Complexity Theory

Part Two

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.
  - $w \in L \quad \text{iff} \quad \exists c \in \Sigma^*.\ V$ 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:

  NP = { *L* | 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!

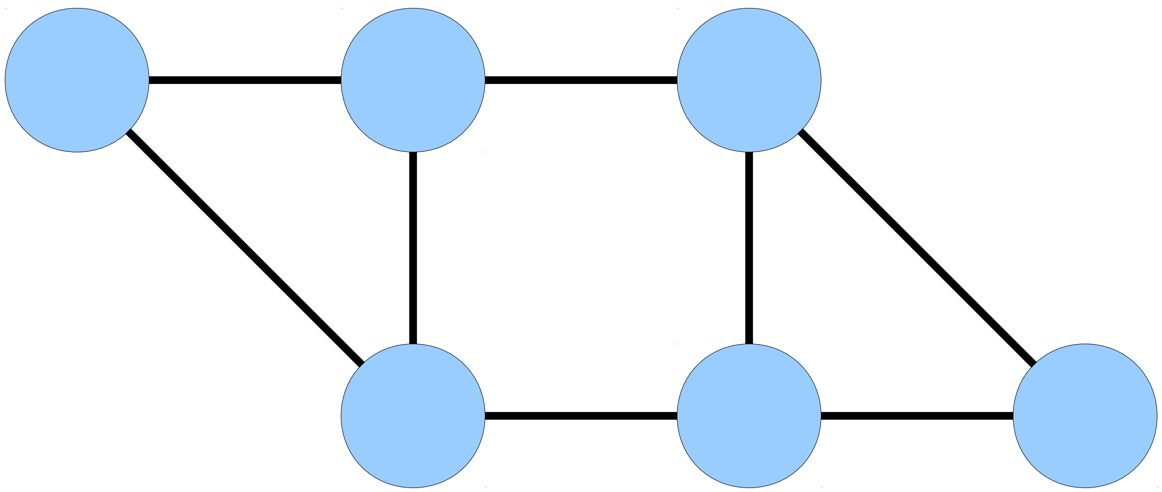
So how *are* we going to reason about **P** and **NP**?

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.

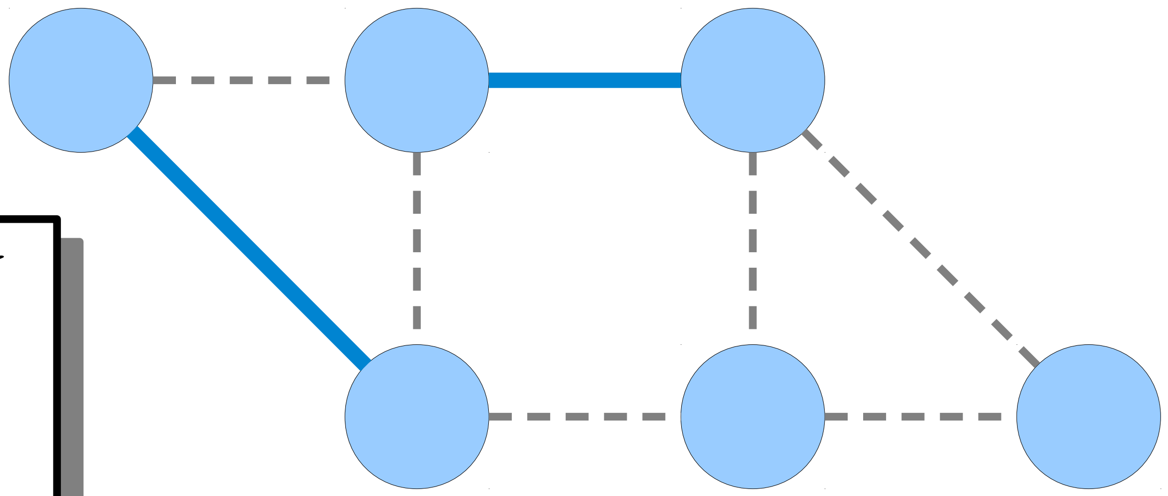
Maximum Matching

- 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.

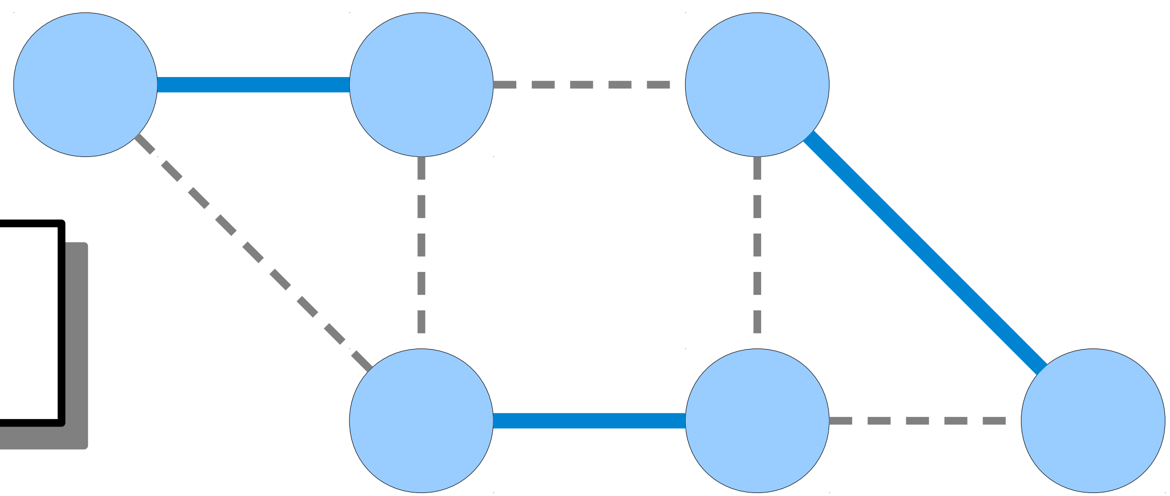


A matching, but not a maximum matching.

