problems are the hardest problems in **NP**.
- All known **NP**-complete problems are enormously hard to solve:
  - All known algorithms for **NP**-complete problems run in worst-case exponential time.
  - Most algorithms for **NP**-complete problems are infeasible for reasonably-sized inputs.

How do we even know NP-complete problems exist in the first place?

Satisfiability

- A propositional logic formula φ is called ***satisfiable*** if there is some assignment to its variables that makes it evaluate to true.
  - $p \land q$ is satisfiable.
  - $p \land \neg p$ is unsatisfiable.
  - $p \to (q \land \neg q)$ is satisfiable.
- An assignment of true and false to the variables of φ that makes it evaluate to true is called a ***satisfying assignment***.

SAT

- The ***boolean satisfiability problem*** (***SAT***) is the following:

Given a propositional logic formula φ, is φ satisfiable?

- Formally:

$$\textbf{\textit{SAT}} = \{ \langle \varphi \rangle \mid \varphi \text{ is a satisfiable PL formula} \}$$

Theorem (Cook-Levin): SAT is **NP**-complete.

Proof: Take CS154!

Finding Additional **NP**-Complete Problems

NP-Completeness

Theorem: Let L_1 and L_2 be languages. If $L_1 \leq_P L_2$ and L_1 is **NP**-hard, then L_2 is **NP**-hard.

Theorem: Let L_1 and L_2 be languages where $L_1 \in$ **NPC** and $L_2 \in$ **NP**. If $L_1 \leq_P L_2$, then $L_2 \in$ **NPC**.