Complexity Theory

Part Two

Recap from Last Time

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:

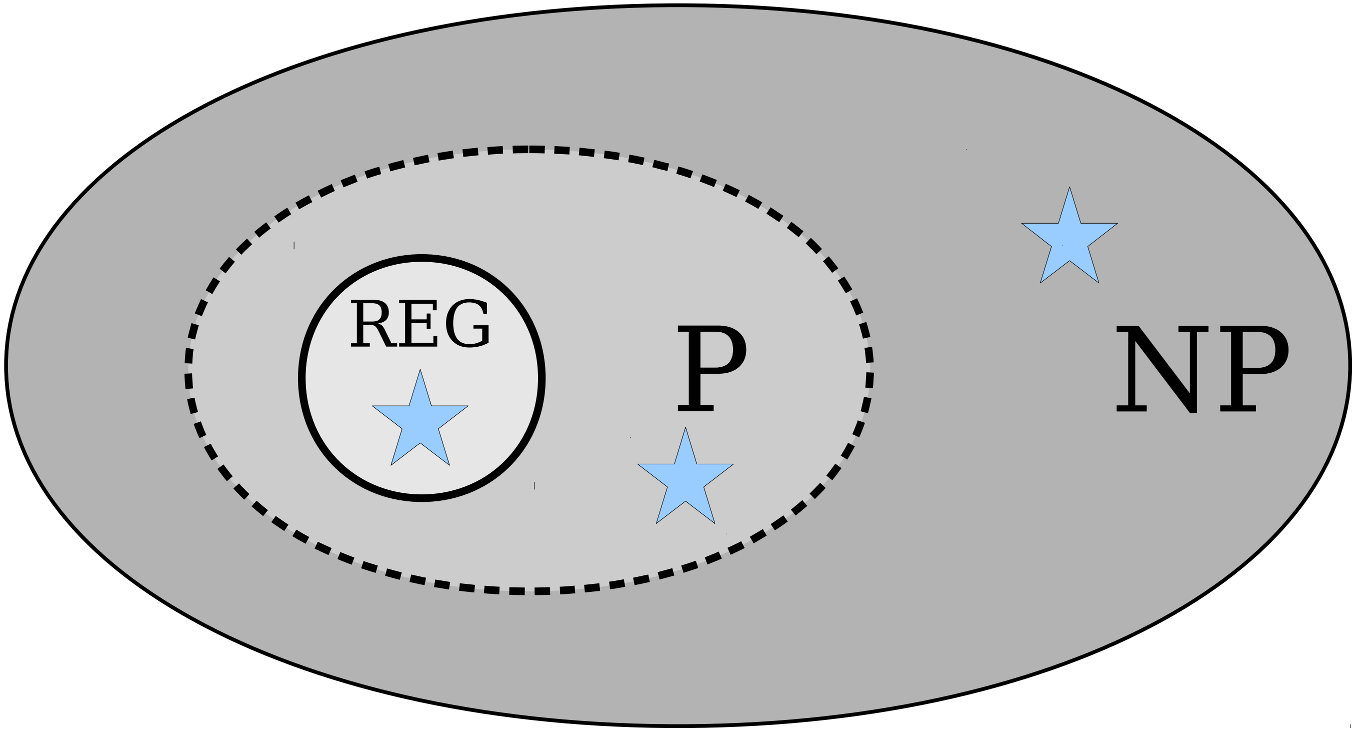
  NP = { L | There is a polynomial-time verifier for L }

- 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).

So how *are* we going to reason about **P** and **NP**?

New Stuff!

A Challenge



Problems in **NP** vary widely in their difficulty, even if **P** = **NP**.

How can we rank the relative difficulties of problems?

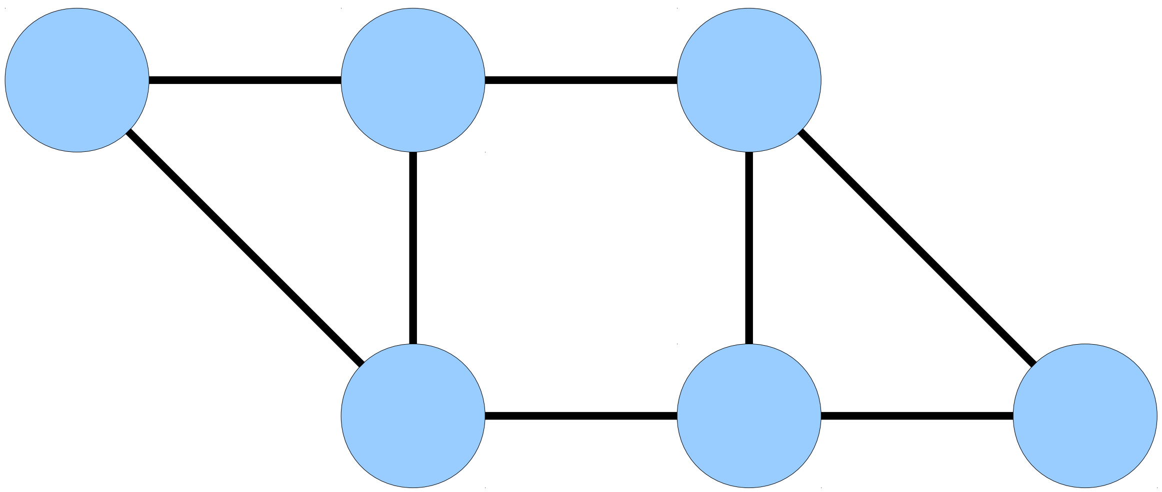
Reducibility

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

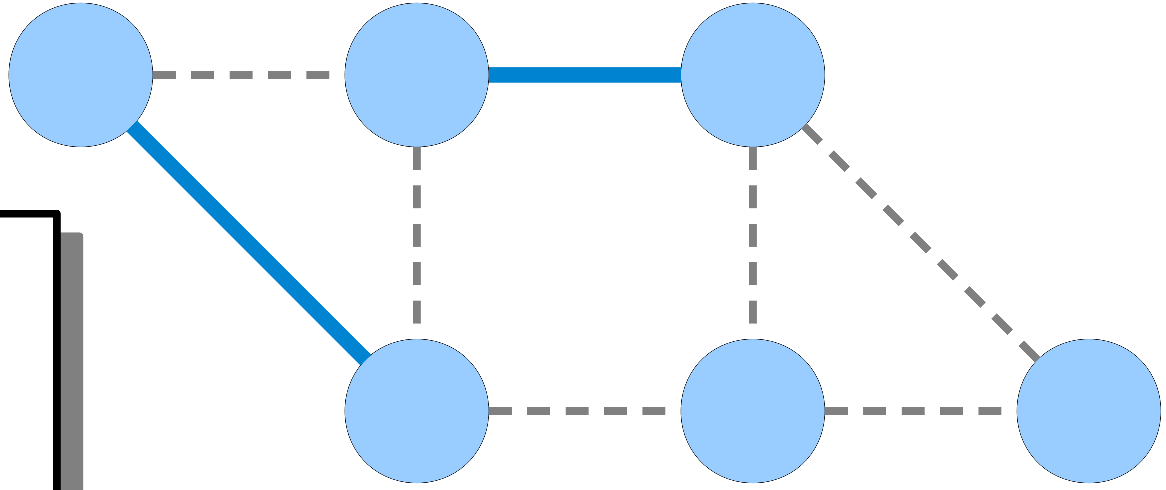
- Given an undirected graph G, a ***matching*** in G is a set of edges such that no two edges share an endpoint.
- A ***maximum matching*** is a matching with the largest number of edges.



Maximum Matching

- 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.

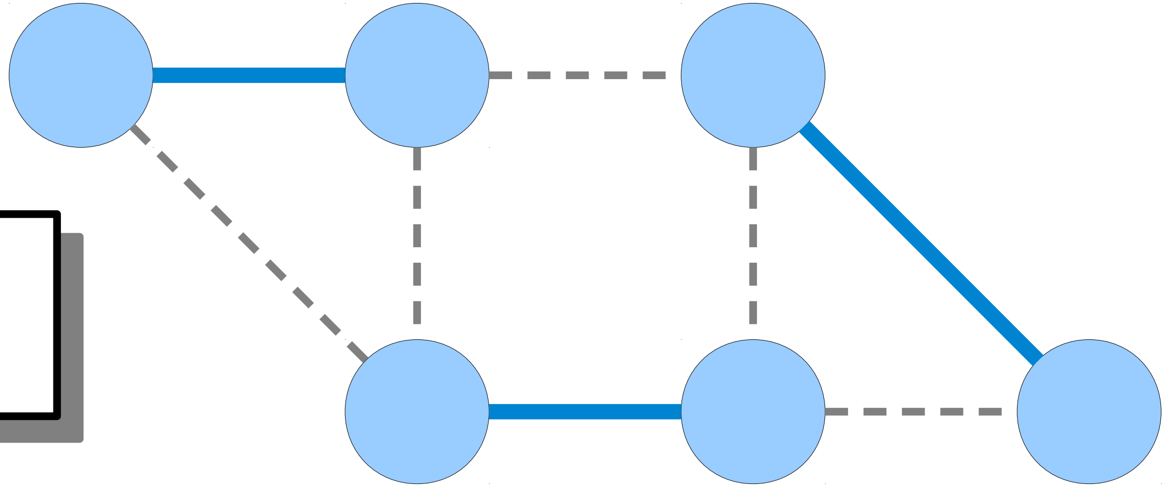
A matching, but not a maximum matching.



Maximum Matching

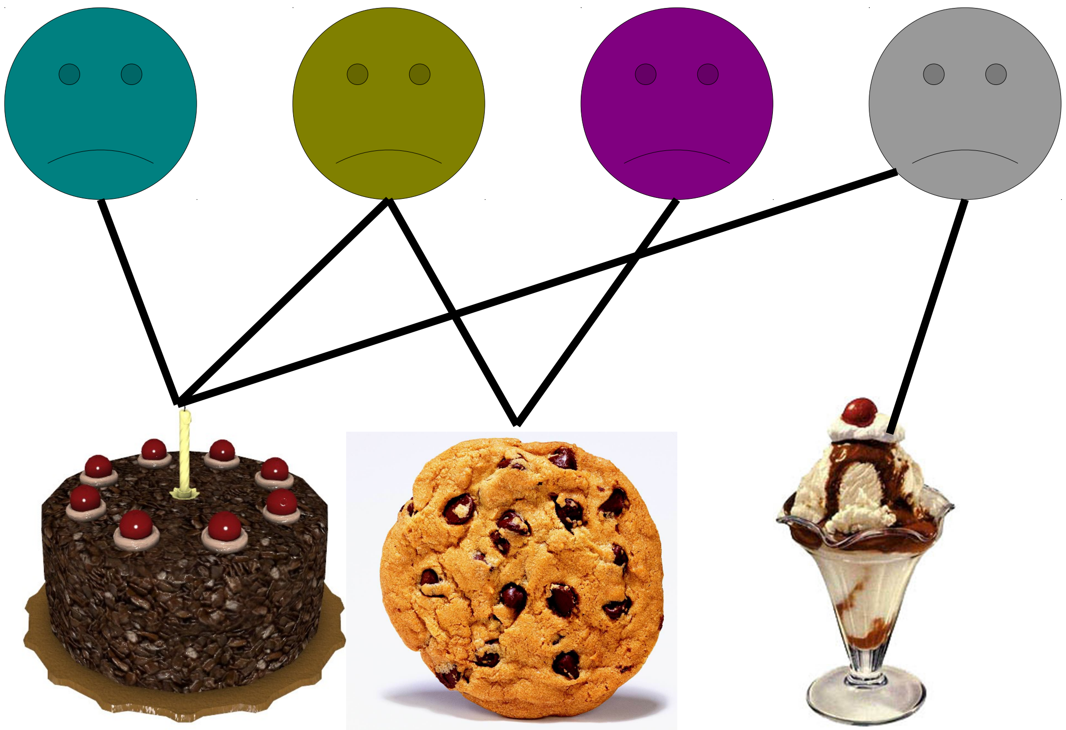
- Given an undirected graph *G*, a ***matching*** in *G* is a set of edges such that no two edges share an endpoint.
- A ***maximum matching*** is a matching with the largest number of edges.

A maximum matching.



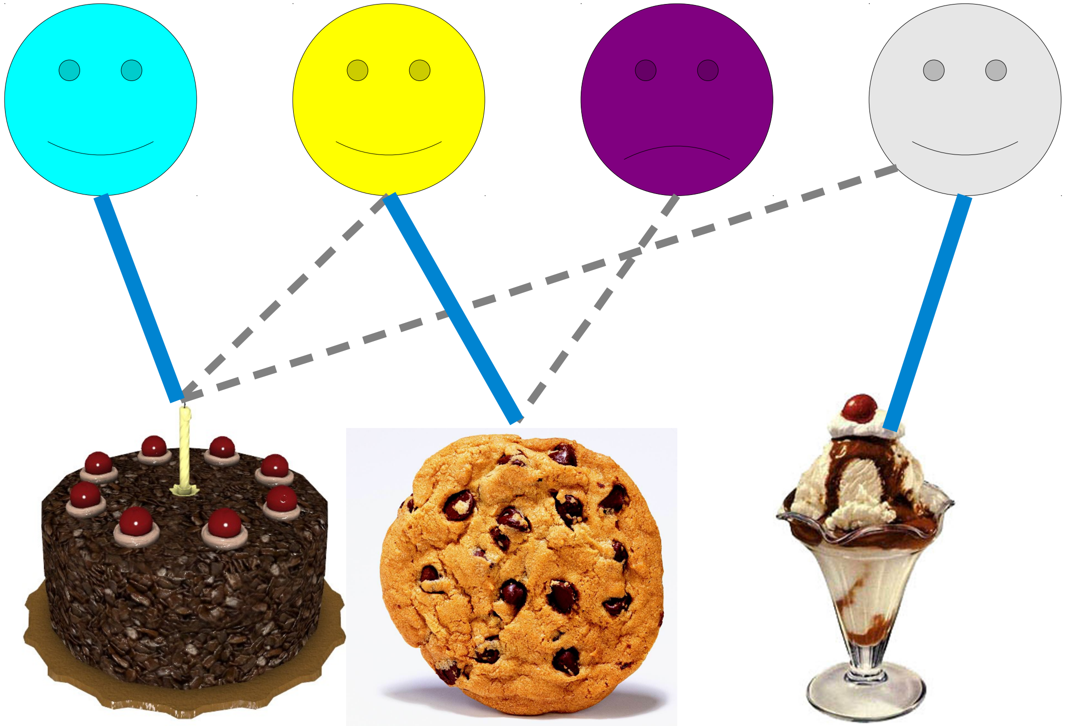
Maximum Matching

- 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

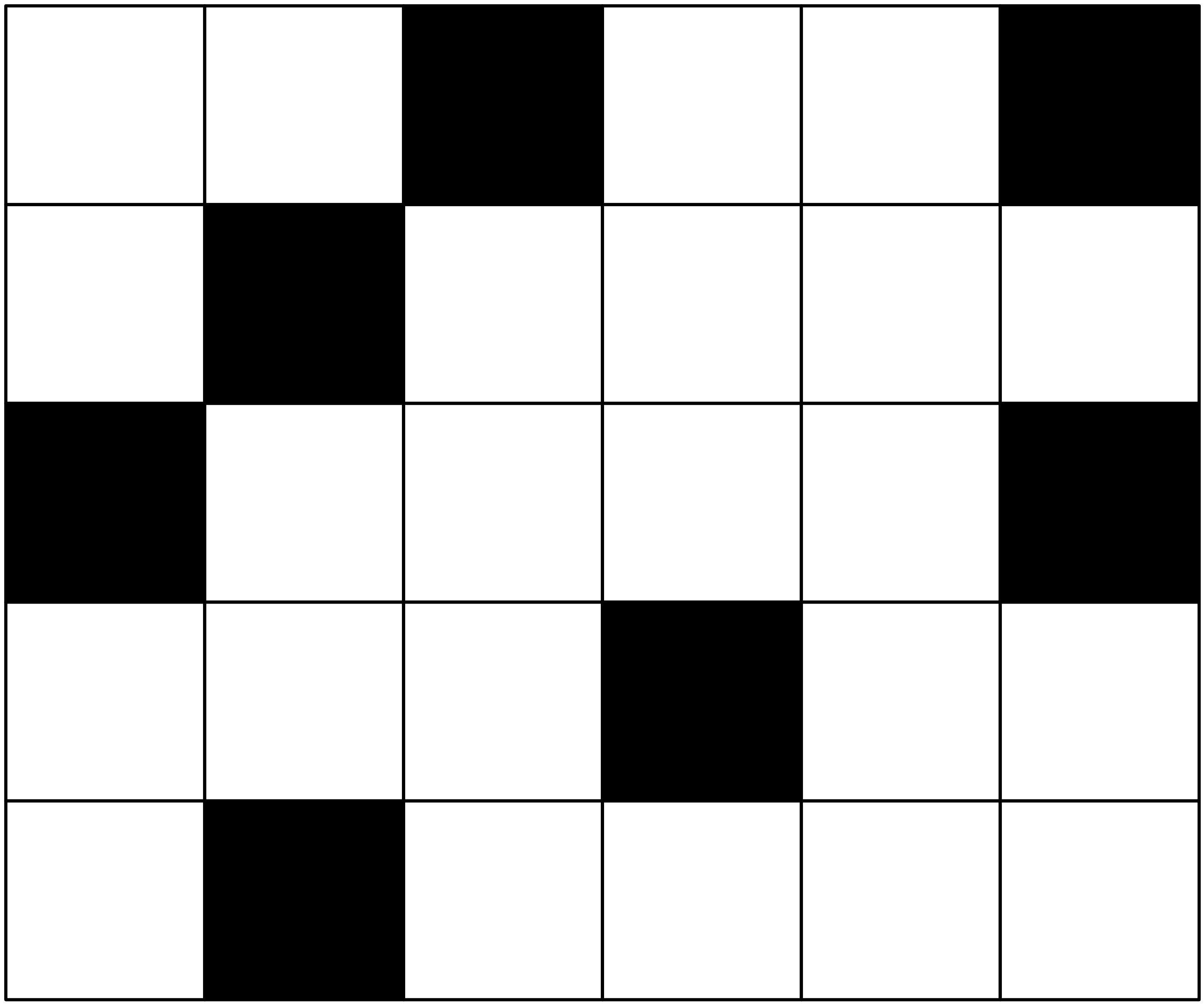
- Given an undirected graph *G*, a ***matching*** in *G* is a set of edges such that no two edges share an endpoint.
- A ***maximum matching*** is a matching with the largest number of edges.



