

Complexity Theory

Part Two

Recap from Last Time

Up to this point:
“*Can we solve this problem?*”
(**Computability Theory**)



Up to this point:

“*Can we solve this problem?*”

(Computability Theory)

Starting today:

“Ok, even if we *can*, we need to consider whether the time/resources required actually make practical/feasible sense.”

(Complexity Theory)

The Complexity Class **P**

- The ***complexity class P*** (for *p*olynomial time) contains all problems that can be solved in polynomial time.
- Formally:

$$\mathbf{P} = \{ L \mid \text{There is a polynomial-time decider for } L \}$$

The Complexity Class **NP**

- The complexity class **NP** (*nondeterministic polynomial time*) contains all problems that can be verified in polynomial time.

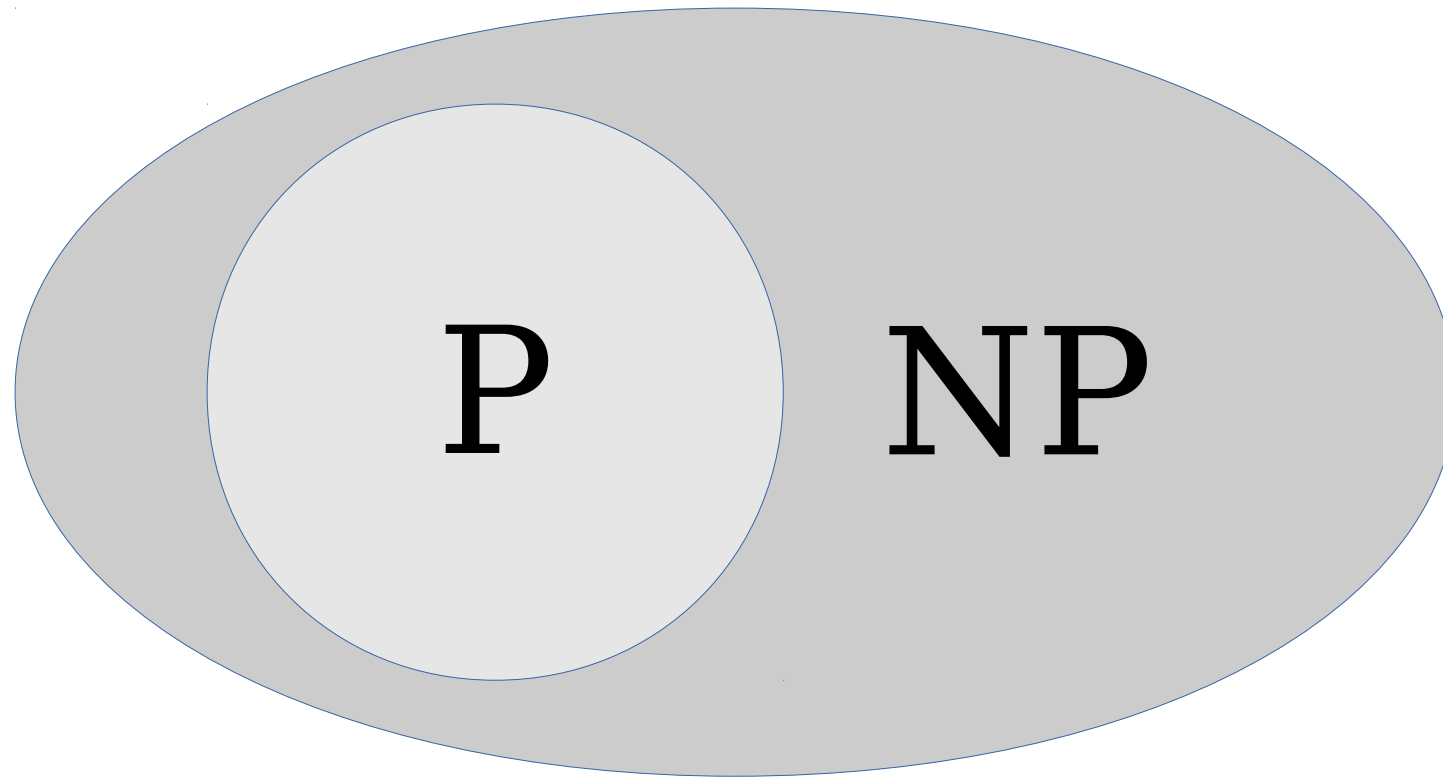
- Formally:

$$\mathbf{NP} = \{ L \mid \text{There is a polynomial-time verifier for } L \}$$

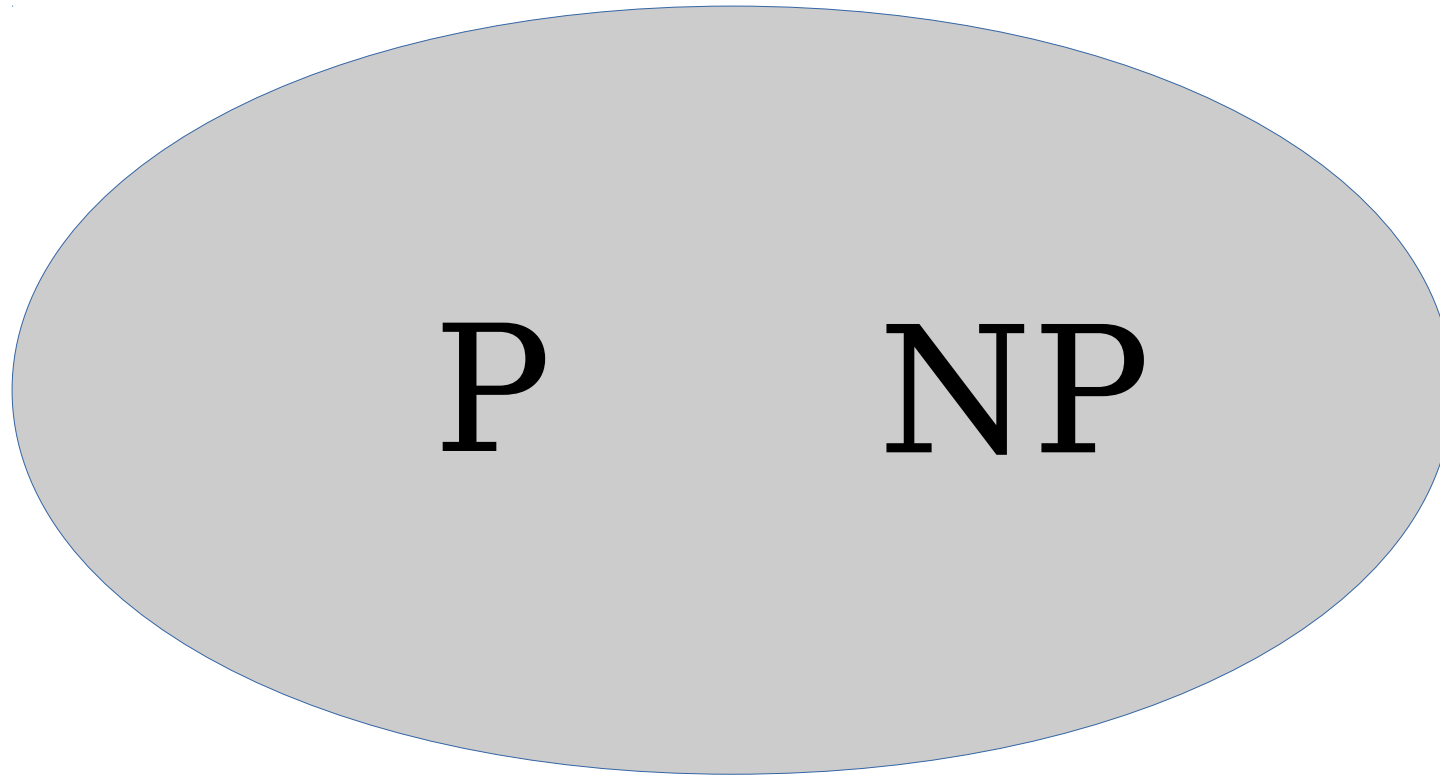
- This means a verifier V 's runtime is a polynomial in $|w|$ (that is, V 's runtime is $O(|w|^k)$ for some integer k).

Does **P** = **NP**?

Which Picture is Correct?

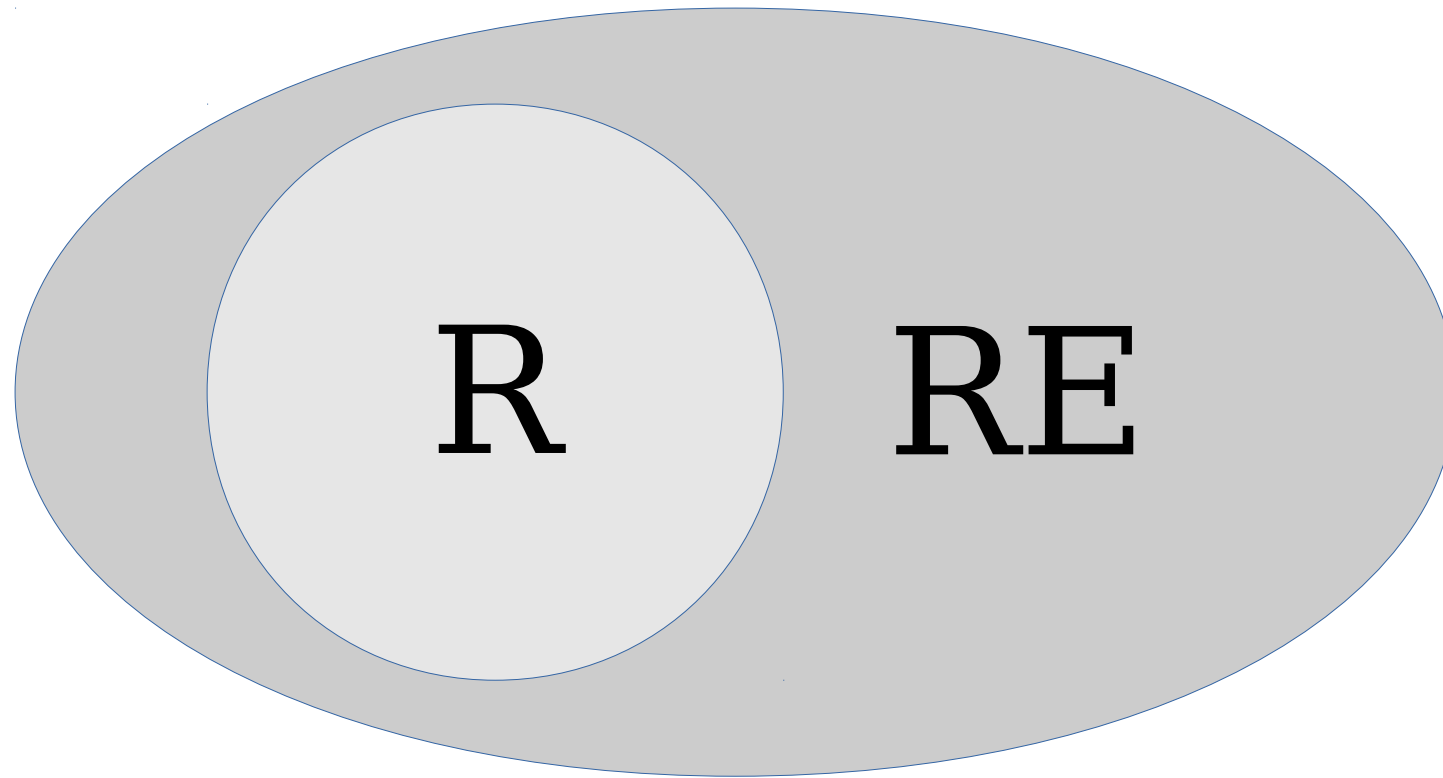


Which Picture is Correct?



In CS103 so far, we have
seen strong precedent for
both outcomes!