Maximum Matching

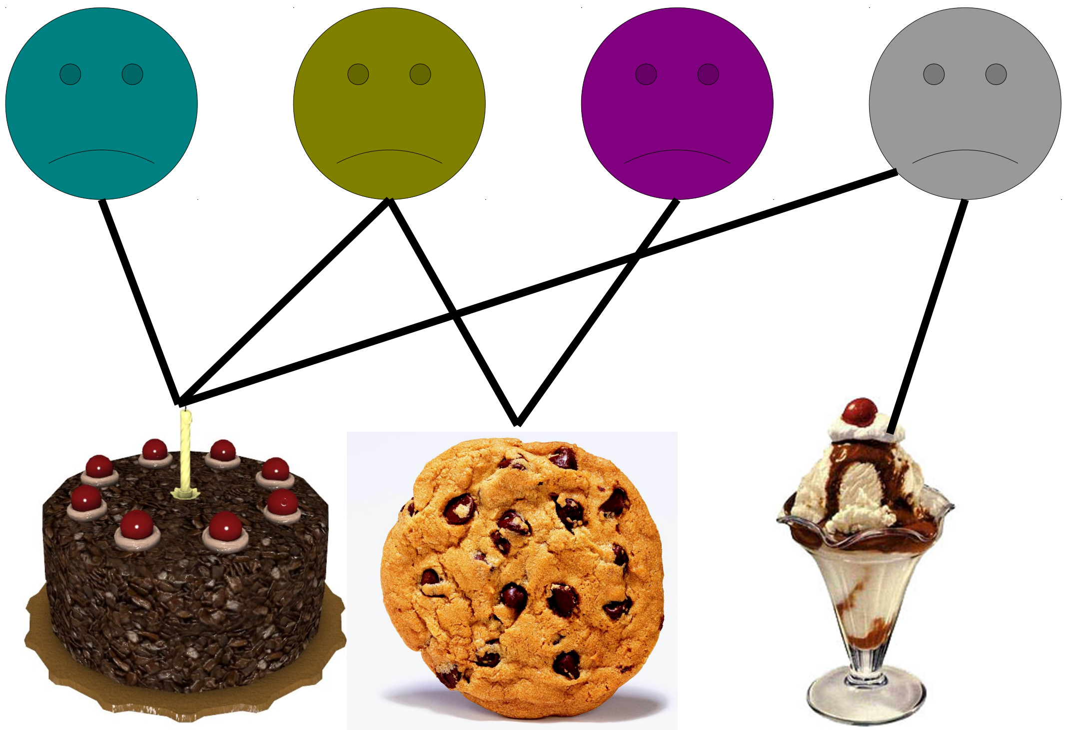
- Given an undirected graph *G*, a ***matching*** in *G* is a set of edges such that no two edges share an endpoint.
- A ***maximum matching*** is a matching with the largest number of edges.

A maximum matching.