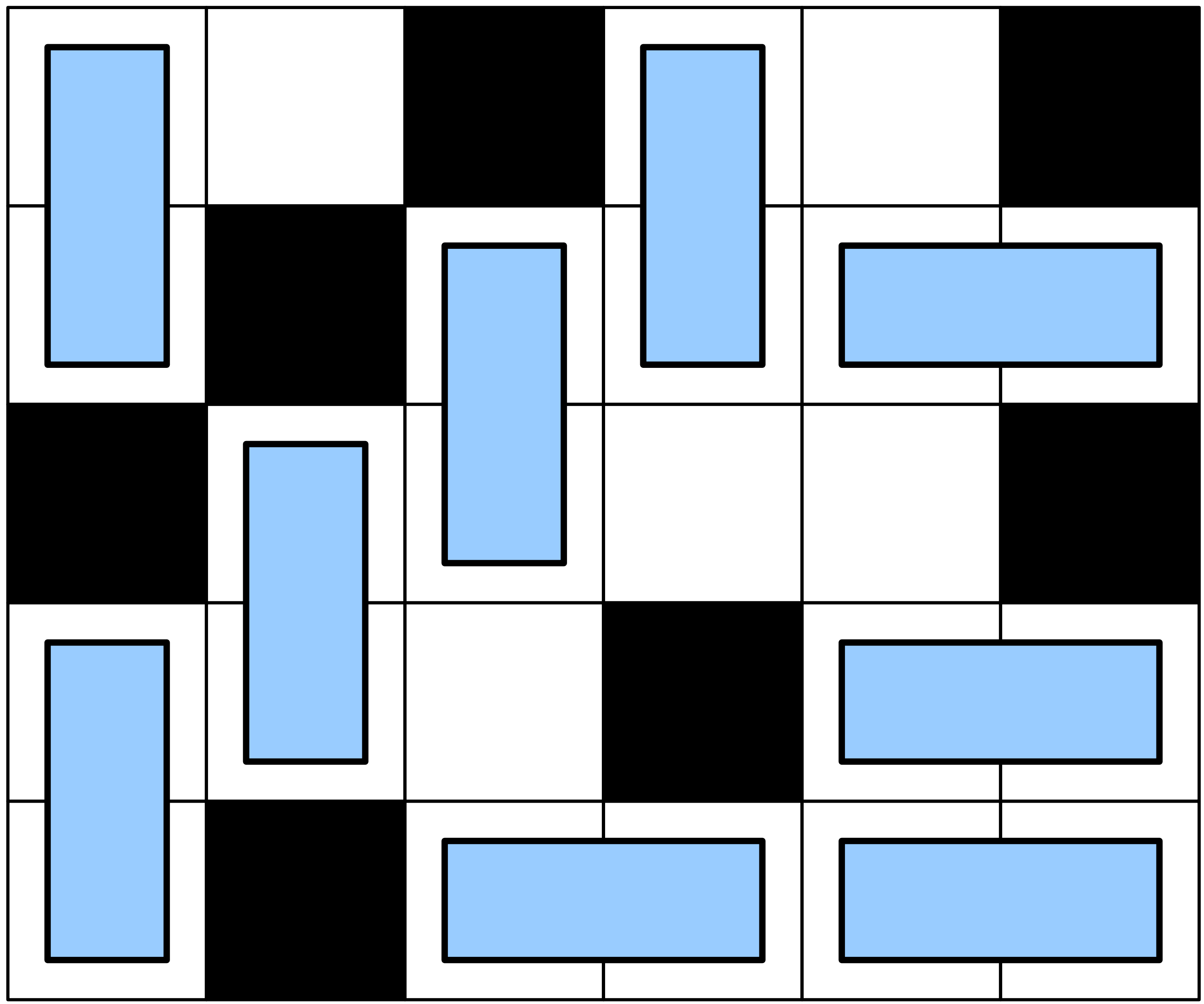
Maximum Matching

- 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?

