Complexity Theory

Part Two

Outline for Today

- ***Recap from Last Time***
  - Where are we, again?
- ***Reducibility***
  - Linking problems together.
- ***NP-Hardness***
  - Defining hardness via reducibility.
- ***NP-Completeness***
  - A mighty interesting class of problems!
- ***The Cook-Levin Theorem***
  - A surprisingly counterintuitive result.

Recap from Last Time

The Cobham-Edmonds Thesis

A language L can be ***decided efficiently*** if there is a TM that decides it in polynomial time.

Equivalently, L can be decided efficiently if it can be decided in time $O(n^k)$ for some $k \in \mathbb{N}$.

Like the Church-Turing thesis, this is ***not*** a theorem!

It's an assumption about the nature of efficient computation, and it is somewhat controversial.

The Complexity Class **P**

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

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

- Assuming the Cobham-Edmonds thesis, a language is in **P** if it can be decided efficiently.

Polynomial-Time Verifiers

- A ***polynomial-time verifier*** for L is a TM V such that
  - V halts on all inputs.
  - $w \in L$ iff $\exists c \in \Sigma^*$. V accepts $\langle w, c \rangle$.
  - V's runtime is a polynomial in $|w|$ (that is, V's runtime is $O(|w|^k)$ for some integer k)

The Complexity Class **NP**

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

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

- The name **NP** comes from another way of characterizing **NP**. If you introduce *nondeterministic Turing machines* and appropriately define "polynomial time," then **NP** is the set of problems that an NTM can solve in polynomial time.

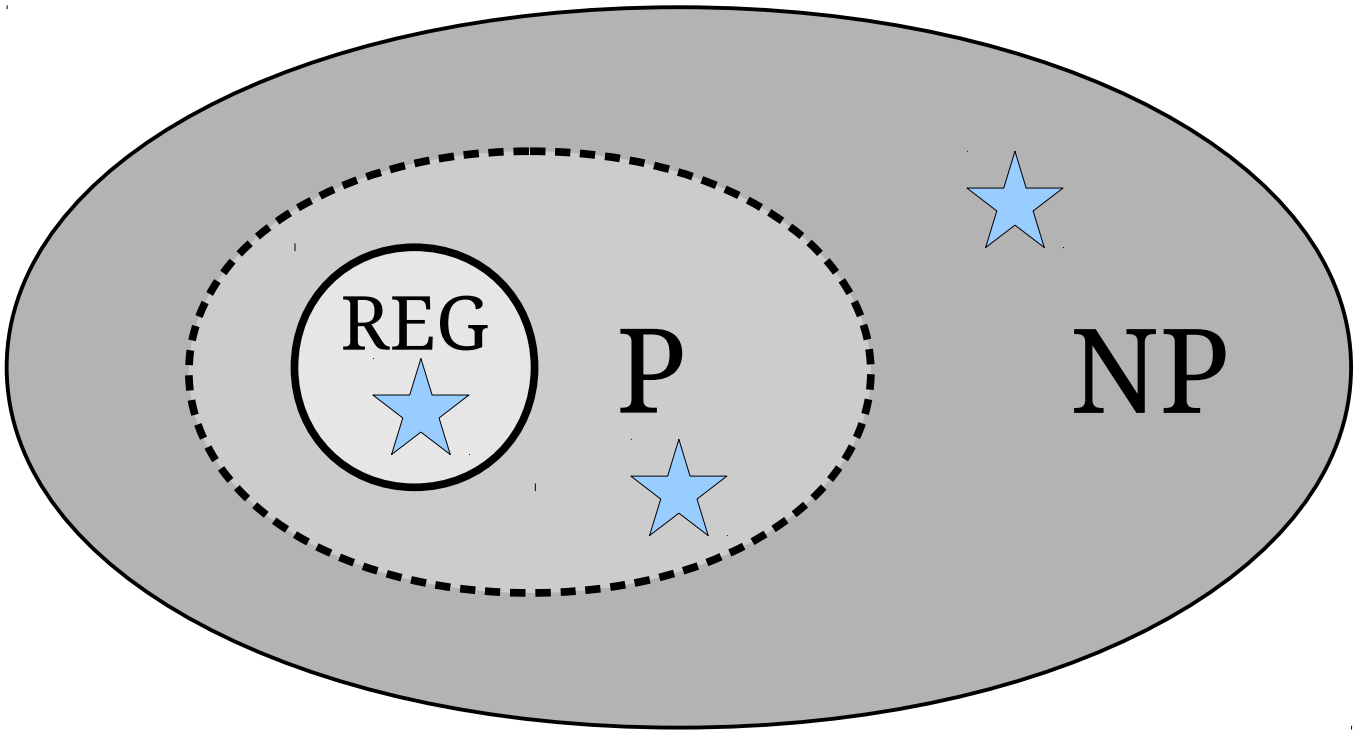
Theorem (Baker-Gill-Solovay): Any proof that purely relies on universality and self-reference cannot resolve $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$.

Proof: Take CS154!

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?

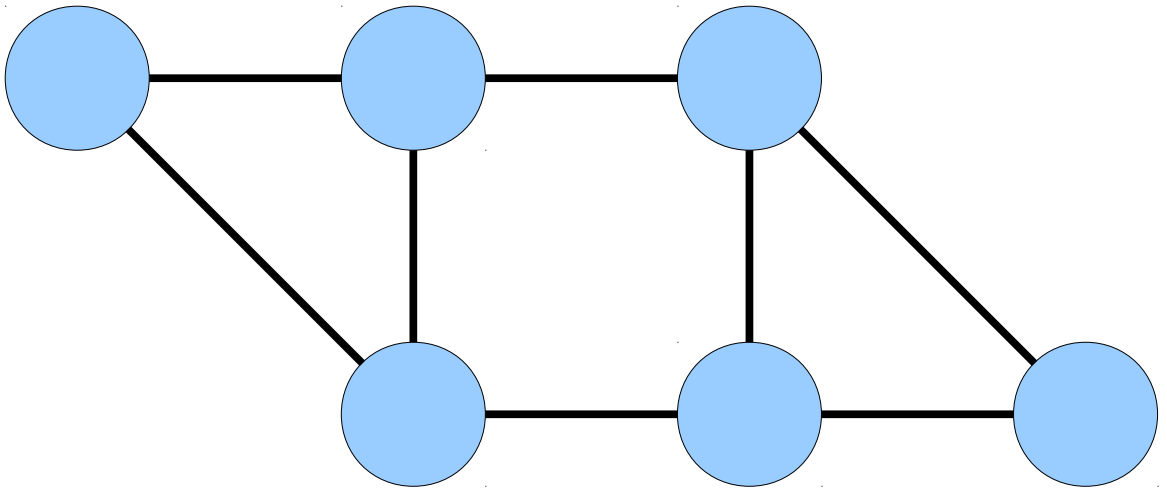
Reducibility

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.

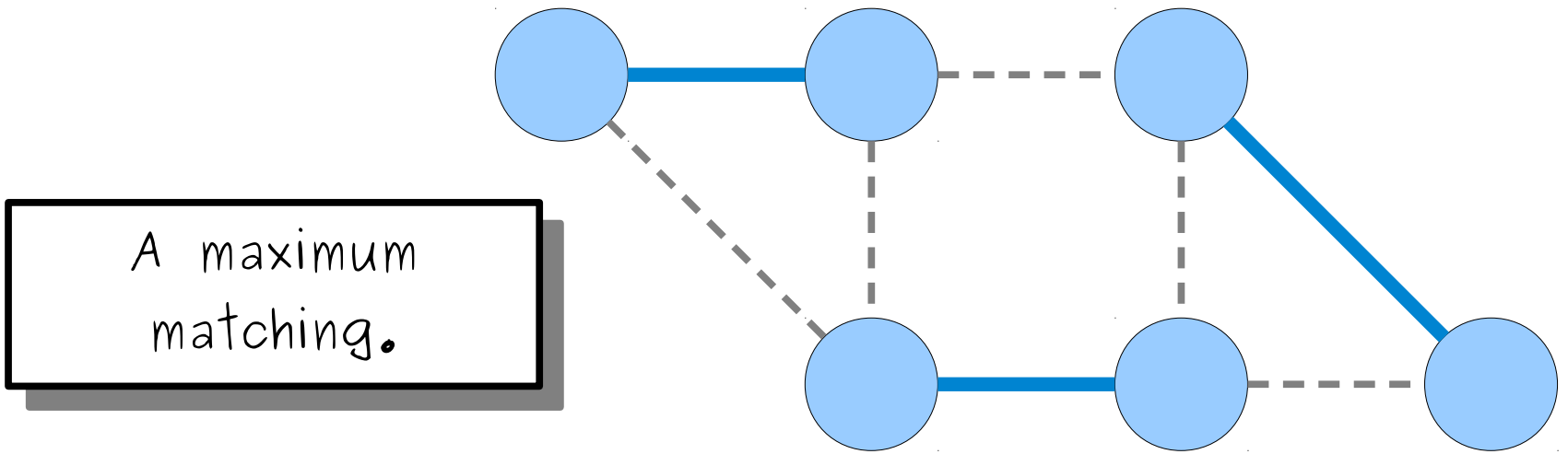
Maximum Matching

- 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.



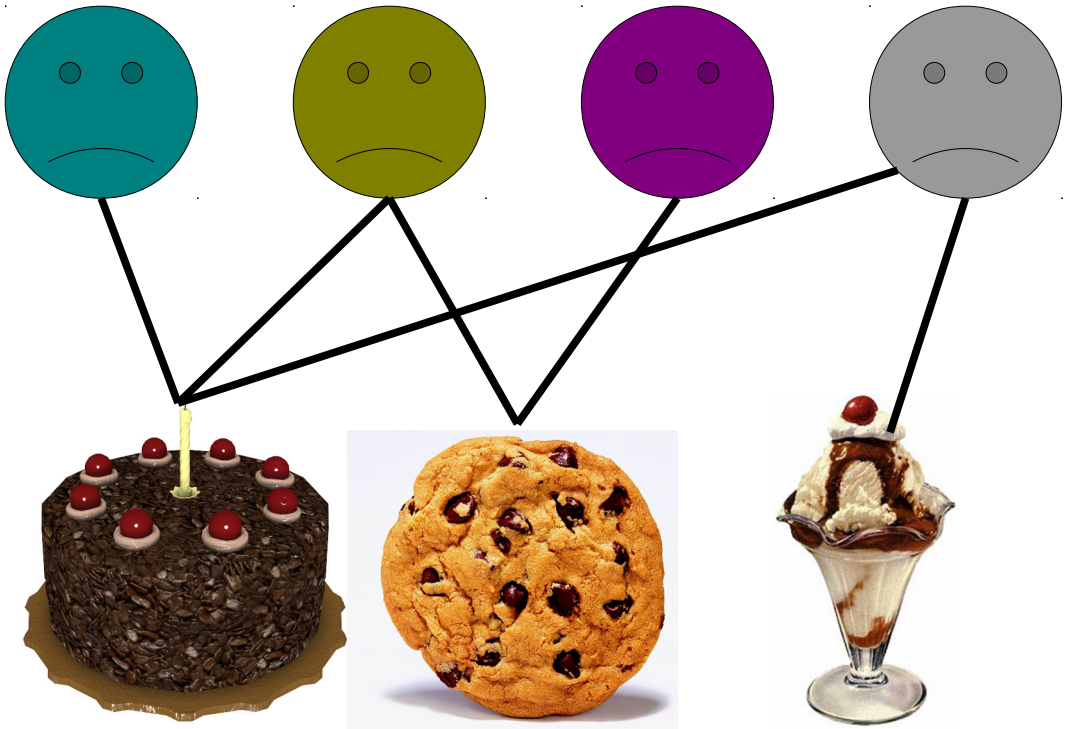
Maximum Matching

- 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.



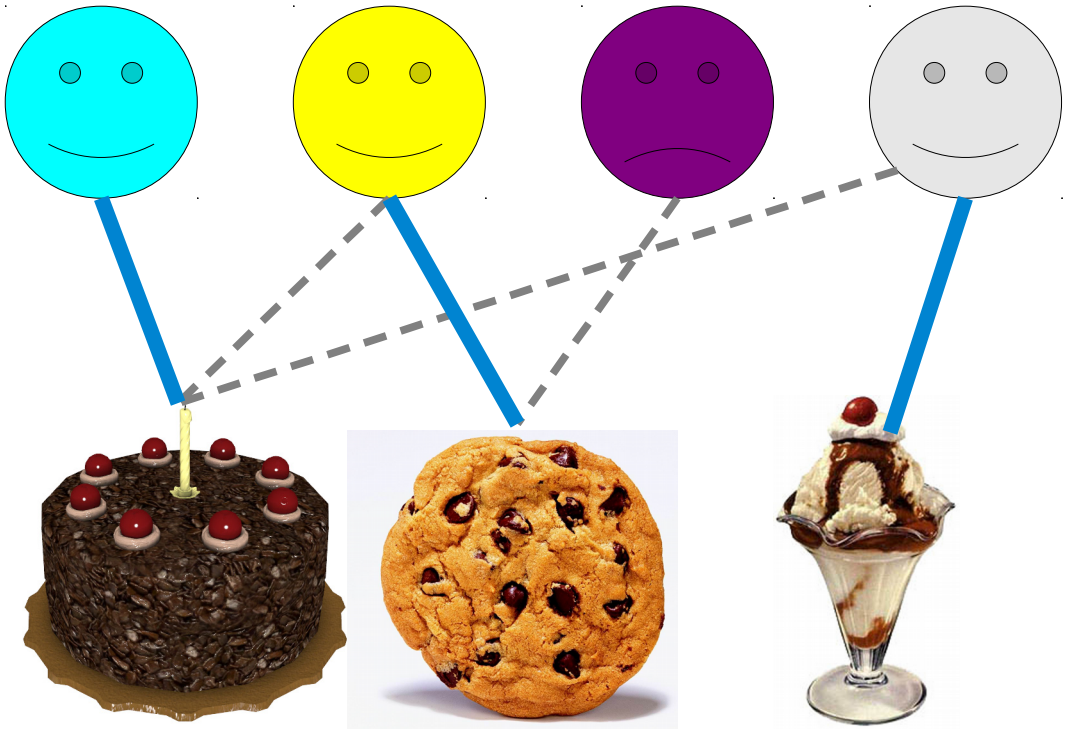
Maximum Matching

- 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.