(remember: if you take out the timing aspect, this is the picture of languages with a decider and languages with a verifier)

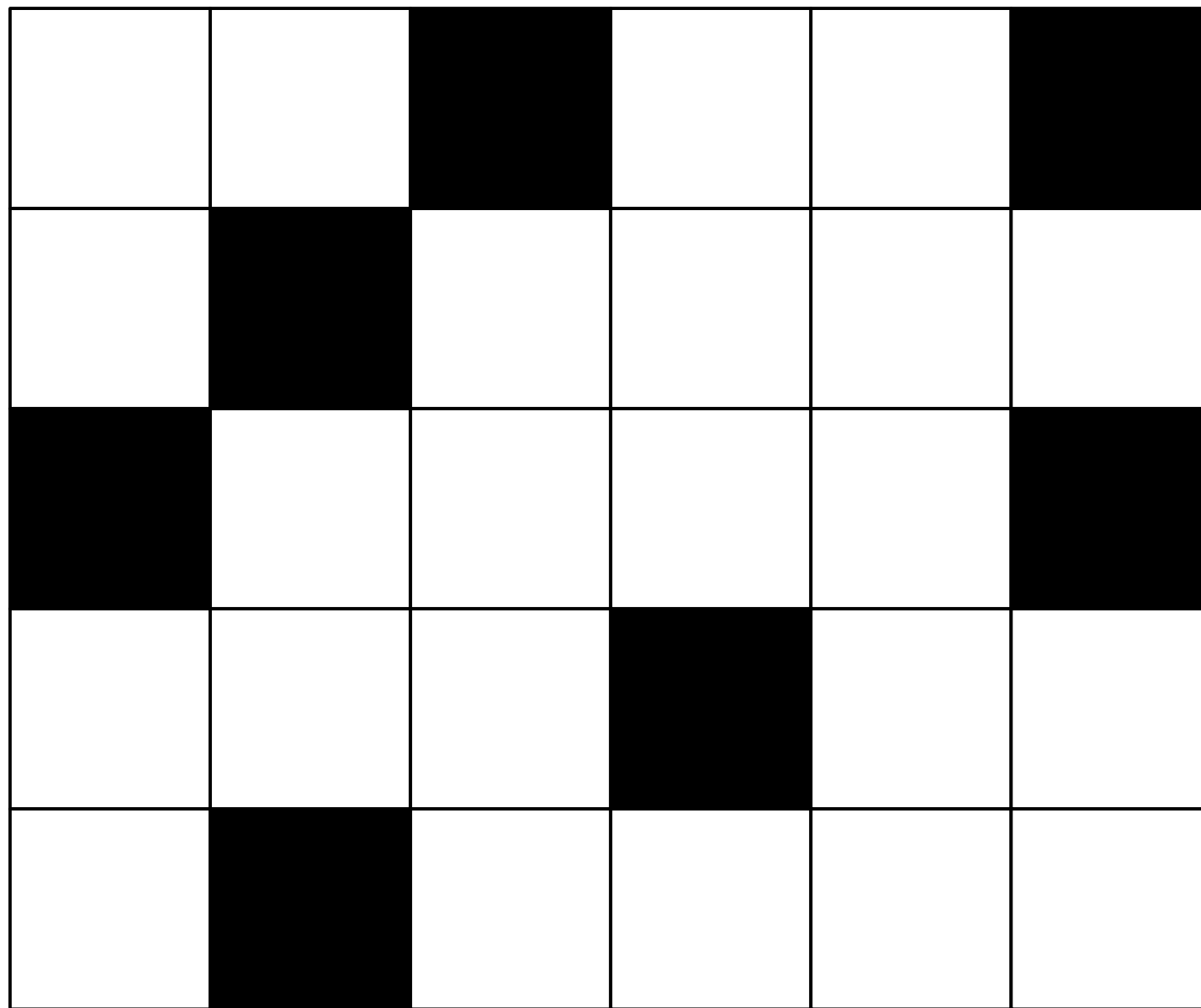


(remember: this is the picture of languages
with a deterministic finite automaton and
languages a nondeterministic finite automaton)