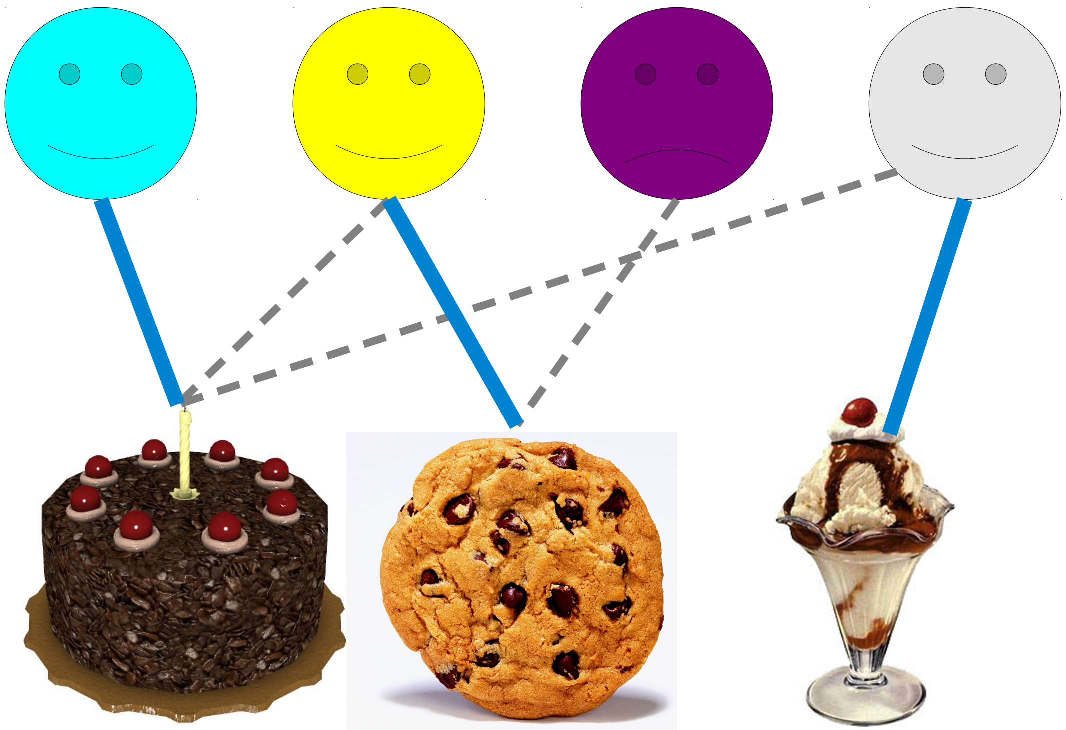
Maximum Matching

- 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

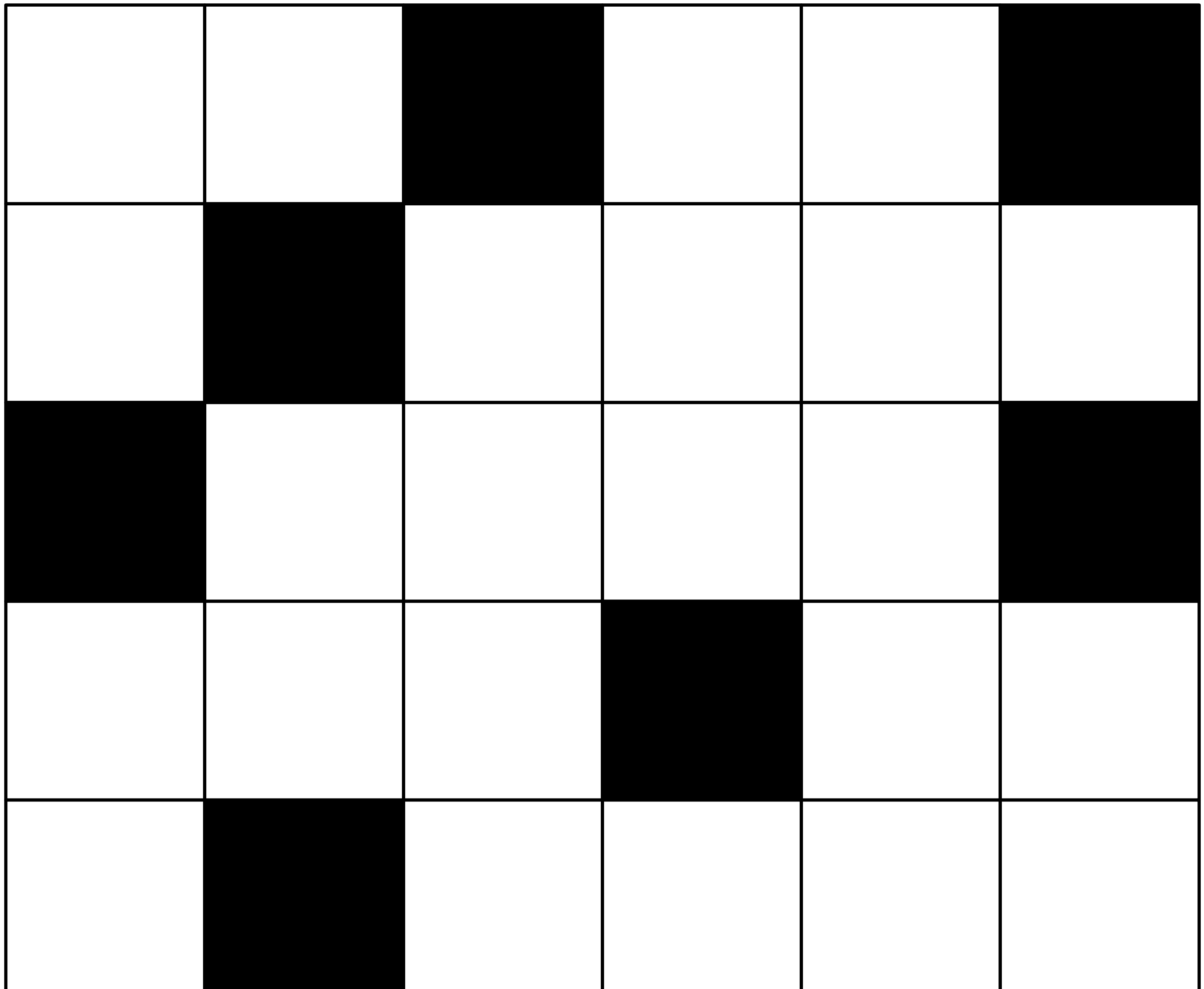
- Given an undirected graph *G*, a ***matching*** in *G* is a set of edges such that no two edges share an endpoint.
- A ***maximum matching*** is a matching with the largest number of edges.