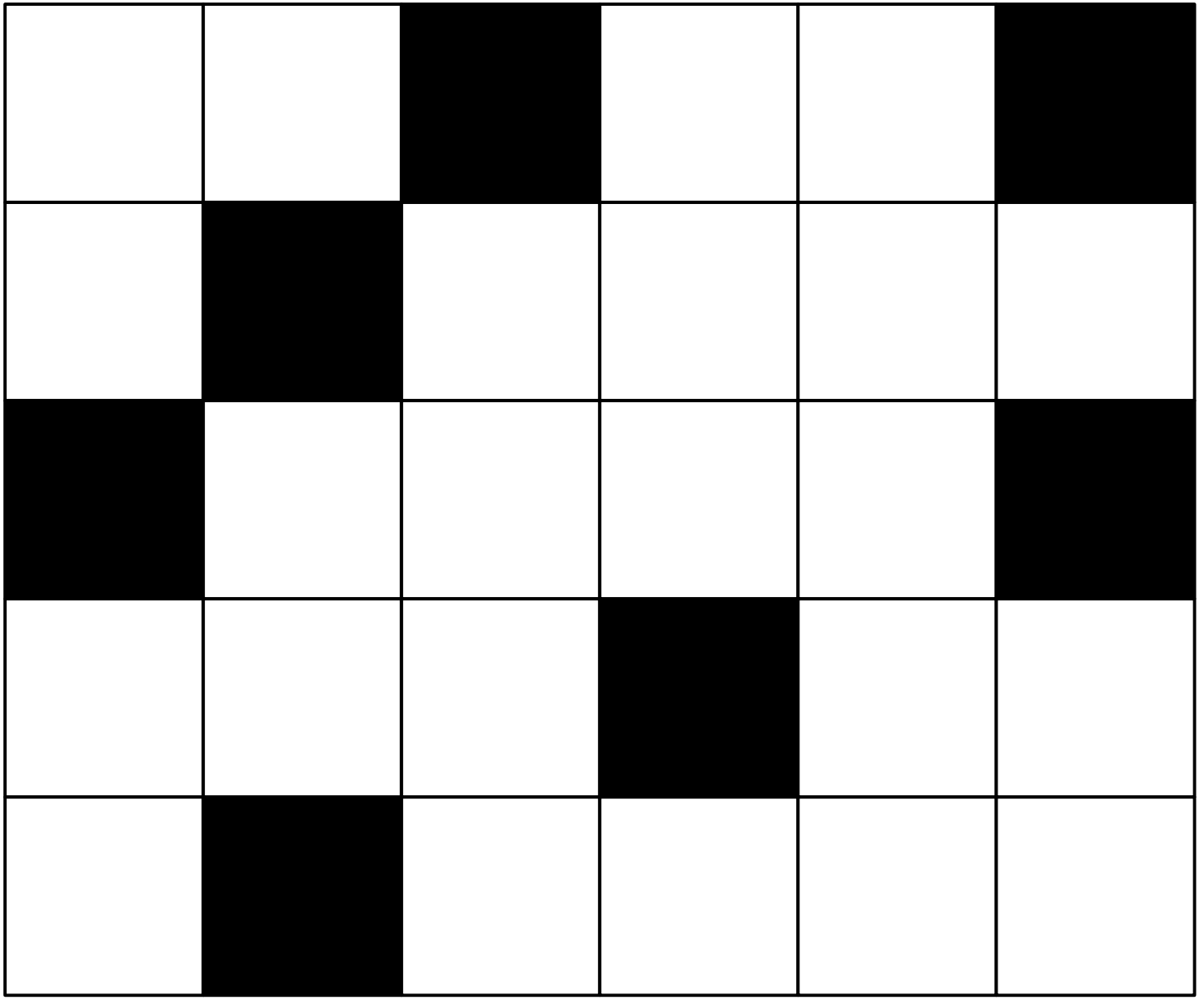


Maximum Matching

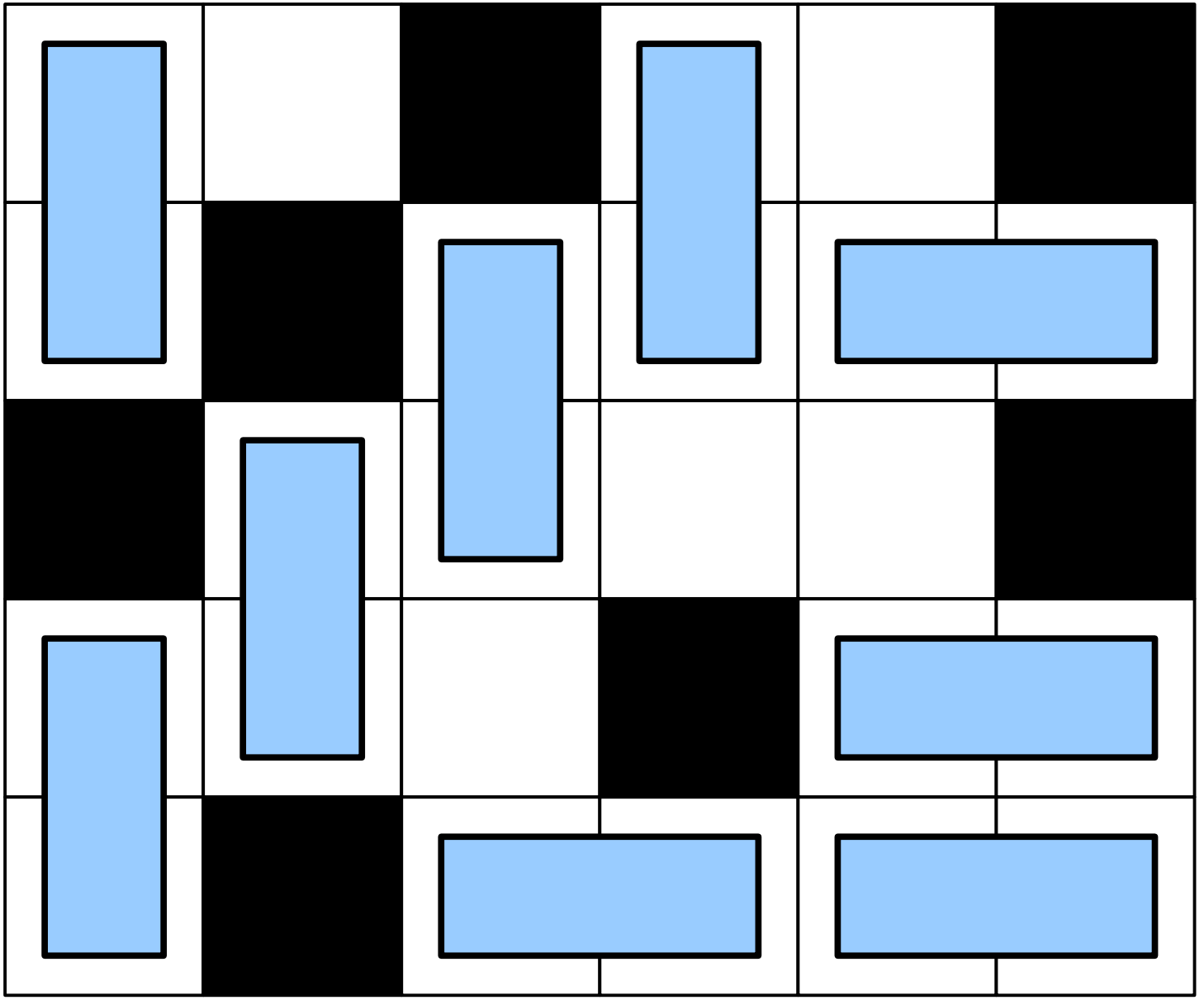
- Given an undirected graph G, a ***matching*** in G is a set of edges such that no two edges share an endpoint.
- A ***maximum matching*** is a matching with the largest number of edges.