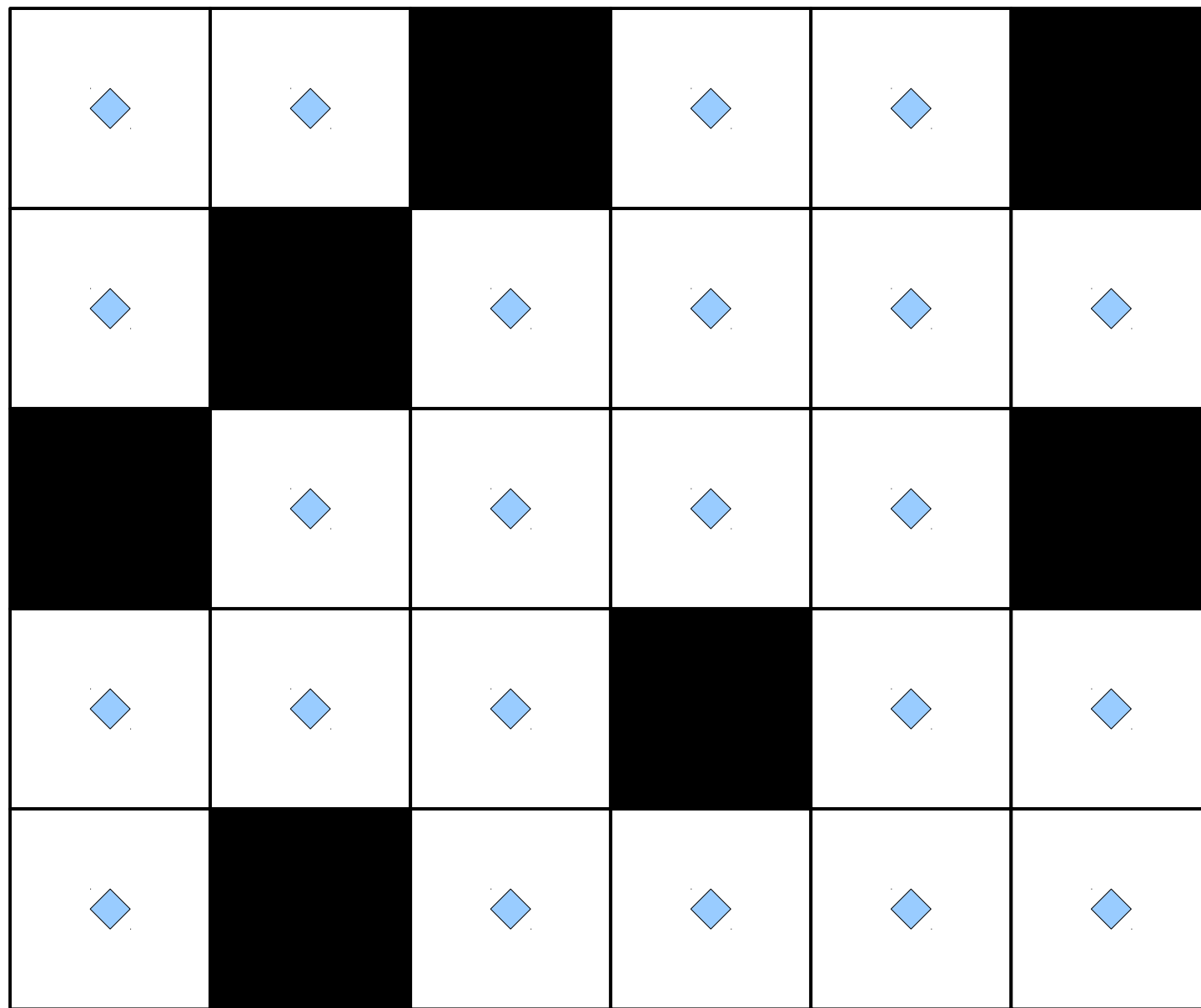


DFA NFA

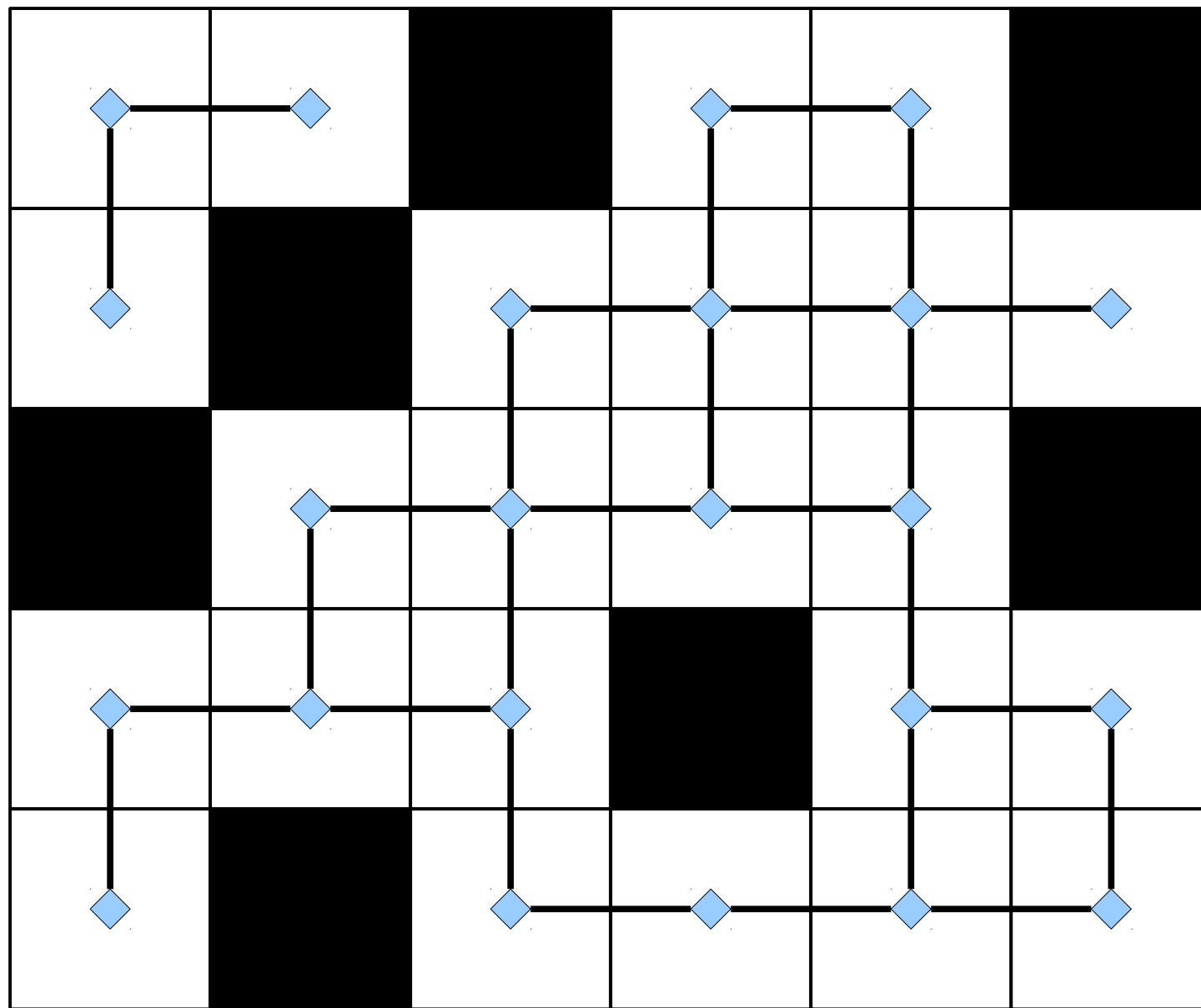
Solving Domino Tiling



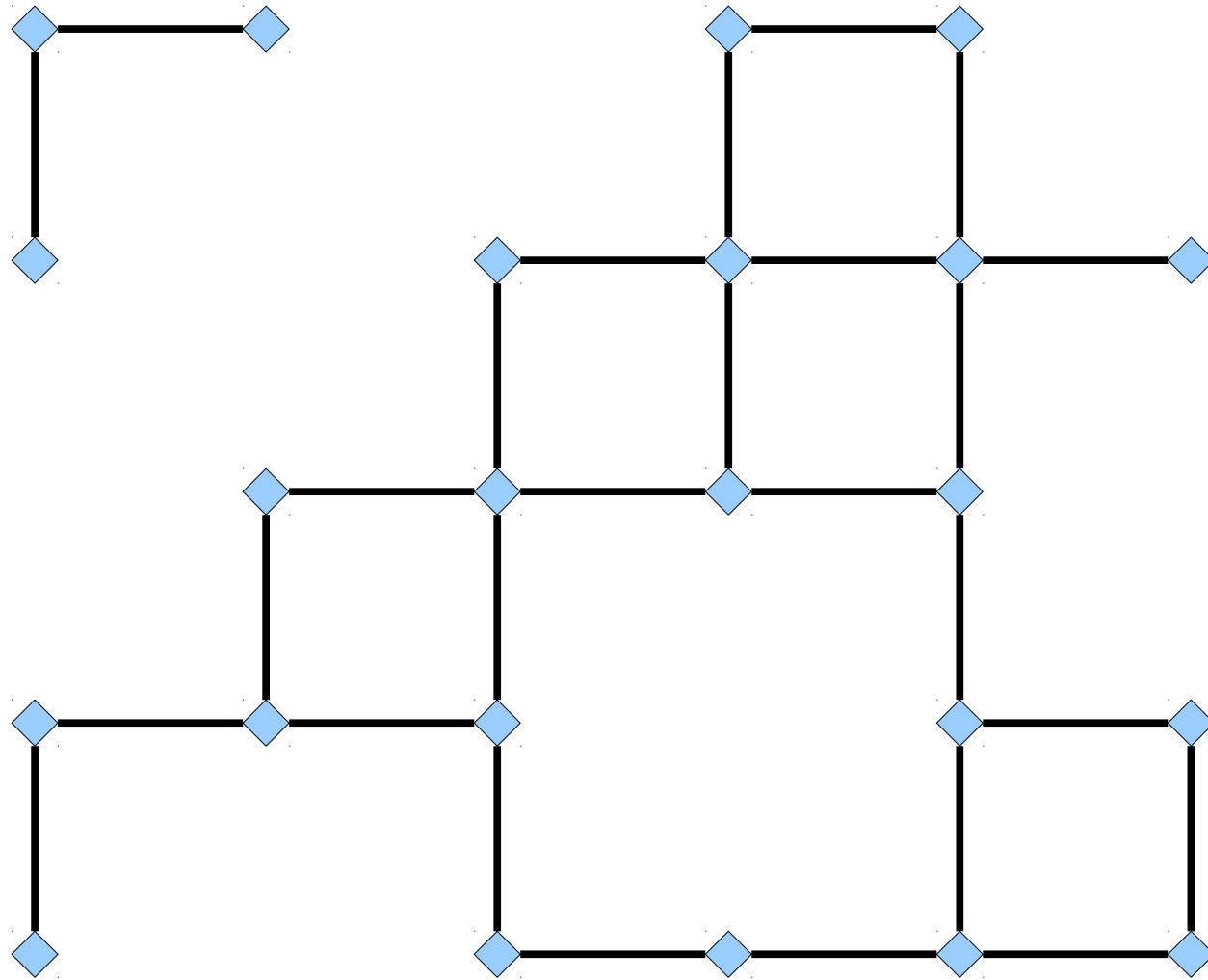
Solving Domino Tiling



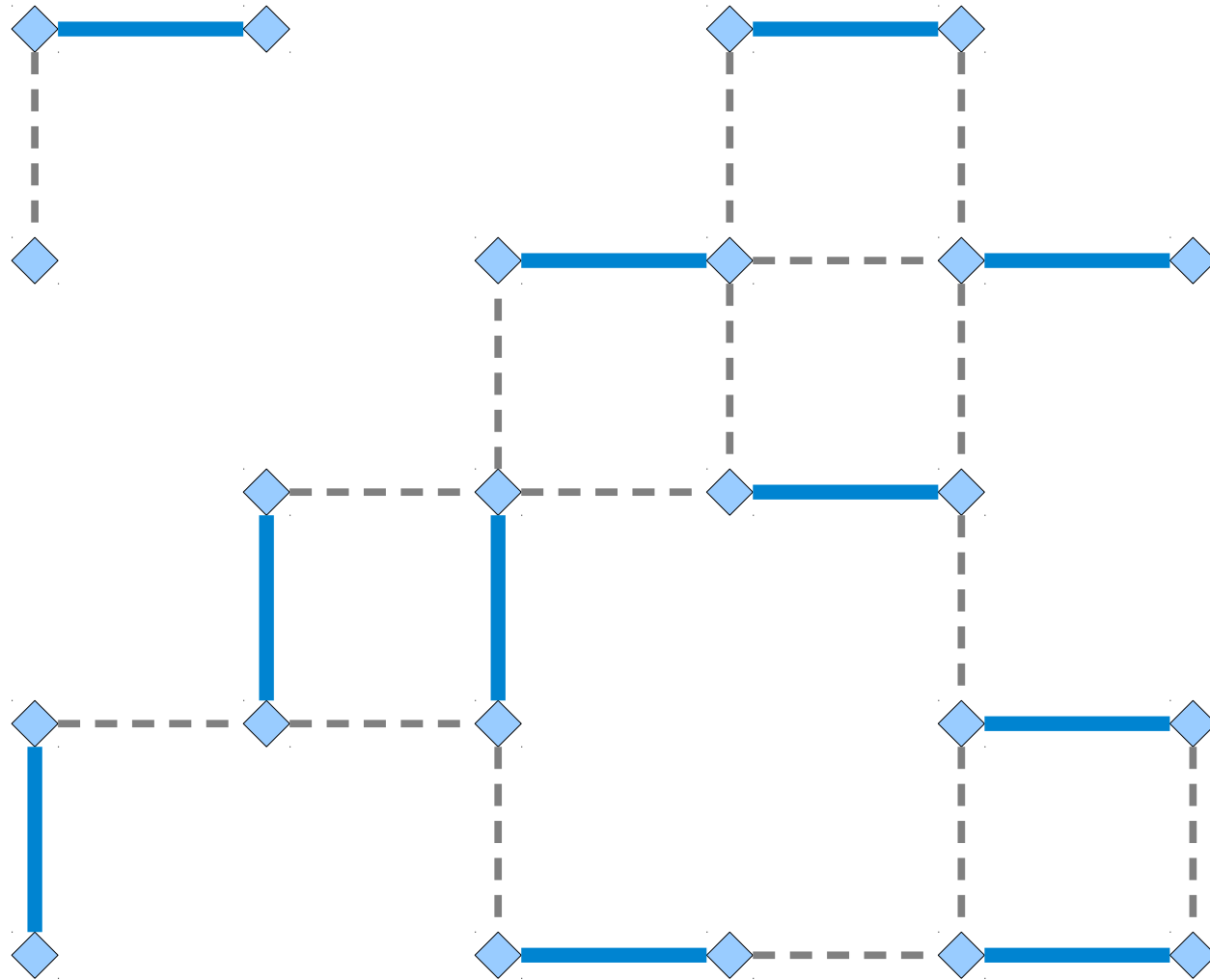
Solving Domino Tiling



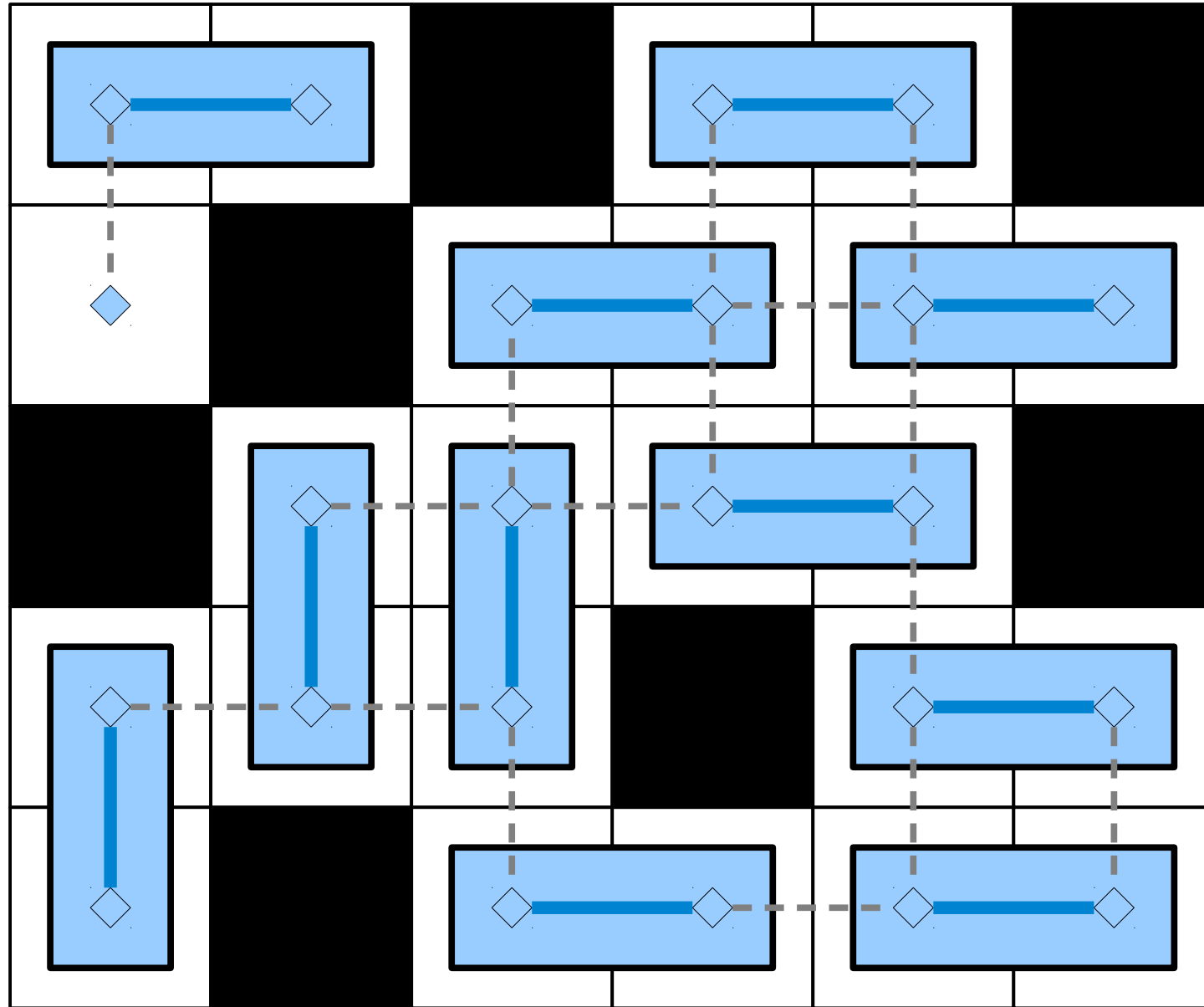
Solving Domino Tiling



Solving Domino Tiling



Solving Domino Tiling



Intuition:

Tiling a grid with dominoes can't be “harder” than solving maximum matching, because if we can solve maximum matching efficiently, we can solve domino tiling efficiently.

New Stuff!

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 ?
- It's known that this problem can be solved in polynomial time (use DFS or BFS).
- 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