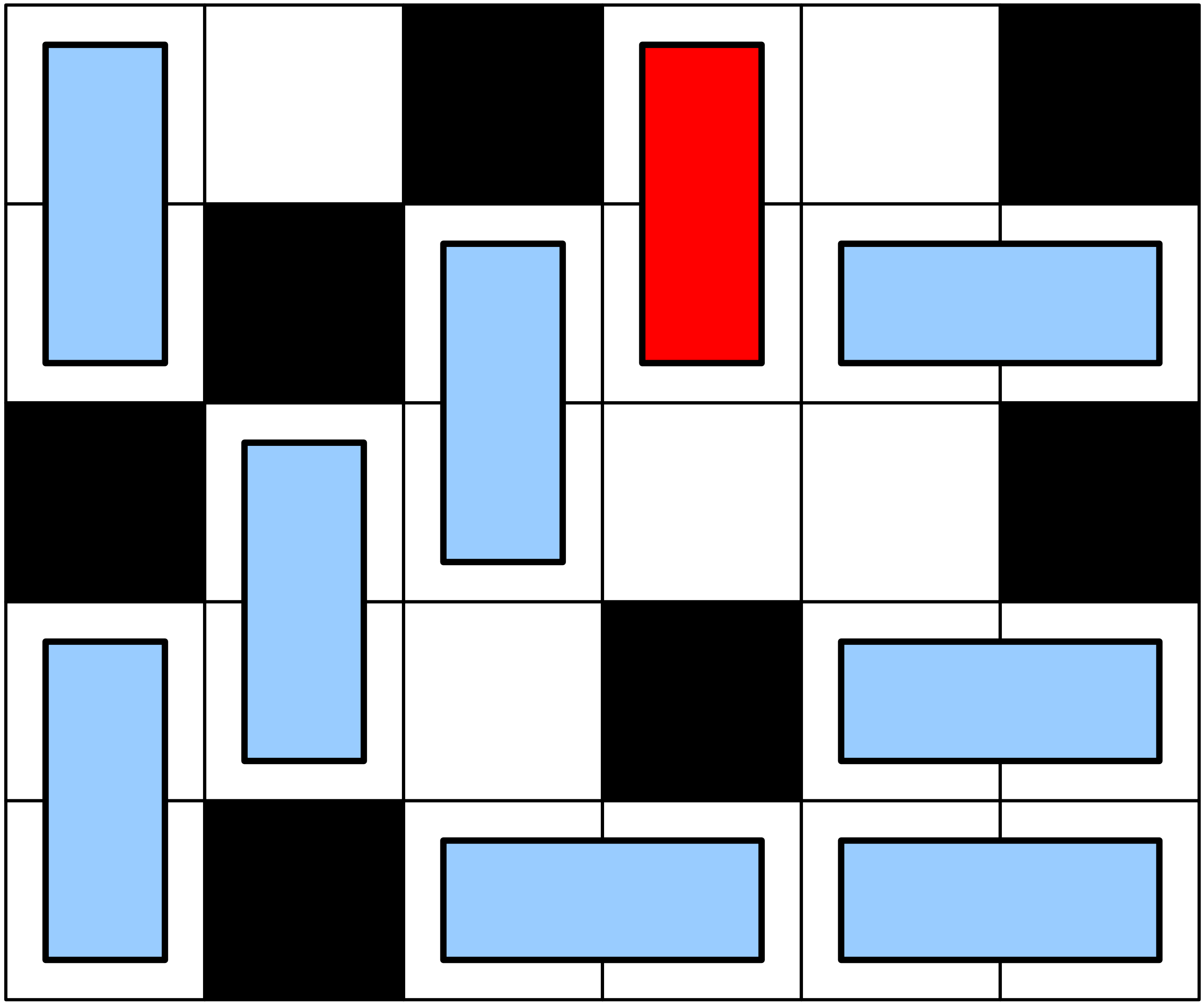
Domino Tiling



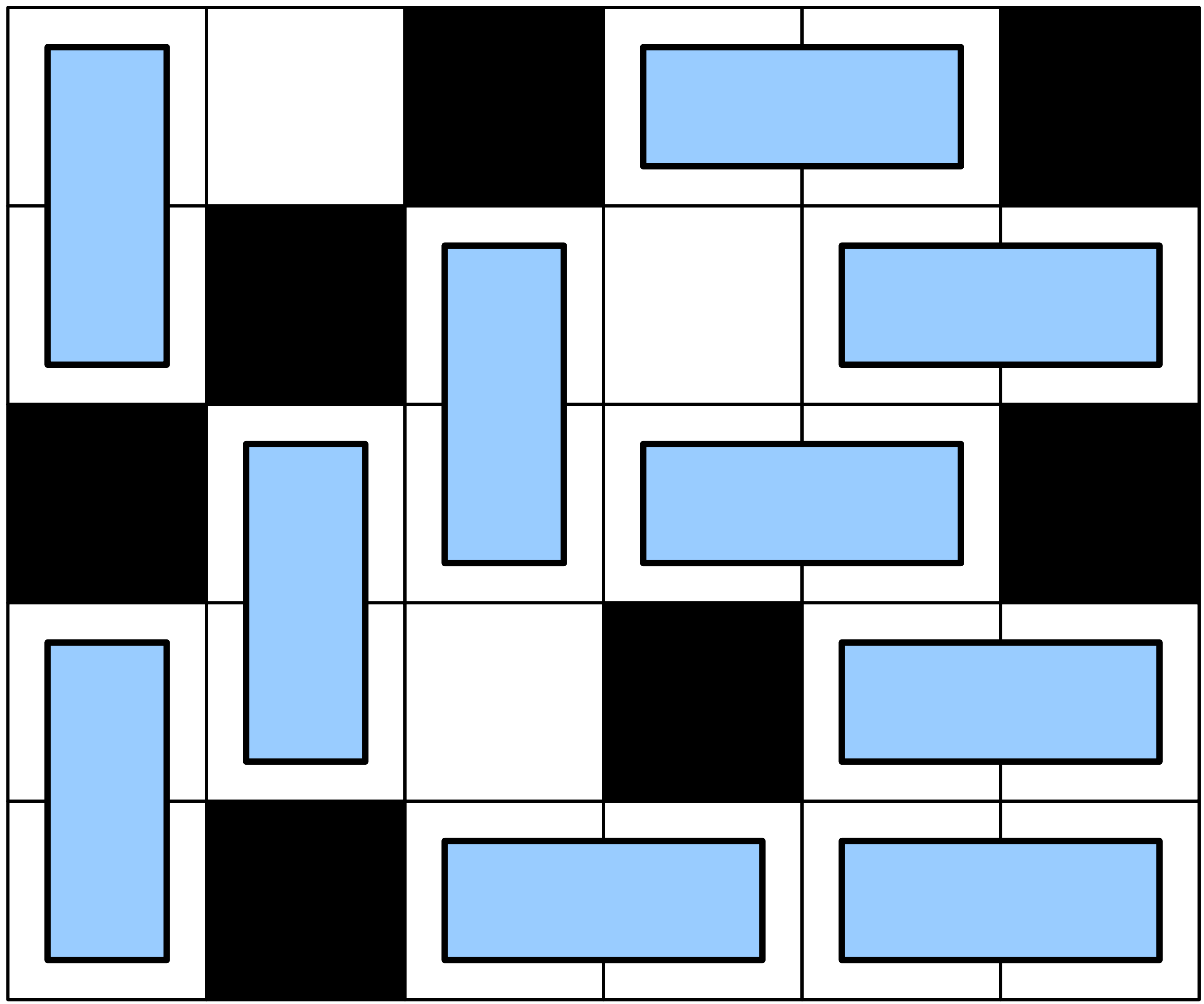
Domino Tiling



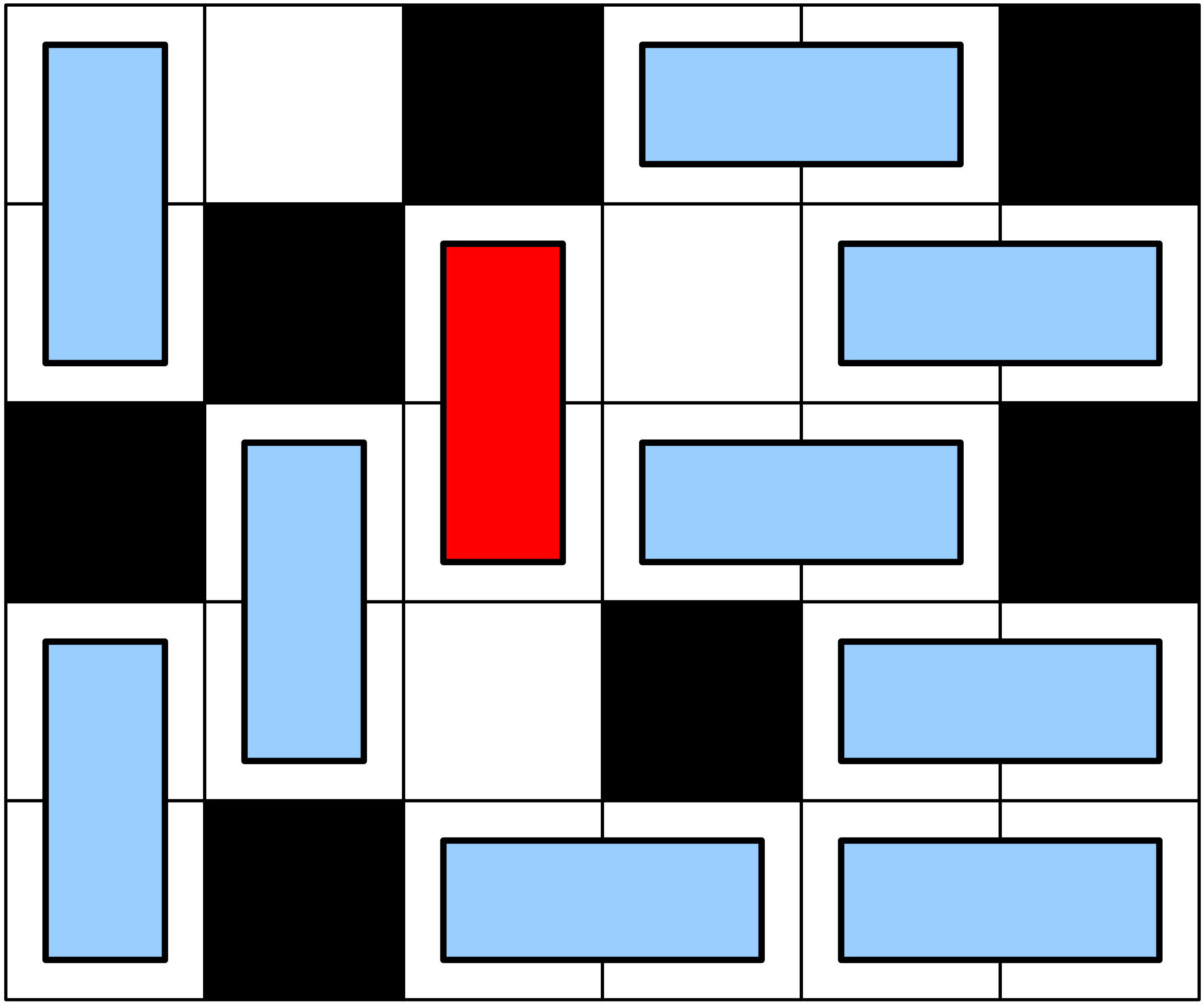
Domino Tiling



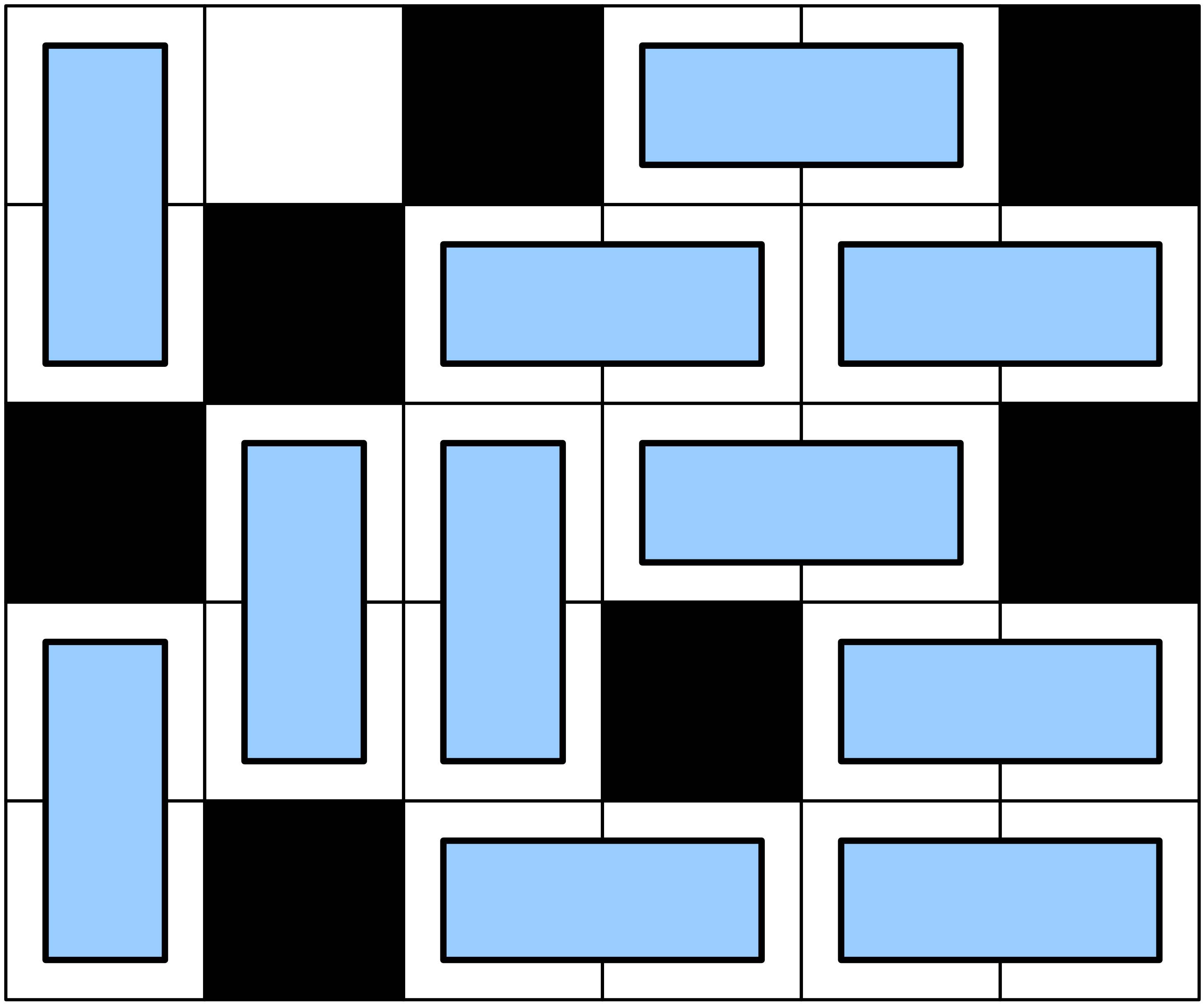
Domino Tiling



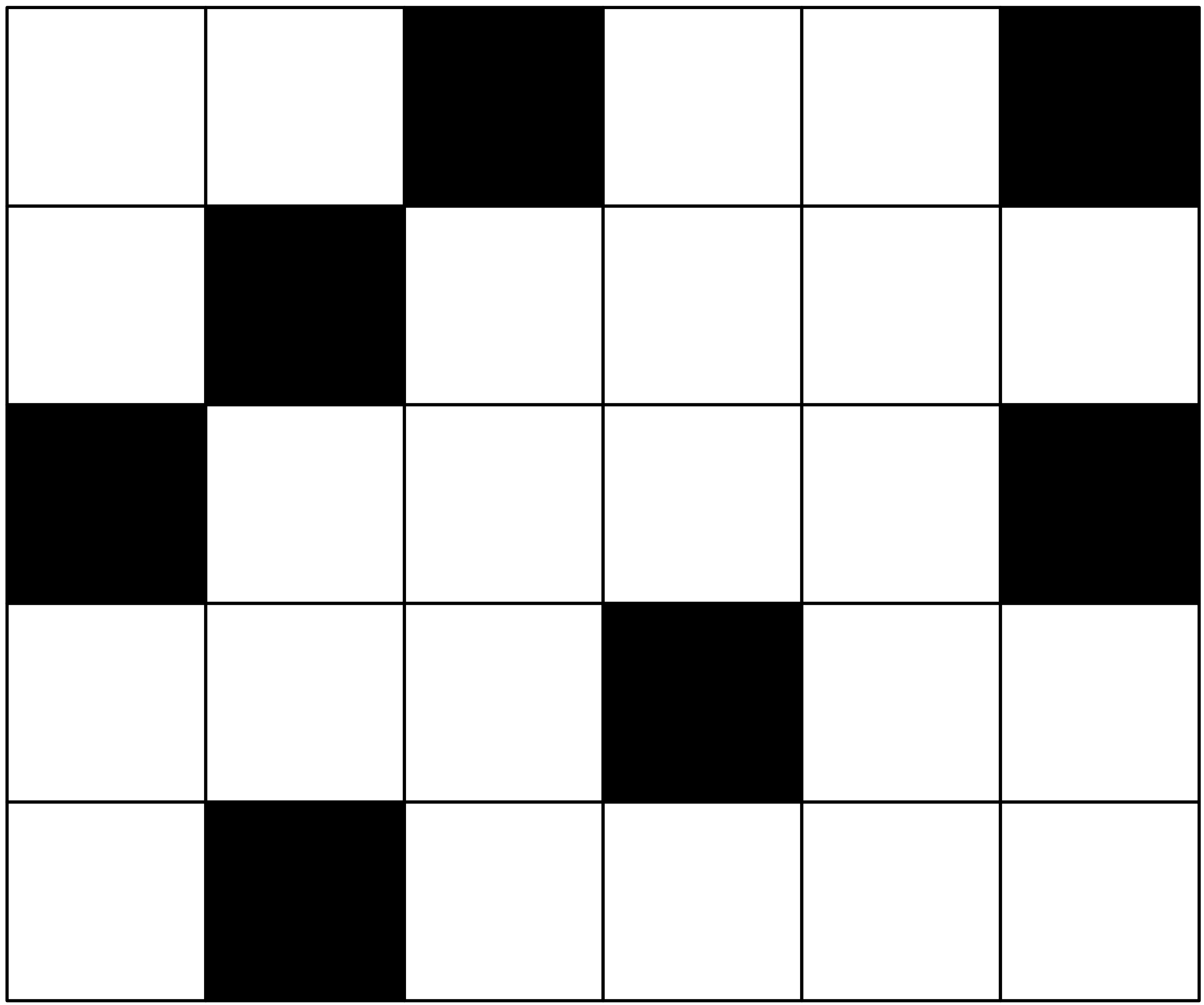
Domino Tiling



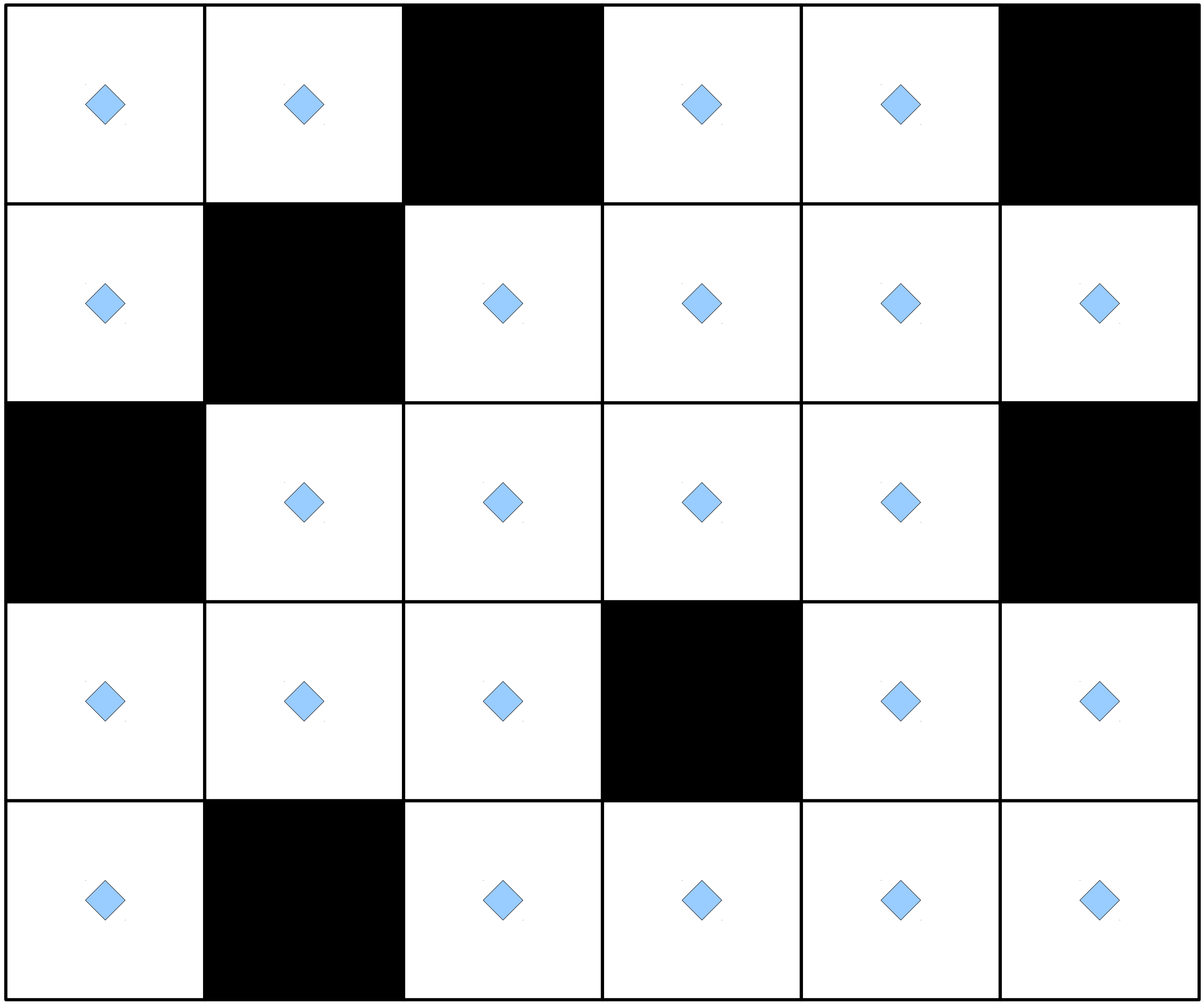
Domino Tiling



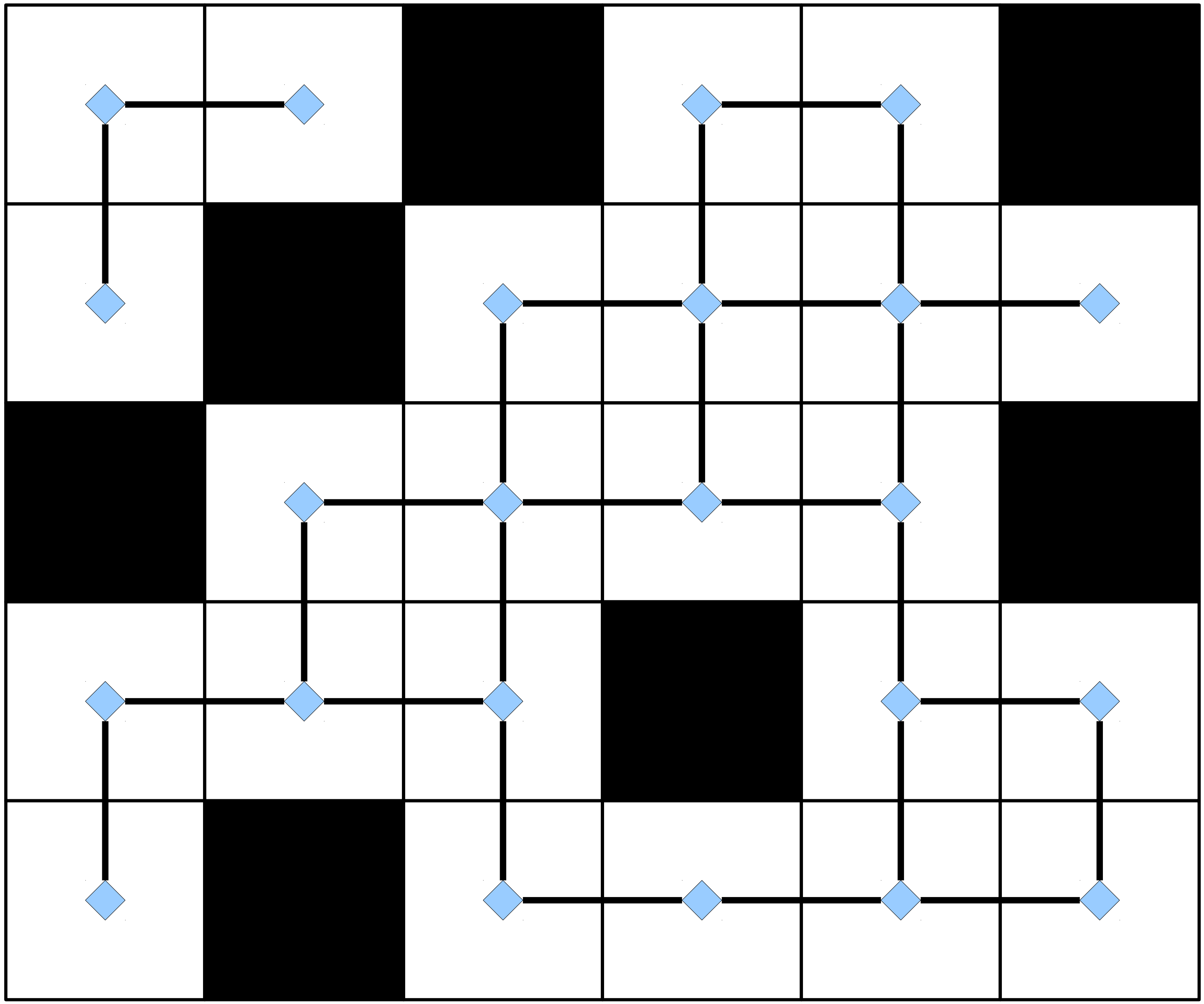
Solving Domino Tiling



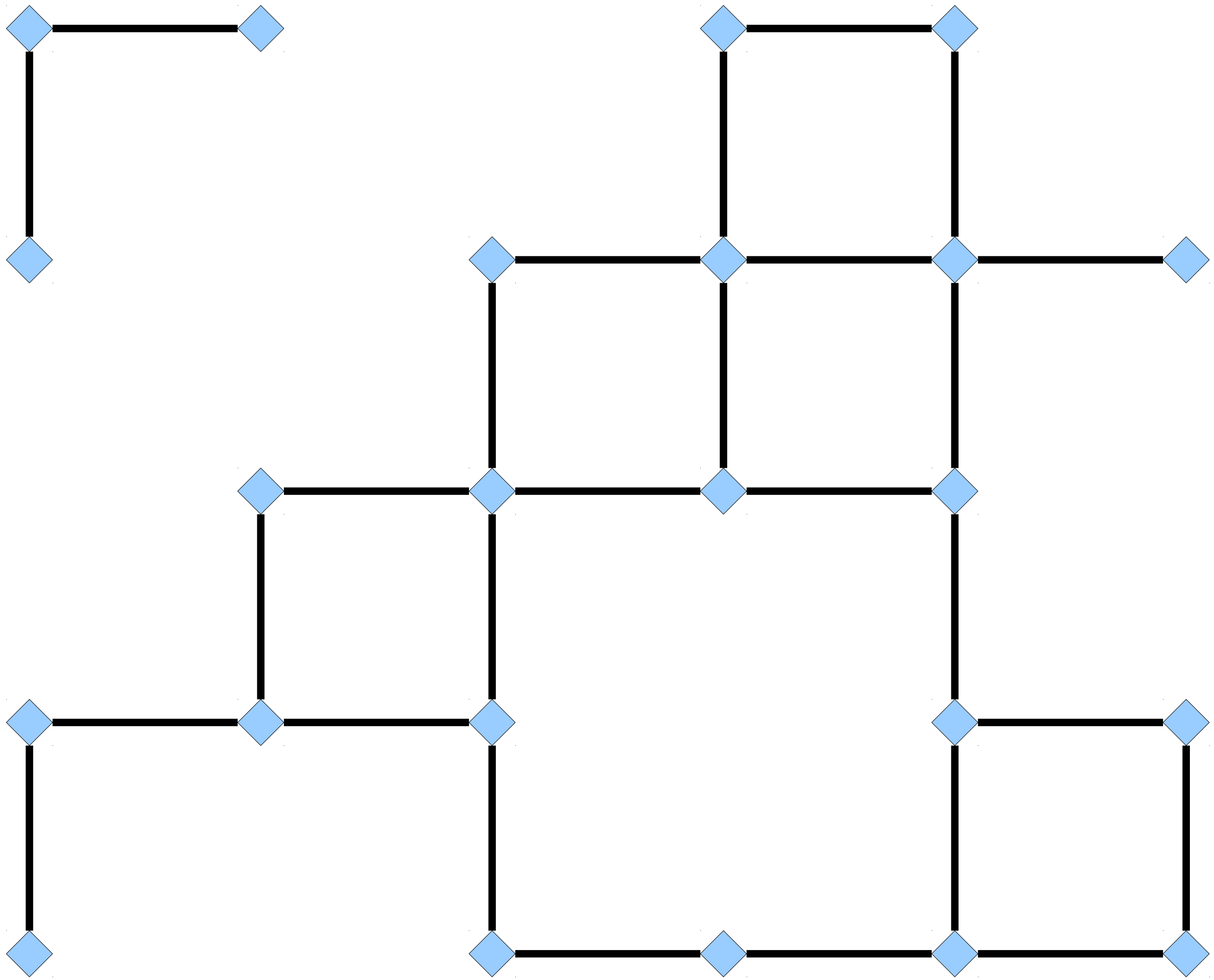
Solving Domino Tiling



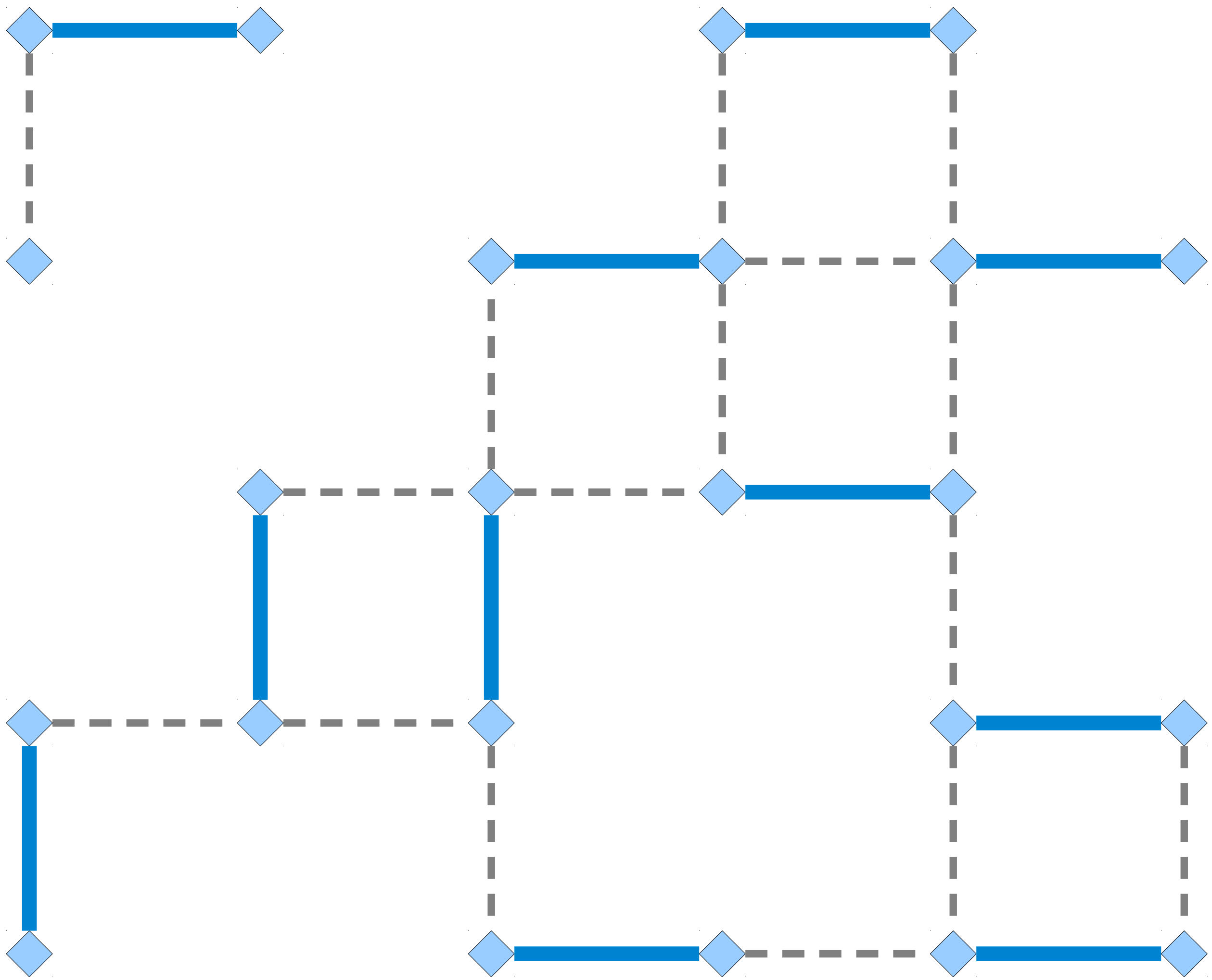
Solving Domino Tiling



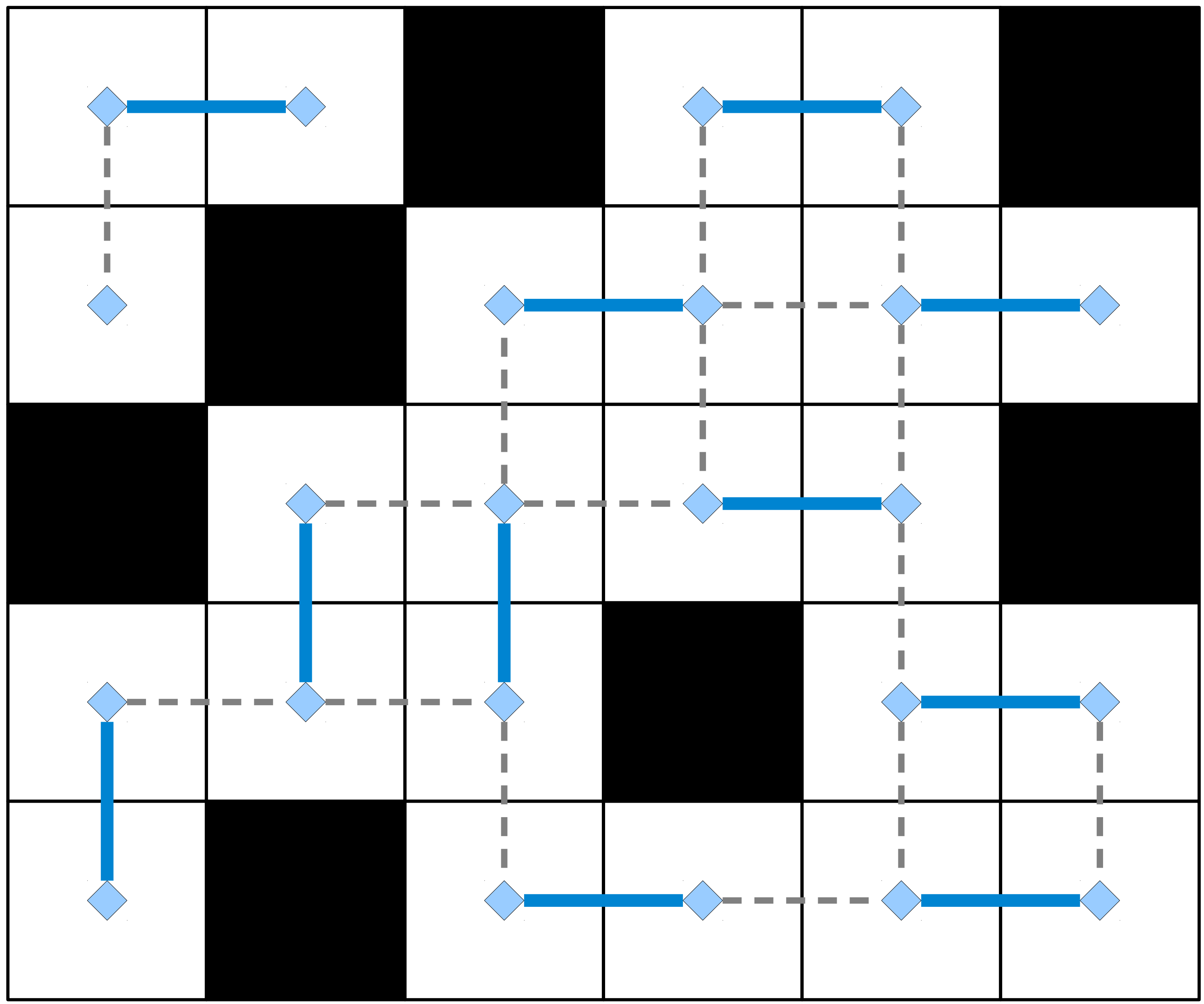
Solving Domino Tiling



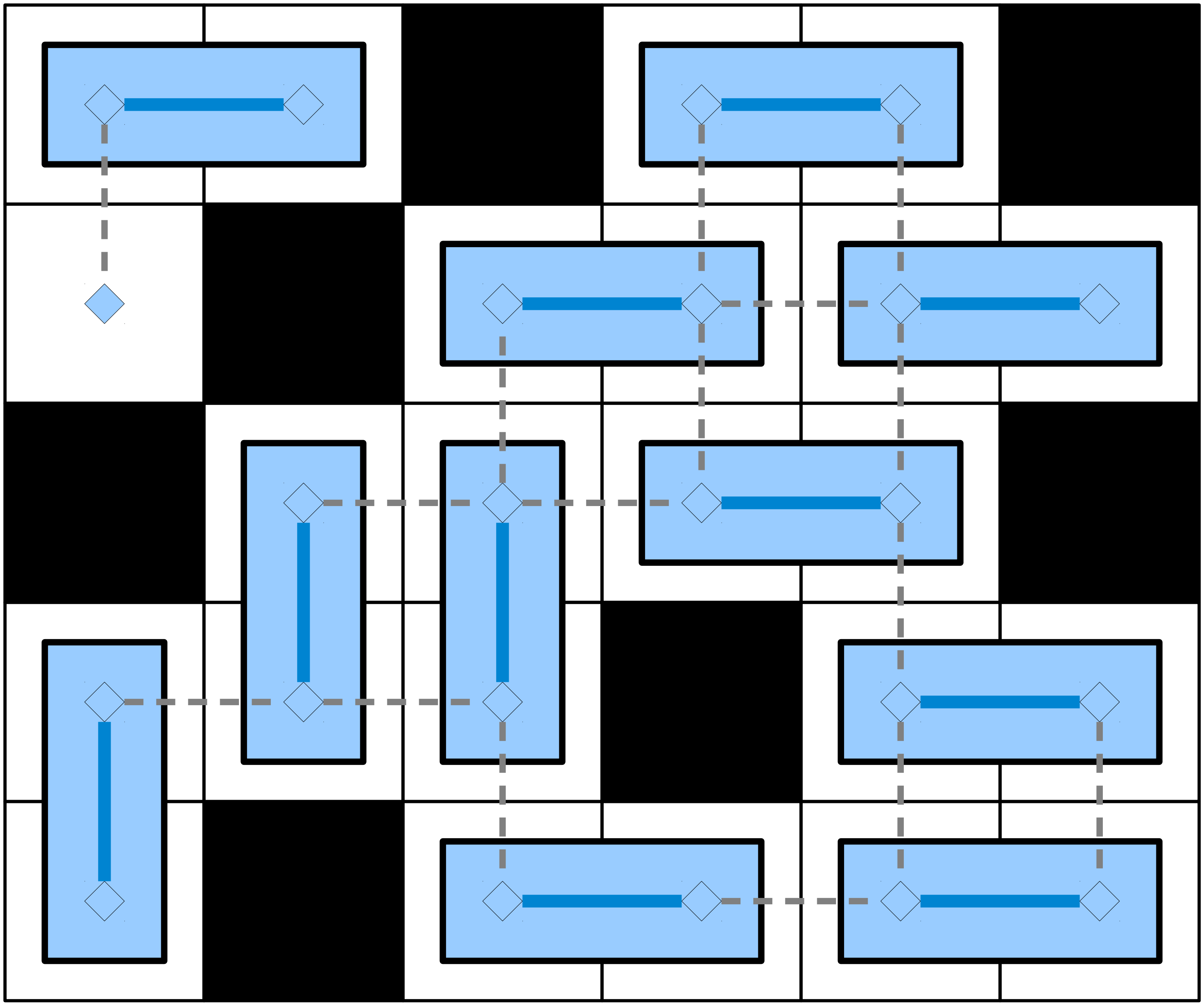
Solving Domino Tiling