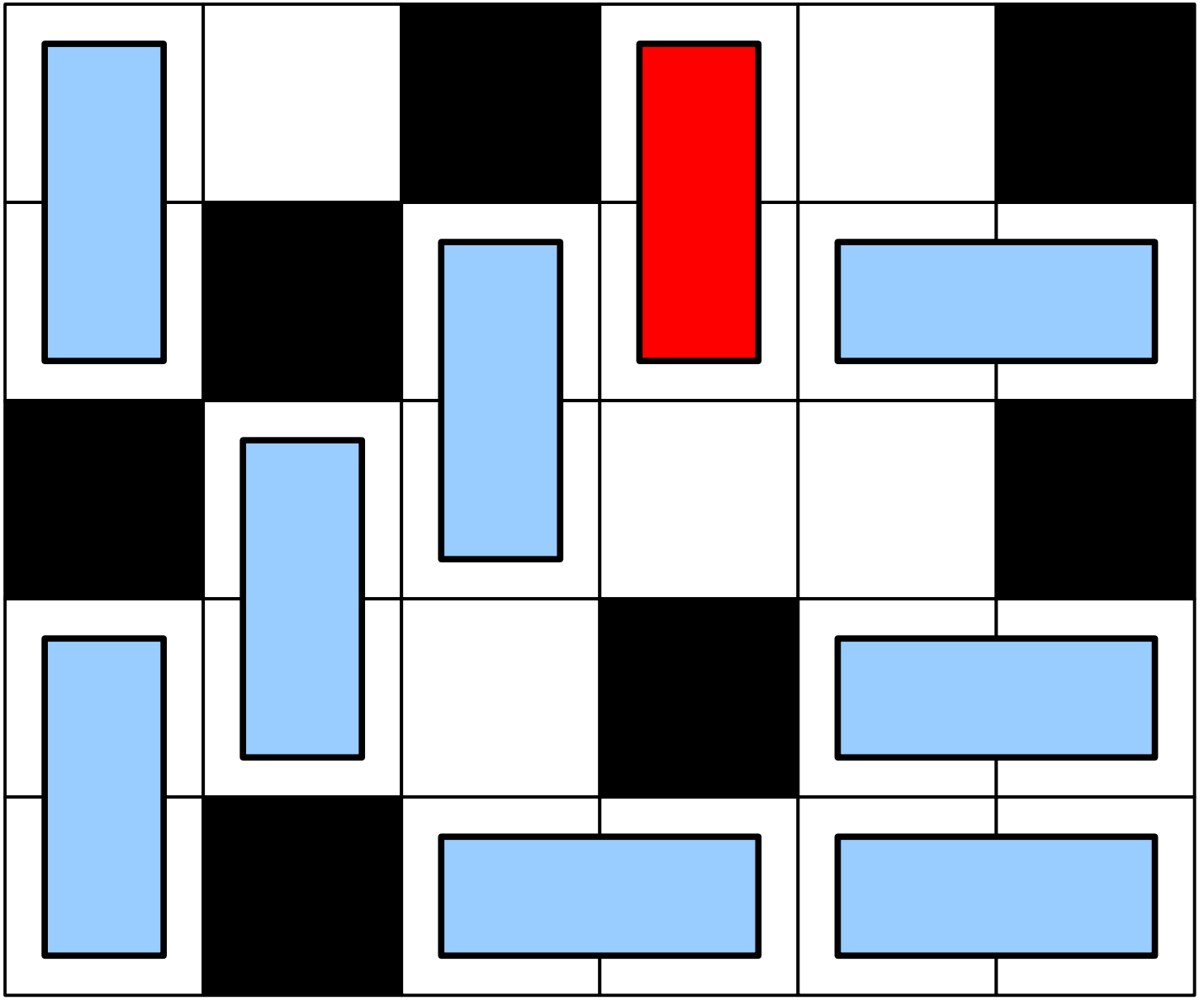
Domino Tiling



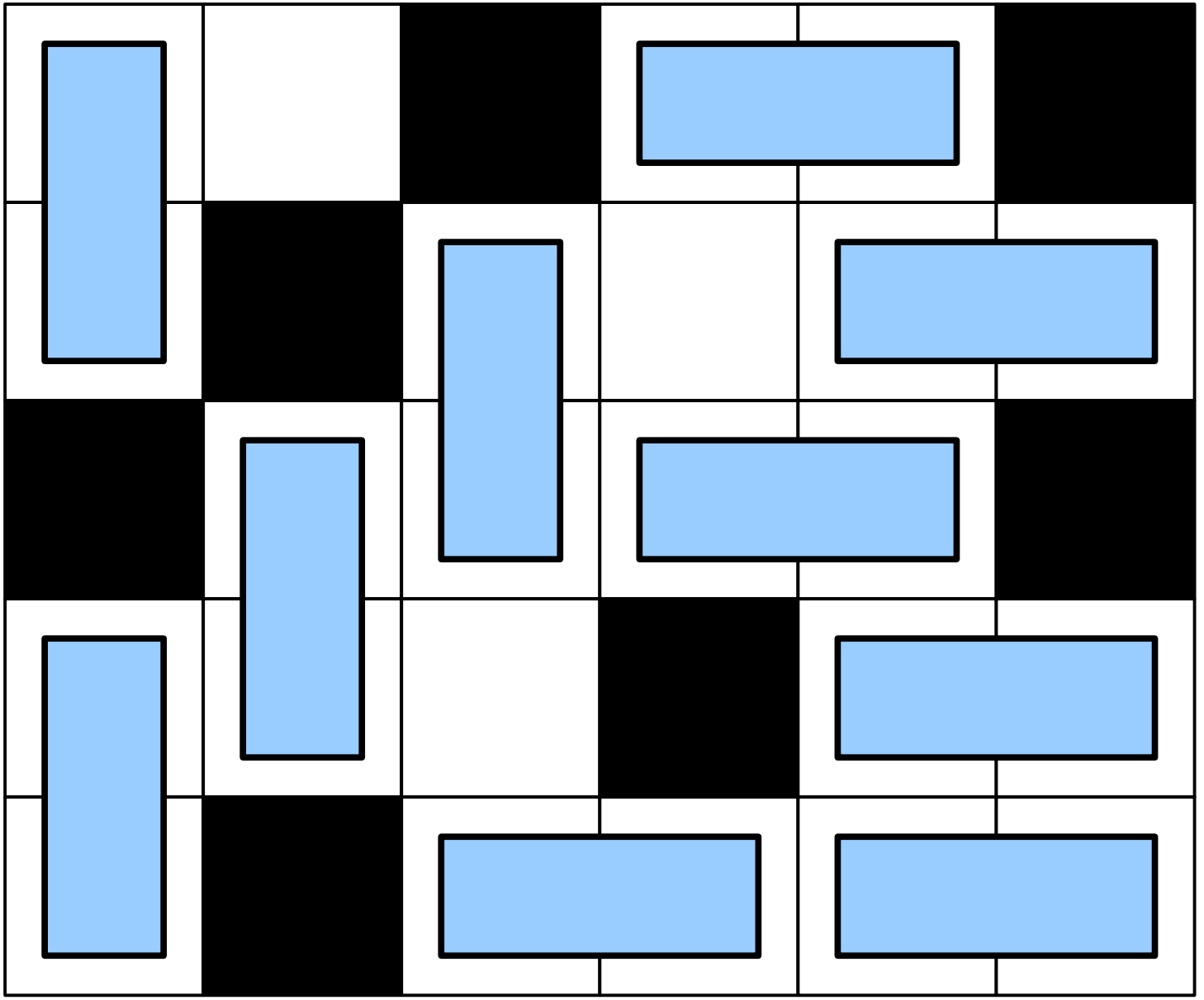
Domino Tiling



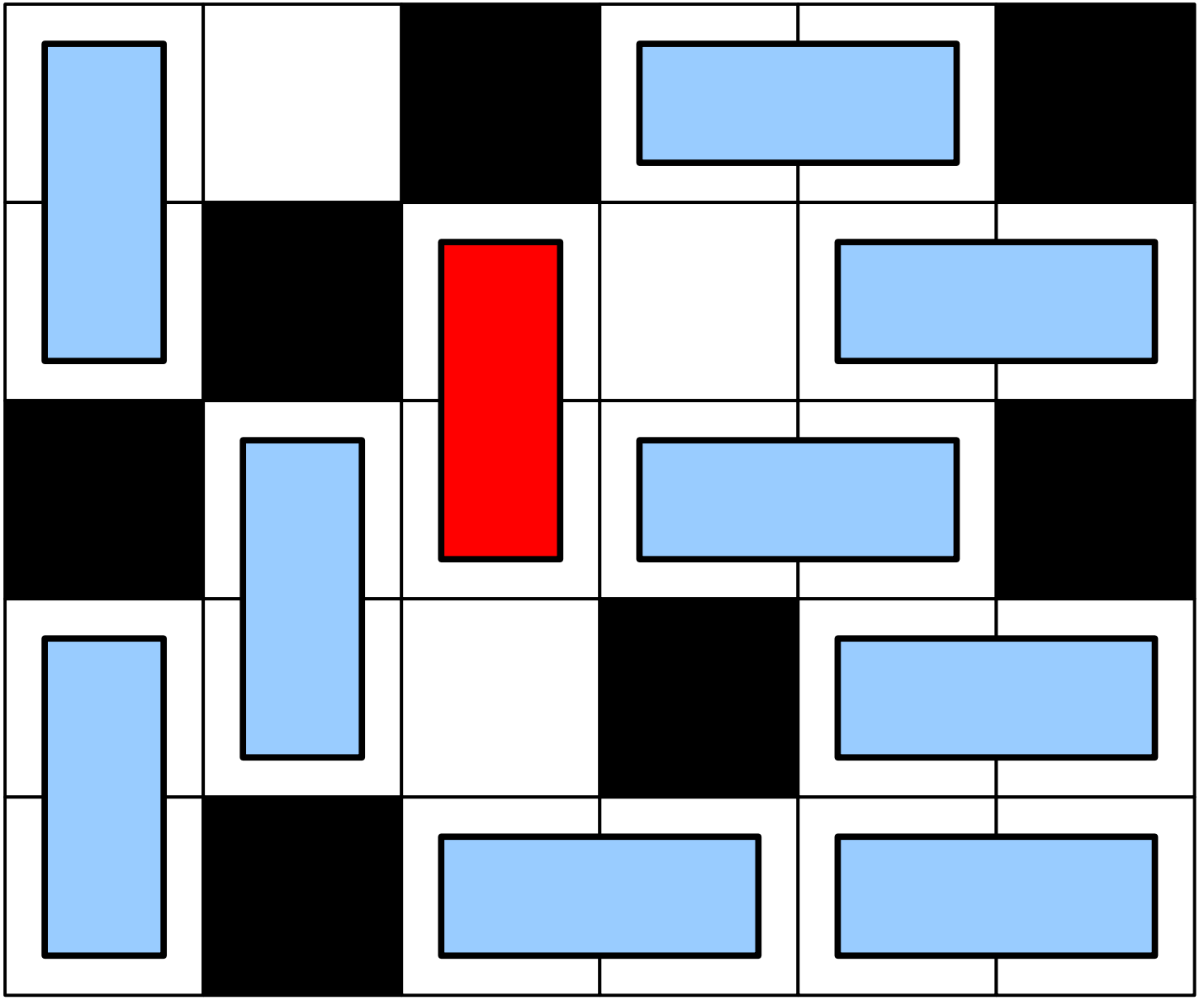
Domino Tiling



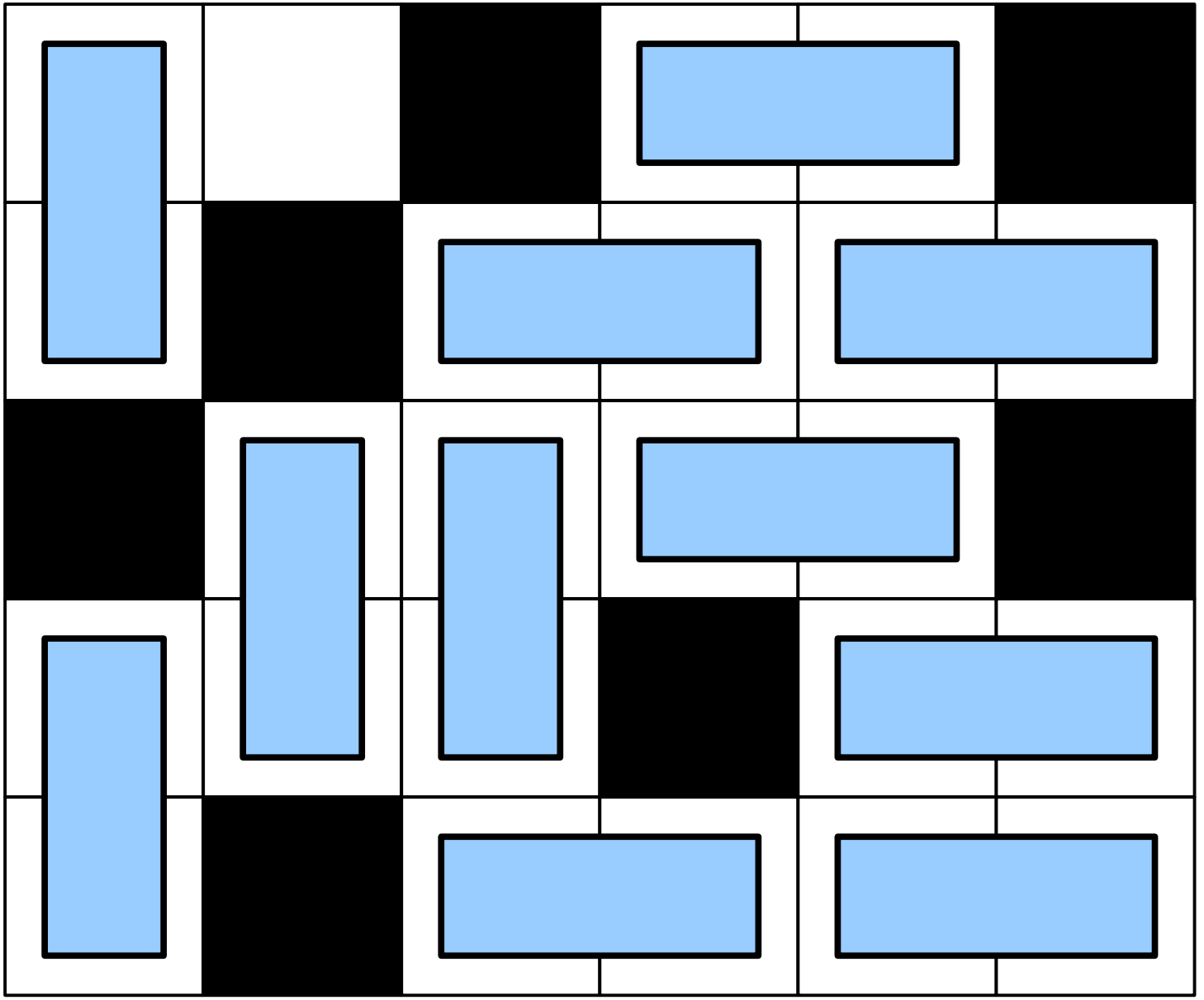
Domino Tiling



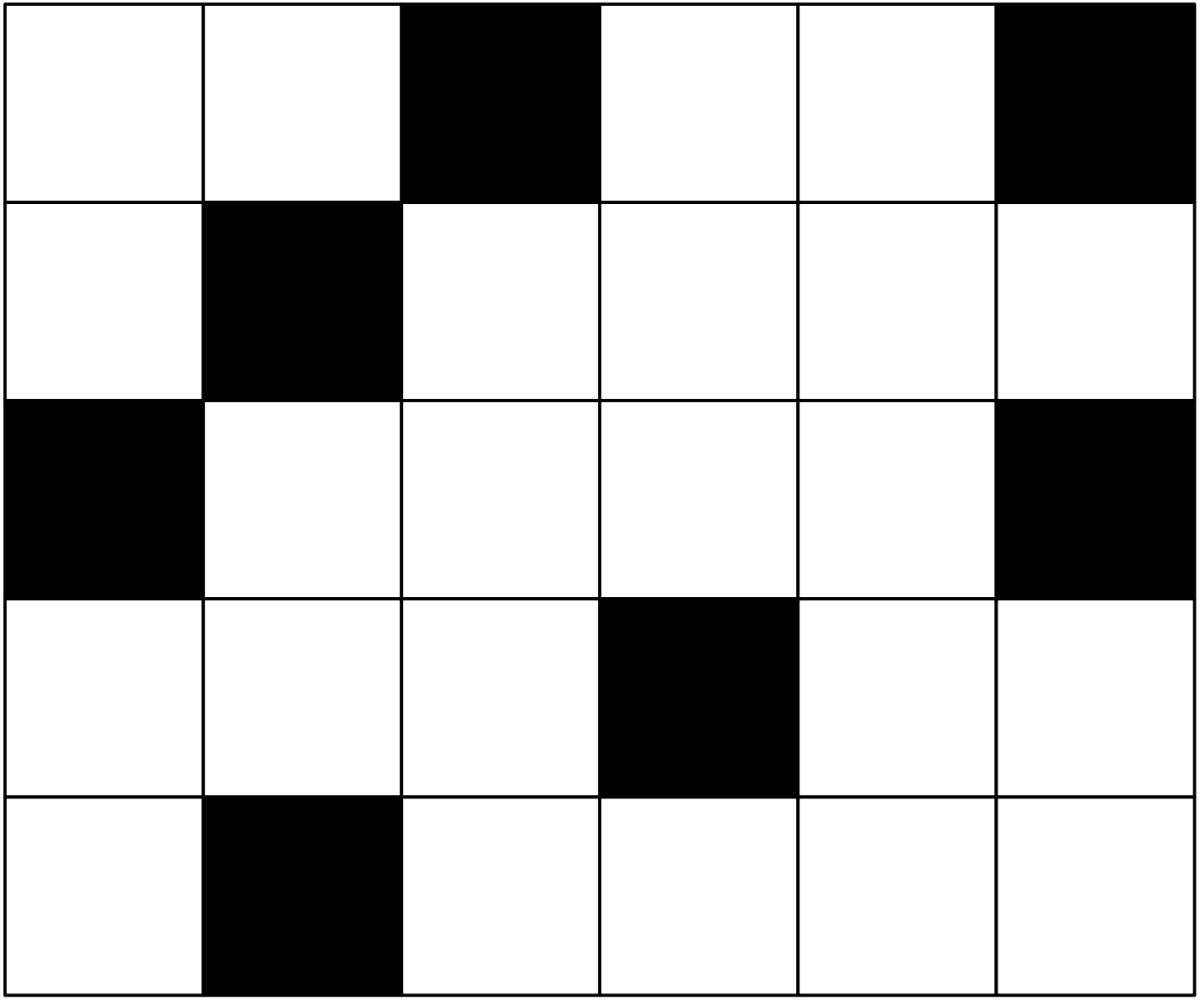
Domino Tiling



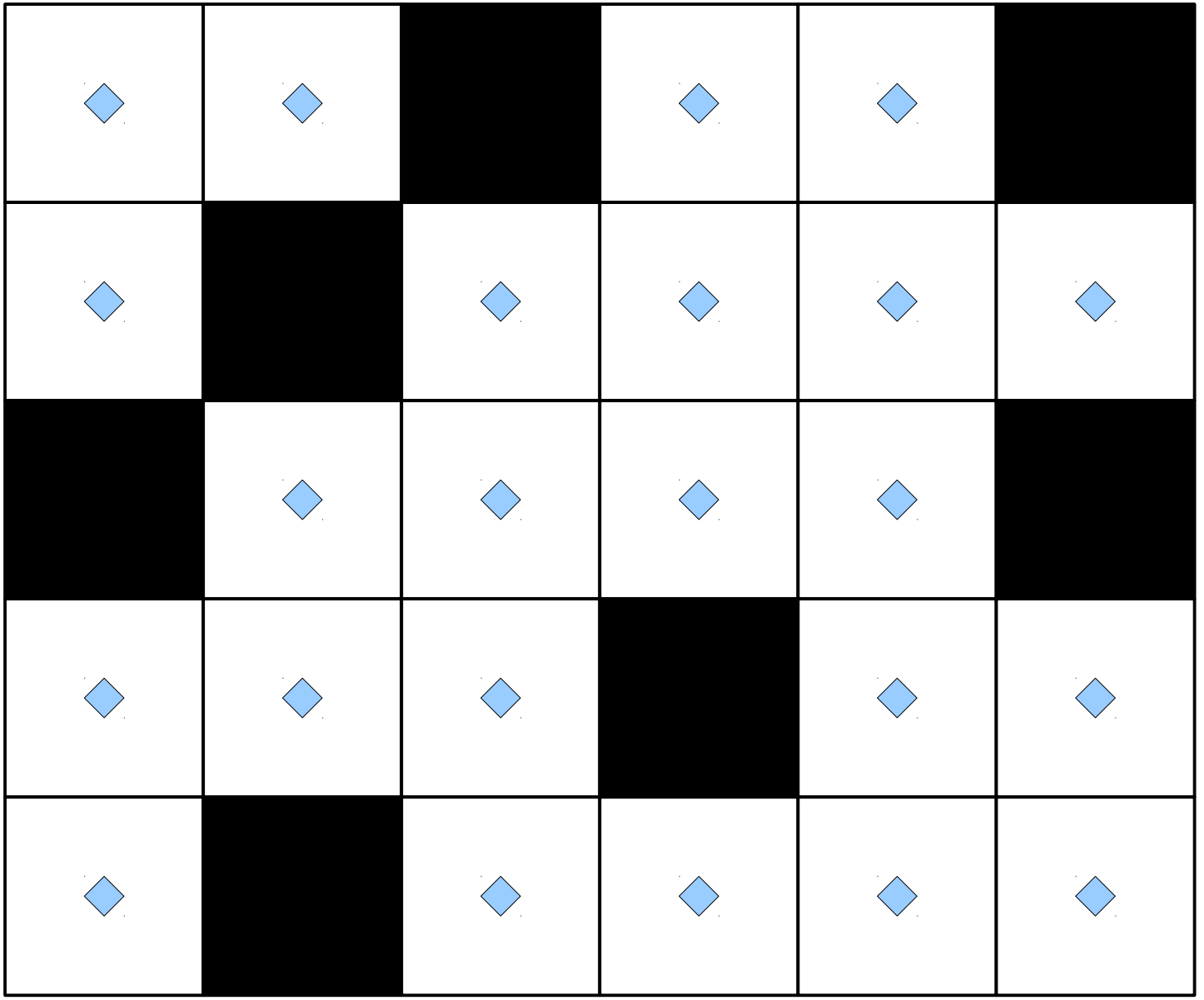
Domino Tiling



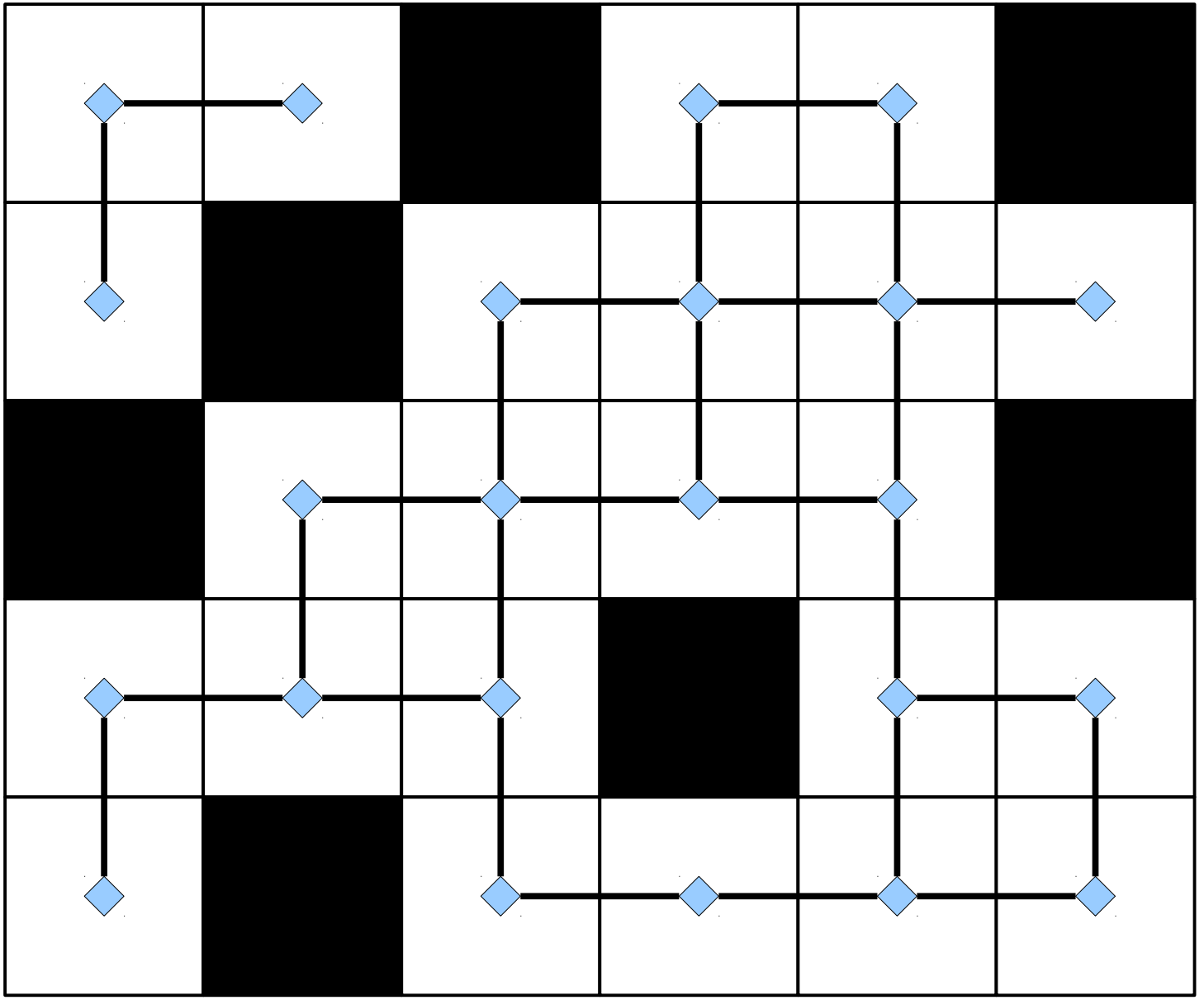
Solving Domino Tiling



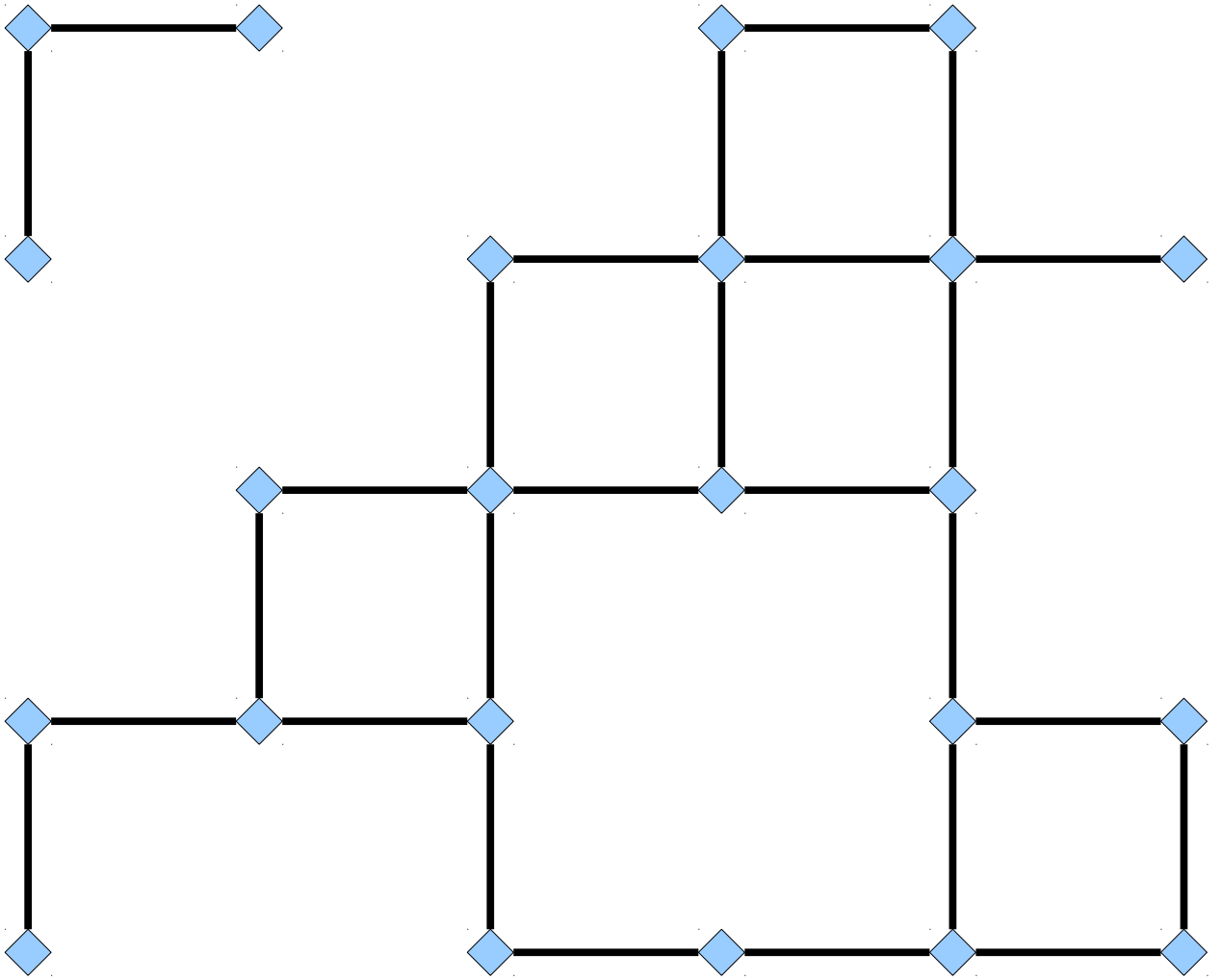
Solving Domino Tiling