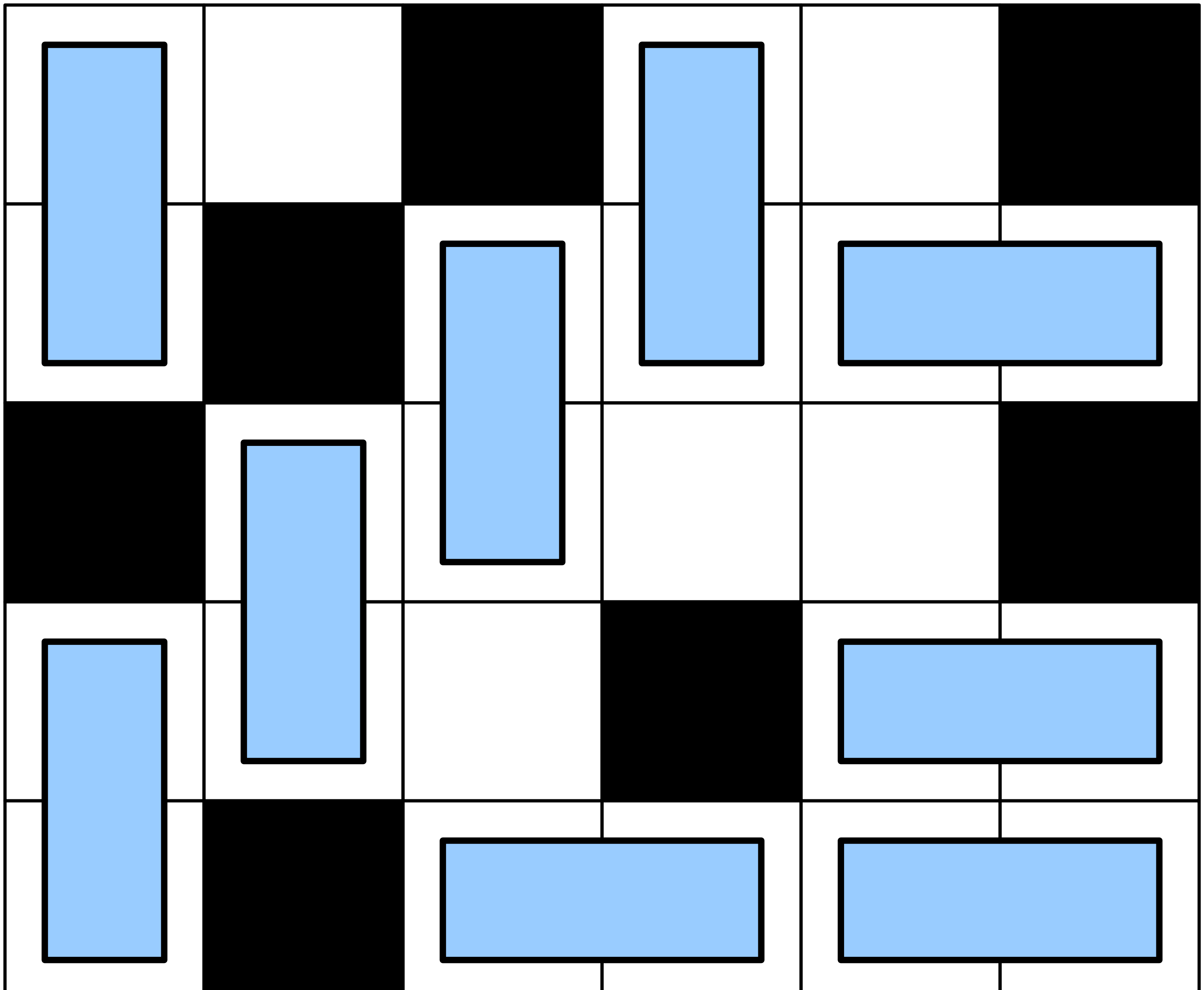
Maximum Matching

- 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?