Connectors

RGB to USB

VGA to DisplayPort

DB13W3 to CATV

DisplayPort to RGB

DB13W3 to HDMI

DVI to DB13W3

S-Video to DVI

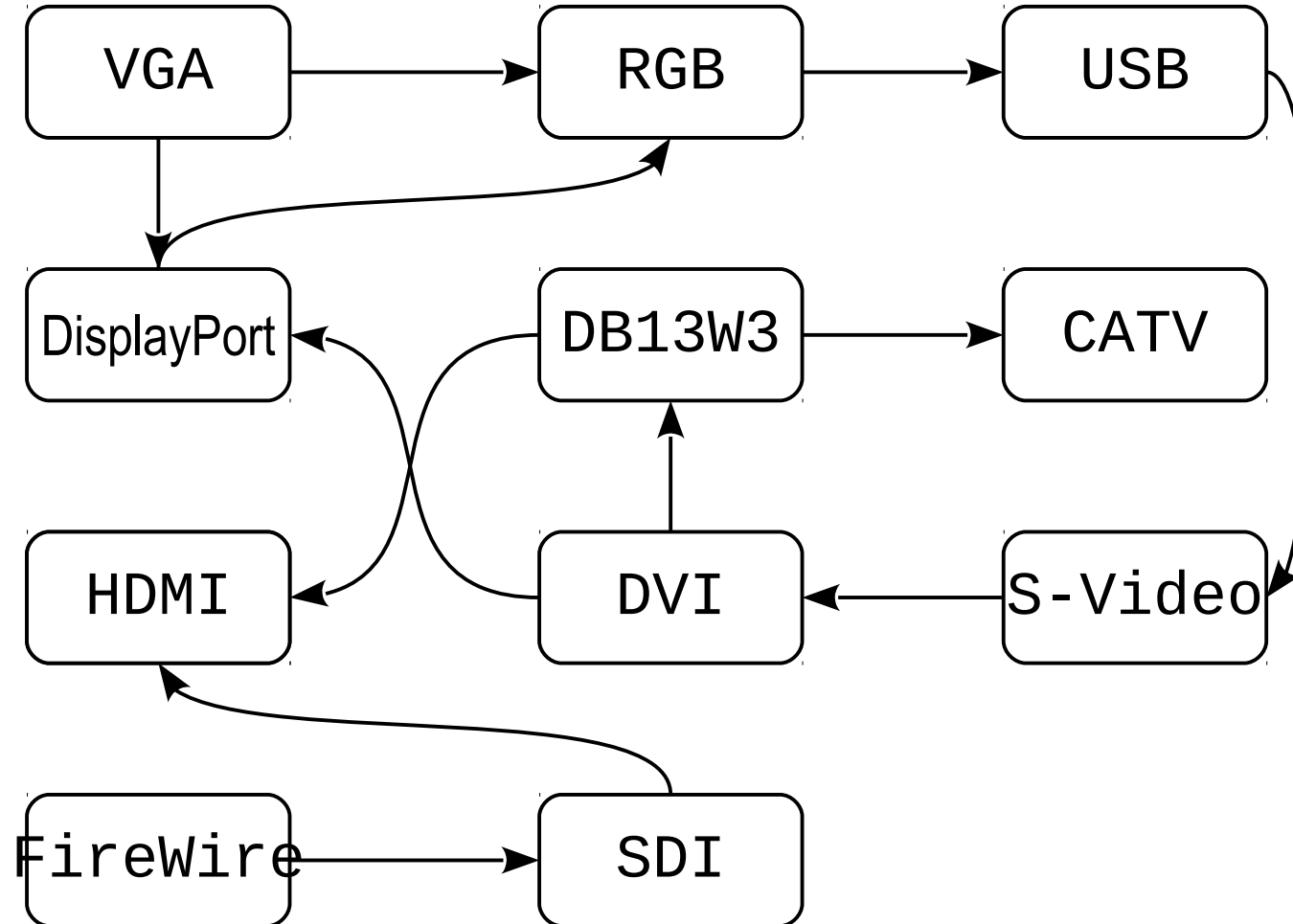
FireWire to SDI

VGA to RGB

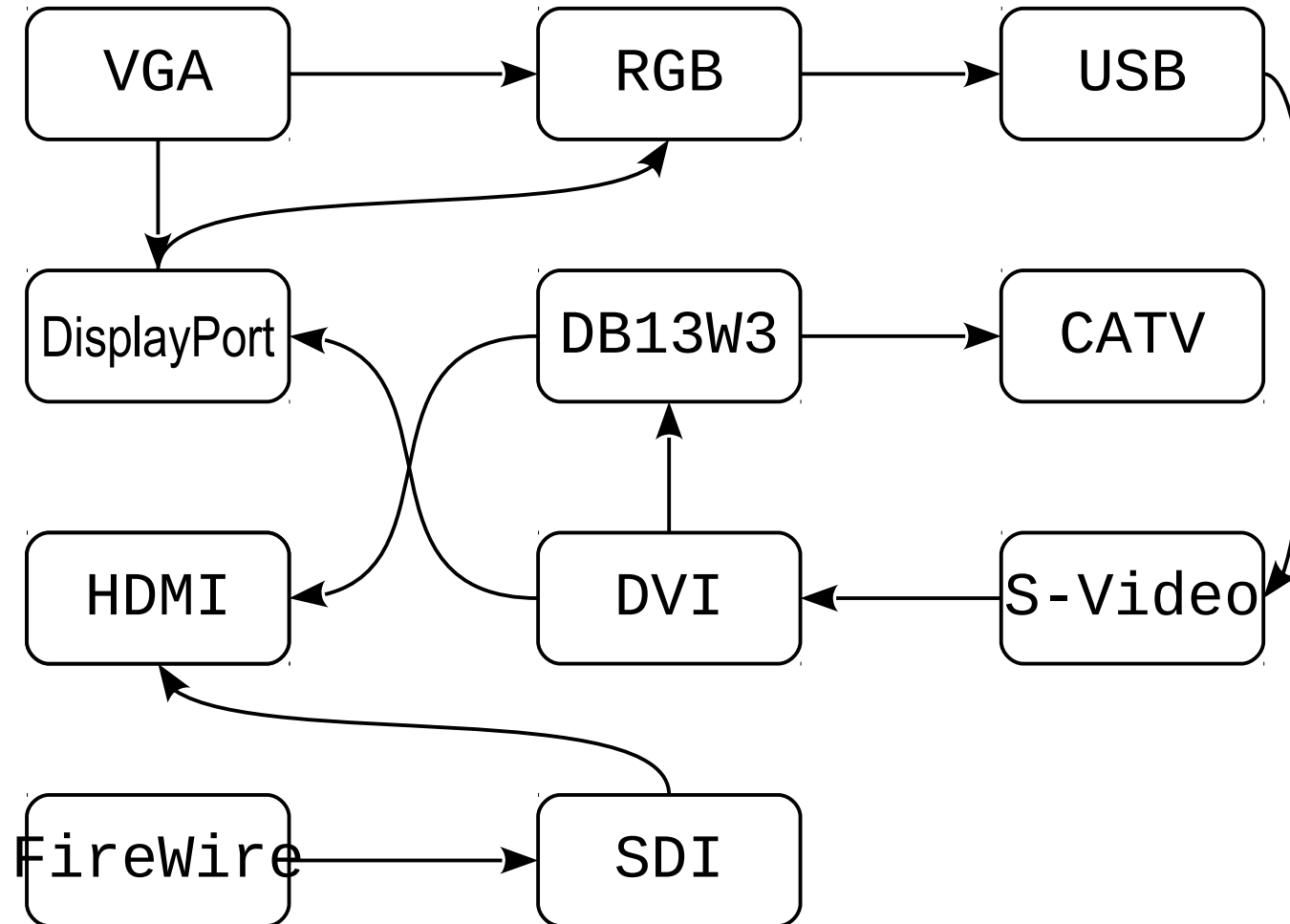
DVI to DisplayPort

USB to S-Video

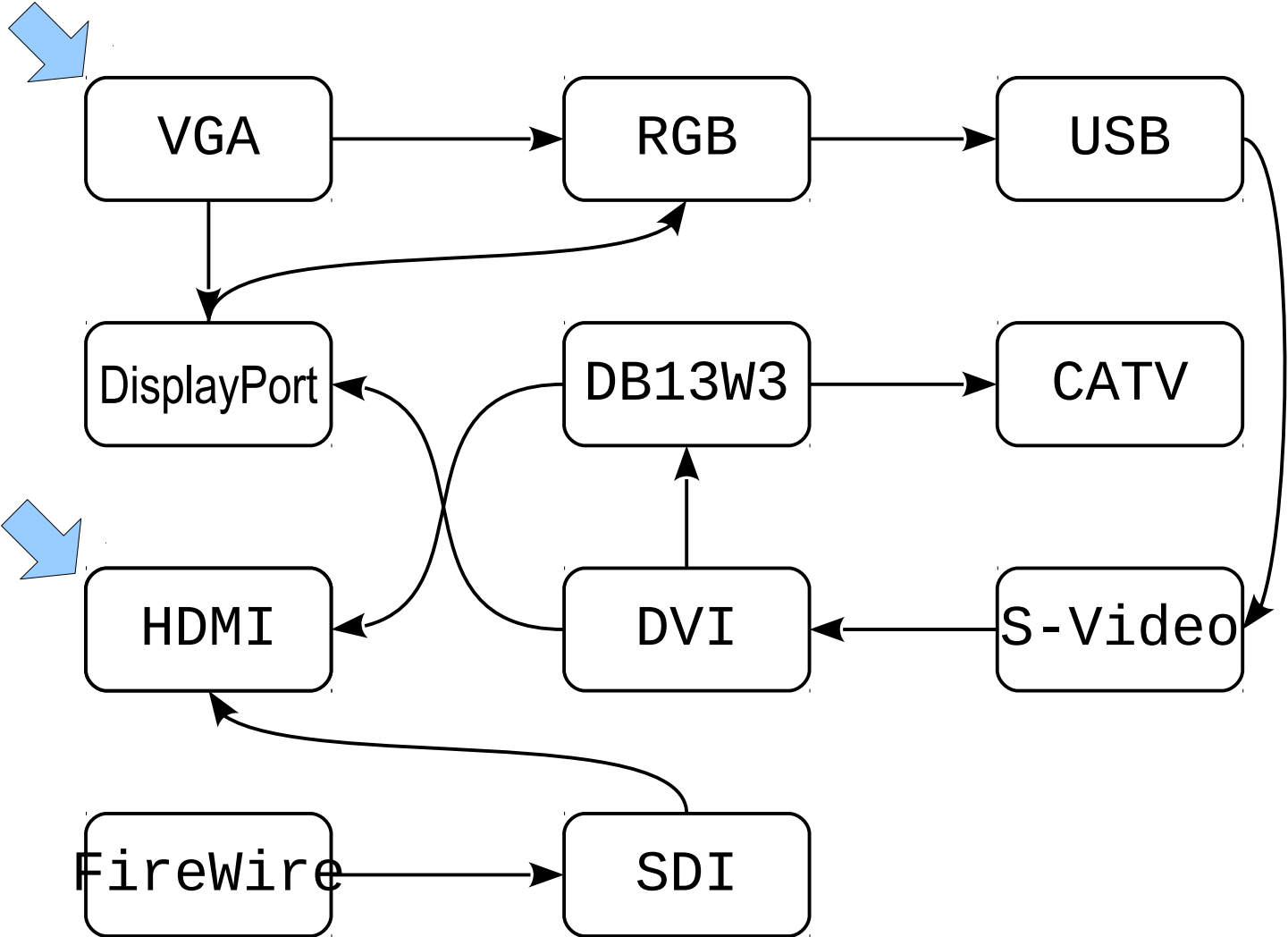
SDI to HDMI



Converter Conundrums



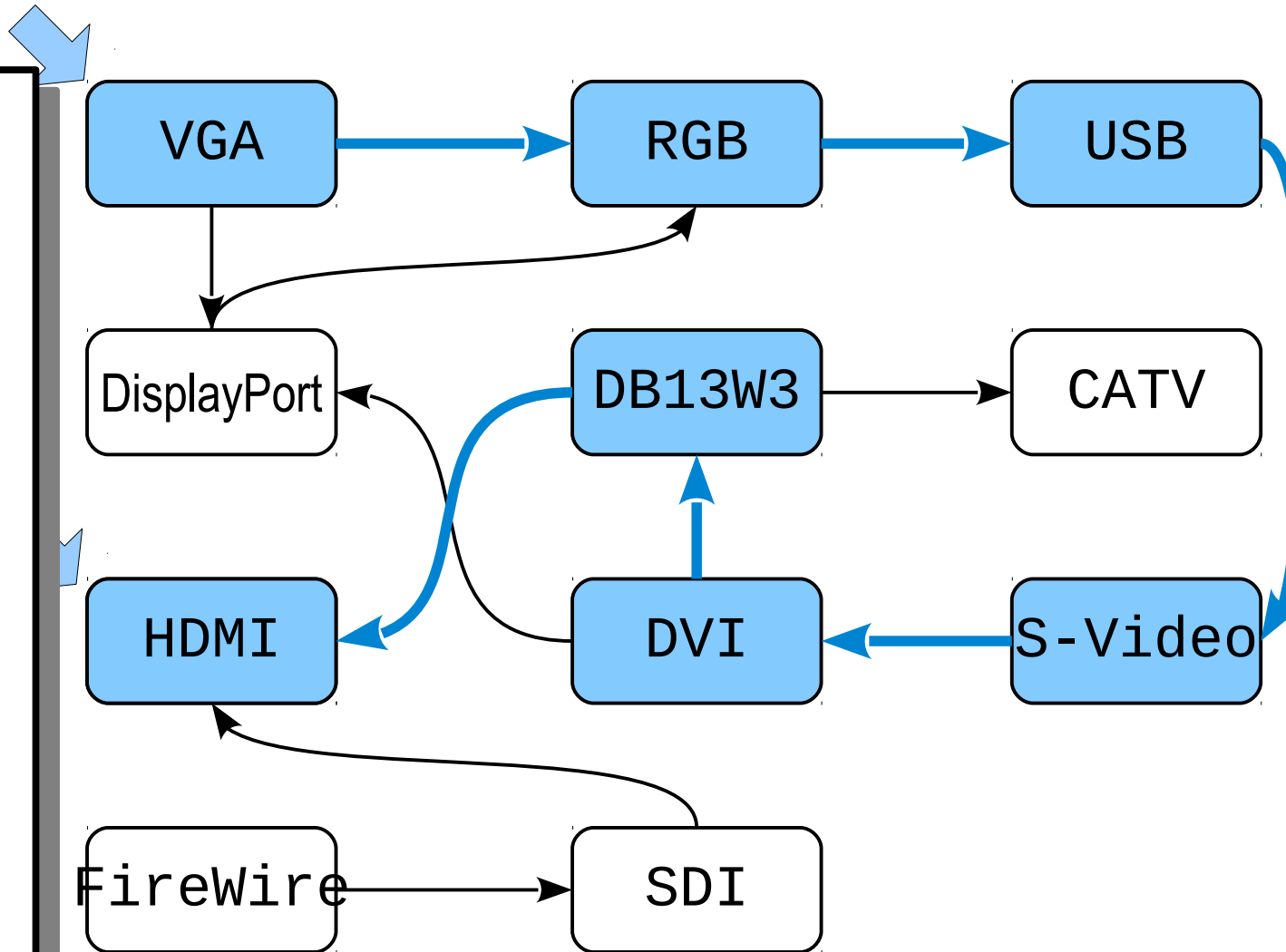
Converter Conundrums



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



In Pseudocode

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

Based on this connection between plugging a laptop into a projector and determining reachability, which of the following statements would be more proper to conclude?

- A. Plugging a laptop into a projector isn't any more difficult than computing reachability in a directed graph.
- B. Computing reachability in a directed graph isn't any more difficult than plugging a laptop into a projector.

Answer at **PolleEv.com/cs103** or
text **CS103** to **22333** once to join, then **A** or **B**.

Intuition:

Finding a way to plug a computer into a projector can't be “harder” than determining reachability in a graph, since if we can determine reachability in a graph, we can find a way to plug a computer into a projector.

```
bool solveProblemA(string input) {  
    return solveProblemB(transform(input));  
}
```

Intuition:

Problem *A* can't be “harder” than problem *B*, because solving problem *B* lets us solve problem *A*.

```
bool solveProblemA(string input) {  
    return solveProblemB(transform(input));  
}
```

- If A and B are problems where it's possible to solve problem A using the strategy shown above*, we write

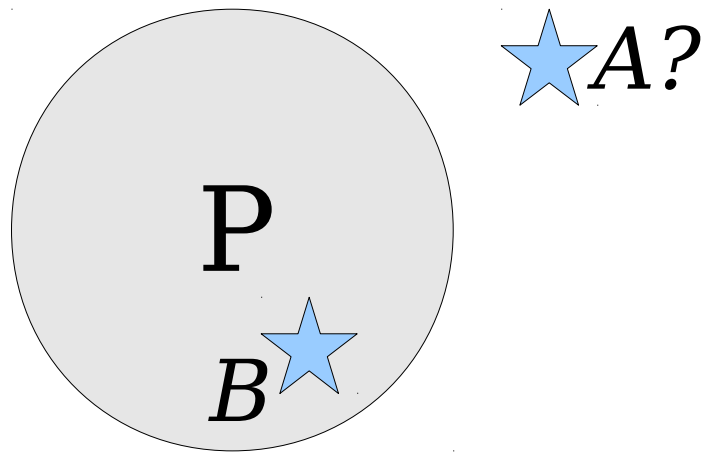
$$A \leq_p B.$$

- We say that ***A is polynomial-time reducible to B.***

* Assuming that transform runs in polynomial time.

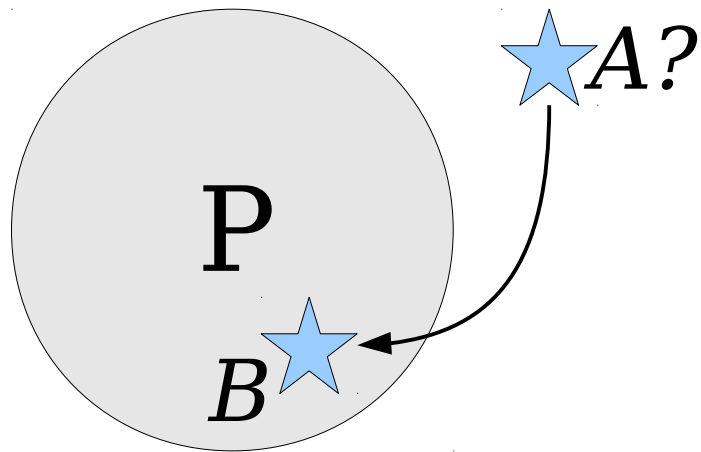
Polynomial-Time Reductions

- If $A \leq_p B$ and $B \in \mathbf{P}$, then $A \in \mathbf{P}$.



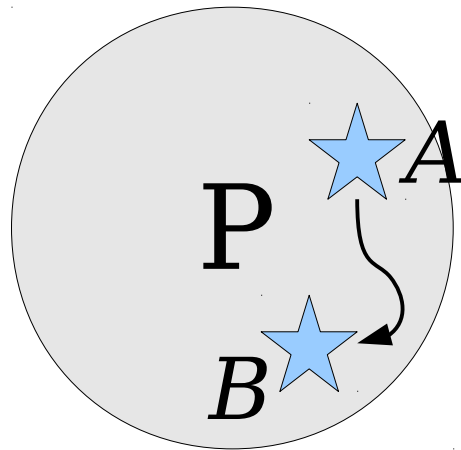
Polynomial-Time Reductions

- If $A \leq_p B$ and $B \in \mathbf{P}$, then $A \in \mathbf{P}$.