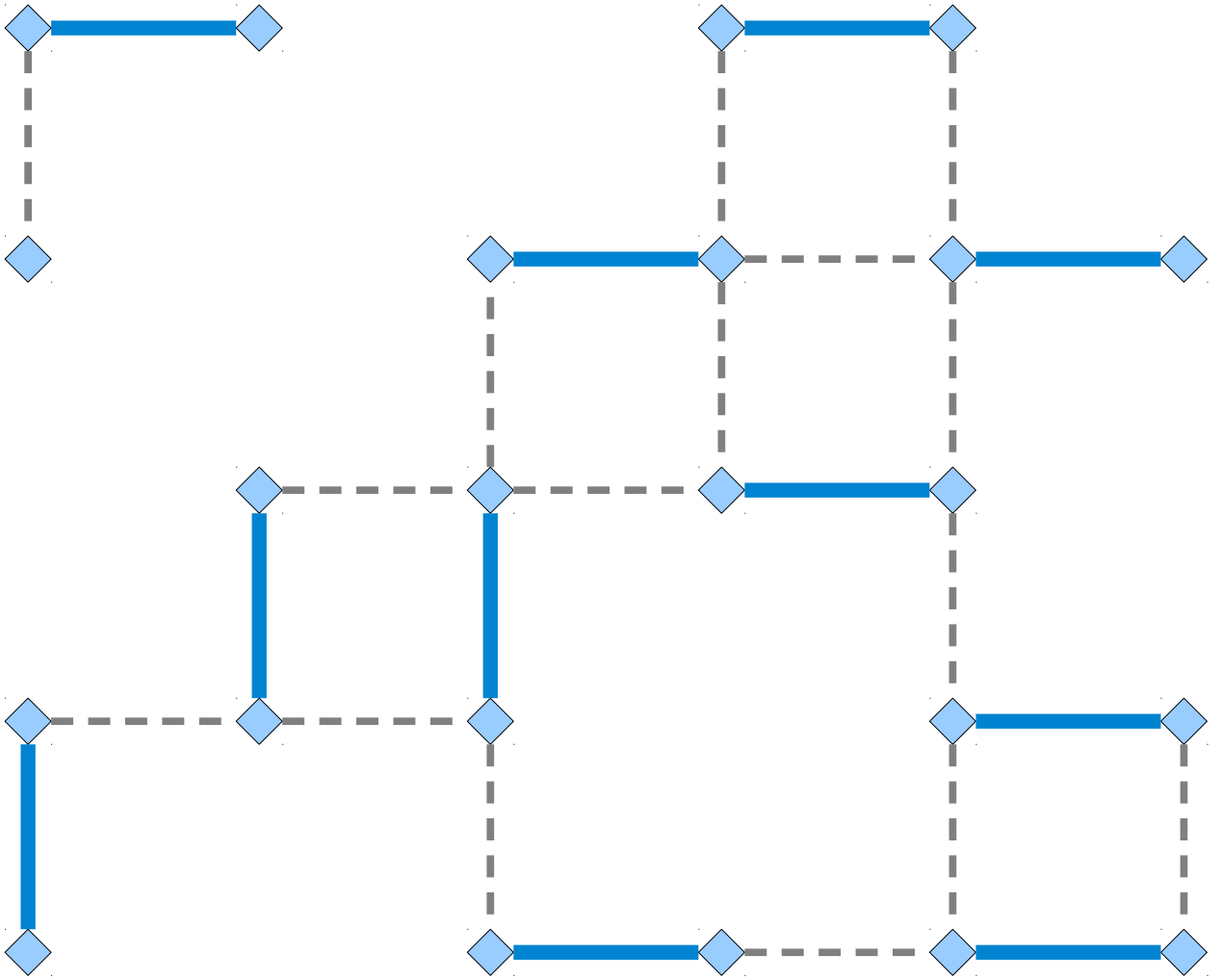
Solving Domino Tiling



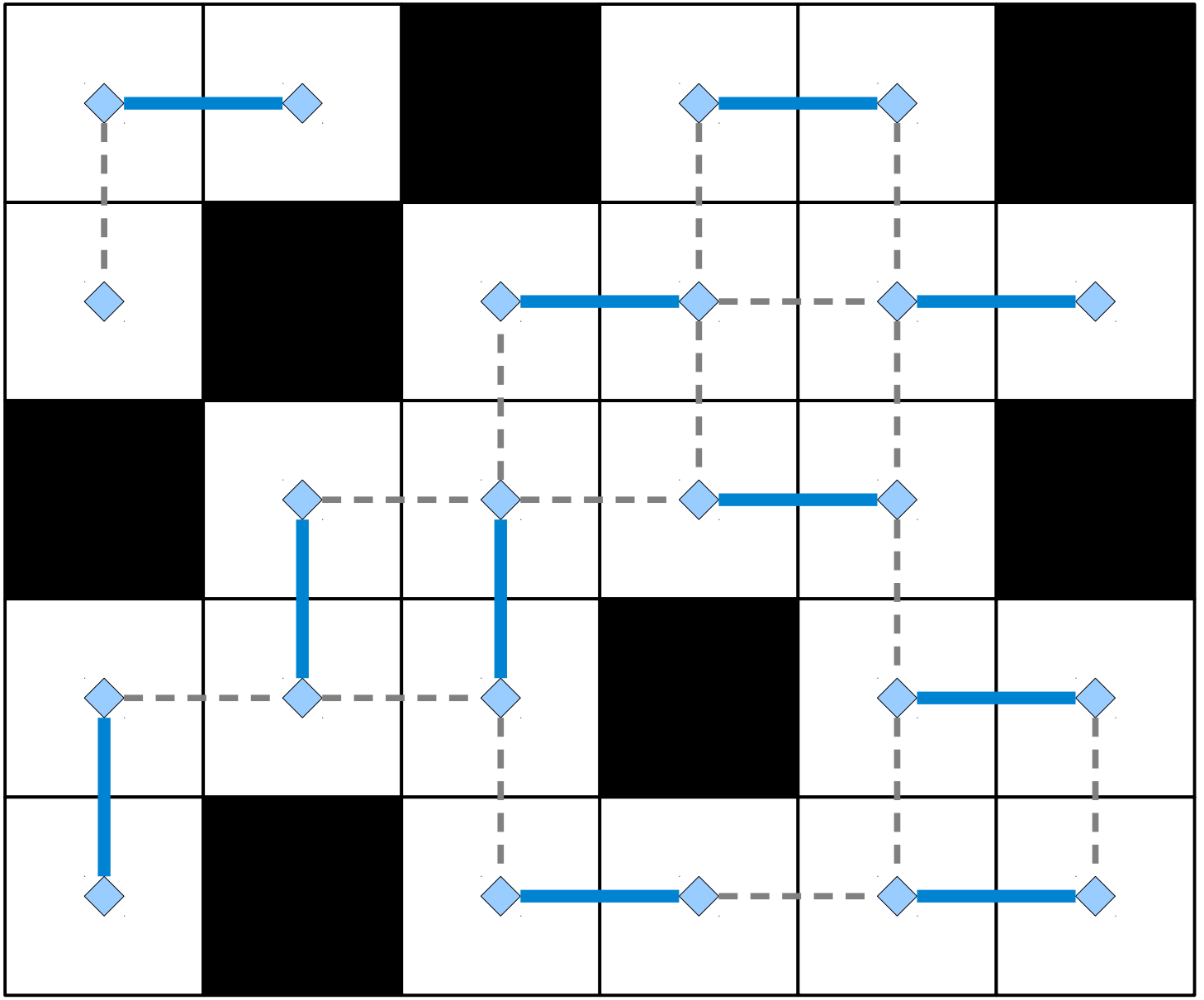
Solving Domino Tiling



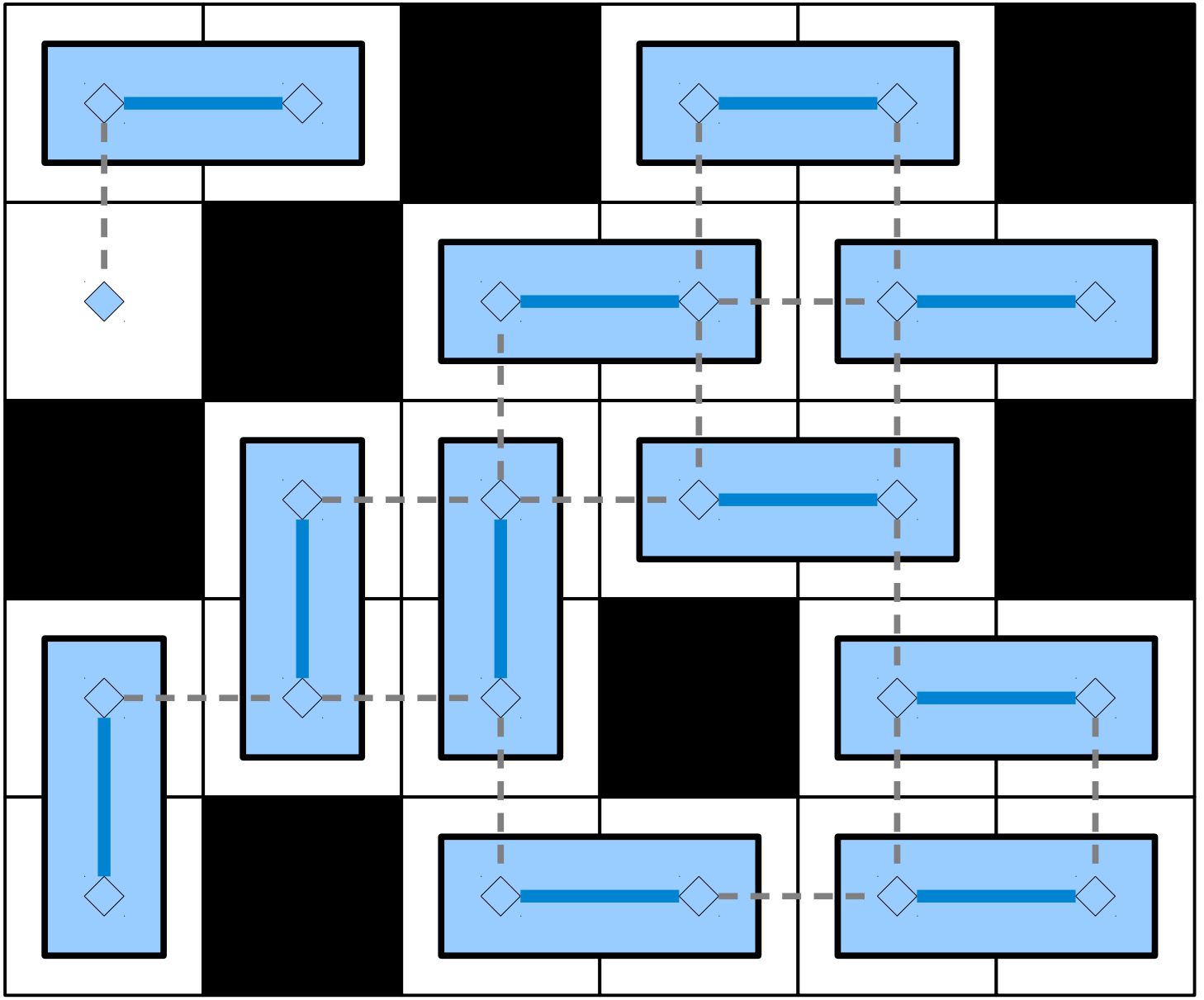
Solving Domino Tiling



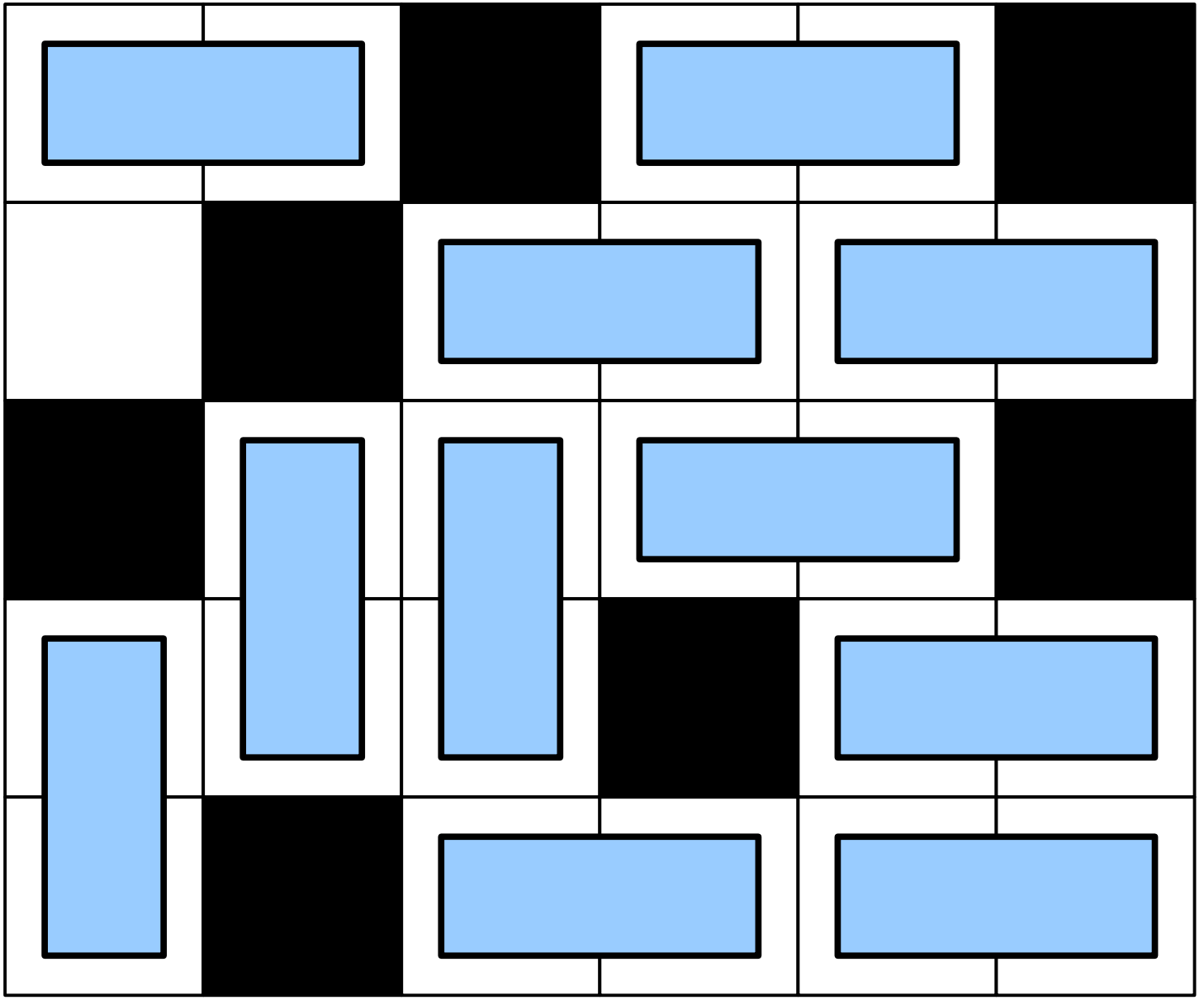
Solving Domino Tiling



Solving Domino Tiling



Solving Domino Tiling



In Pseudocode

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

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 *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