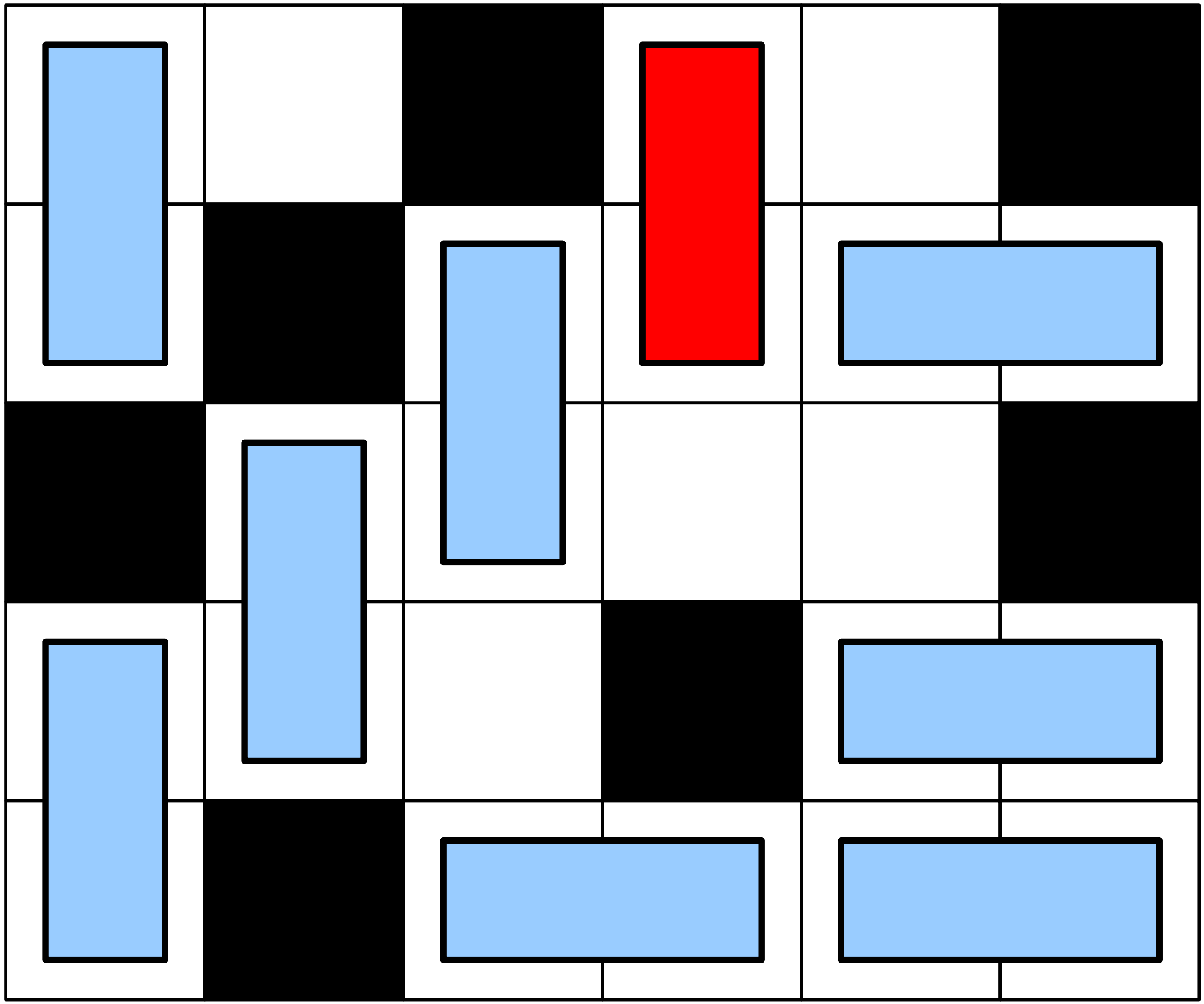
Domino Tiling



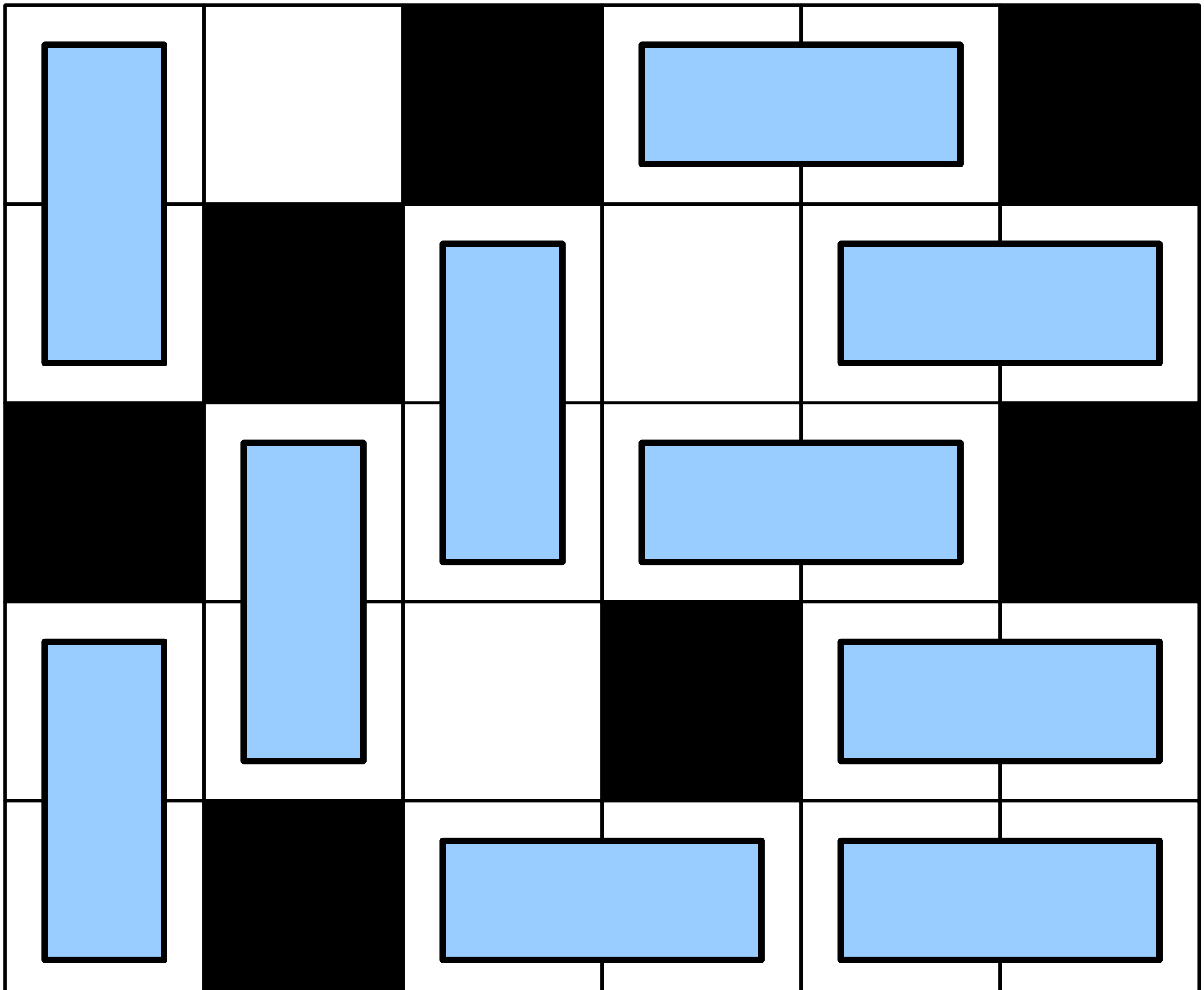