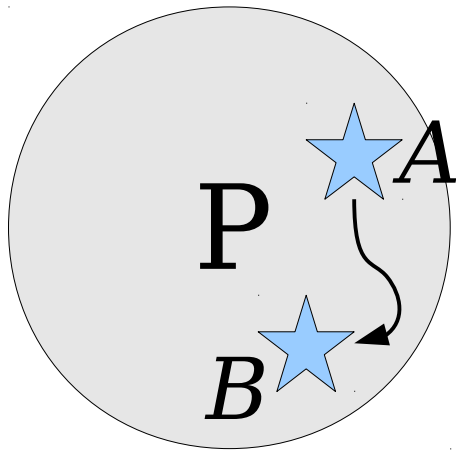
Polynomial-Time Reductions

- If $A \leq_p B$ and $B \in \mathbf{P}$, then $A \in \mathbf{P}$.



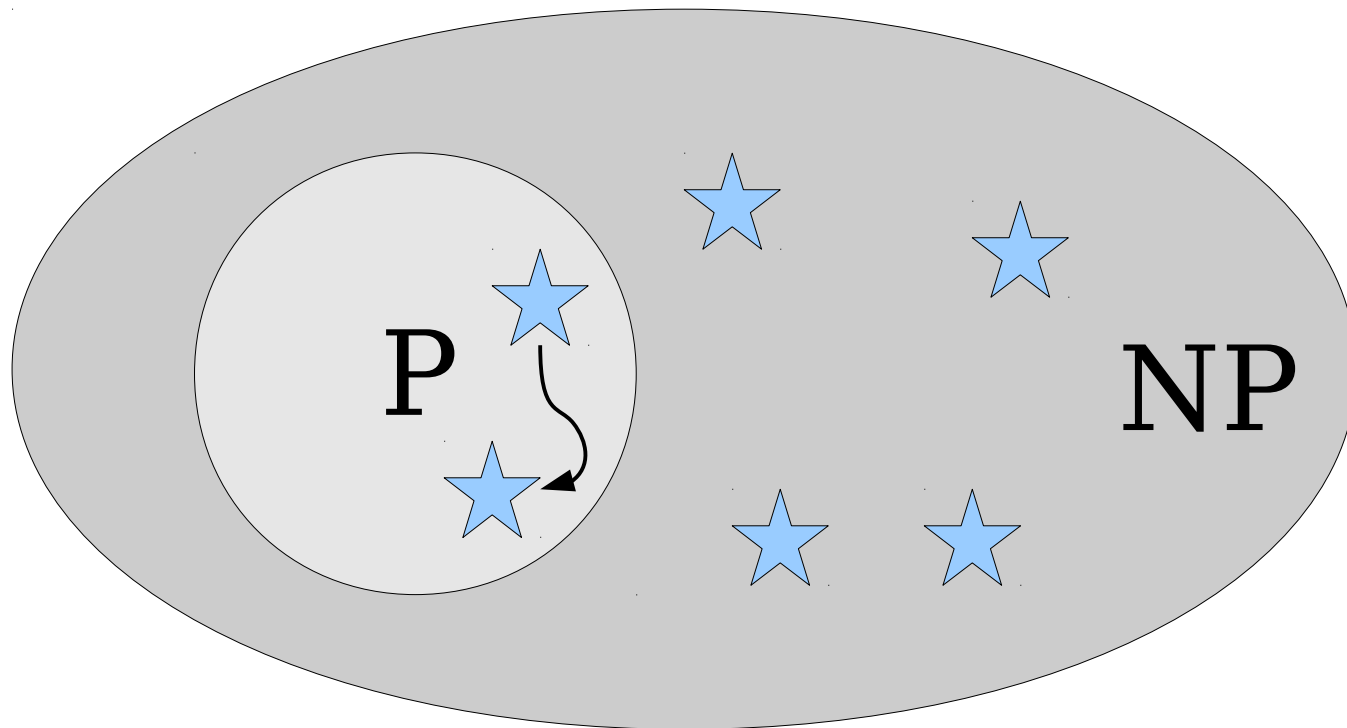
Polynomial-Time Reductions

- If $A \leq_p B$ and $B \in \mathbf{P}$, then $A \in \mathbf{P}$.
- If $A \leq_p B$ and $B \in \mathbf{NP}$, then $A \in \mathbf{NP}$.



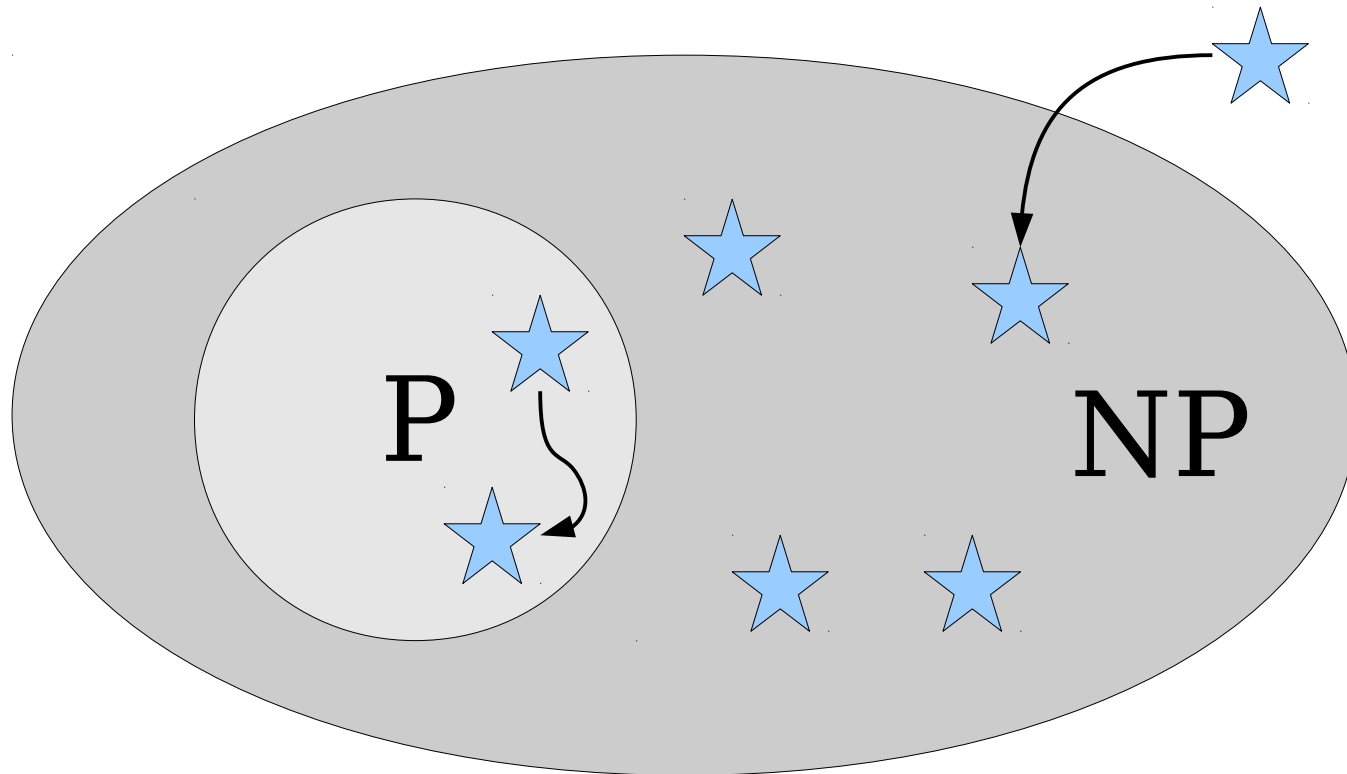
Polynomial-Time Reductions

- If $A \leq_p B$ and $B \in \mathbf{P}$, then $A \in \mathbf{P}$.
- If $A \leq_p B$ and $B \in \mathbf{NP}$, then $A \in \mathbf{NP}$.



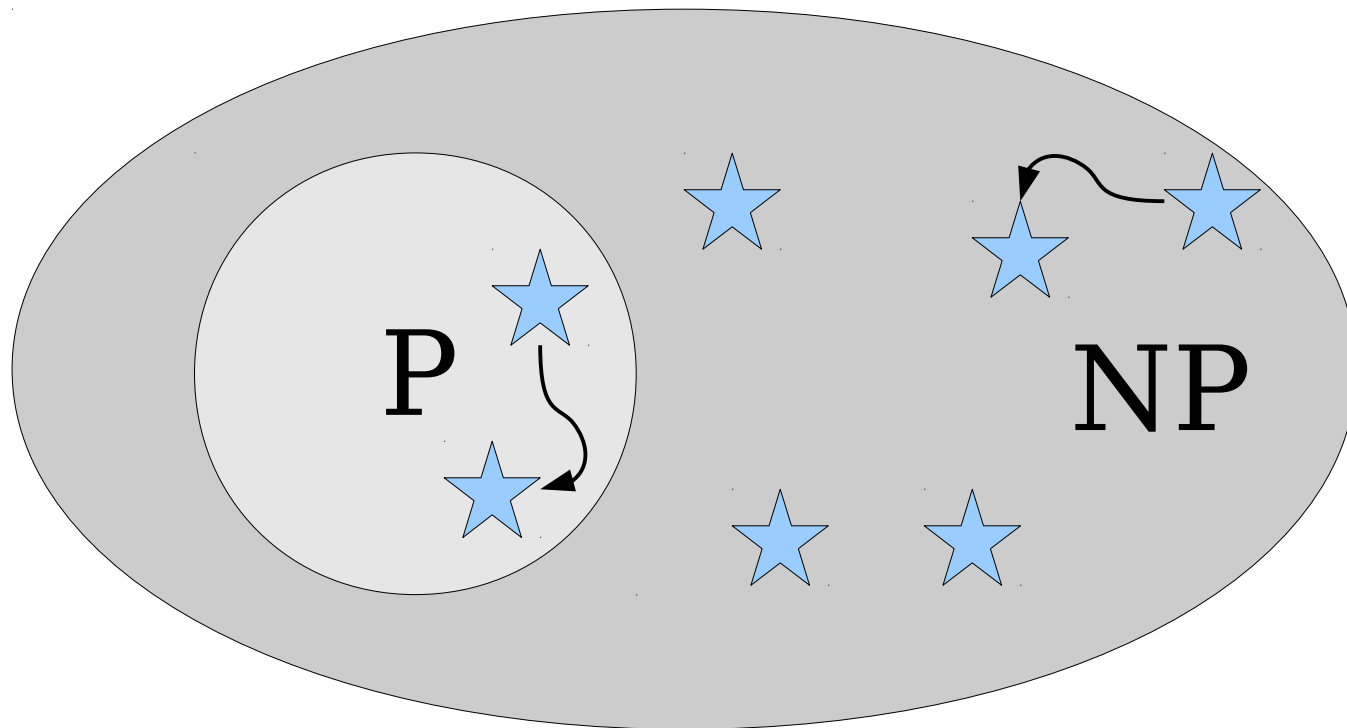
Polynomial-Time Reductions

- If $A \leq_p B$ and $B \in \mathbf{P}$, then $A \in \mathbf{P}$.
- If $A \leq_p B$ and $B \in \mathbf{NP}$, then $A \in \mathbf{NP}$.



Polynomial-Time Reductions

- If $A \leq_p B$ and $B \in \mathbf{P}$, then $A \in \mathbf{P}$.
- If $A \leq_p B$ and $B \in \mathbf{NP}$, then $A \in \mathbf{NP}$.



This \leq_p relation lets us rank the relative difficulties of problems in **P** and **NP**.

What else can we do with it?

NP-Hardness and **NP**-Completeness

Question: What makes a problem
hard to solve?

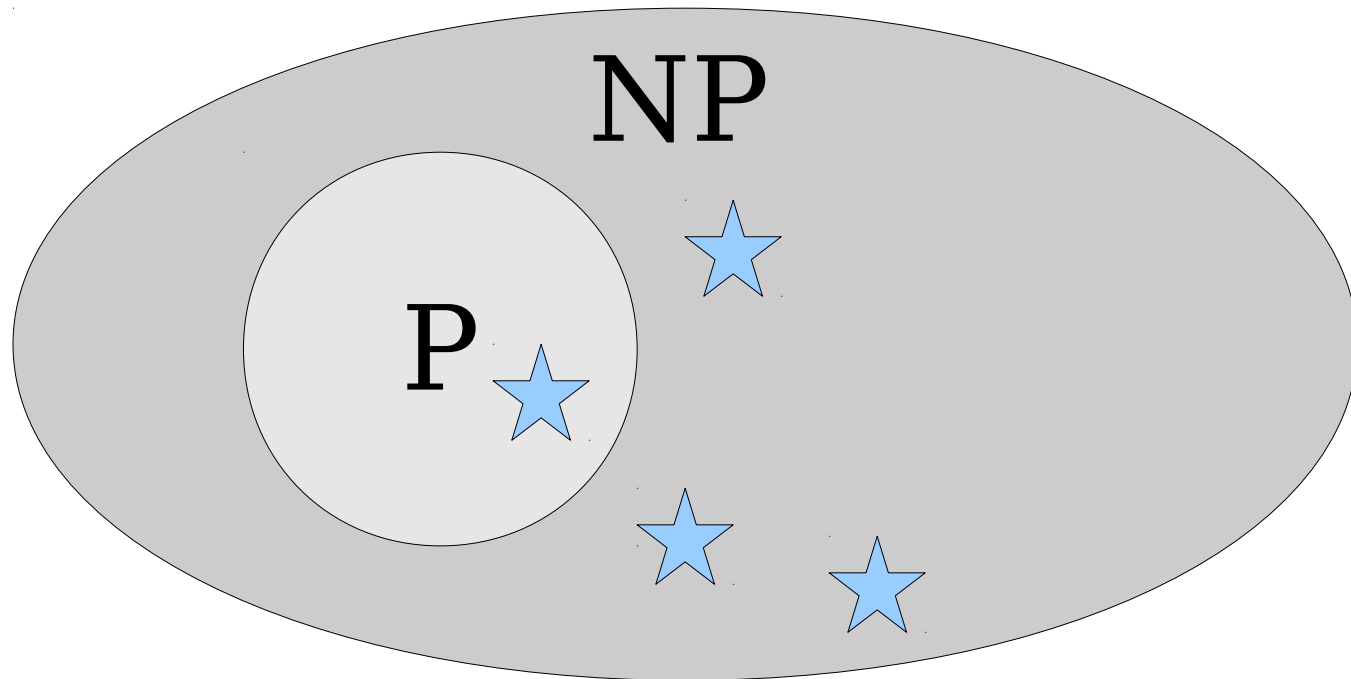
Intuition: If $A \leq_p B$, then problem B is at least as hard* as problem A .