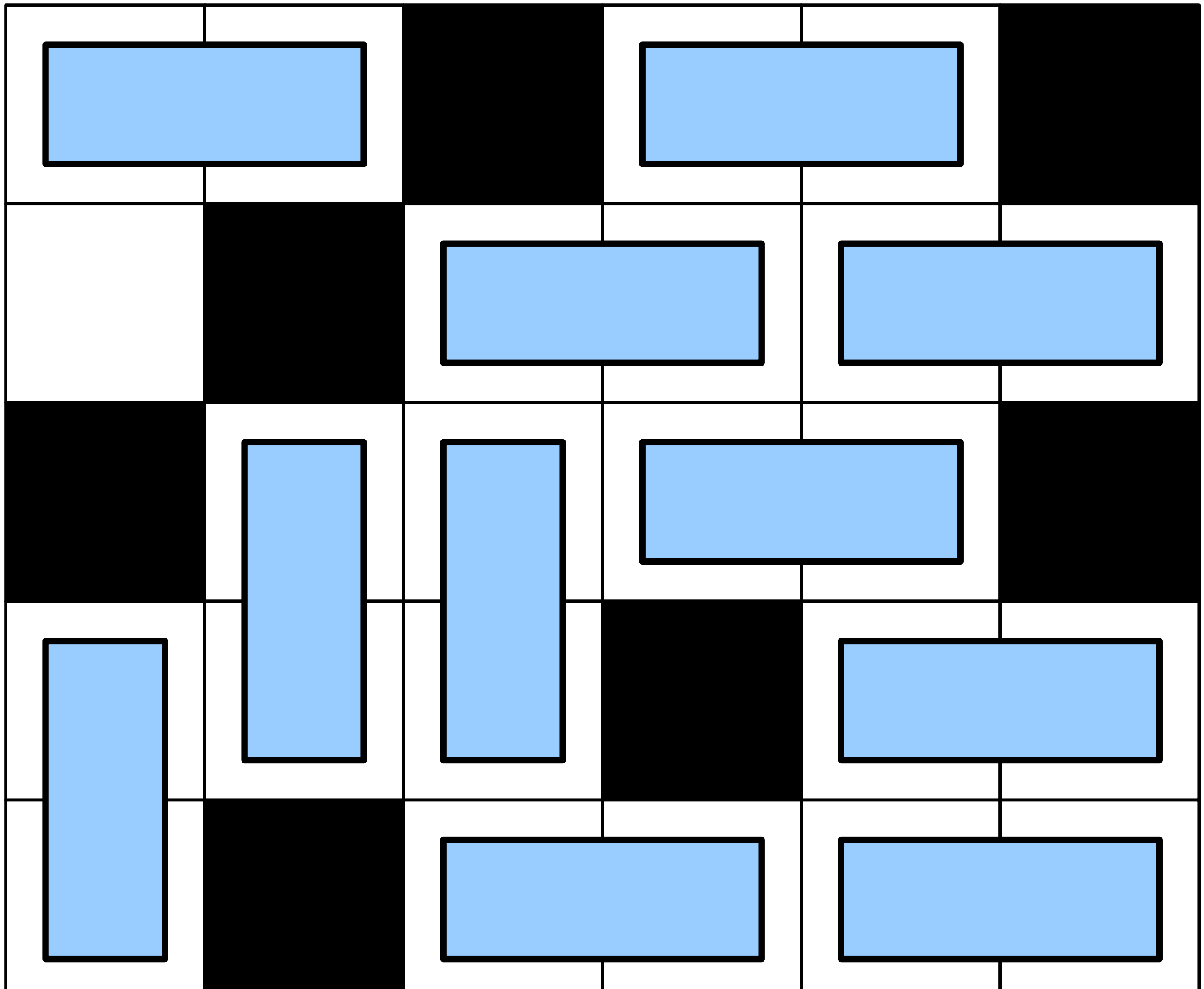
Solving Domino Tiling



Solving Domino Tiling



Solving Domino Tiling



In Pseudocode

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

Based on this connection between maximum matching and domino tiling, which of the following statements would be more proper to conclude?

A. Finding a maximum matching isn't any more difficult (in BigO/P-NP terms) than tiling a grid with dominoes.

B. Tiling a grid with dominoes isn't any more difficult (in BigO/P-NP terms) than finding a maximum matching.

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

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.