Domino Tiling



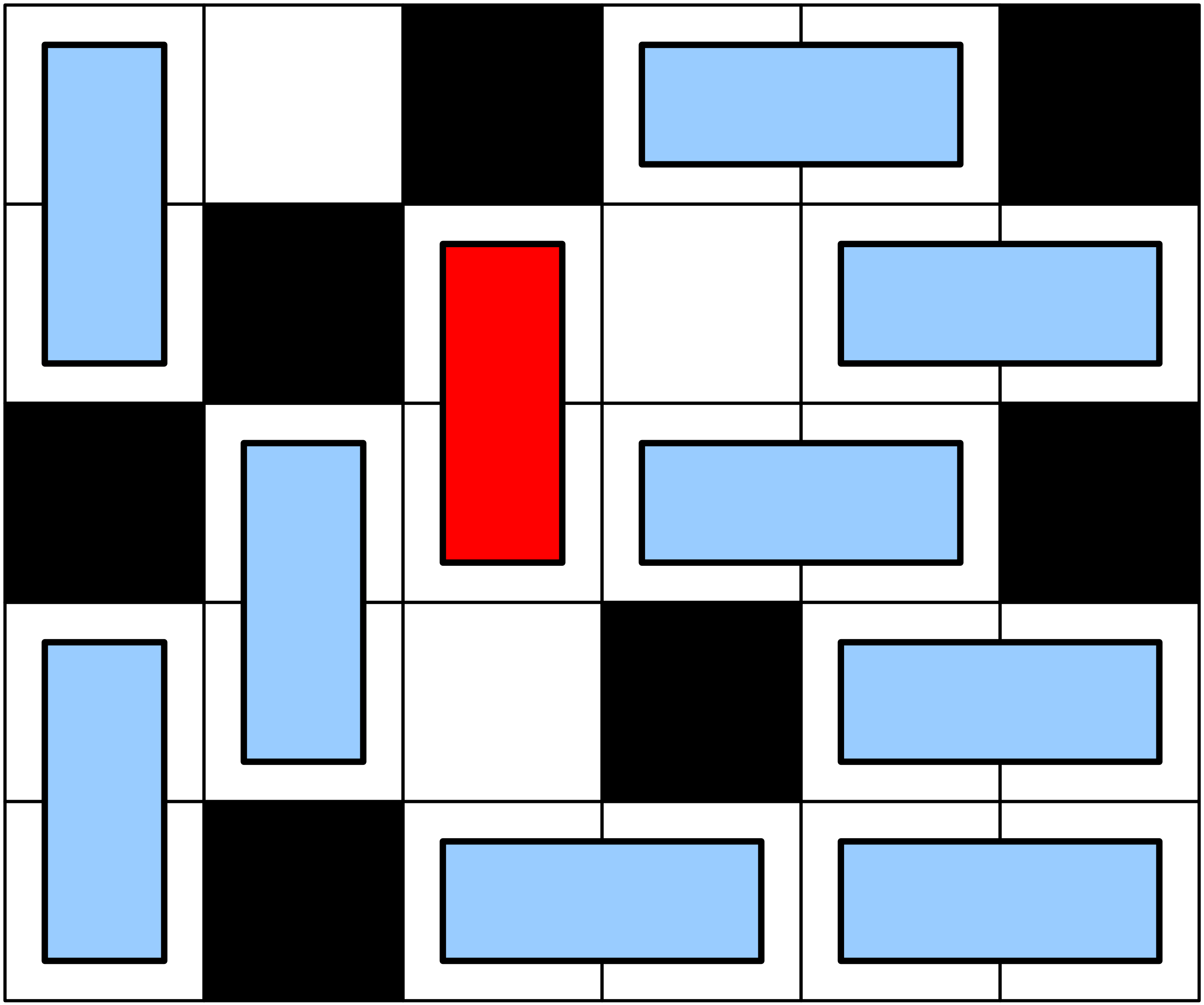
Domino Tiling