* for some definition of “at least as hard as.”

Intuition: To show that some problem is hard, show that lots of other problems reduce to it.

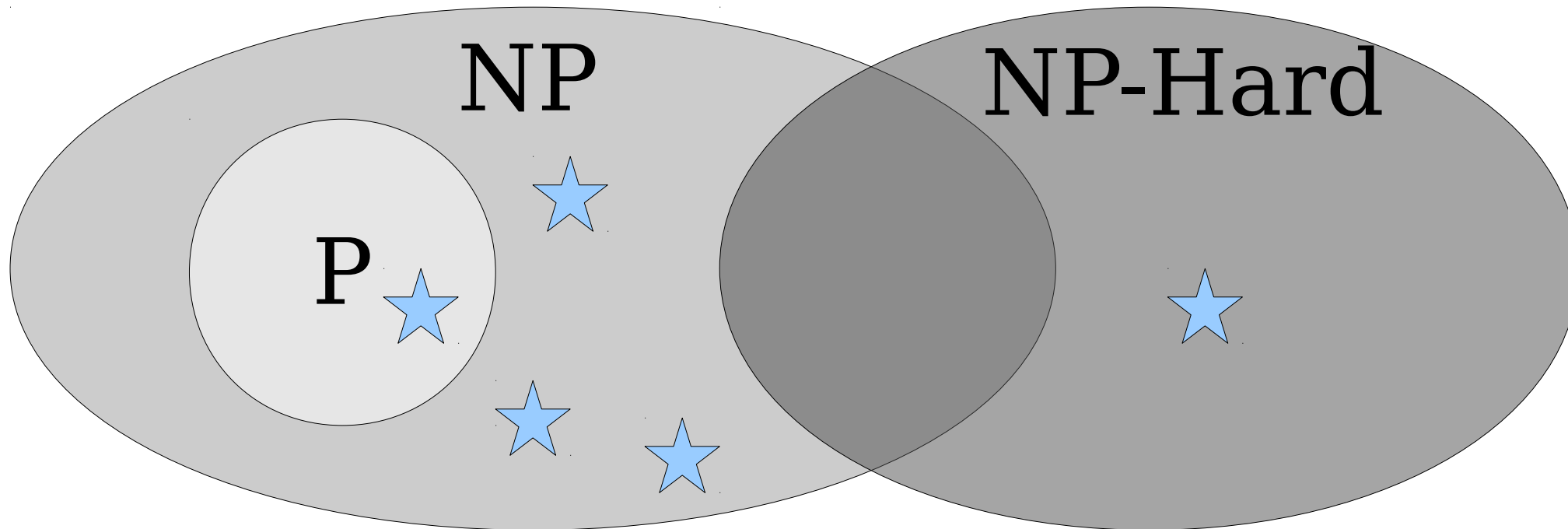
NP-Hardness

- A language L is called **NP-hard** if for *every* $A \in \mathbf{NP}$, we have $A \leq_p L$.



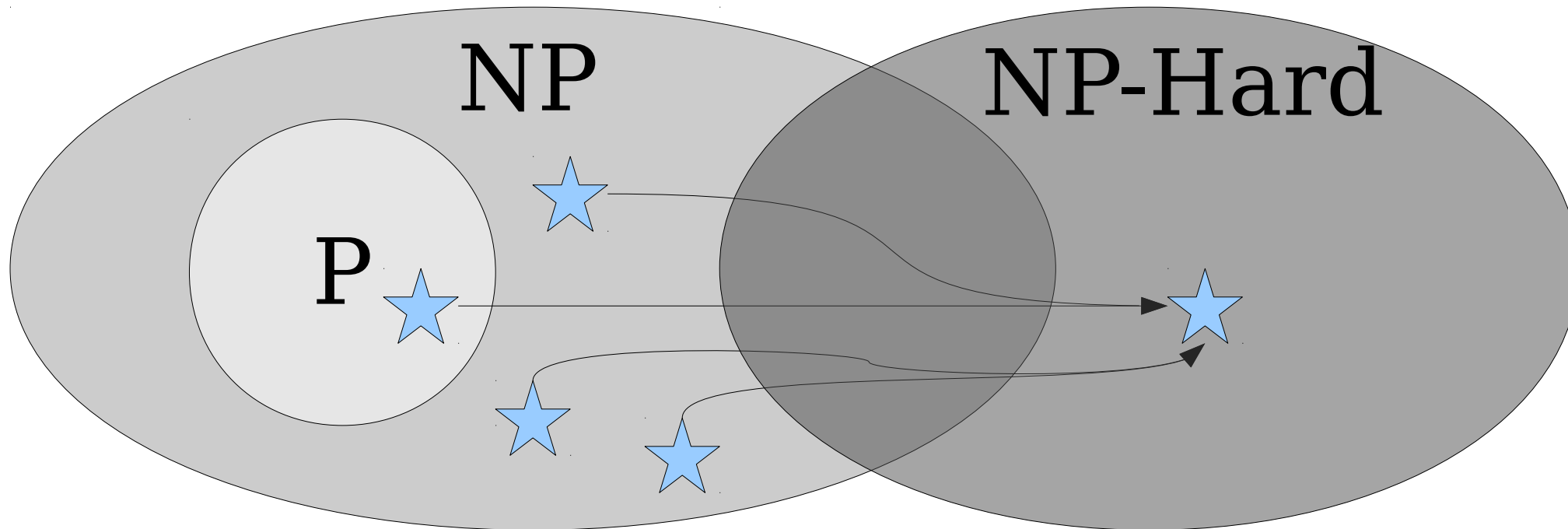
NP-Hardness

- A language L is called **NP-hard** if for *every* $A \in \mathbf{NP}$, we have $A \leq_p L$.



NP-Hardness

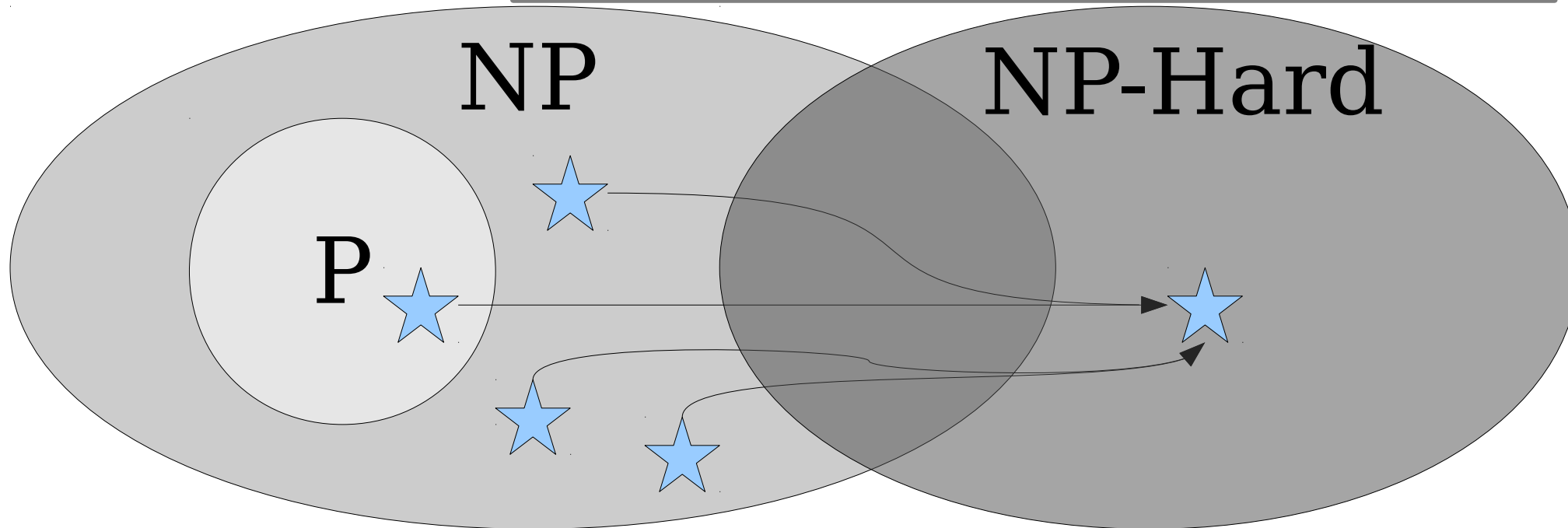
- A language L is called **NP-hard** if for *every* $A \in \mathbf{NP}$, we have $A \leq_p L$.



NP-Hardness

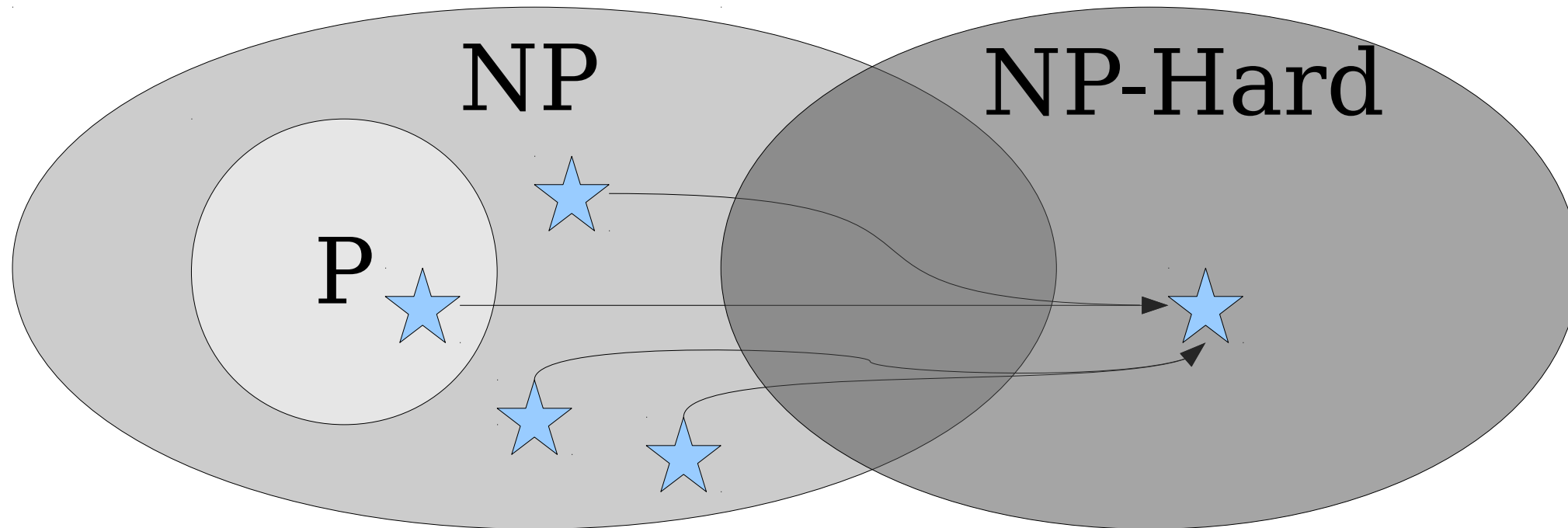
- A language L is called **NP-hard** if for every $A \in \mathbf{NP}$, we have $A \leq_p L$.