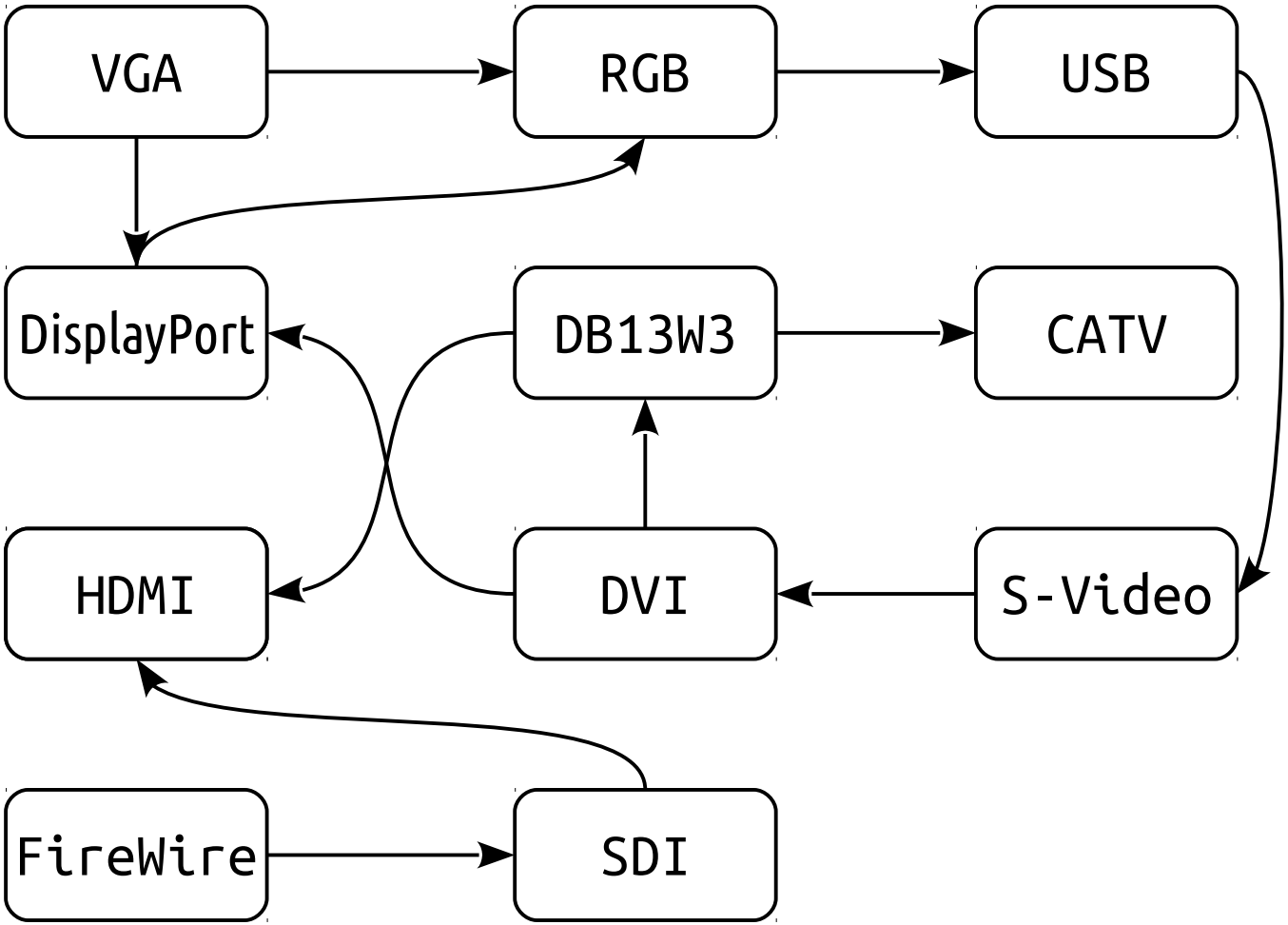
Connectors

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

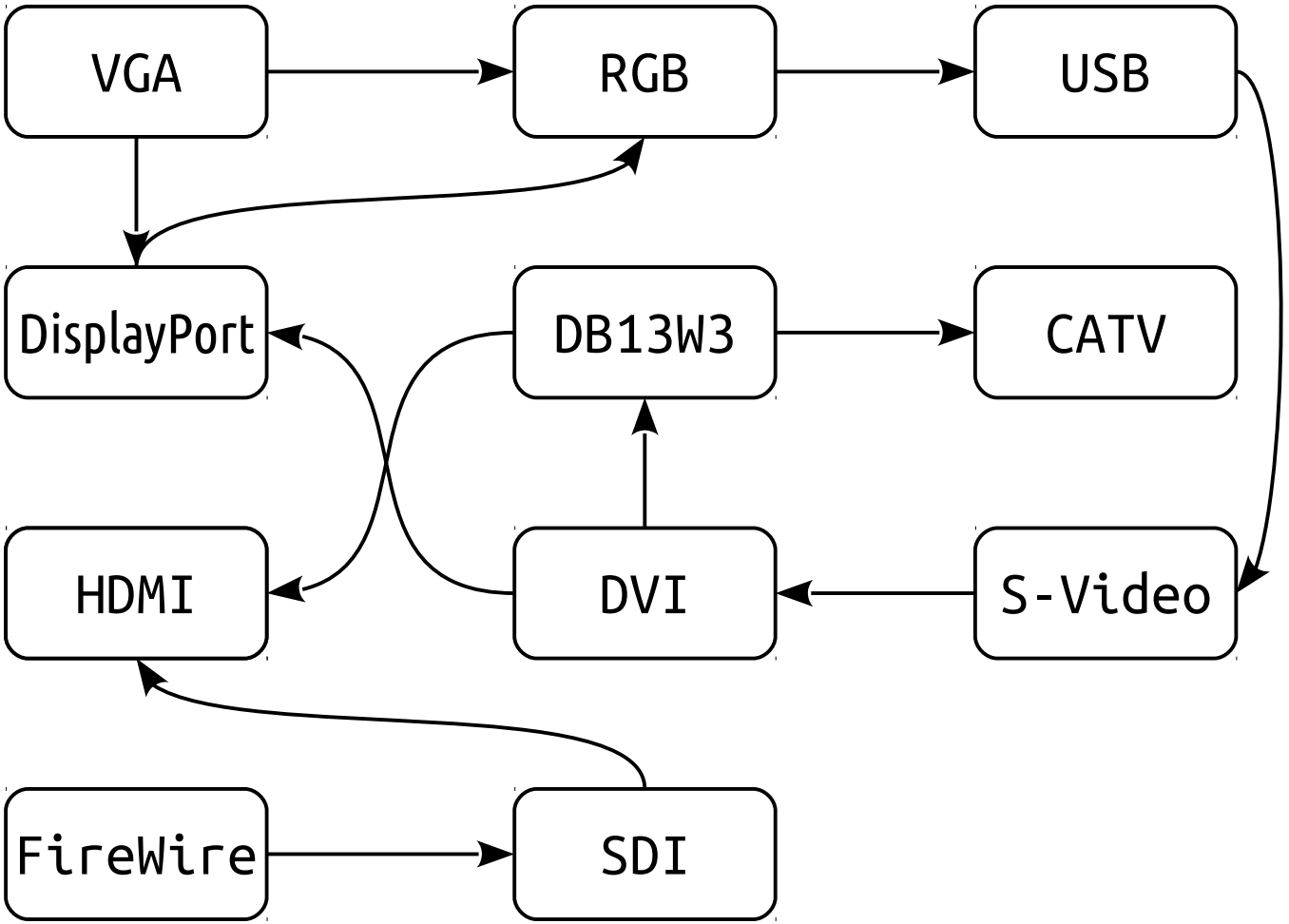
Converter Conundrums

Connectors

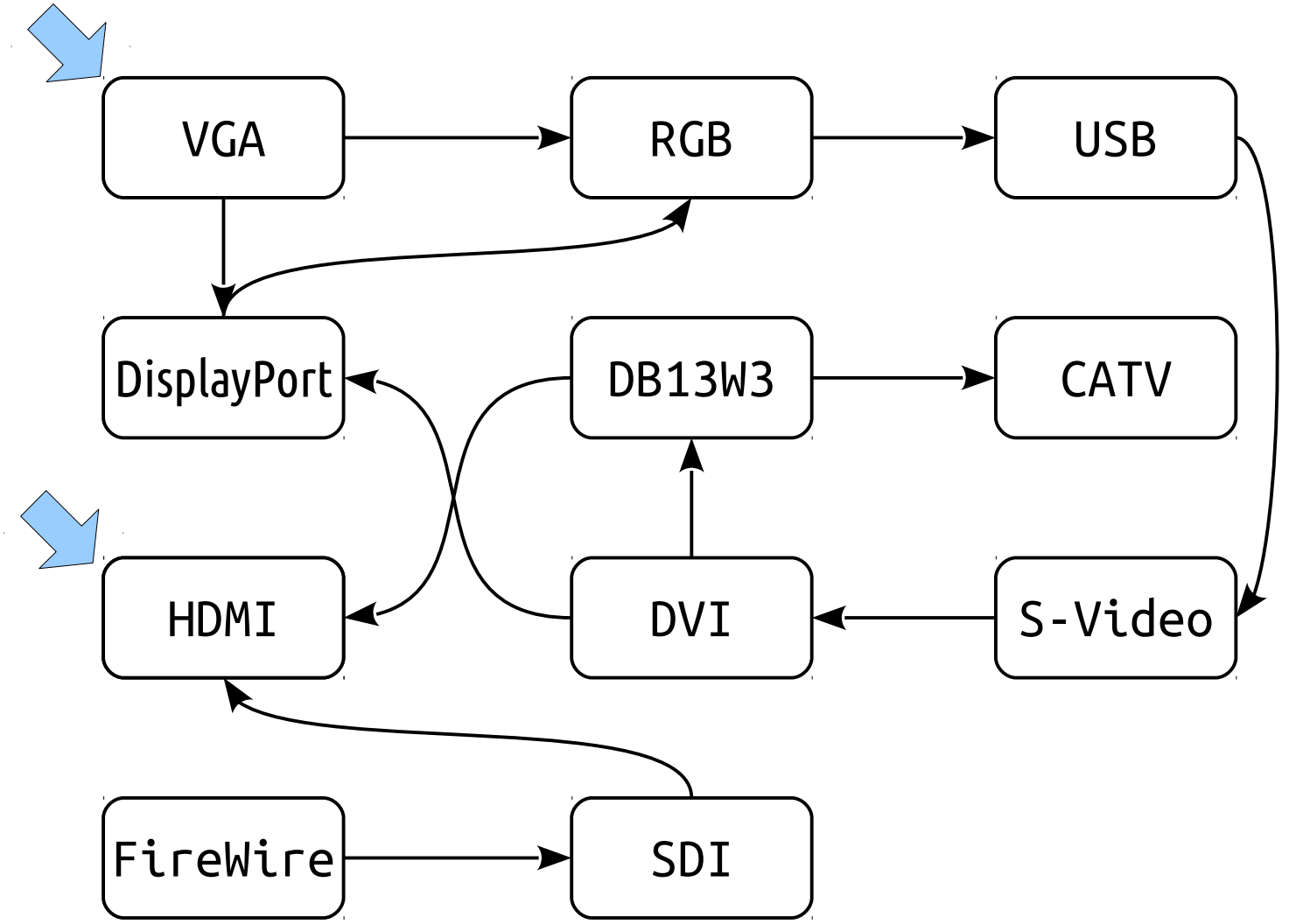
RGB to USB
VGA to DisplayPort
DB13W3 to CATV
DisplayPort to RGB
DB13W3 to HDMI
DVI to DB13W3
S-Video to DVI
FireWire to SDI
VGA to RGB
DVI to DisplayPort
USB to S-Video
SDI to HDMI