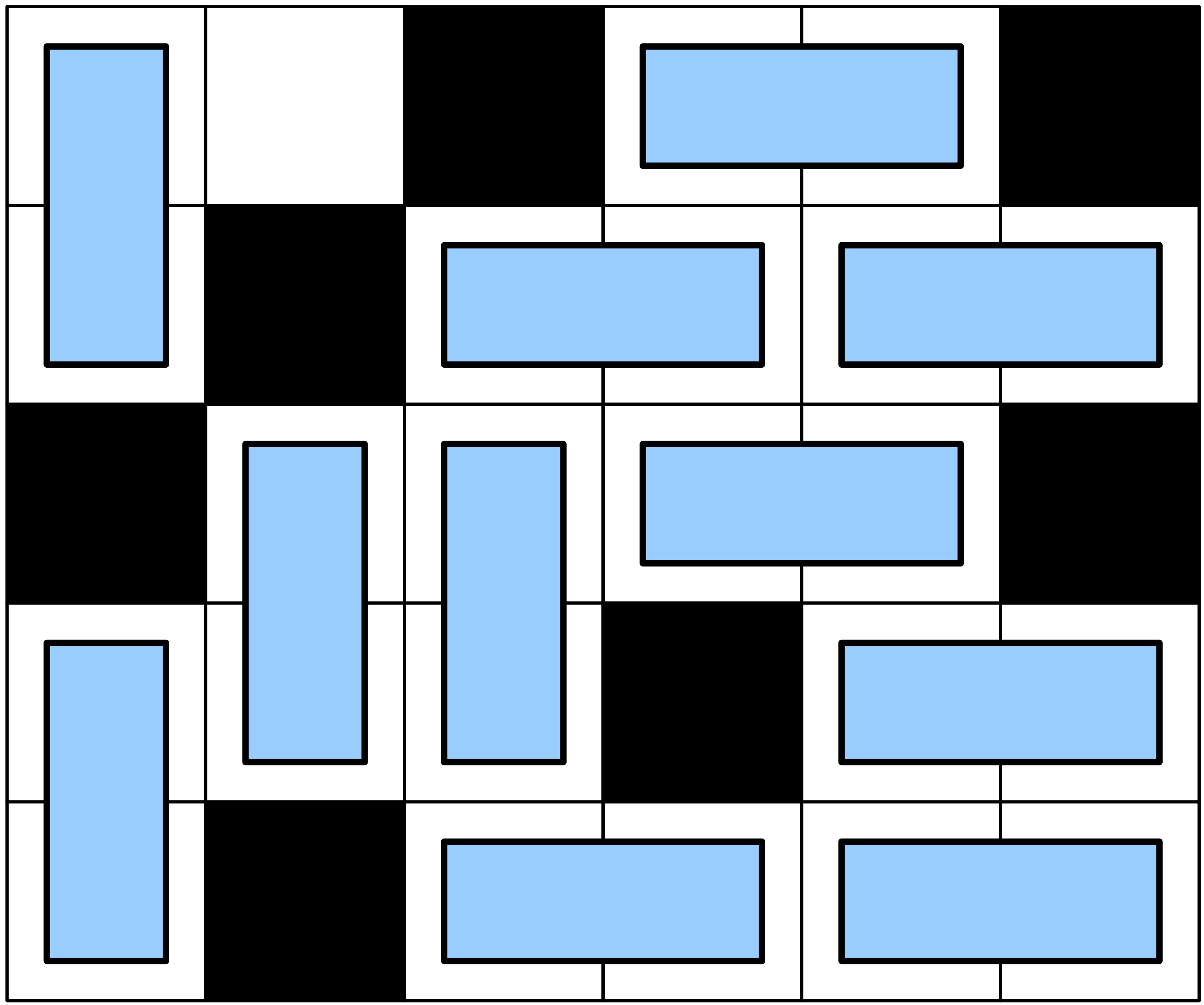
Domino Tiling



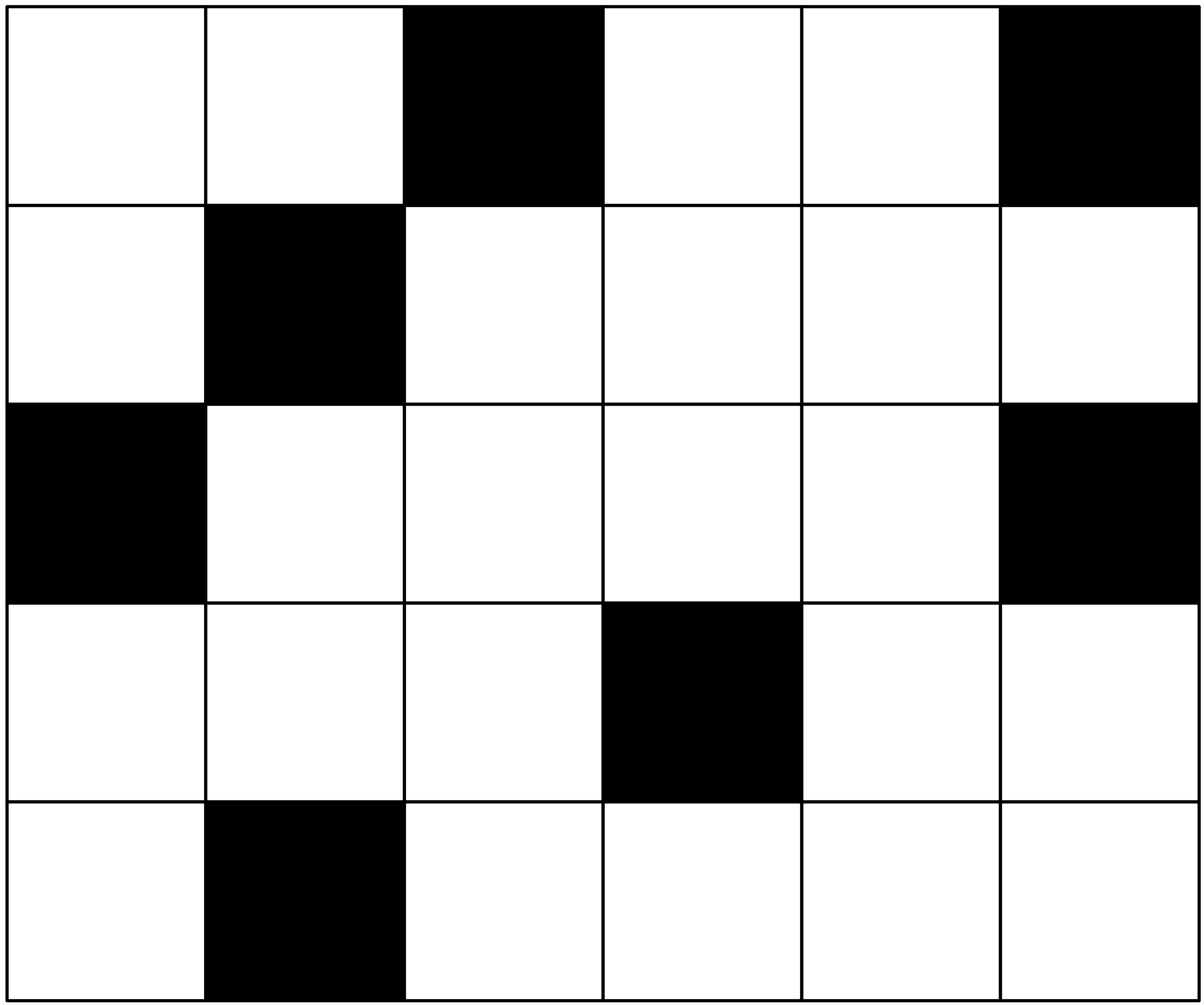