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



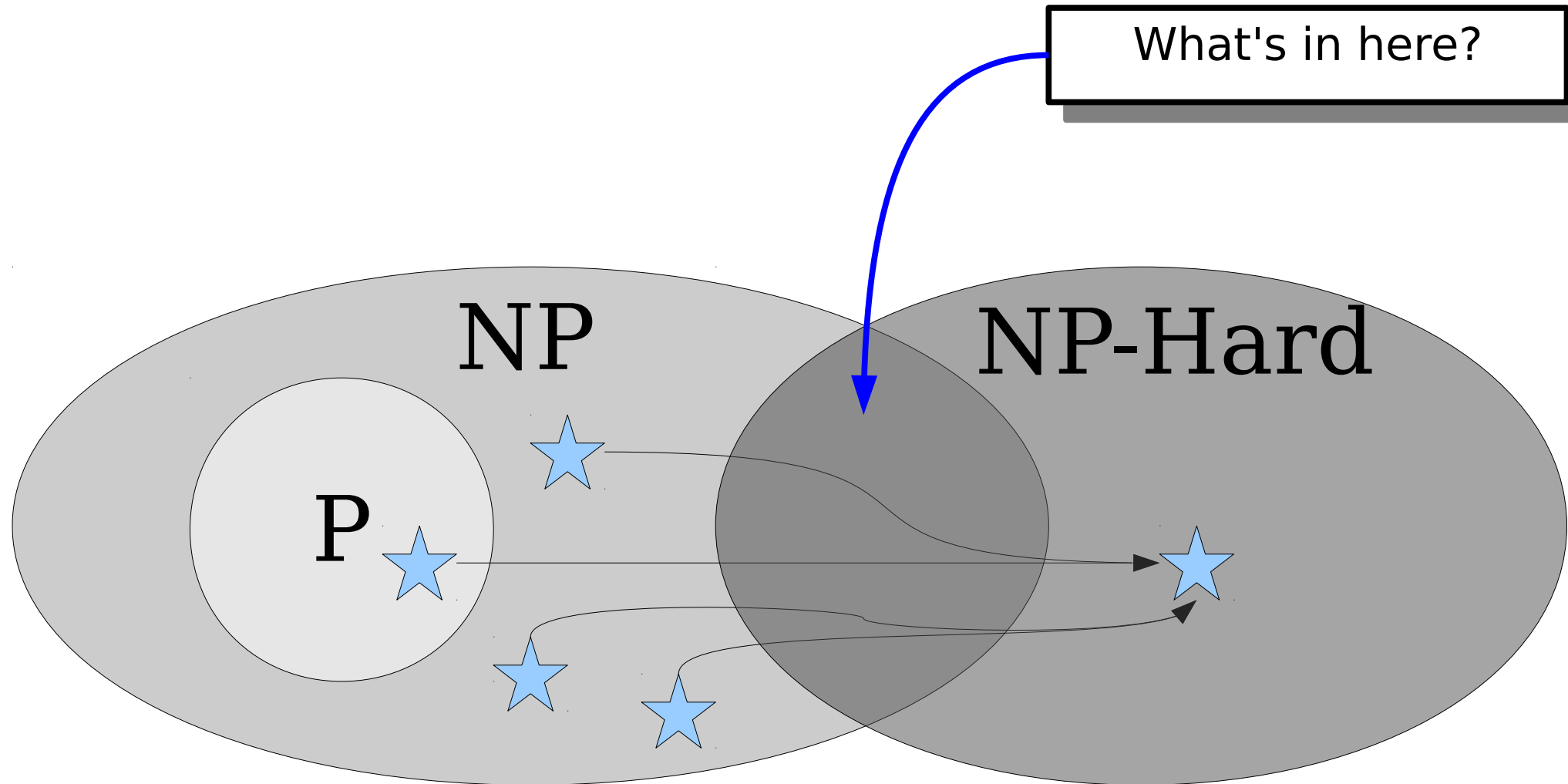
NP-Hardness

- A language L is called **NP-hard** if for *every* $A \in \mathbf{NP}$, we have $A \leq_p L$.



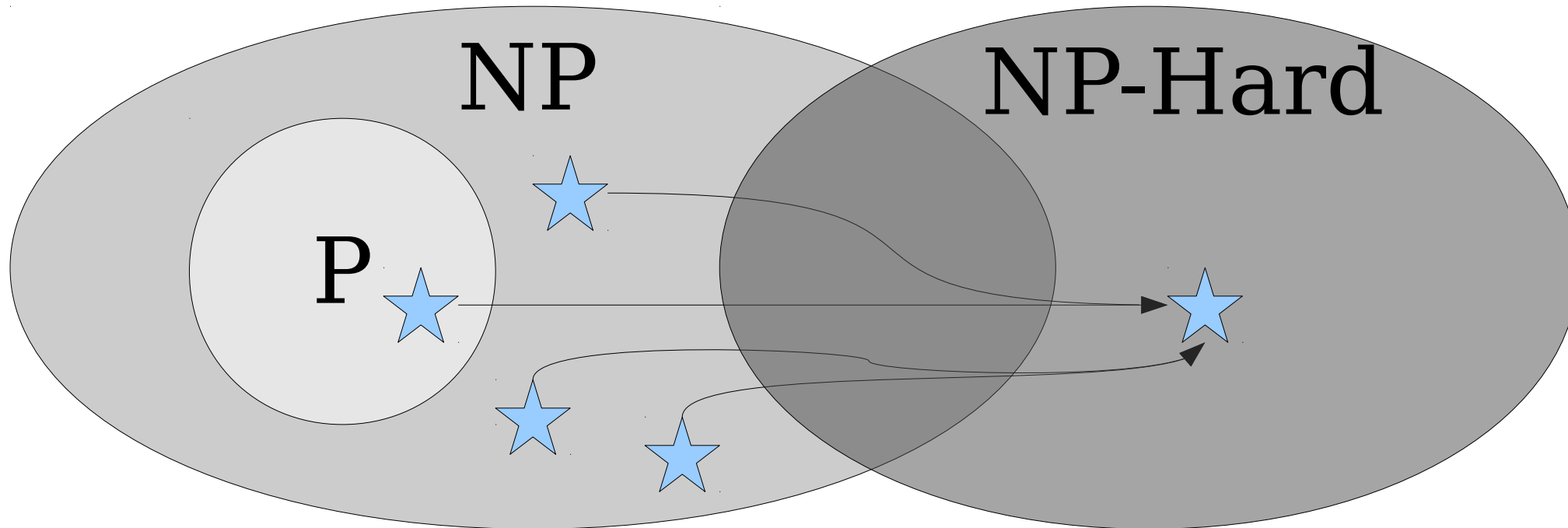
NP-Hardness

- A language L is called **NP-hard** if for every $A \in \mathbf{NP}$, we have $A \leq_p L$.



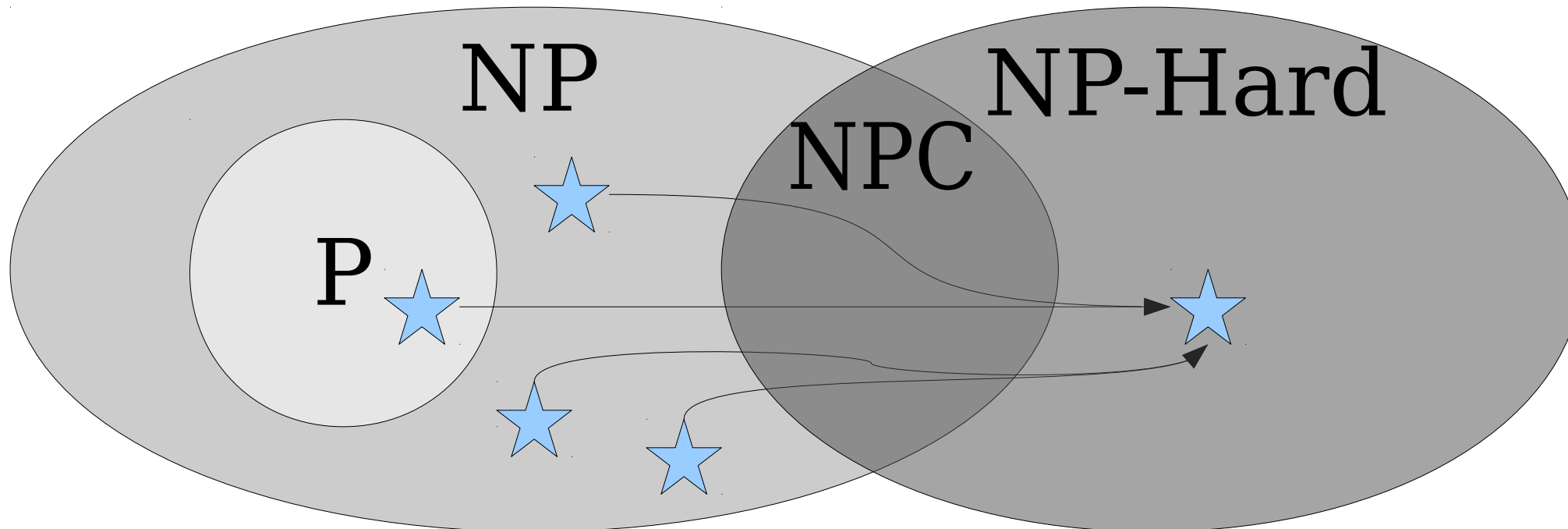
NP-Hardness

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



NP-Hardness

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

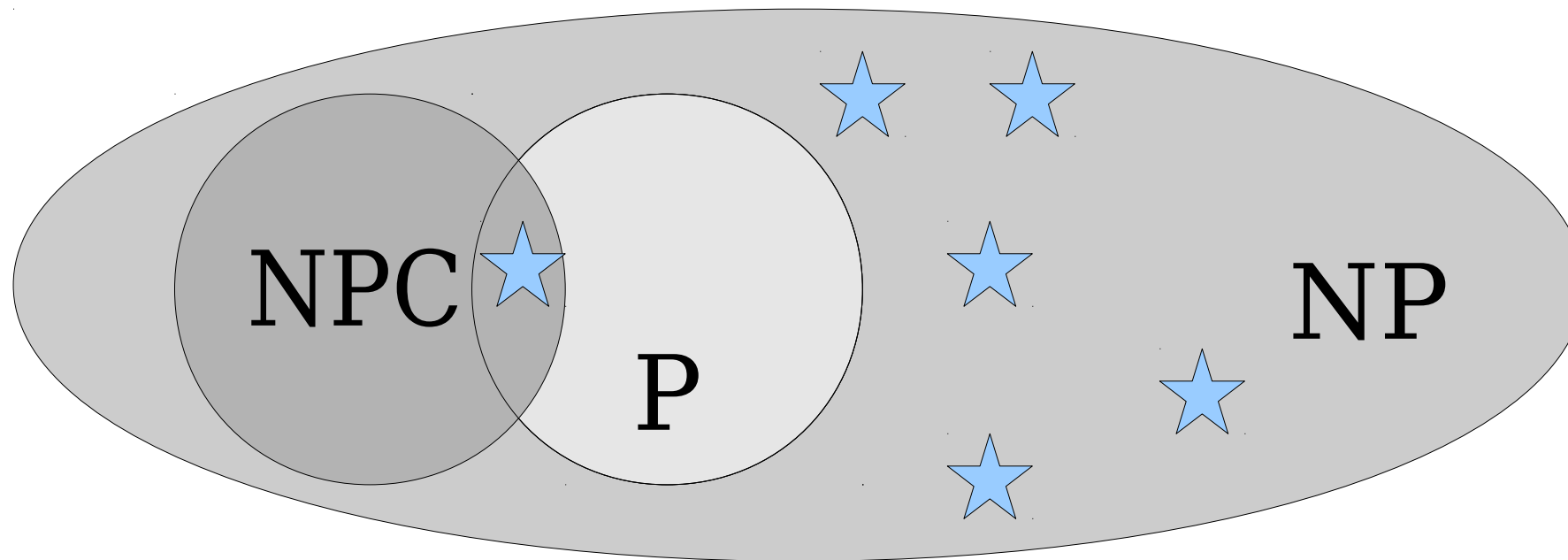


The Tantalizing Truth

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

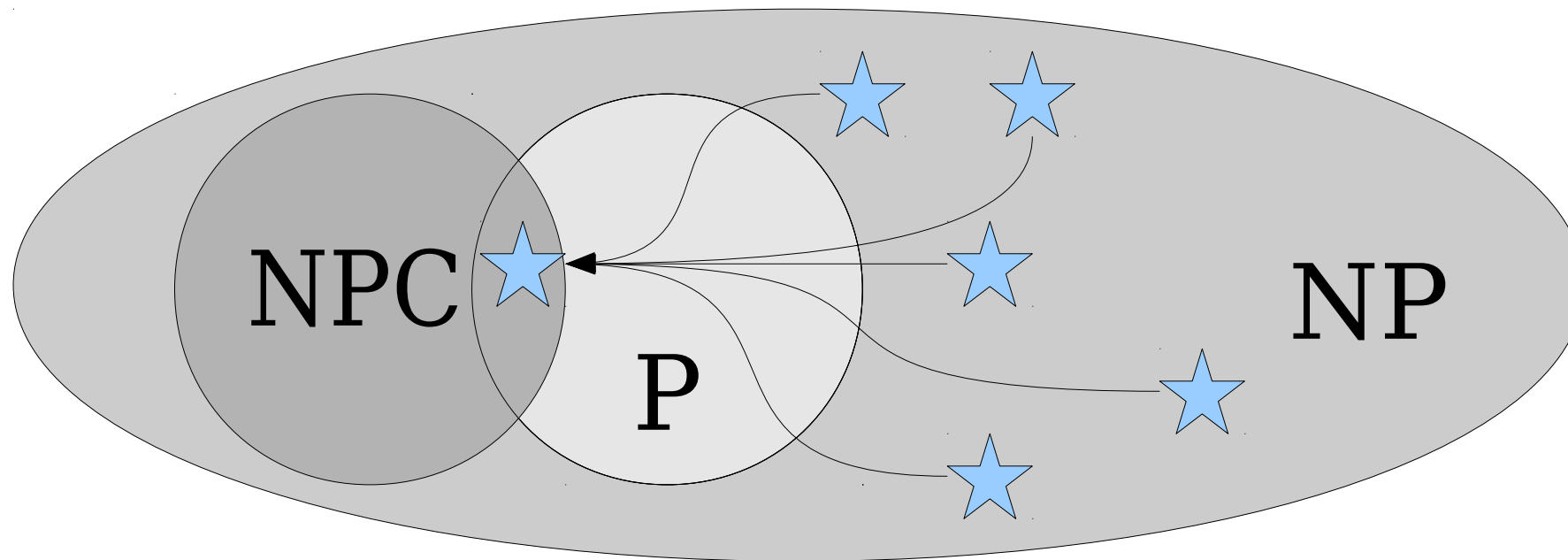
The Tantalizing Truth

Theorem: If *any* NP-complete language is in \mathbf{P} , then $\mathbf{P} = \mathbf{NP}$.