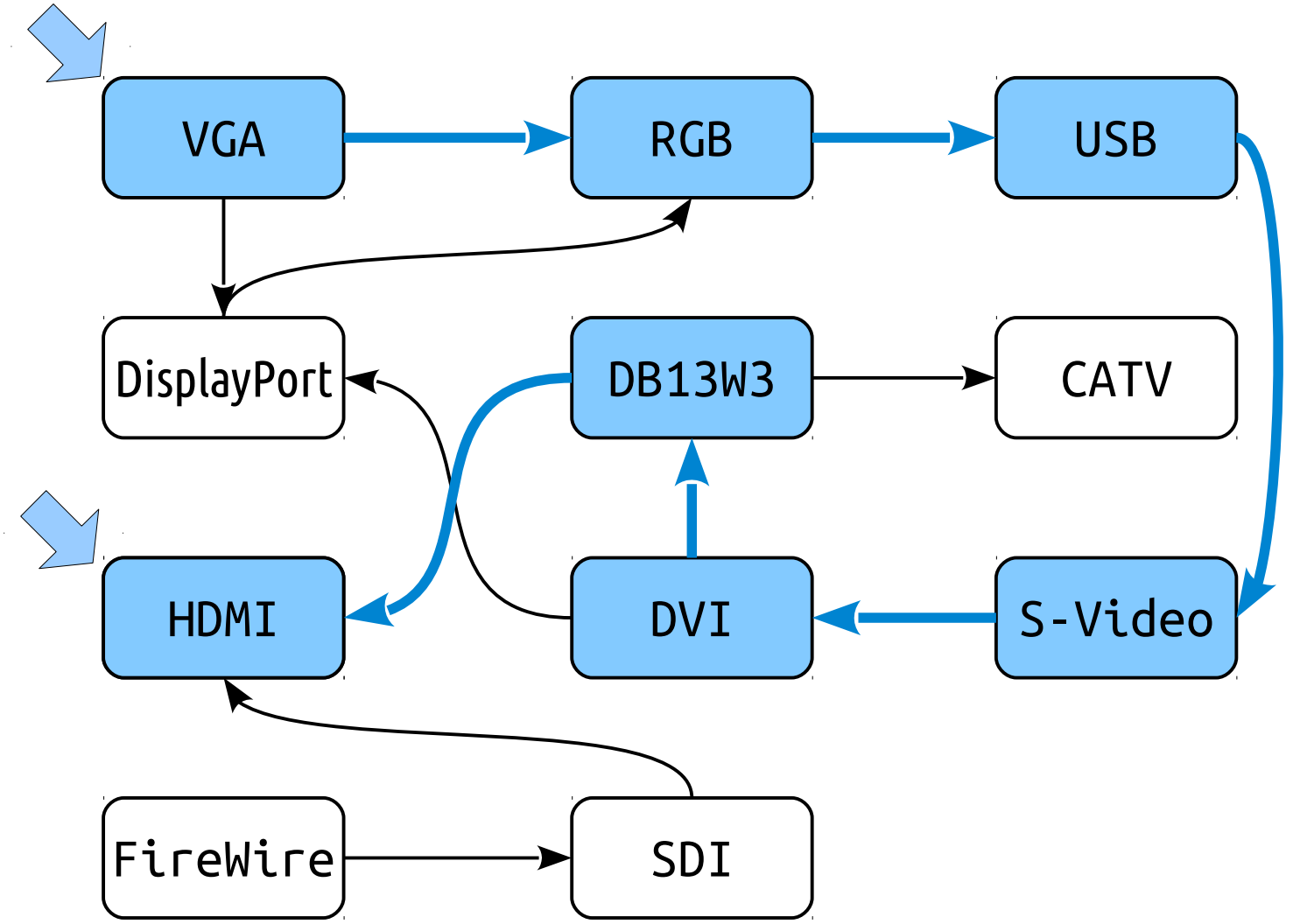
Converter Conundrums



Converter Conundrums



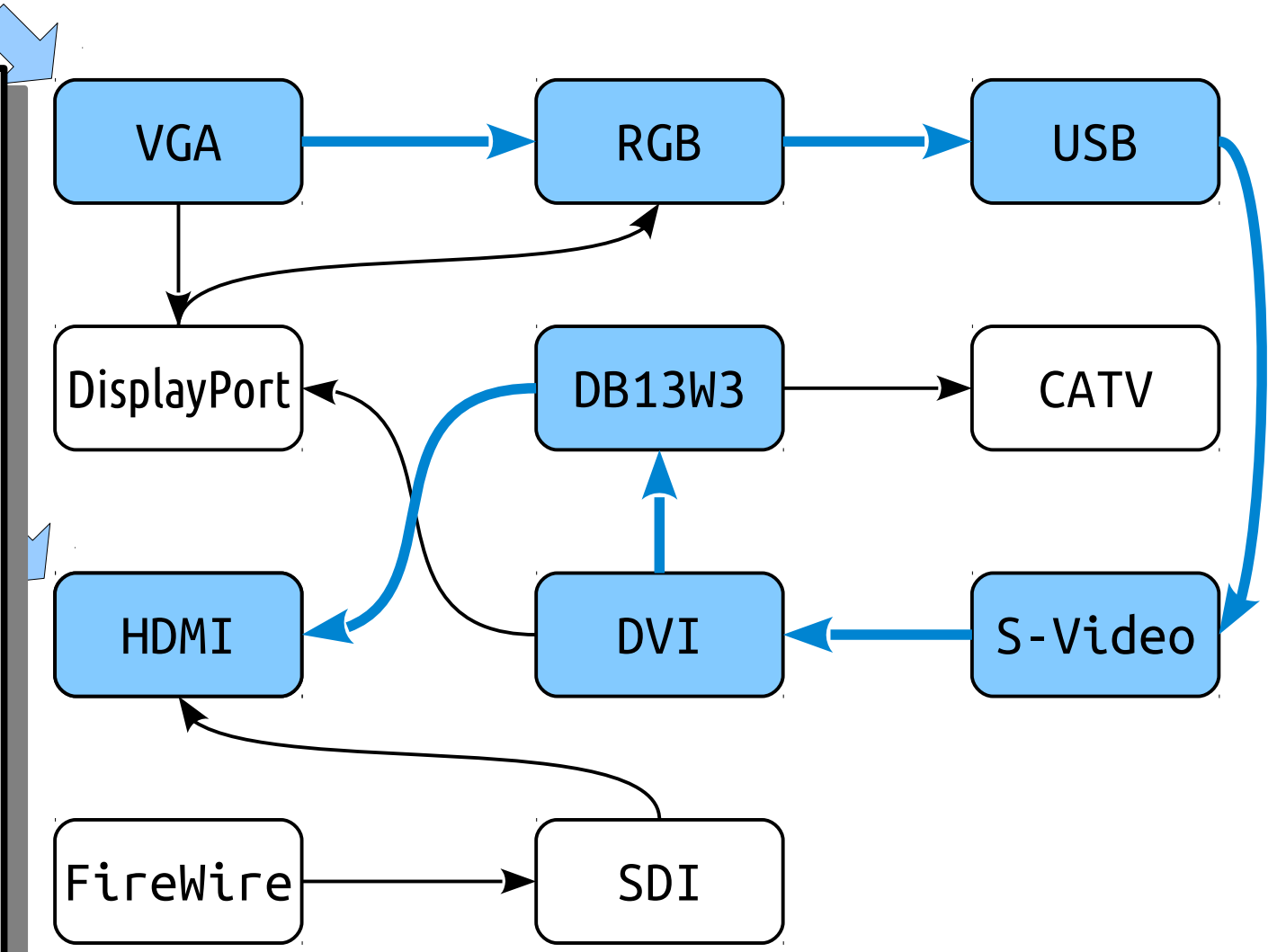
Converter Conundrums



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);
}
```

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

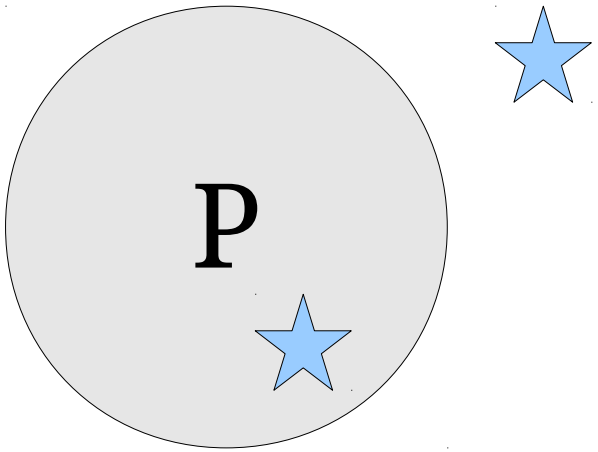
$$A \leq_{\mathrm{p}} B.$$

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

* Assuming that `transform` runs in polynomial time.

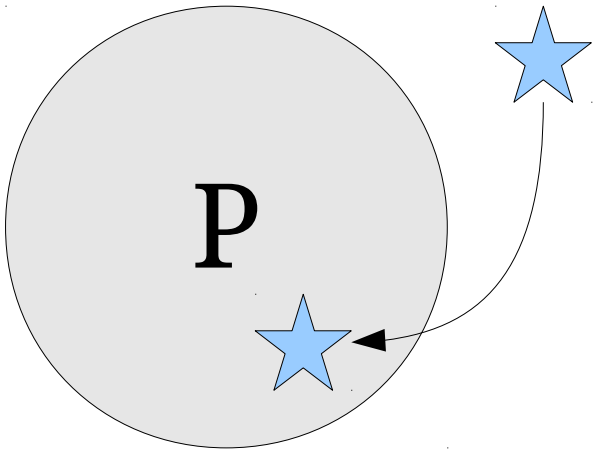
Polynomial-Time Reductions

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



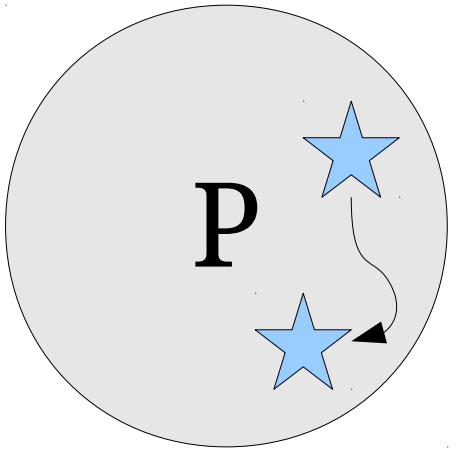
Polynomial-Time Reductions

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



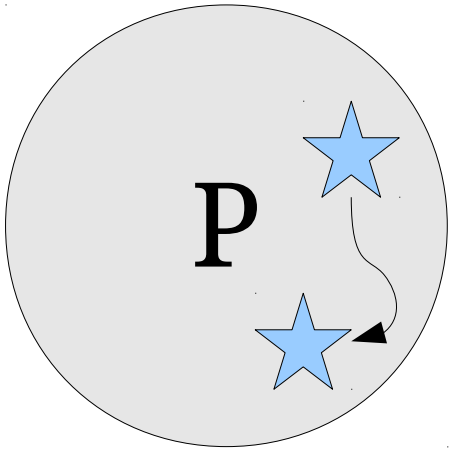
Polynomial-Time Reductions

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



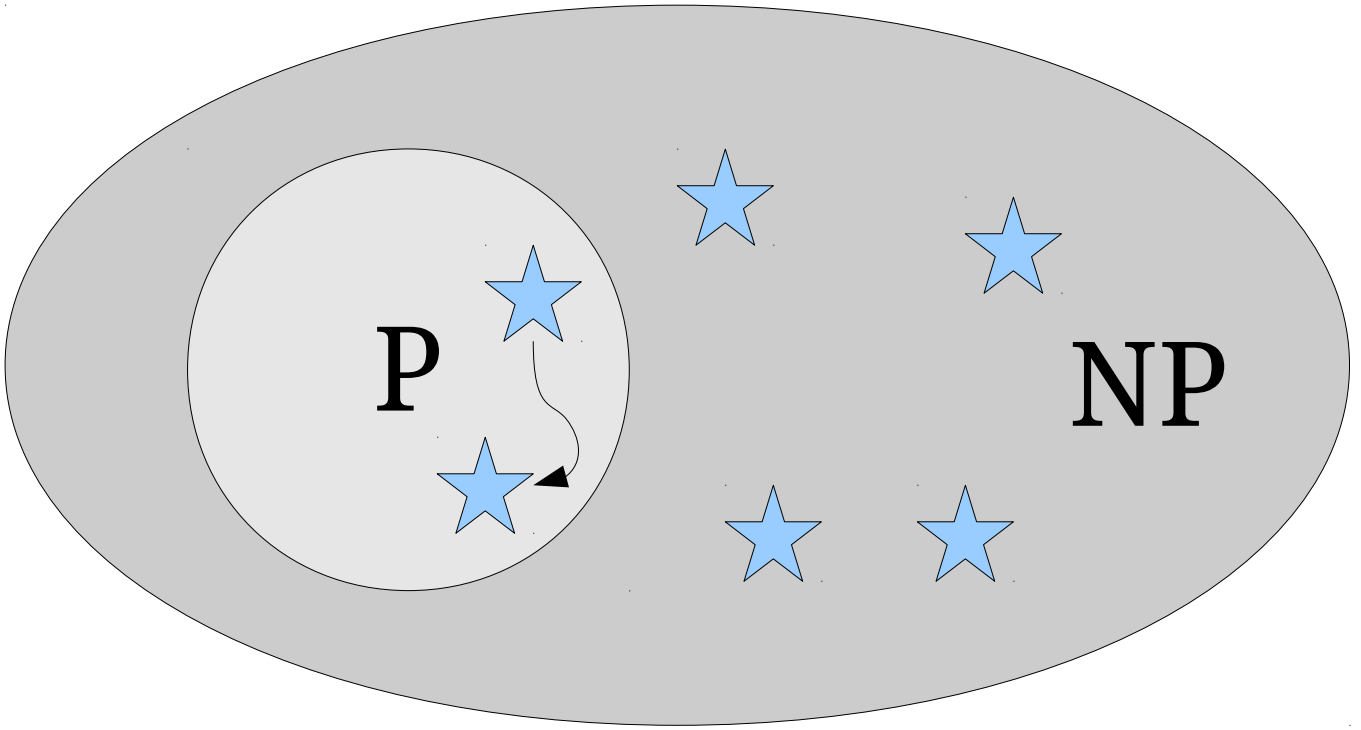
Polynomial-Time Reductions

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



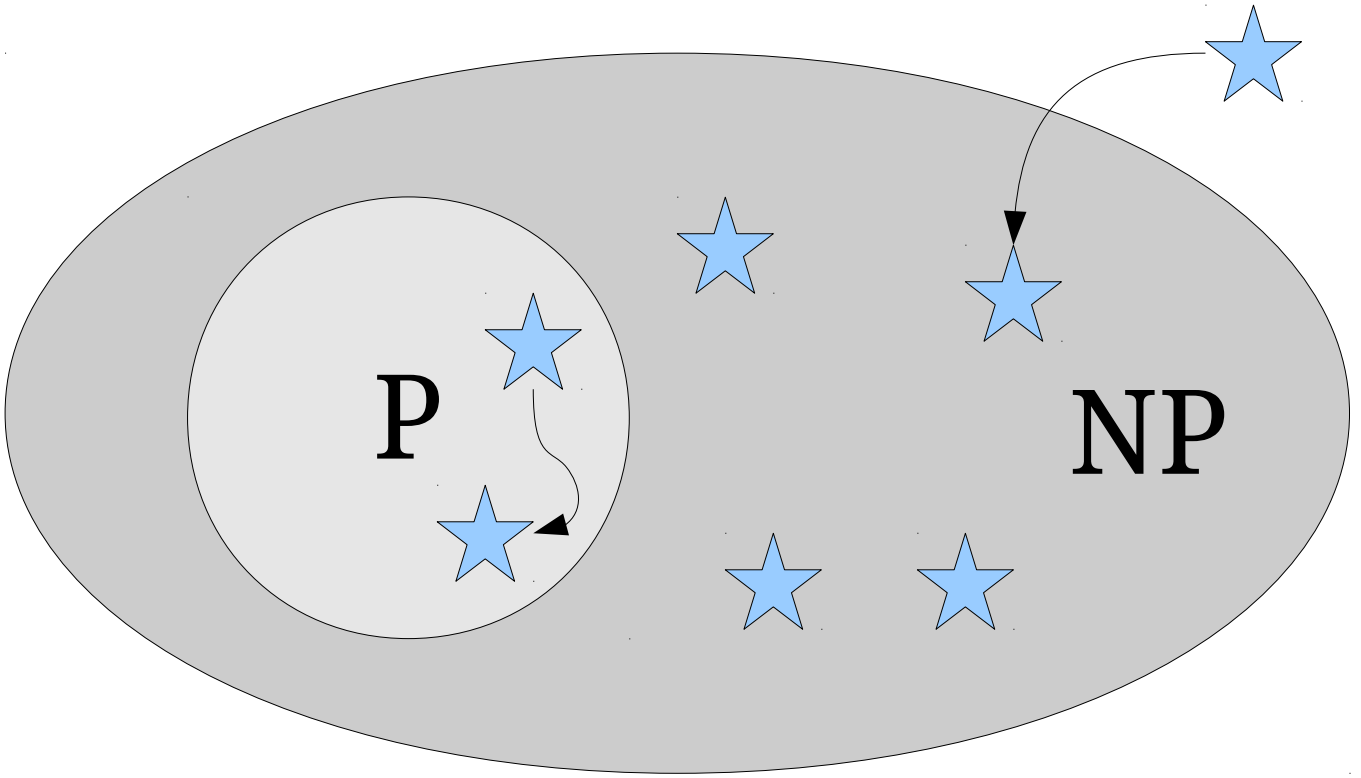
Polynomial-Time Reductions

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



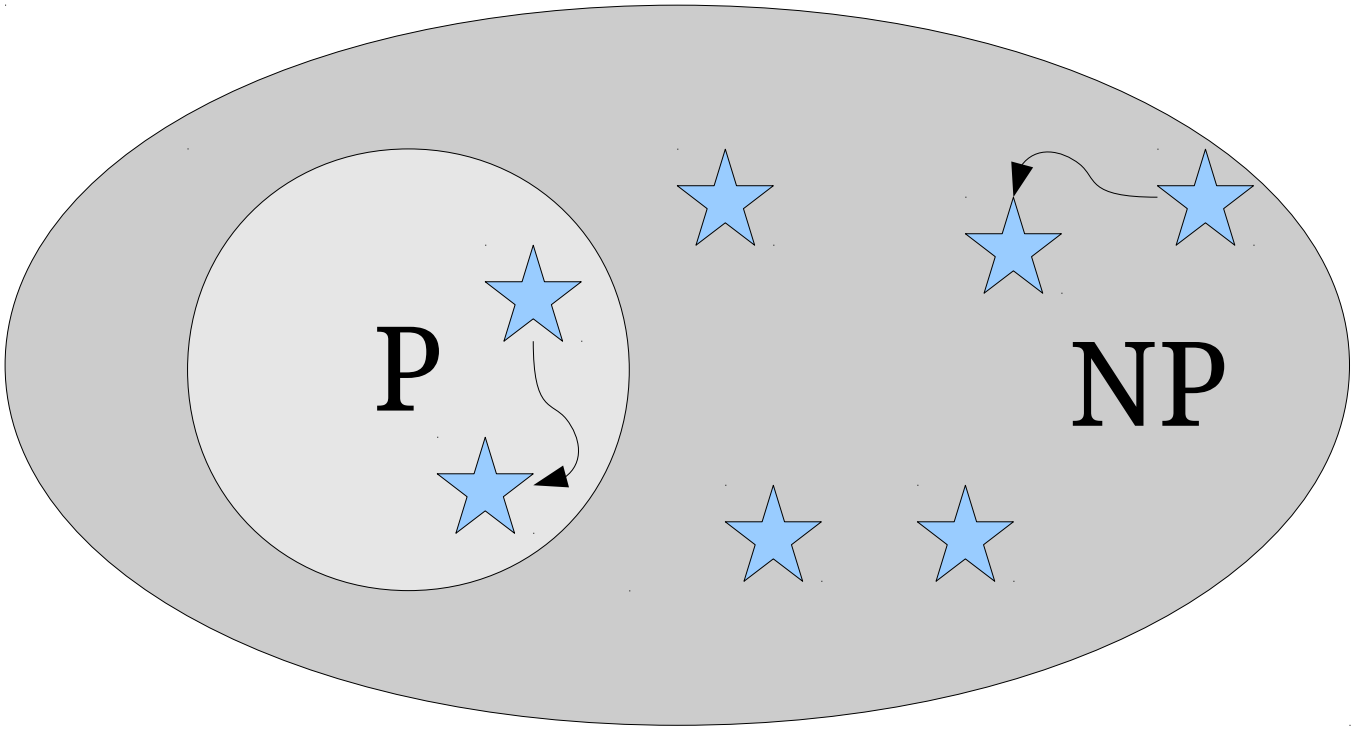
Polynomial-Time Reductions

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



Time-Out for Announcements!

Reminder: PS9

- As a friendly reminder, Problem Set 9 cannot be submitted late. The hard deadline, enforced by computers, is 3:00PM on Friday.
  - Please don't attract the Electronic Evil Eye by waiting until the last minute to submit!
- Please keep asking questions on Piazza and in office hours!

Practice Final Exam

- We will be holding a proctored practice final exam on Saturday at 2PM, tentatively in Hewlett 200.
- We'll be using “Practice Final Exam 3,” so you may want to hold off on using that one in the meantime.
- Everyone is welcome!

Your Questions

“How would materials from 103 be useful in the future? I haven't seen them in the 200 level classes I've taken, nor have I seen them in interview questions.”

“How would materials from 103 be useful in the future? I haven't seen them in the 200 level classes I've taken, nor have I seen them in interview questions.”

“How would materials from 103 be useful in the future? I haven't seen them in the 200 level classes I've taken, nor have I seen them in interview questions.”

CS221 talks about A* search, and it used to ask you to formally prove correctness using a proof by contradiction / induction. It also talks about propositional logic, first-order logic, and automated theorem proving.

CS224W uses a ton of graph theory, and many of the problems asked there can be solved purely with CS103 and CS109 material.

CS227B uses a modified version of FOL to represent and encode games, and uses propositional formulas to simulate them.

CS240 talks about different algorithms for scheduling processes on a multiprocessor system, which runs into concerns from today's lecture!

CS243 talks about program optimization, where basically every problem is either undecidable or computationally intractable.

CS248 explores triangle meshes, which are essentially 3D representations of planar graphs.