Another Example

Reachability

- Consider the following problem:

  Given an directed graph $\boldsymbol{G}$ ***and nodes*** $\boldsymbol{s}$ ***and*** $\boldsymbol{t}$ ***in*** $\boldsymbol{G}$***, is there a path from*** $\boldsymbol{s}$ ***to*** $\boldsymbol{t}$***?***

- 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

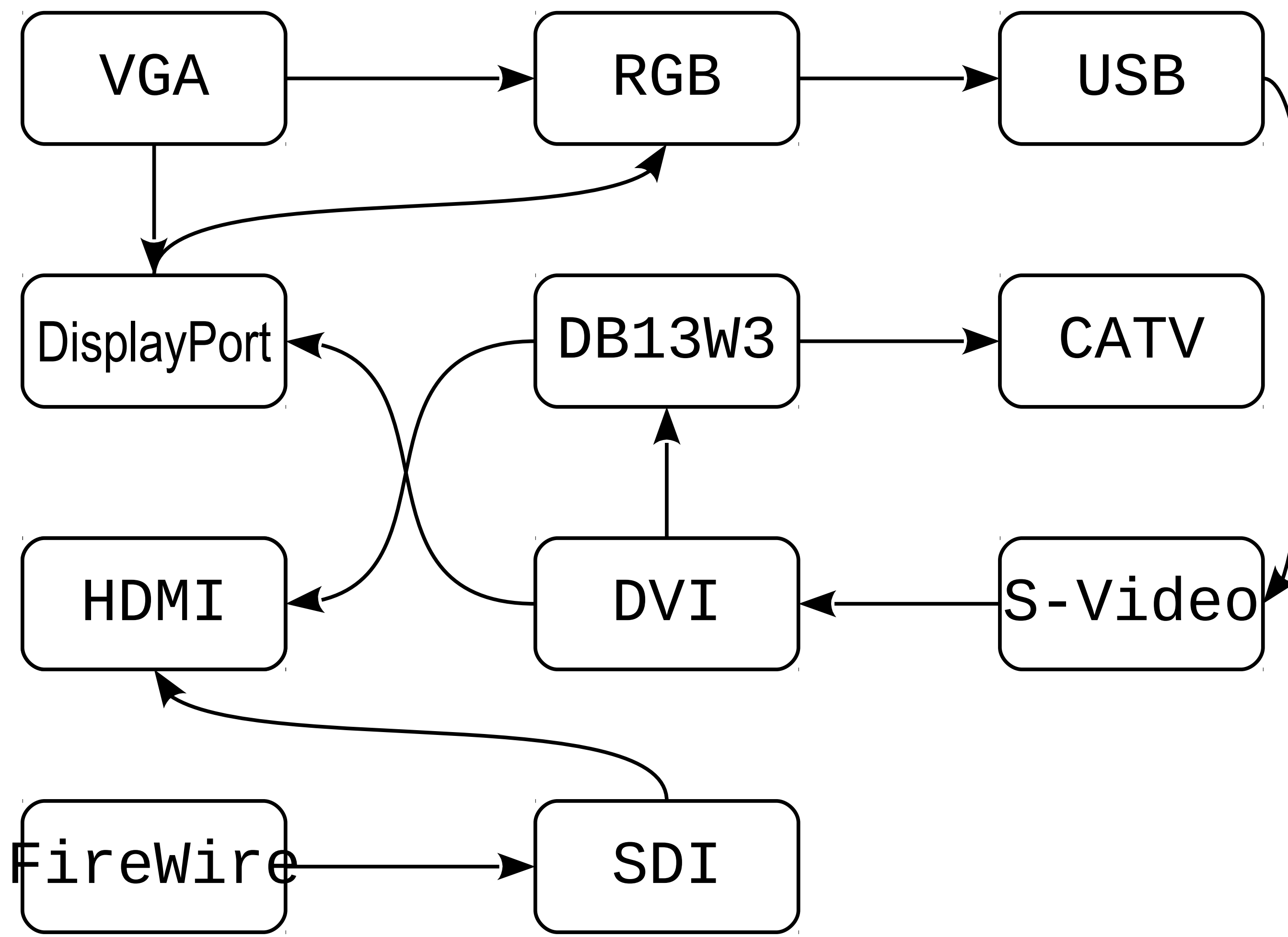
Connectors

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

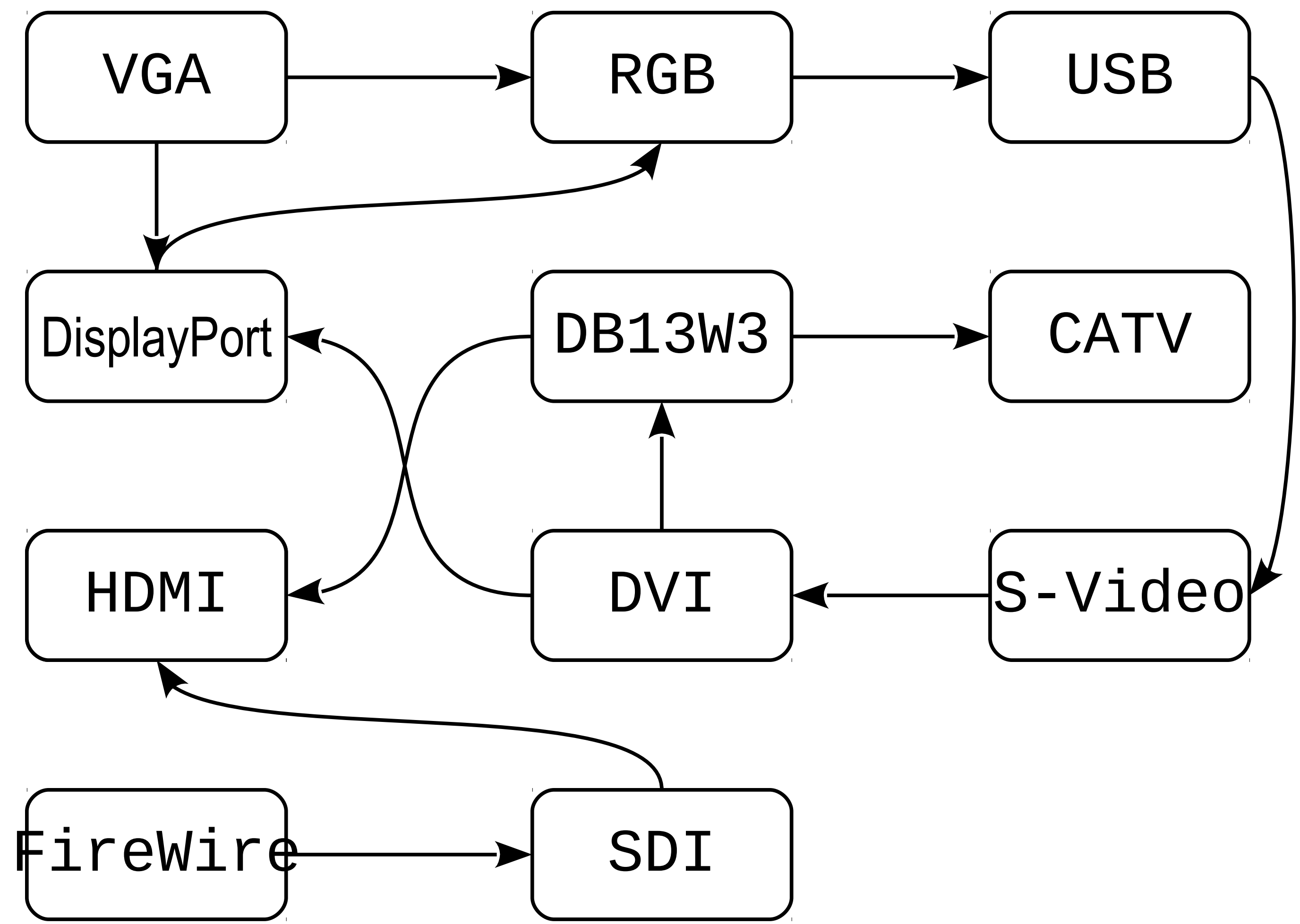
Converter Conundrums

Connectors

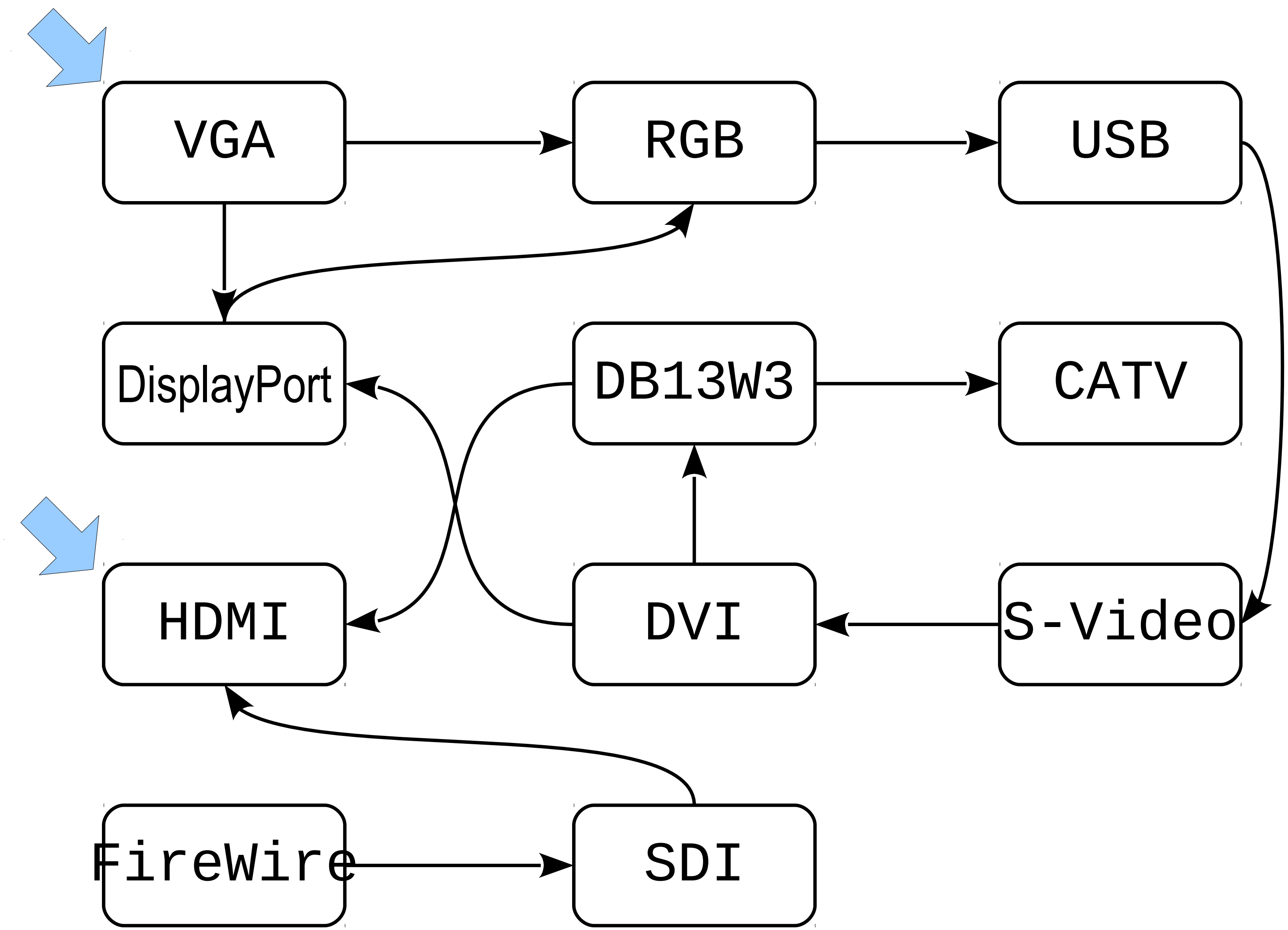
RGB to USB
VGA to DisplayPort
DB13W3 to CATV
DisplayPort to RGB
DB13W3 to HDMI
DVI to DB13W3
S-Video to DVI
FireWire to SDI
VGA to RGB
DVI to DisplayPort
USB to S-Video
SDI to HDMI



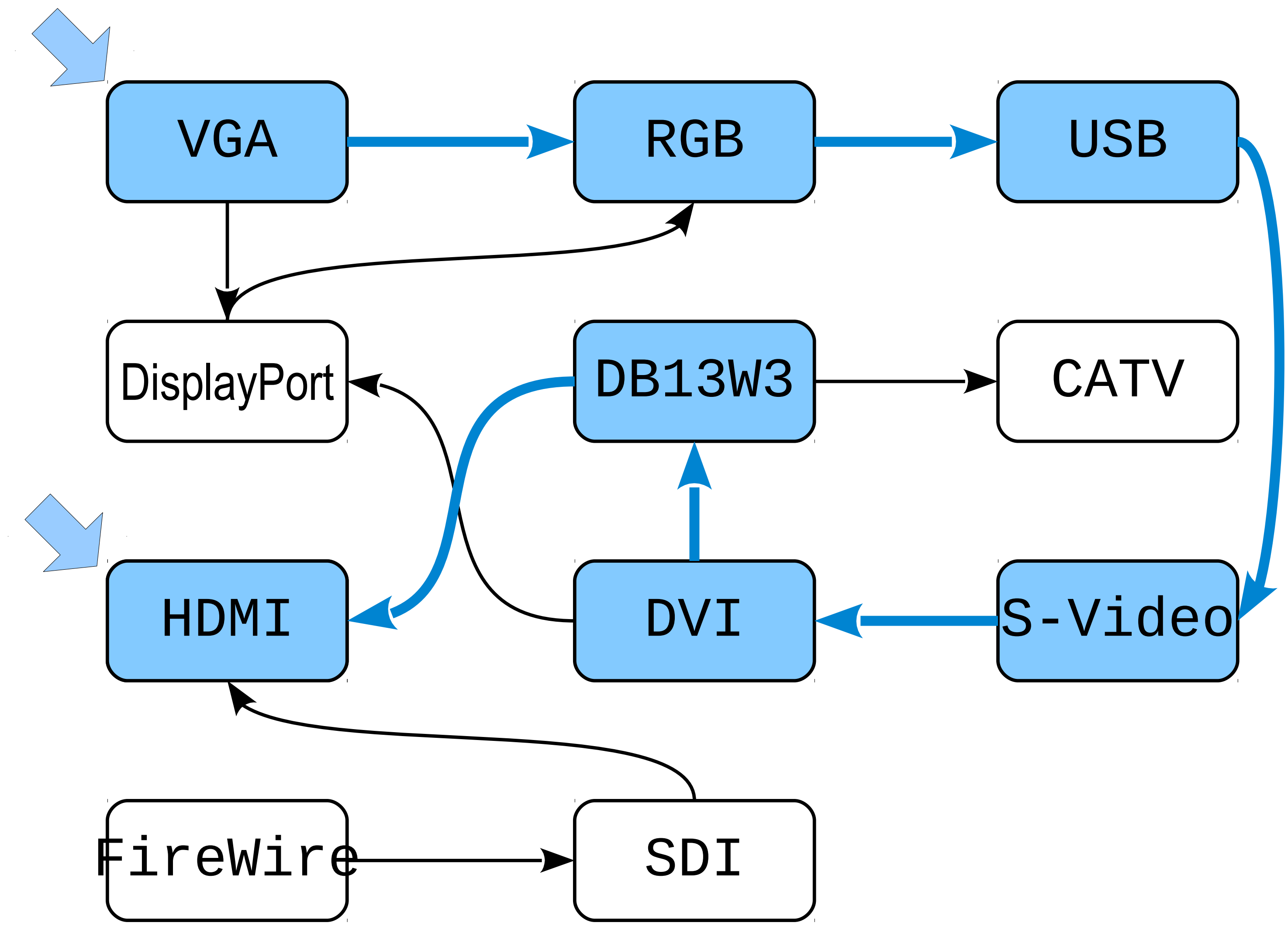
Converter Conundrums



Converter Conundrums



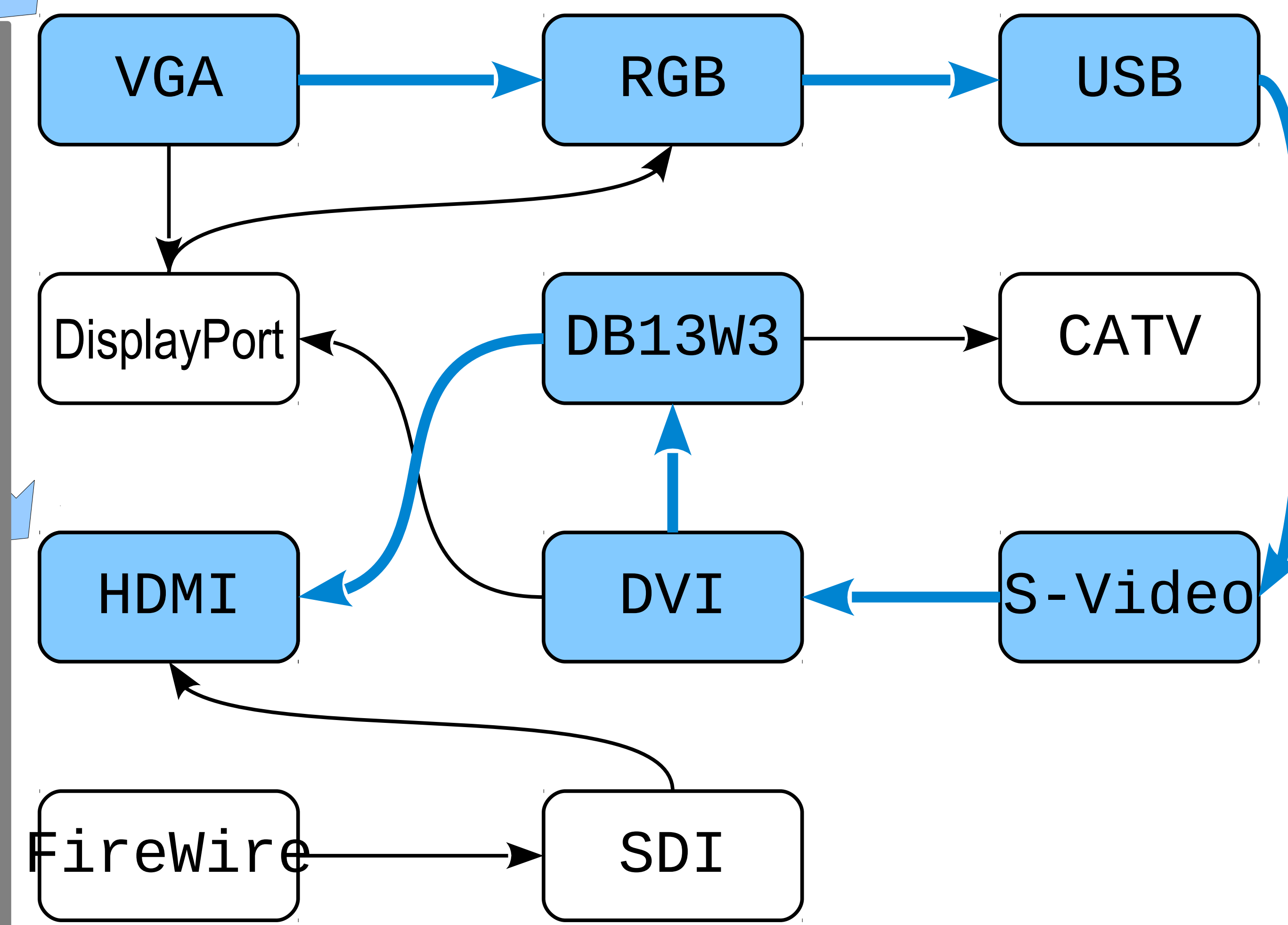
Converter Conundrums



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 that 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 **PollEv.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

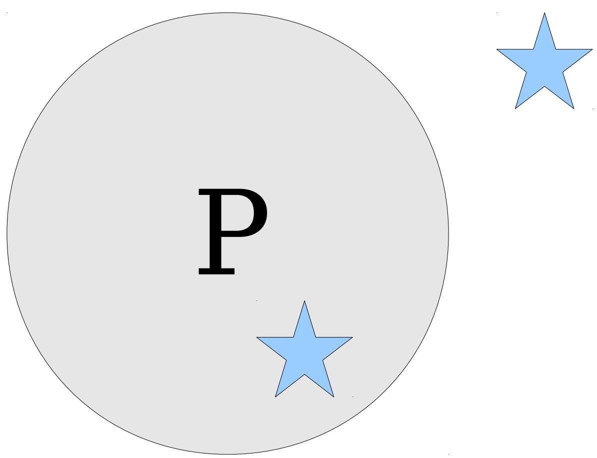
$$A \leq_p B.$$

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

* Assuming that `transform` runs in polynomial time.