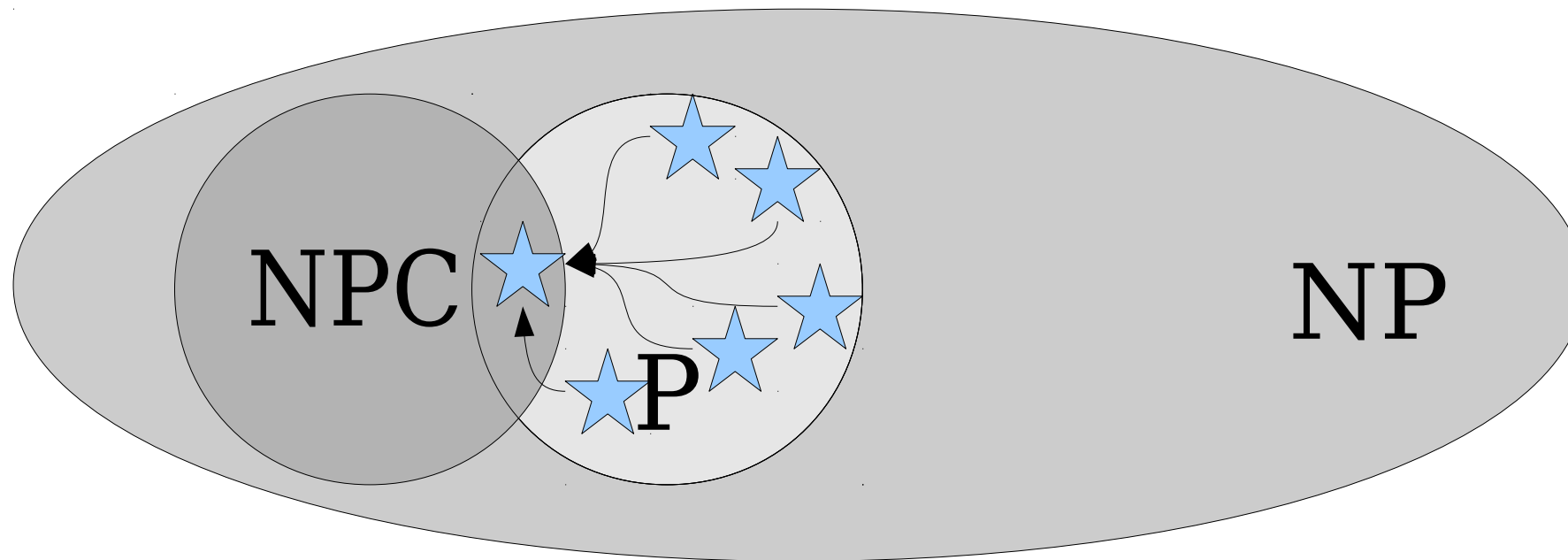
The Tantalizing Truth

Theorem: If *any* NP-complete language is in \mathbf{P} , then $\mathbf{P} = \mathbf{NP}$.



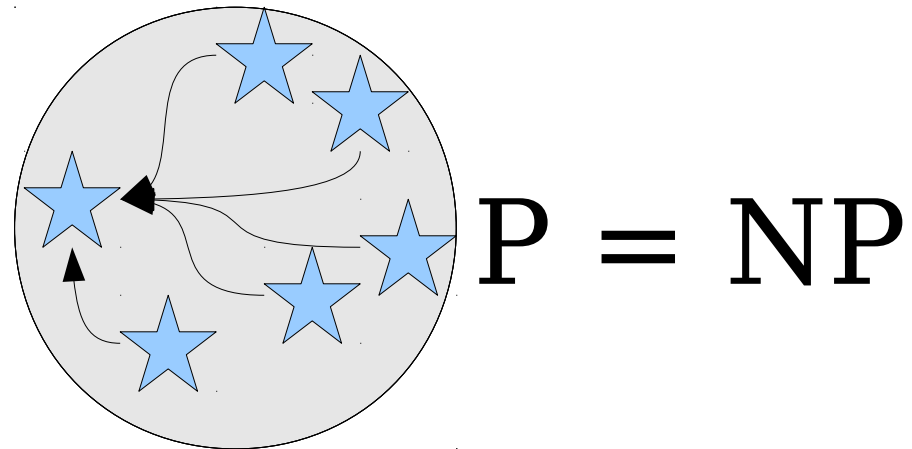
The Tantalizing Truth

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



The Tantalizing Truth

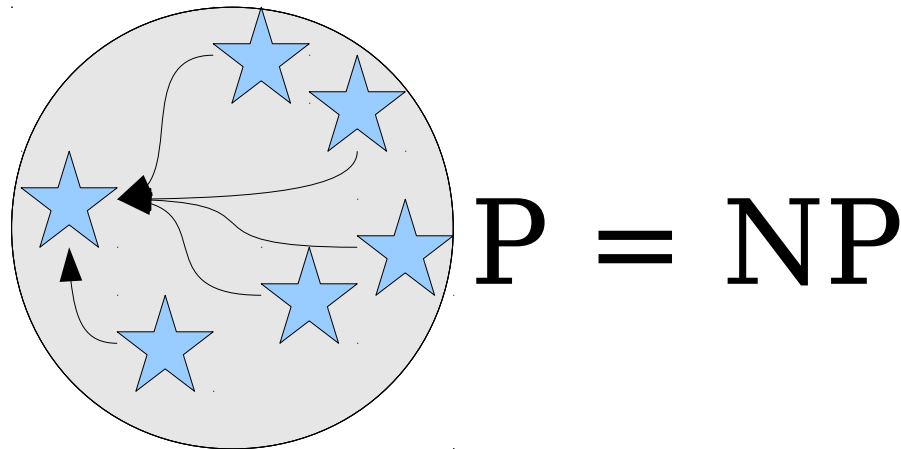
Theorem: If *any* NP-complete language is in **P**, then **P** = **NP**.



The Tantalizing Truth

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

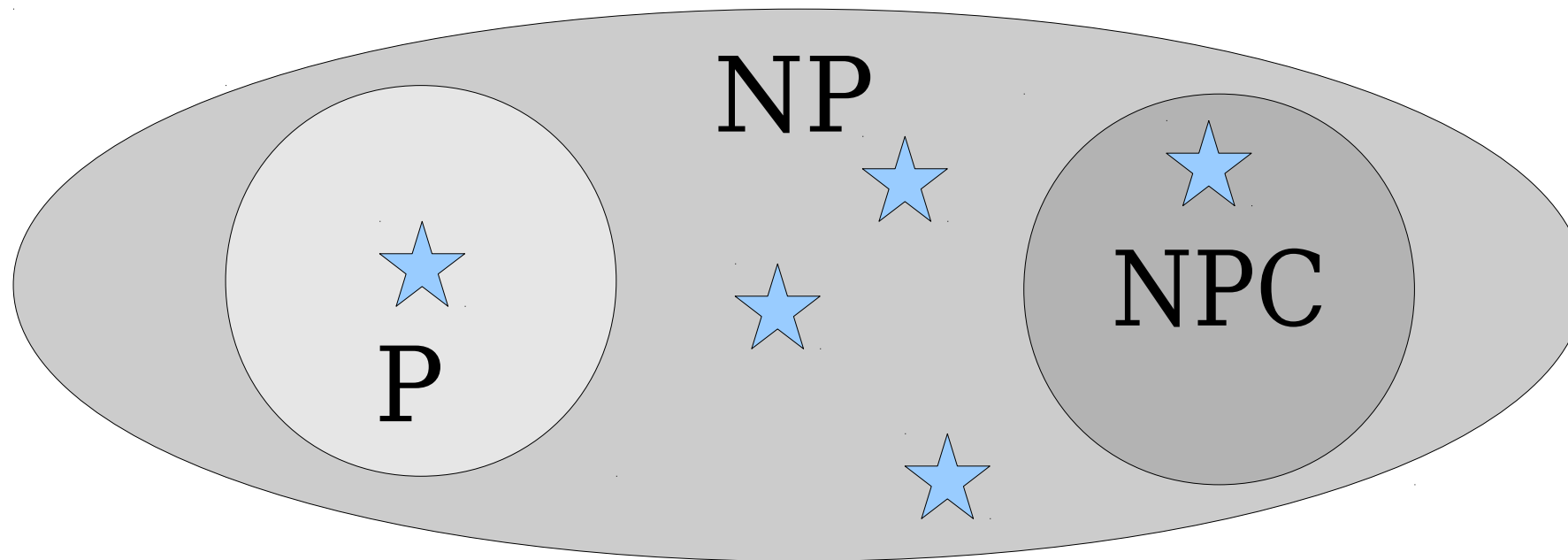
Proof: Suppose that L is **NP**-complete and $L \in \mathbf{P}$. Now consider any arbitrary **NP** problem A . Since L is **NP**-complete, we know that $A \leq_p L$. Since $L \in \mathbf{P}$ and $A \leq_p L$, we see that $A \in \mathbf{P}$. Since our choice of A was arbitrary, this means that $\mathbf{NP}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



The Tantalizing Truth

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

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



An NP-Complete Problem

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:

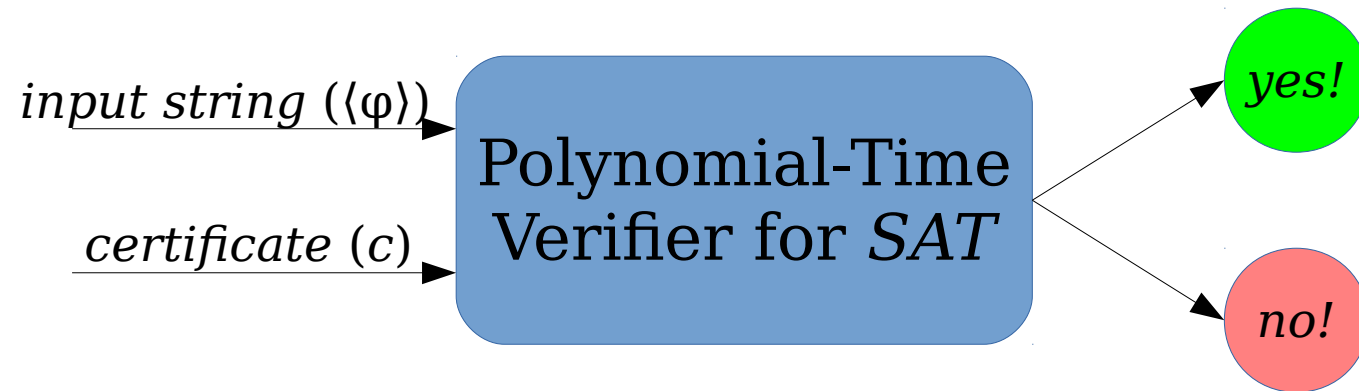
Given a propositional logic formula φ , is φ satisfiable?

- Formally:

$SAT = \{ \langle \varphi \rangle \mid \varphi \text{ is a satisfiable propositional logic formula } \}$

Is SAT in NP?

- To show that SAT is in NP, we have to build a verifier TM for SAT, and we have to be sure that our verifier TM runs in polynomial time (i.e., $O(|\langle\varphi\rangle|)$).



$SAT = \{ \langle\varphi\rangle \mid \varphi \text{ is a satisfiable propositional logic formula} \}$

$SAT = \{ \langle \varphi \rangle \mid \varphi \text{ is a satisfiable propositional logic formula } \}$

The language SAT happens to be in **NP**. Think about how a polynomial-time verifier for SAT might work. Which of the following would work as certificates for such a verifier, given that the input is a propositional formula $\langle \varphi \rangle$?

- A. The truth table of φ .
- B. One possible variable assignment to φ .
- C. A list of all possible variable assignments for φ .
- D. None of the above, or two or more of the above.

Answer at **PollEv.com/cs103** or
text **CS103** to **22333** once to join, then **A**, **B**, **C**, or **D**.

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

Proof Idea: To see that **SAT** \in **NP**, show how to make a polynomial-time verifier for it. Key idea: have the certificate be a satisfying assignment.

To show that **SAT** is **NP**-hard, given a polynomial-time verifier V for an arbitrary **NP** language L , for any string w you can construct a polynomially-sized formula $\varphi(w)$ that says “there is a certificate c where V accepts $\langle w, c \rangle$.” This formula is satisfiable if and only if $w \in L$, so deciding whether the formula is satisfiable decides whether w is in L .

Proof: Take CS154!

Why All This Matters

- Resolving $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ is equivalent to just figuring out how hard SAT is.
 - If $\text{SAT} \in \mathbf{P}$, then $\mathbf{P} = \mathbf{NP}$.
 - If $\text{SAT} \notin \mathbf{P}$, then $\mathbf{P} \neq \mathbf{NP}$.

Other Sample NP-Hard Problems

- **Computational biology:** Given a set of genomes, what is the most probable evolutionary tree that would give rise to those genomes? (*Maximum parsimony problem*)
- **Game theory:** Given an arbitrary perfect-information, finite, two-player game, who wins? (*Generalized geography problem*)
- **Operations research:** Given a set of jobs and workers who can perform those tasks in parallel, can you complete all the jobs within some time bound? (*Job scheduling problem*)
- **Machine learning:** Given a set of data, find the simplest way of modeling the statistical patterns in that data (*Bayesian network inference problem*)
- **Medicine:** Given a group of people who need kidneys and a group of kidney donors, find the maximum number of people who can end up with kidneys (*Cycle cover problem*)
- **Systems:** Given a set of processes and a number of processors, find the optimal way to assign those tasks so that they complete as soon as possible (*Processor scheduling problem*)