CS255 uses hard problems to build cryptographic systems and does a ton with sets of functions and first-order definitions.

CS261 spends most of its time talking about how to handle hard problems.

CS274 explores algorithms in evolutionary biology, which runs into topics from today's lecture!

CS295 uses results about undecidability to classify different approaches to automated testing and verification.

“How would materials from 103 be useful in the future? I haven't seen them in the 200 level classes I've taken, nor have I seen them in interview questions.”

“How would materials from 103 be useful in the future? I haven't seen them in the 200 level classes I've taken, nor have I seen them in interview questions.”

Hey Keith!

I hope you're doing well! I just wanted to let you know something cool I was thinking about. I was in an interview with Salesforce yesterday, and they wanted me to write a program to identify all the words in a file that rhymed.

I started thinking about it and realized that rhyming was just an equivalence relation - so I could basically just build a dictionary where each key was one element of an equivalence class and the values were members of that class. It was really awesome to see this in real life!

Thanks again,

[*signature*]

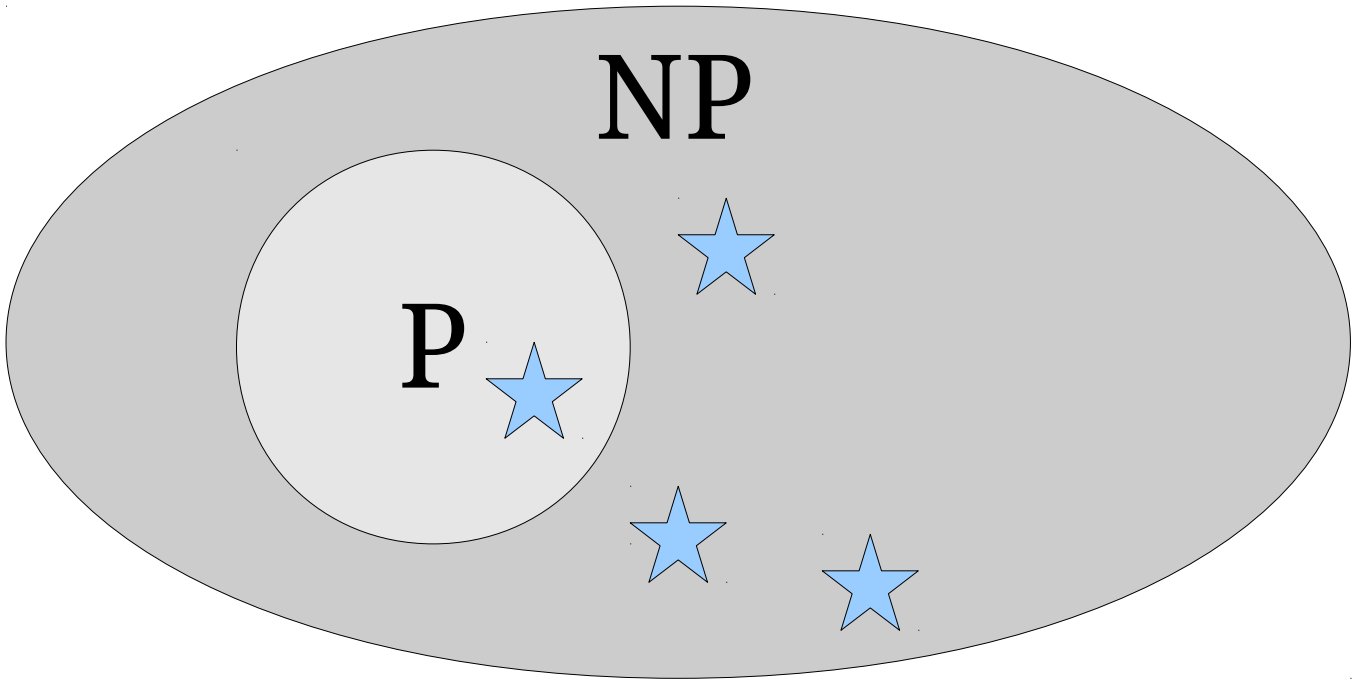
Back to CS103!

Using Reducibility

NP-Hardness

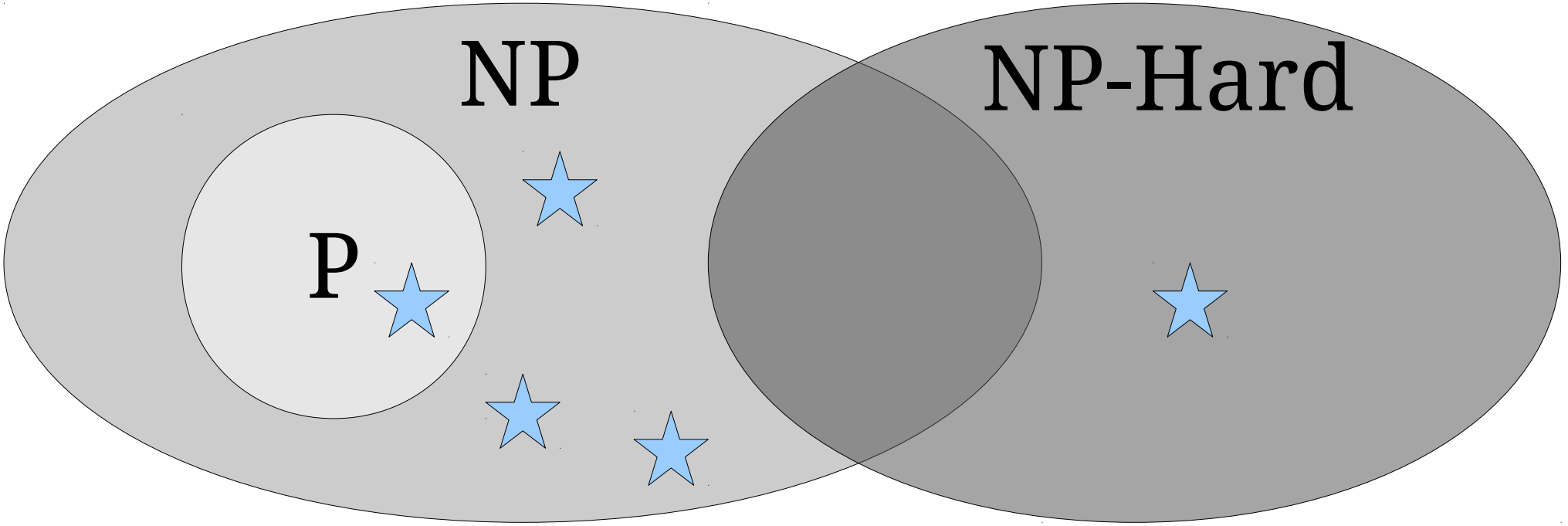
NP-Hardness

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



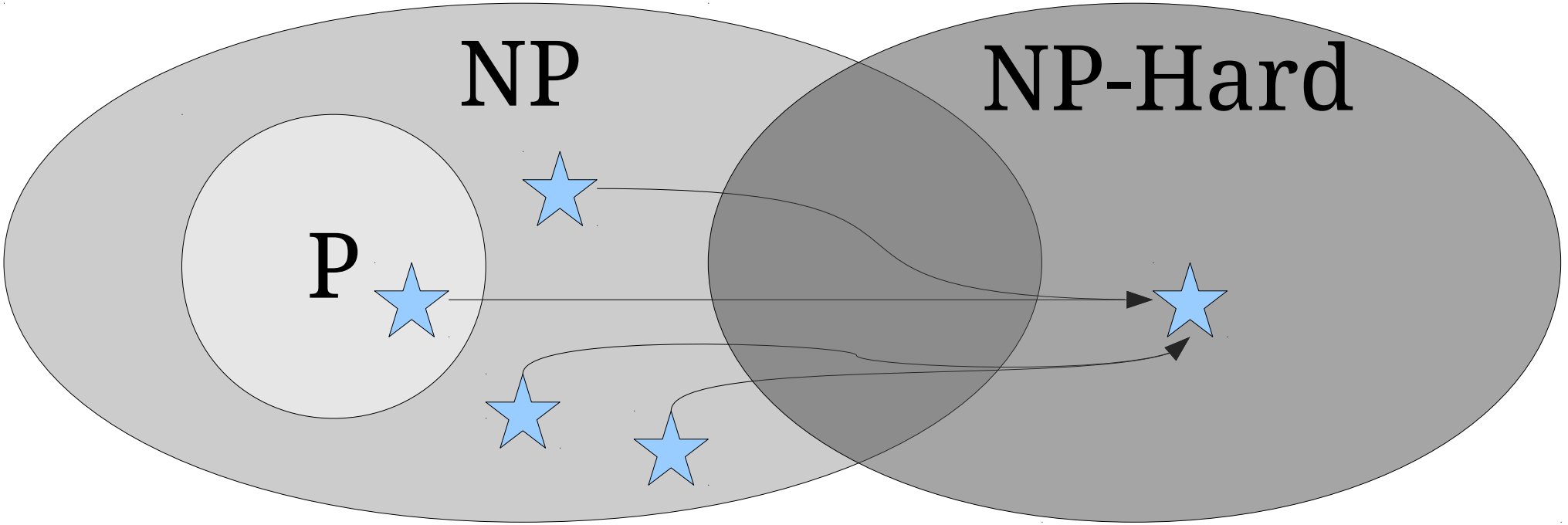
NP-Hardness

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



NP-Hardness

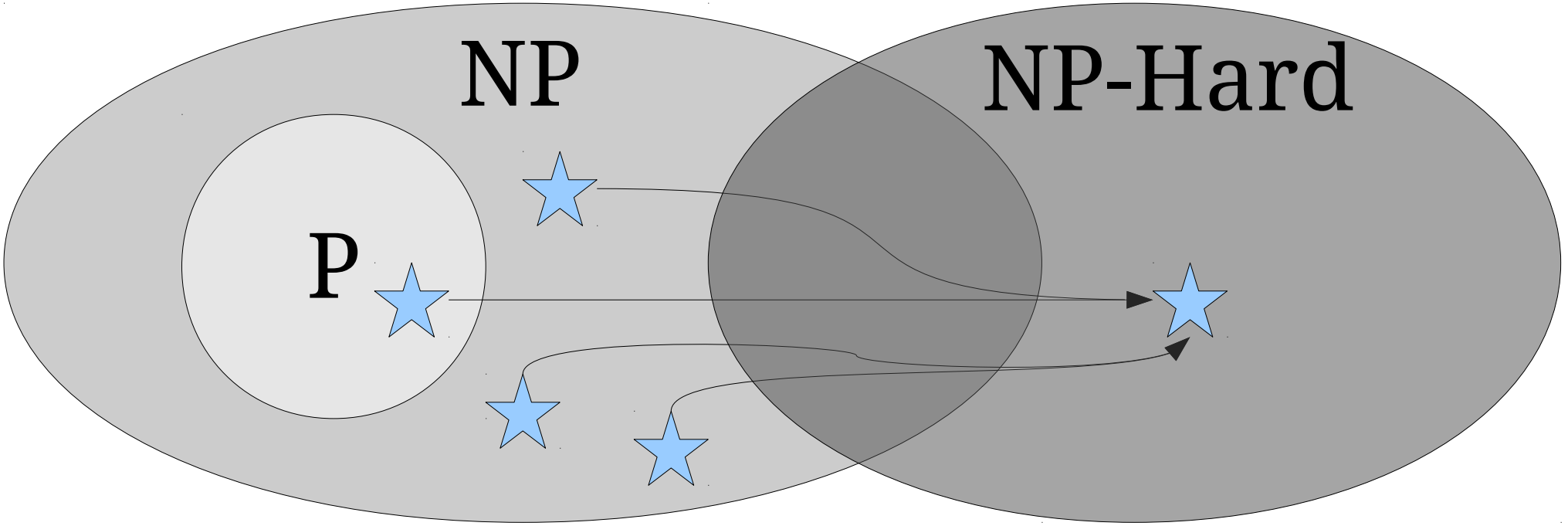
- A language L is called ***NP-hard*** if for *every* $A \in \mathbf{NP}$, we have $A \leq_p L$.



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 **NP**, since an algorithm for L can be used to decide all problems in **NP**.

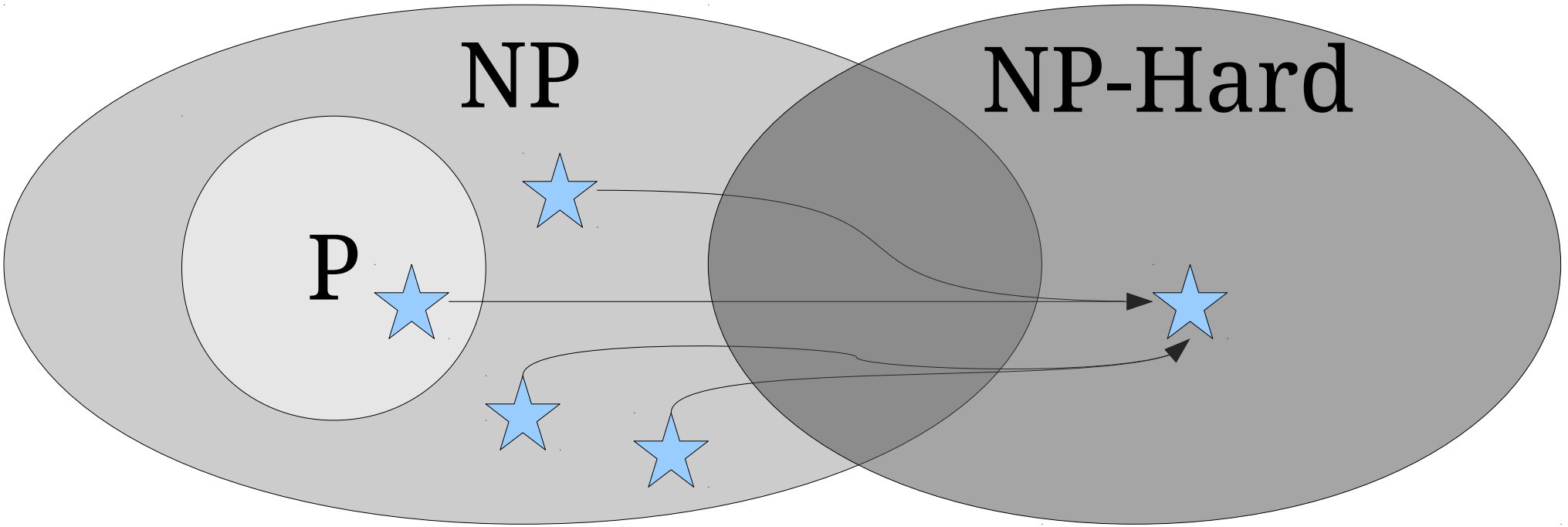


NP-Hardness

- Unlike the other classes of languages we've seen in this class (regular languages, decidable languages, etc.), **NP**-hardness is a *lower bound* on difficulty.
- If we say something is **NP**-hard, intuitively speaking, it is at least as hard as all the problems in **NP**, *and possibly a lot harder!*
- ***Fact we'll prove in a minute:*** if **P** ≠ **NP**, then *no* **NP**-hard problem can be solved in polynomial time!
- ***Useful fact for building an intuition:*** The languages A_{TM} and L_D are **NP**-hard, but they're not in **NP** because they're not even decidable!

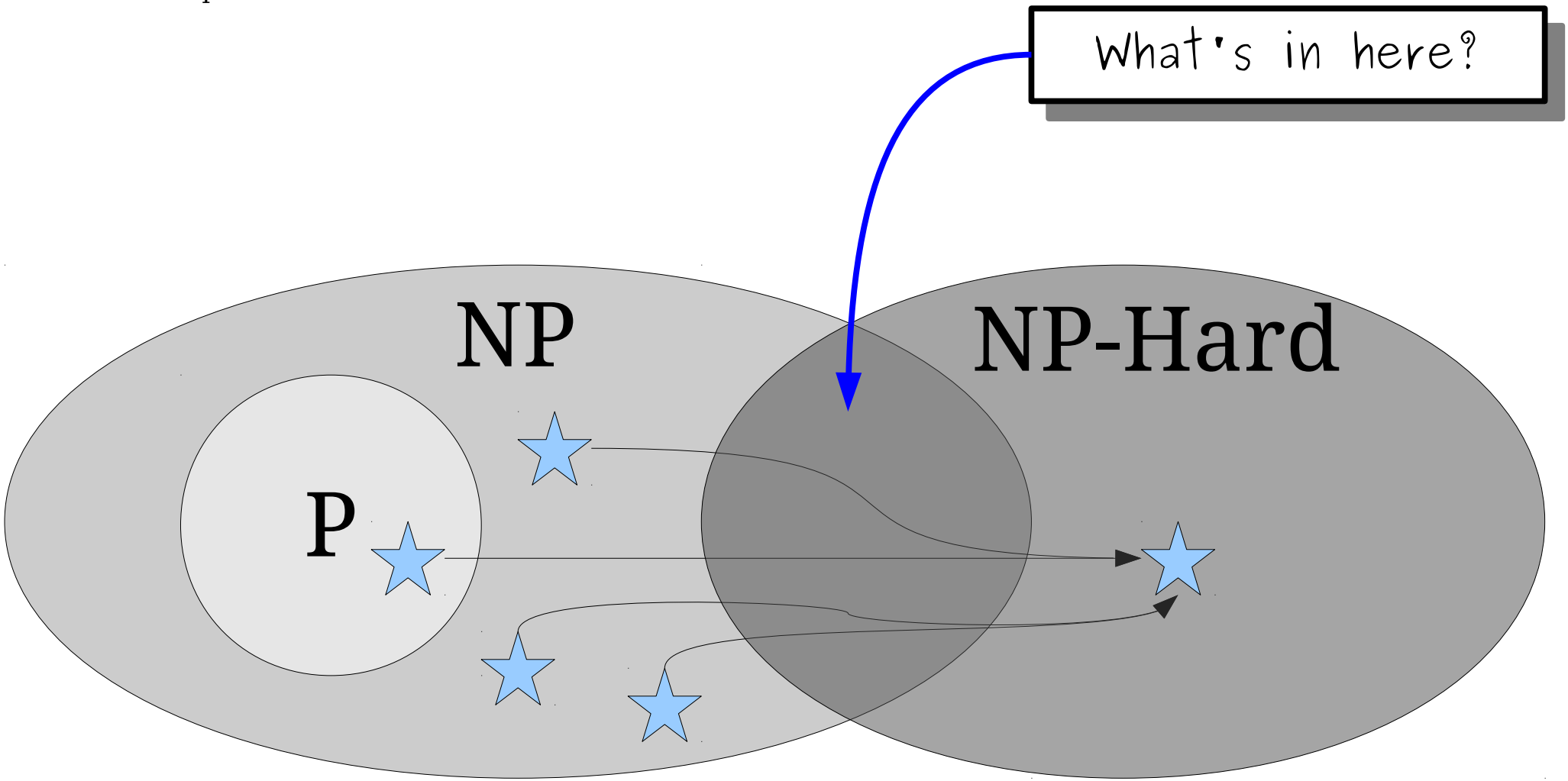
NP-Completeness

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



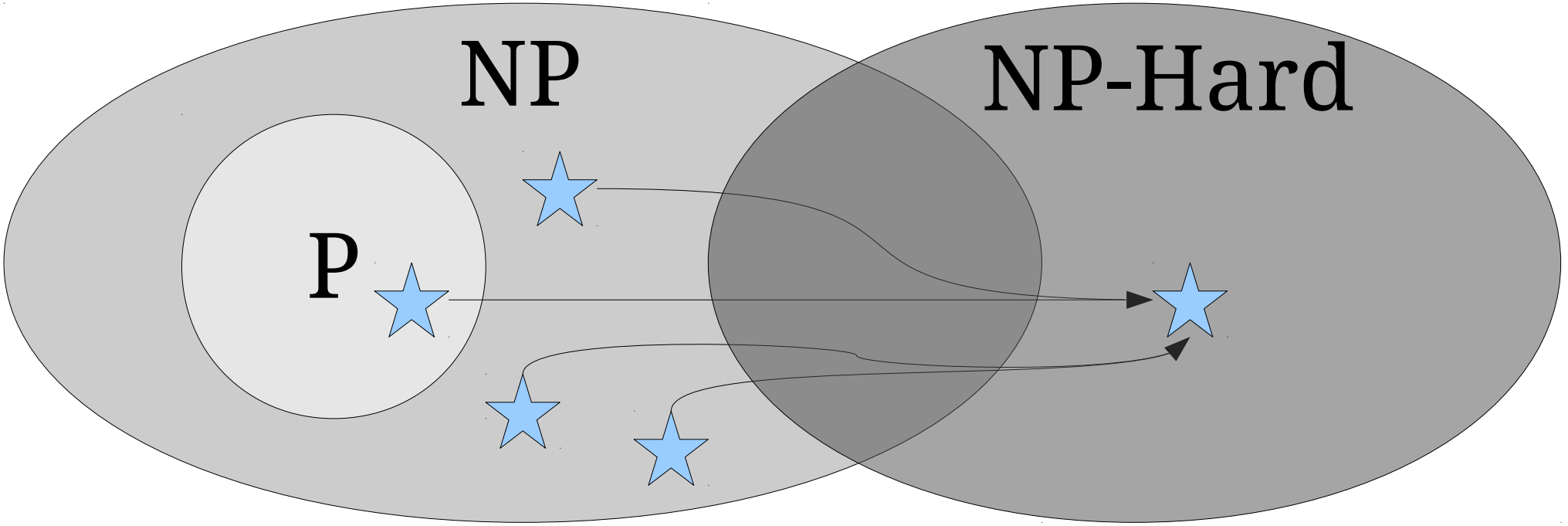
NP-Completeness

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



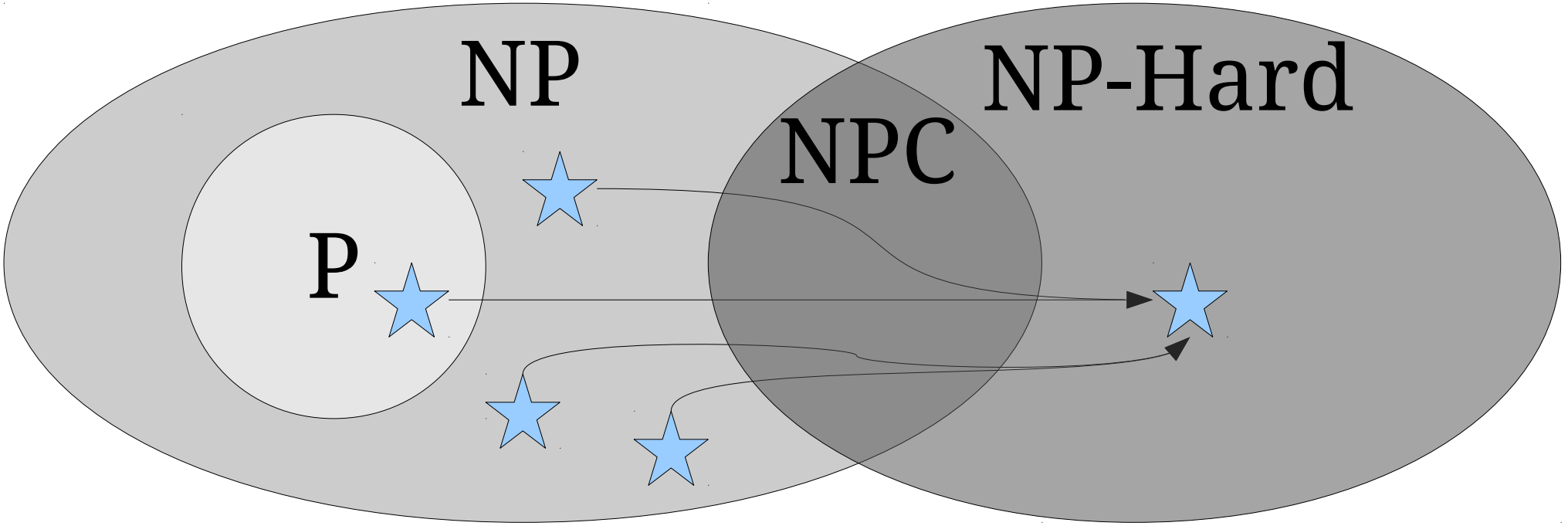
NP-Completeness

- 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-Completeness

- 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-Completeness

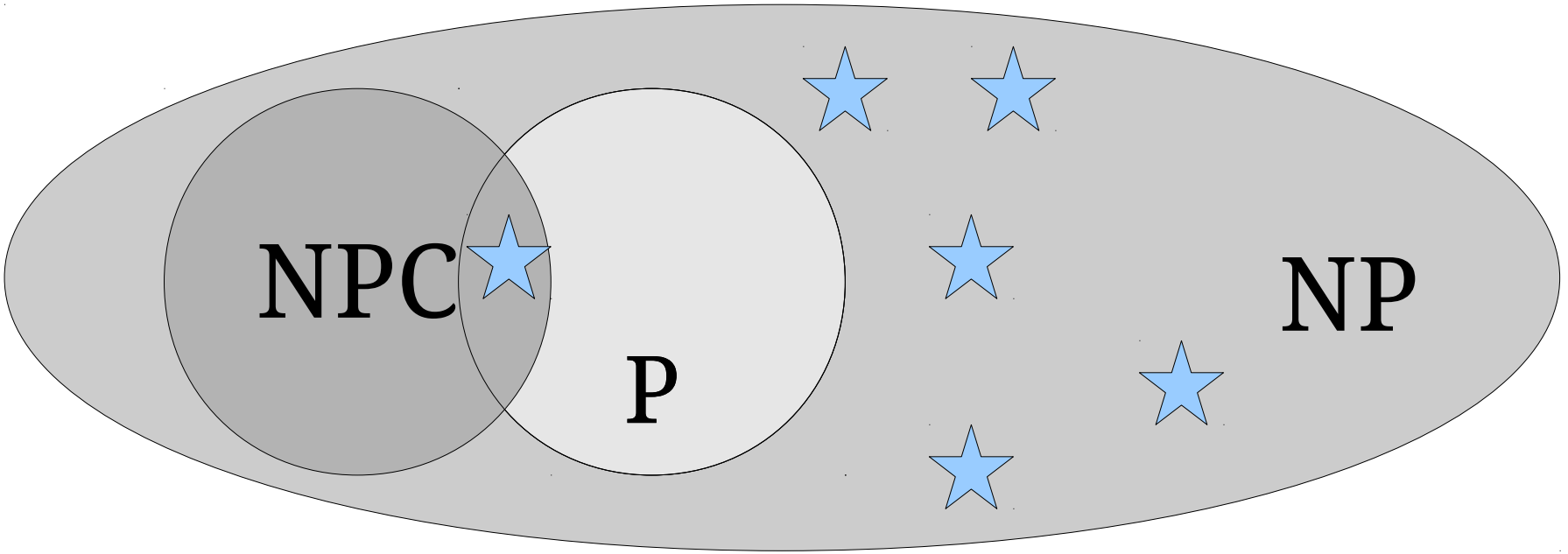
- If a language L is **NP**-complete, then, intuitively:
  - ***It is no harder than the hardest problems in NP***. This is because all **NP**-complete problems are themselves in **NP**, so they can't be “too hard” to be in **NP**.
  - ***It is no easier than the hardest problems in NP***. This is because it's **NP**-hard, meaning it's “at least as hard” as any problem in **NP**.
- Accordingly, the **NP**-complete problems are often described as “the hardest problems in **NP**,” and that's a great mental model to have.

The Tantalizing Truth

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

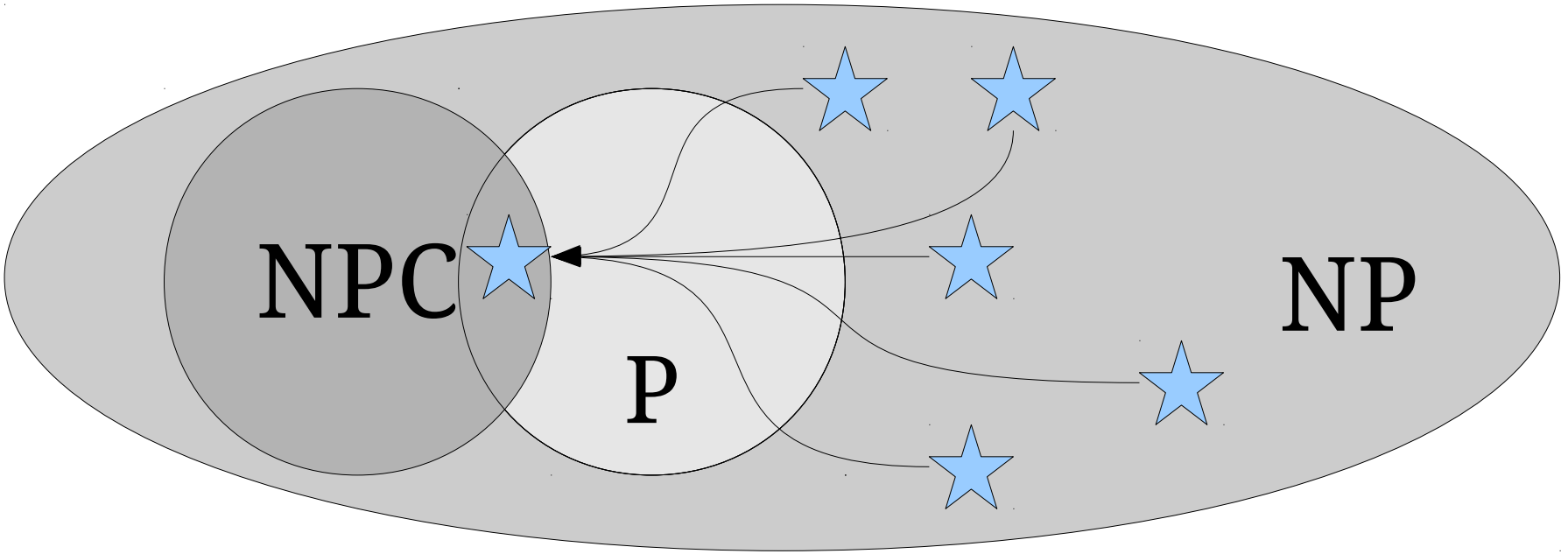
The Tantalizing Truth

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



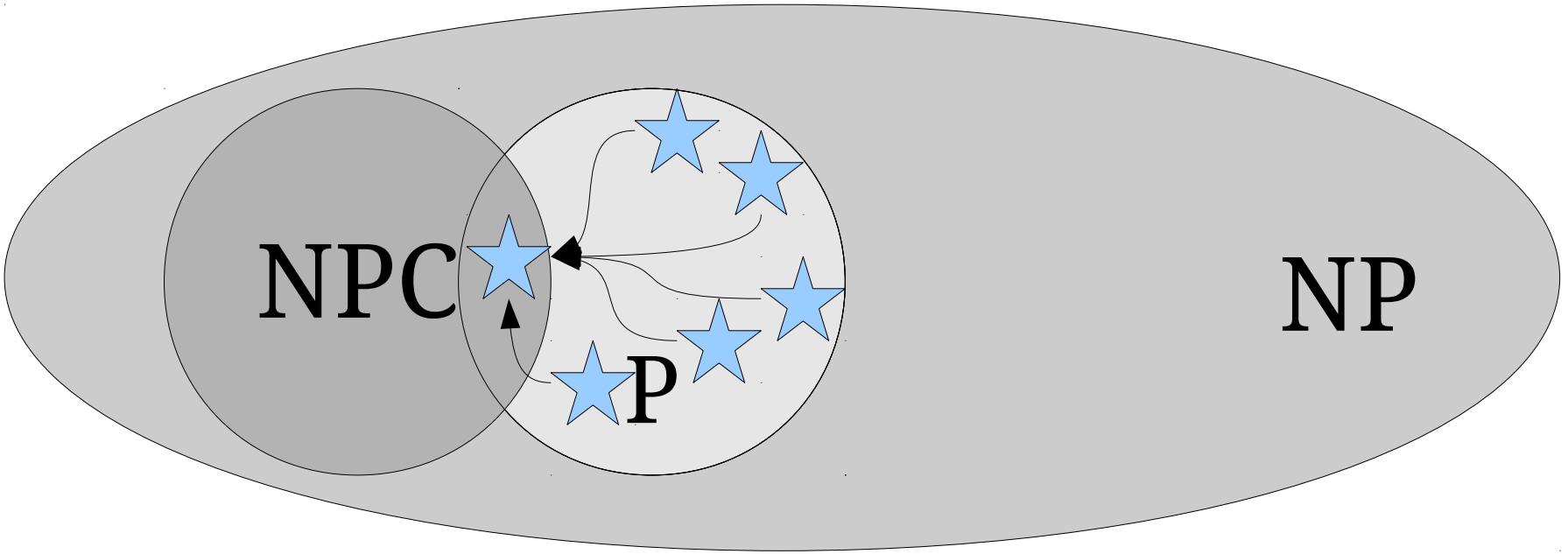
The Tantalizing Truth

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



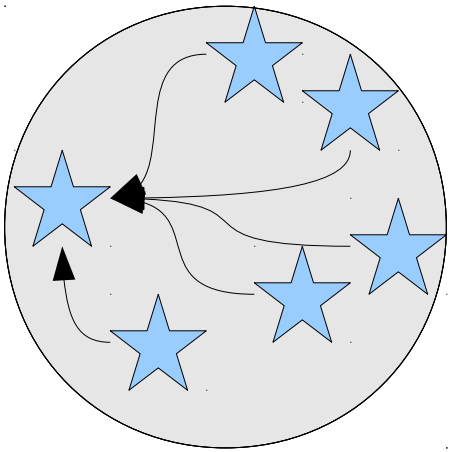
The Tantalizing Truth

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**.

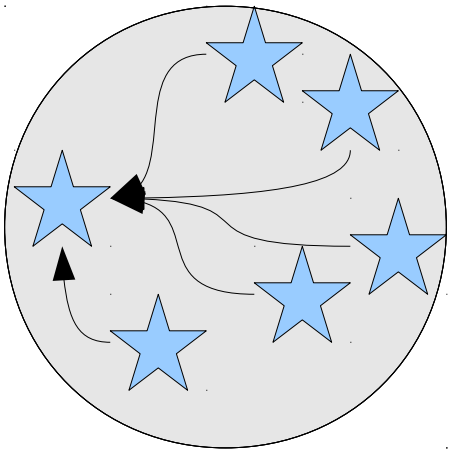


P = NP

The Tantalizing Truth

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

Proof: Suppose that L is **NP**-complete and $L \in \mathbf{P}$. 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**. ■

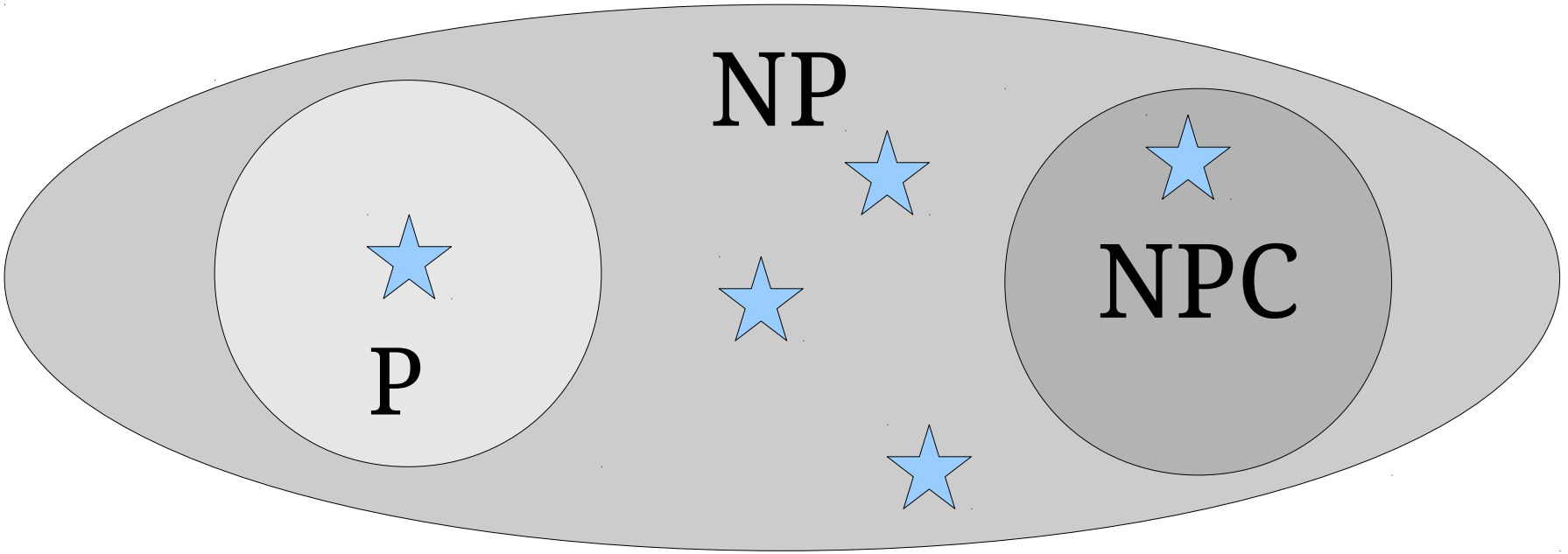


P = NP

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.
  - $p \wedge q$ 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 }**

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

Proof Idea: Given a polynomial-time verifier V for an **NP** language L, for any string w, you can write a gigantic formula $\varphi(w)$ that says "there is a certificate c such that V accepts $\langle w, c \rangle$." This formula is satisfiable iff there is a c where V accepts $\langle w, c \rangle$, which in turn happens precisely if $w \in L$. ■

Proof: Read Sipser or 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 $\mathbf{P} \neq \mathbf{NP}$.
- We've turned a huge, abstract, theoretical problem about solving problems versus checking solutions into the concrete task of seeing how hard one problem is.
- You can get a sense for how little we know about algorithms and computation given that we can't yet answer this question!

Why All This Matters

- You will almost certainly encounter **NP**-hard problems in practice – they're everywhere!
- If a problem is **NP**-hard, then there is no known algorithm for that problem that
  - is efficient on all inputs,
  - always gives back the right answer, and
  - runs deterministically.
- ***Useful intuition:*** If you need to solve an **NP**-hard problem, you will either need to settle for an approximate answer, an answer that's likely but not necessarily right, or have to work on really small inputs.

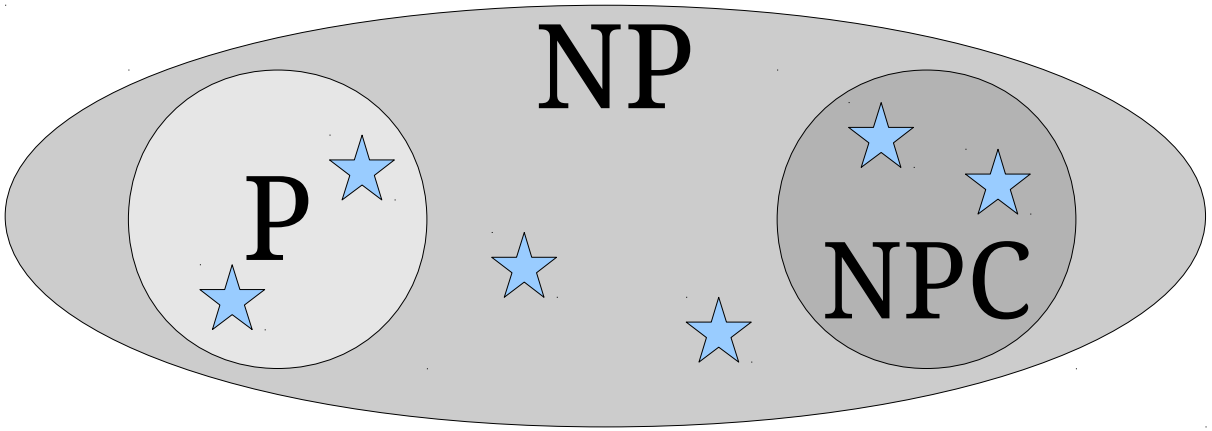
Sample **NP**-Hard Problems

- ***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)*

Coda: What if **P ≠ NP**?

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 **P = NP**?

Practically Speaking

Next Time

- ***The Big Picture***
- ***Where to Go from Here***
- ***A Final "Your Questions"***
- ***Parting Words!***