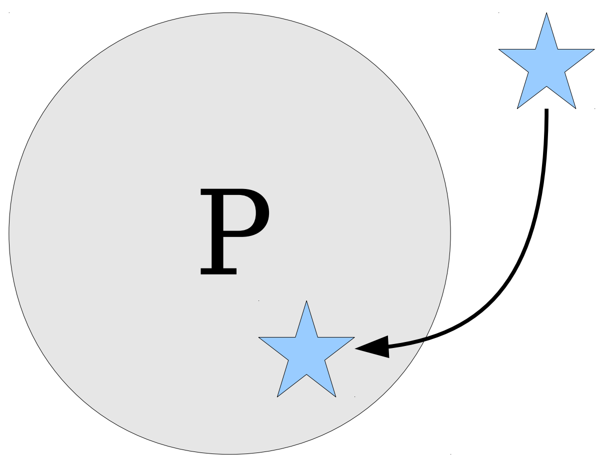
Polynomial-Time Reductions

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



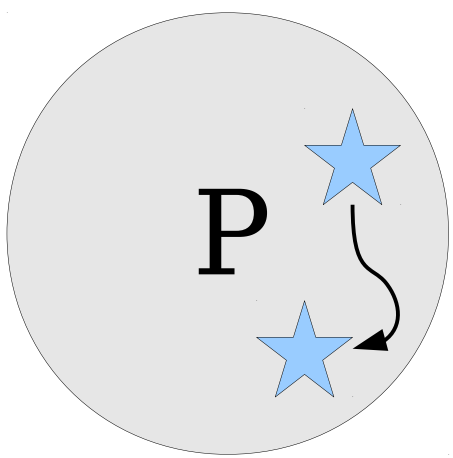
Polynomial-Time Reductions

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



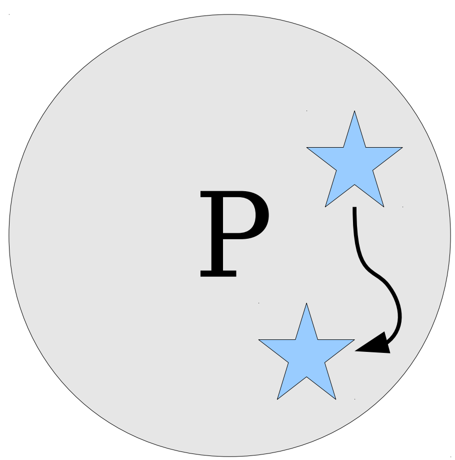
Polynomial-Time Reductions

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



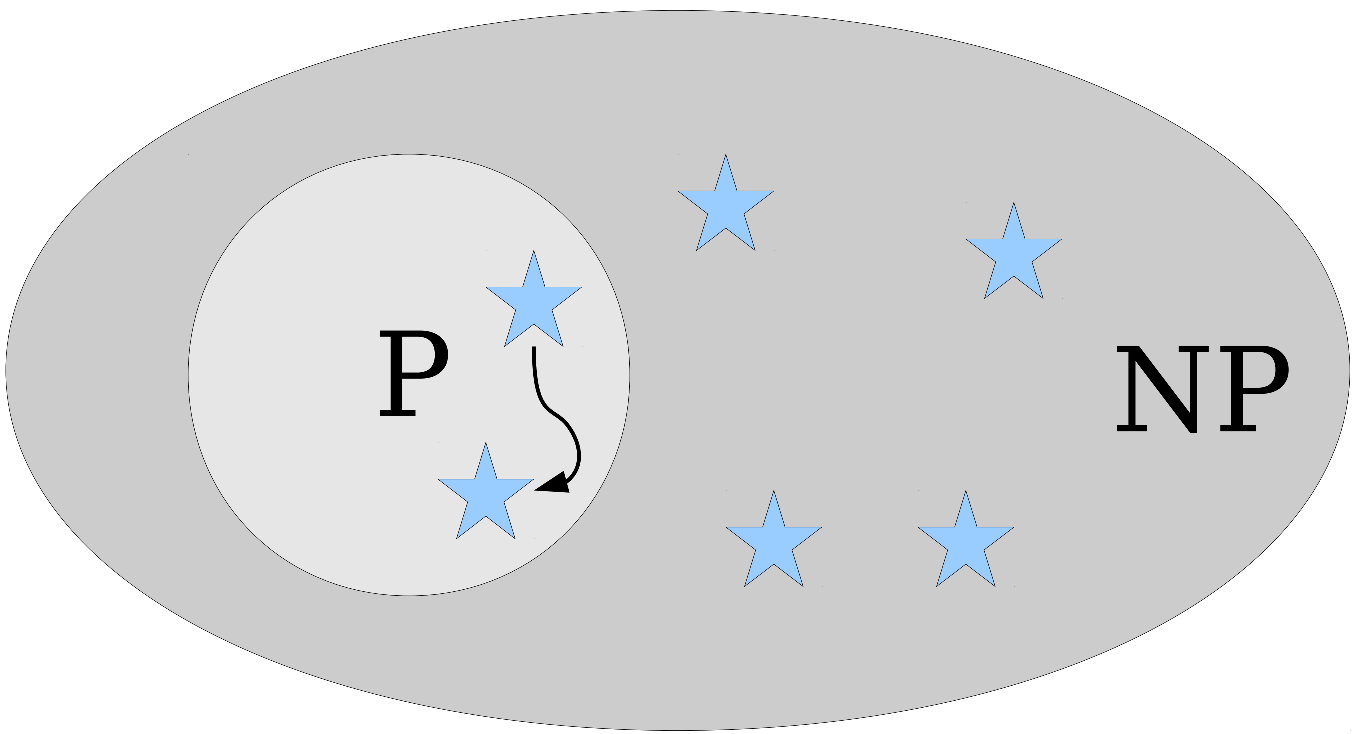
Polynomial-Time Reductions

- 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 $A \in \mathbf{NP}$.



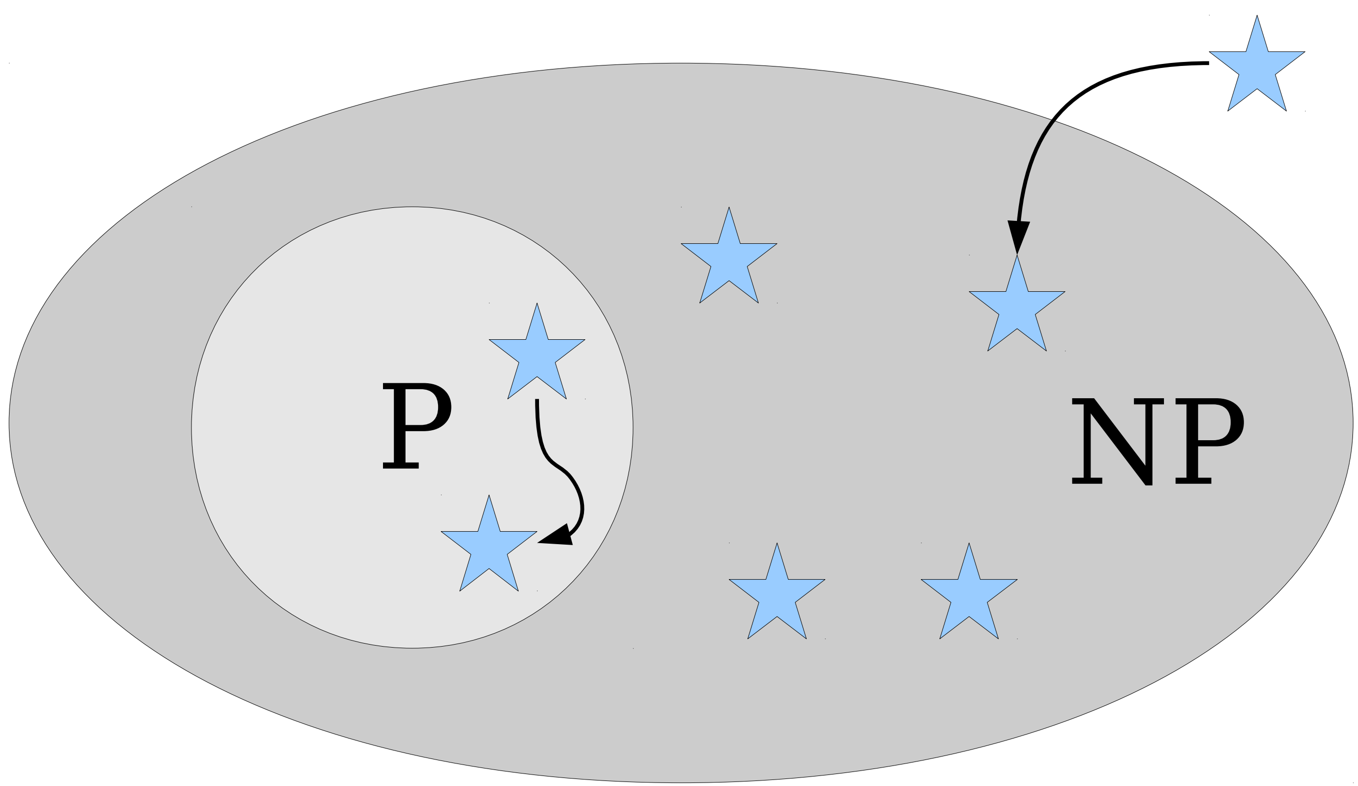
Polynomial-Time Reductions

- 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 $A \in \mathbf{NP}$.



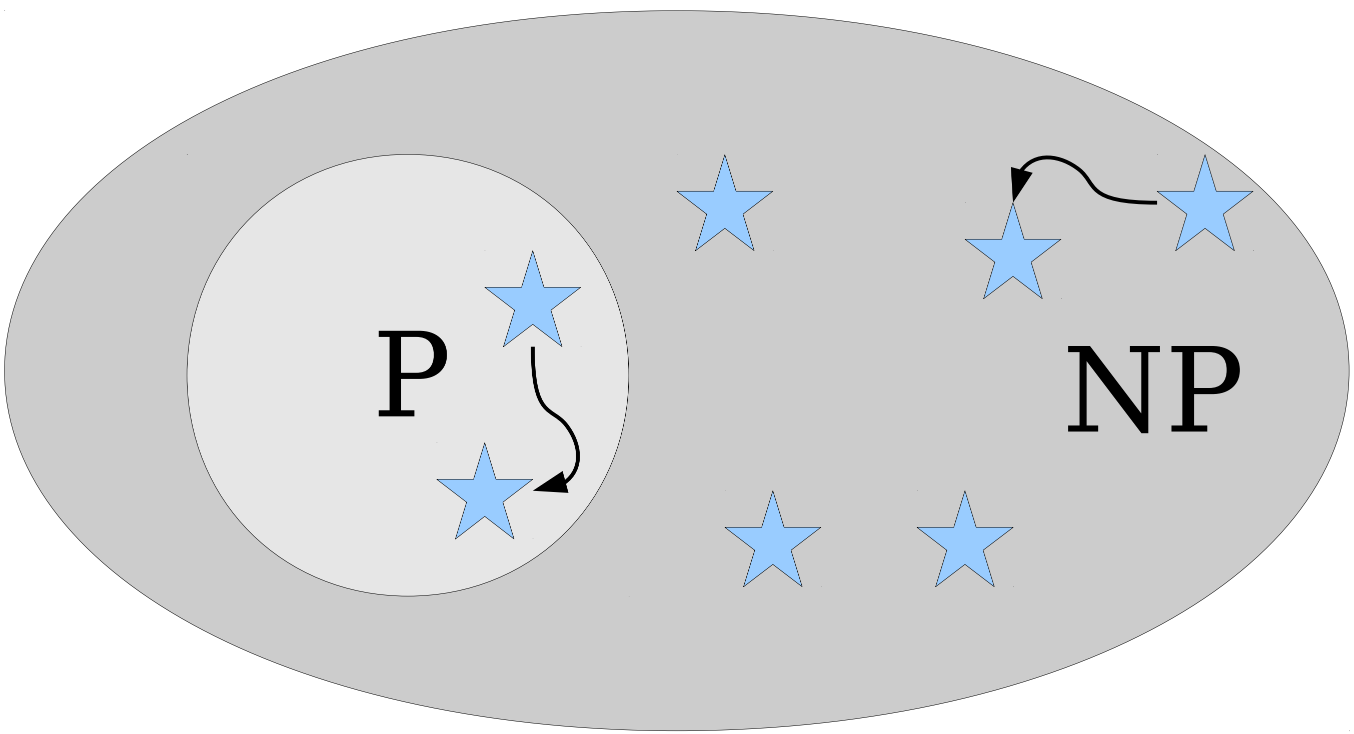
Polynomial-Time Reductions

- 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?

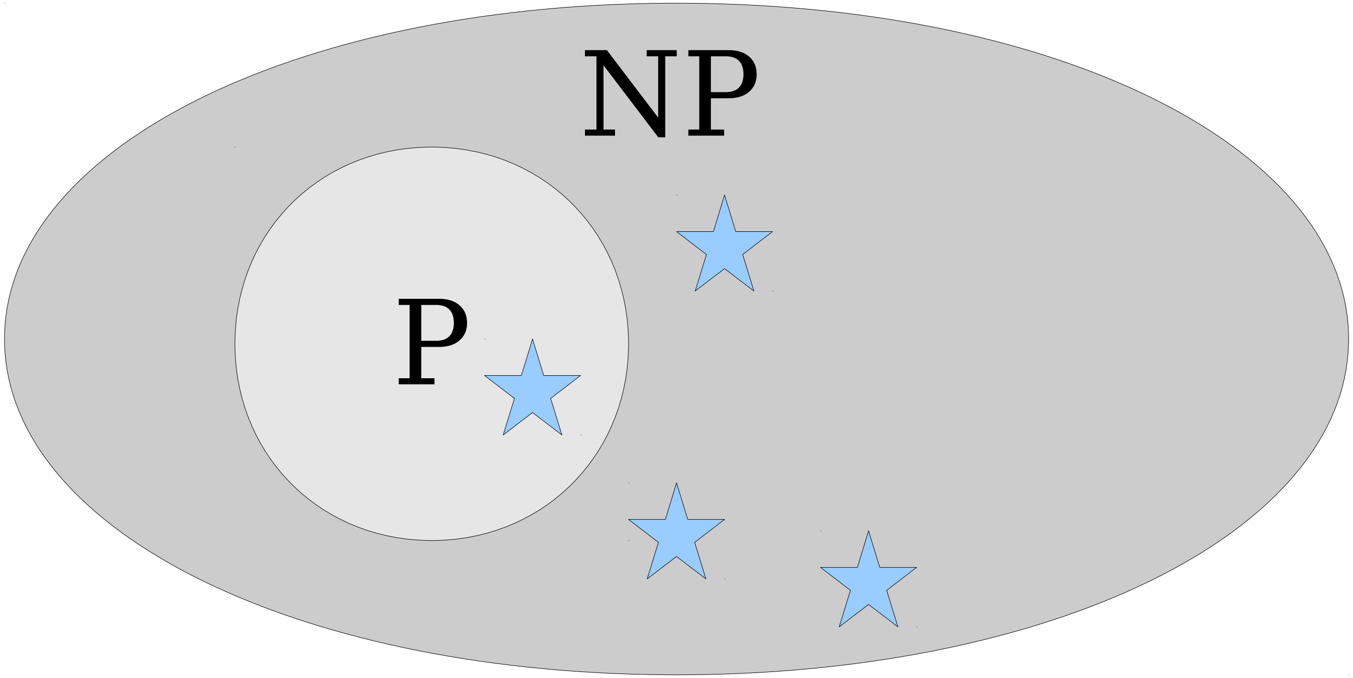
Intuition: If $A \leq_p B$, then problem B is at least as hard* as problem A.

* 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.

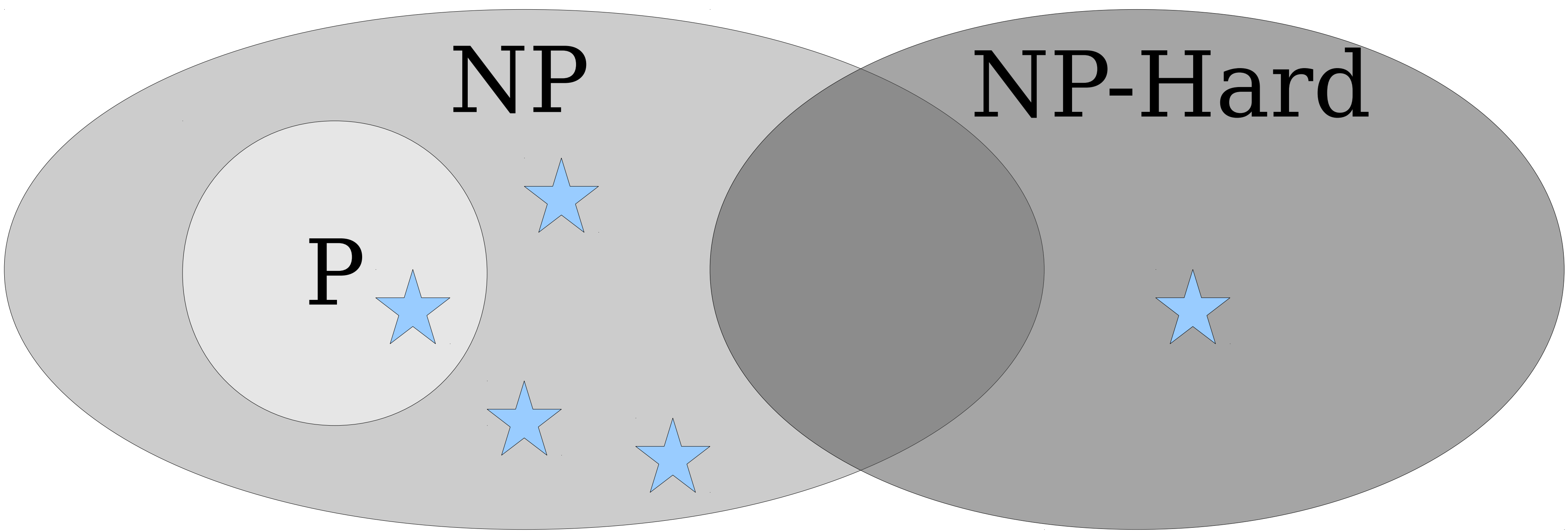
NP-Hardness

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



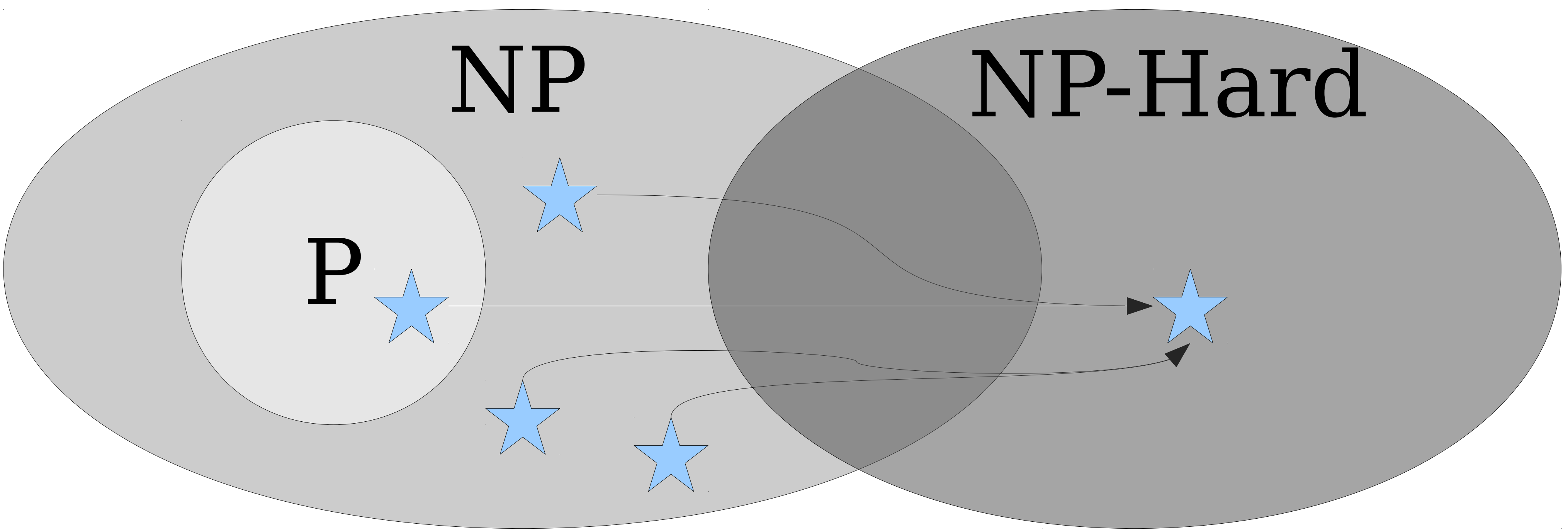
NP-Hardness

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



NP-Hardness

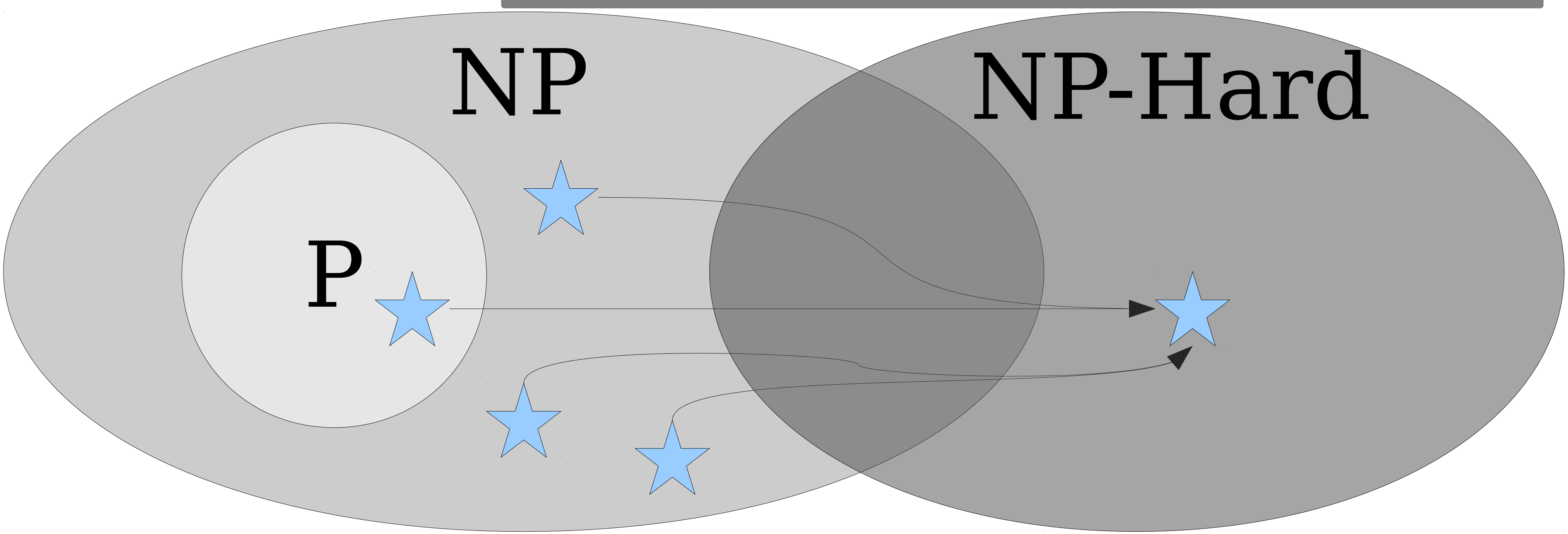
- A language L is called ***NP-hard*** if for *every* $A \in$ **NP**, we have $A \leq_{\mathrm{P}} L$.



NP-Hardness

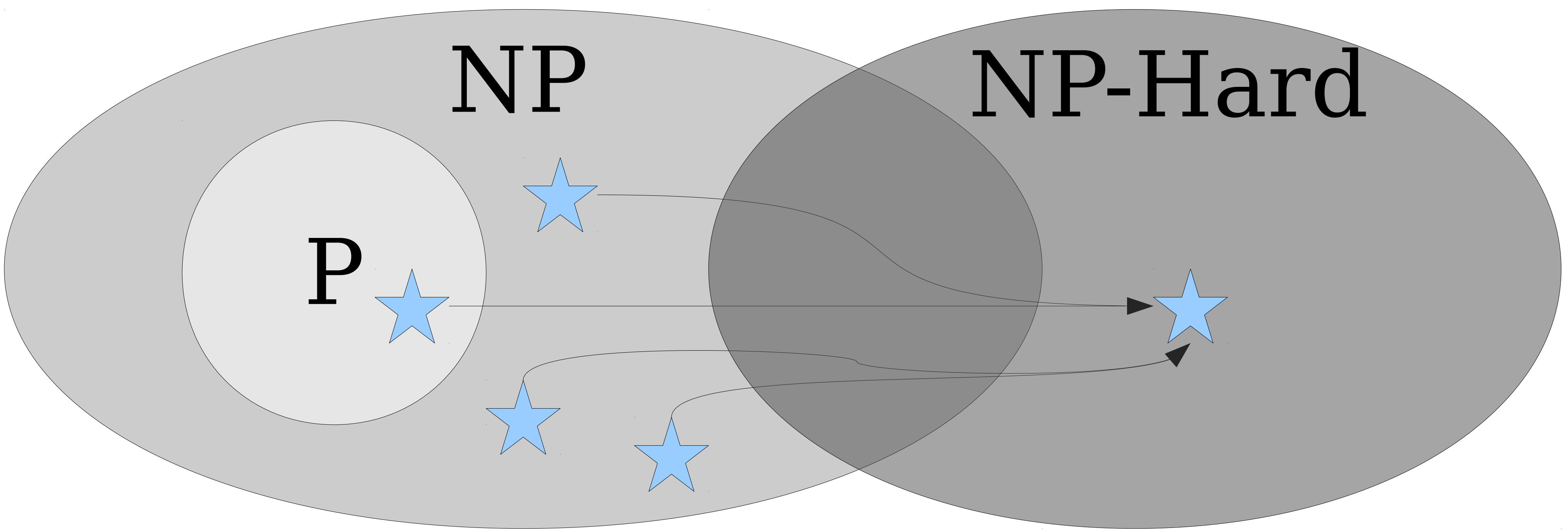
- A language L is called ***NP-hard*** if for *every* $A \in \mathbf{NP}$, we have $A \leq_P L$.

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



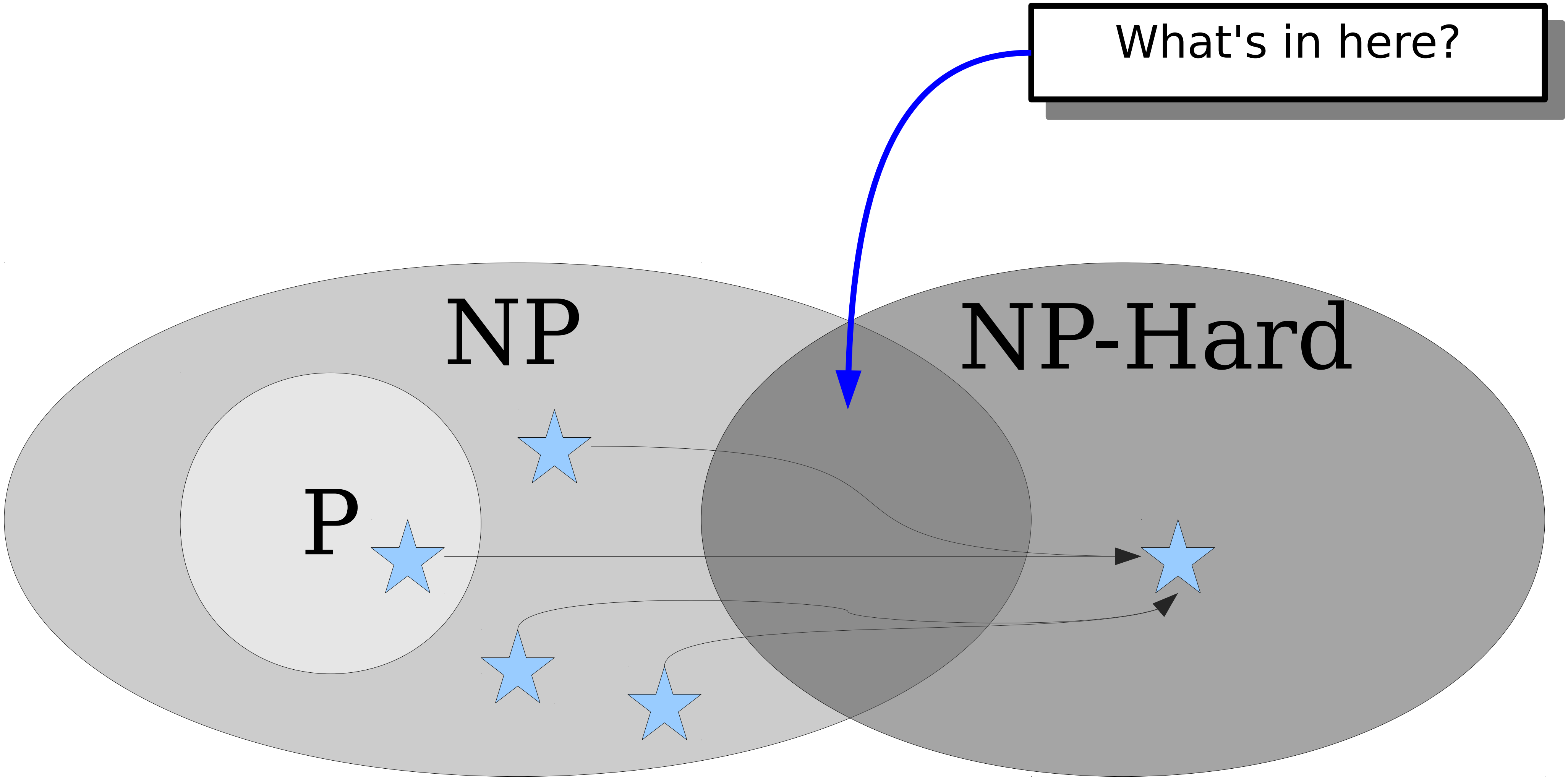
NP-Hardness

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



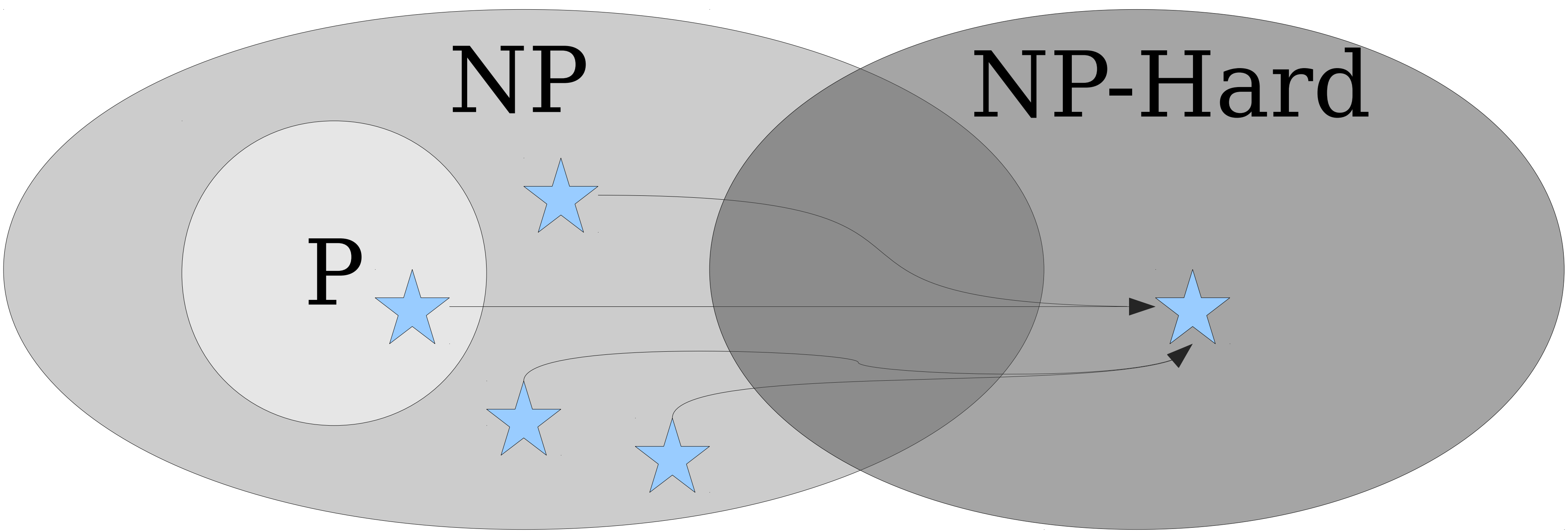
NP-Hardness

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



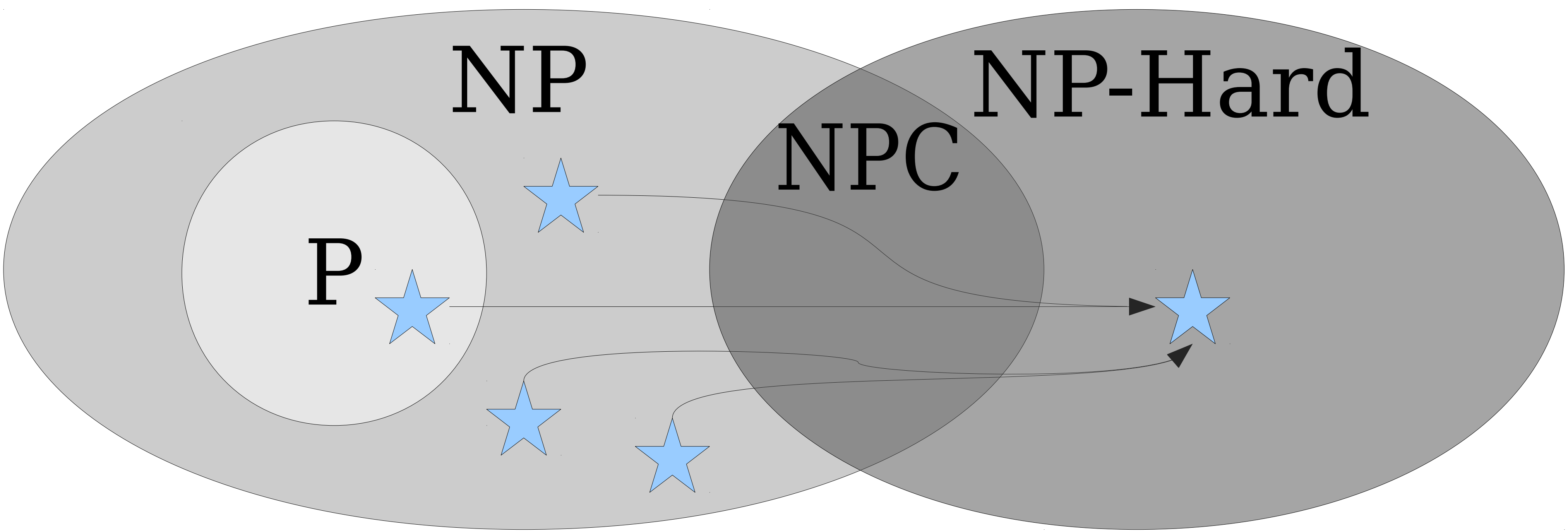
NP-Hardness

- 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* $A \in$ **NP**, we have $A \leq_{\mathrm{P}} L$.
- 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.

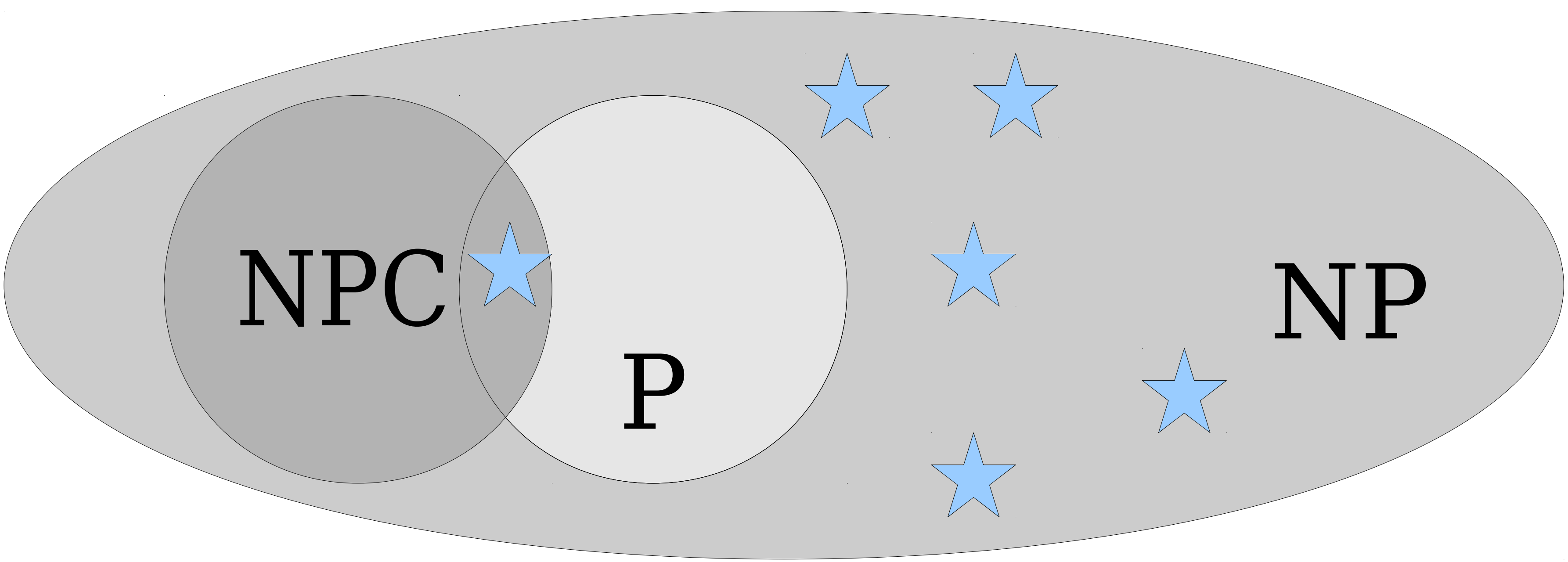


The Tantalizing Truth

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

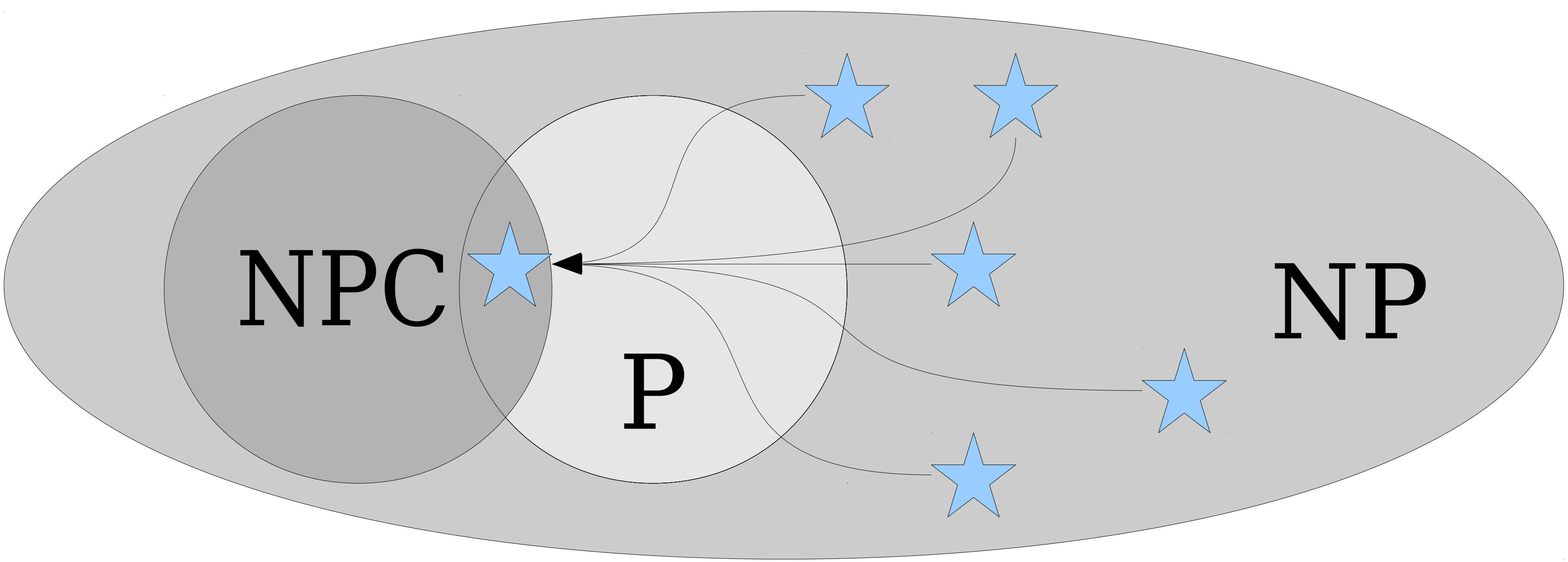
The Tantalizing Truth

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



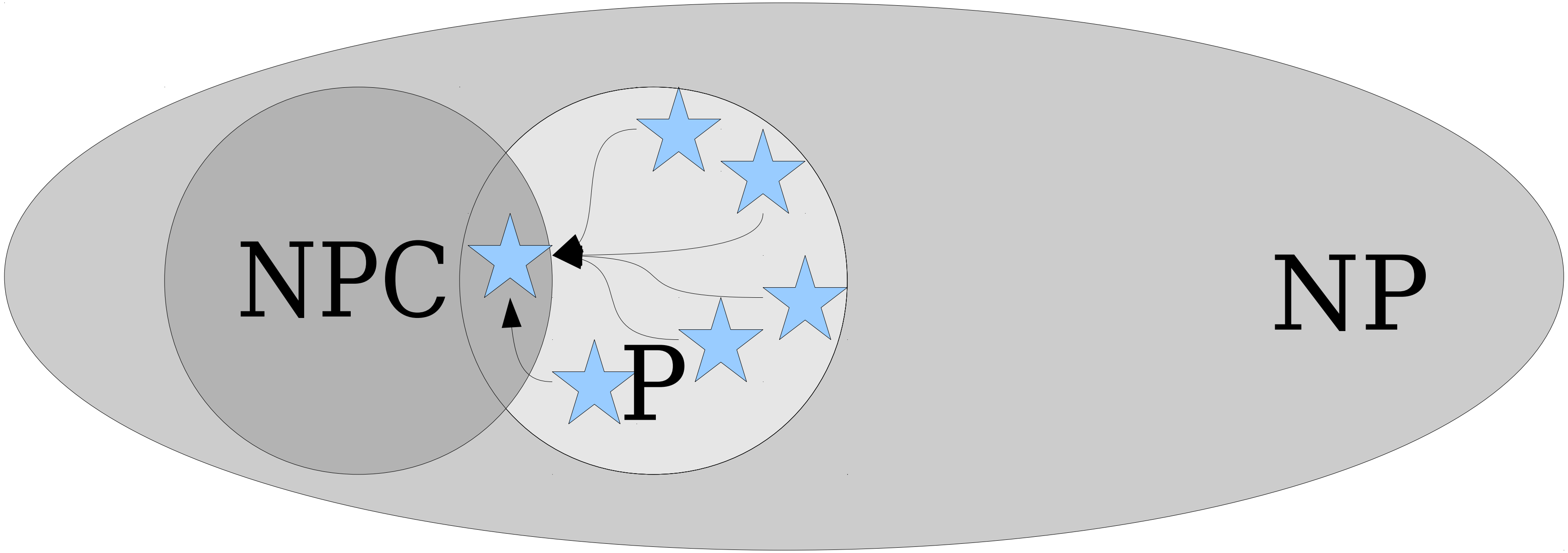
The Tantalizing Truth

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



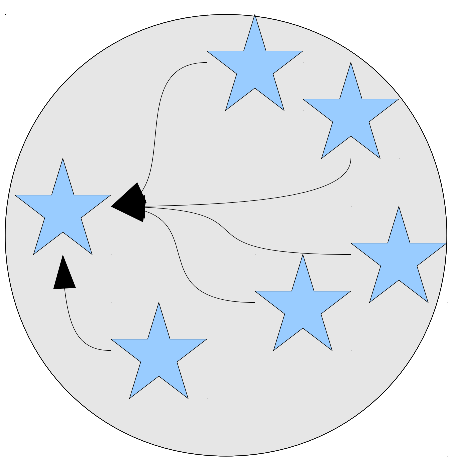
The Tantalizing Truth

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



The Tantalizing Truth

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

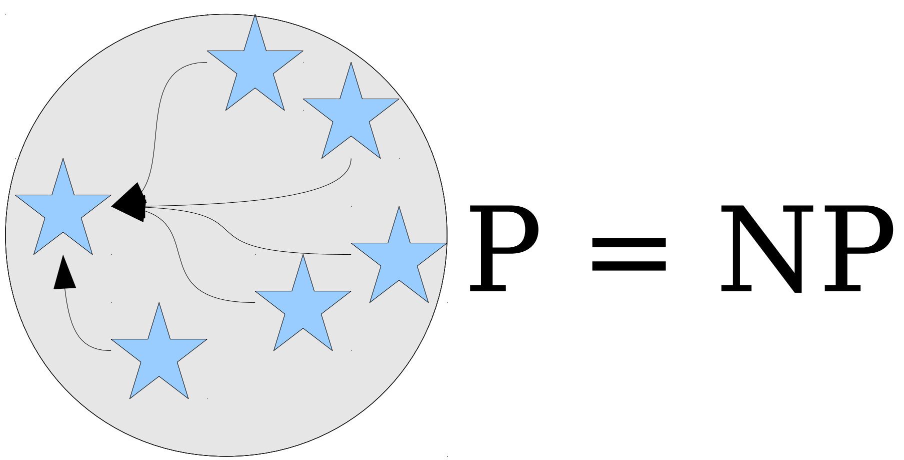


P = NP

The Tantalizing Truth

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

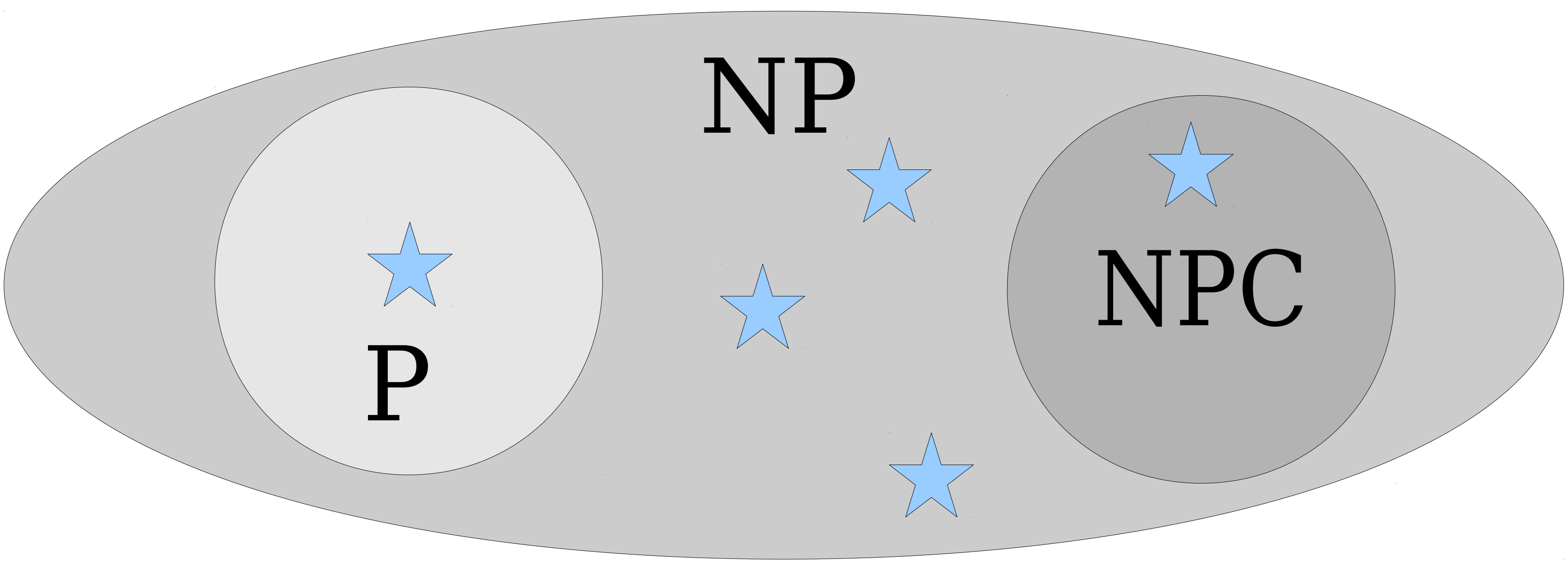
Proof: Suppose that L is **NP**-complete and $L \in$ **P**. Now consider any arbitrary **NP** problem A. Since L is **NP**-complete, we know that $A \leq_p L$. Since $L \in$ **P** and $A \leq_p L$, we see that $A \in$ **P**. Since our choice of A was arbitrary, this means that **NP** $\subseteq$ **P**,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



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:

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

SAT = { ⟨φ⟩ | φ is a satisfiable PL 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 φ?

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** ∈ **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 polymomial-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 SAT $\in \mathbf{P}$, then $\mathbf{P} = \mathbf{NP}$.

    If SAT $\notin \mathbf{P}$, then $\mathbf{P} \neq \mathbf{NP}$.

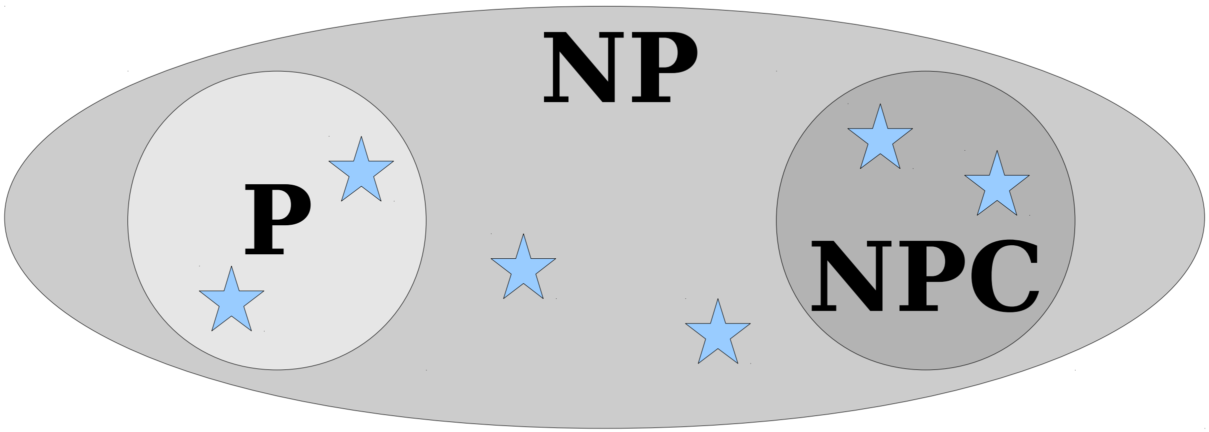
Sample **NP**-Hard Problems

- ***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, twoplayer 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)*

Coda: What if $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ is resolved?

Intermediate Problems

- With few exceptions, every problem we've discovered in **NP** has either
  - definitely been proven to be in **P**, or
  - definitely been proven to be **NP**-complete.
- A problem that's **NP**, not in **P**, but not **NP**-complete is called ***NP-intermediate***.
- ***Theorem (Ladner):*** There are **NP**-intermediate problems if and only if **P** ≠ **NP**.



What if $\mathbf{P} \neq \mathbf{NP}$?

A Good Read:

"A Personal View of Average-Case Complexity" by Russell Impagliazzo

What if $\mathbf{P} = \mathbf{NP}$?

And a Dismal Third Option