Domino Tiling



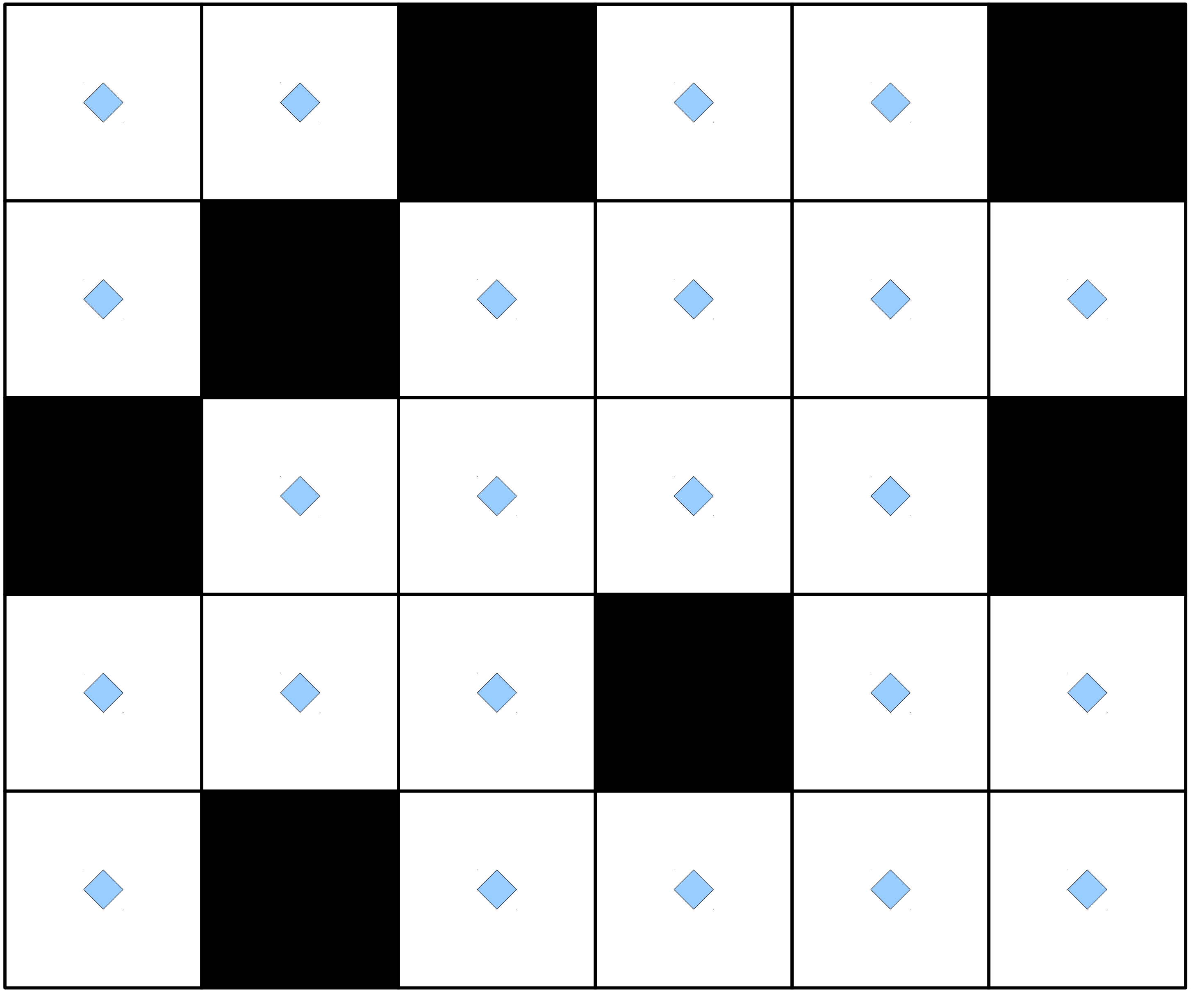
Domino Tiling



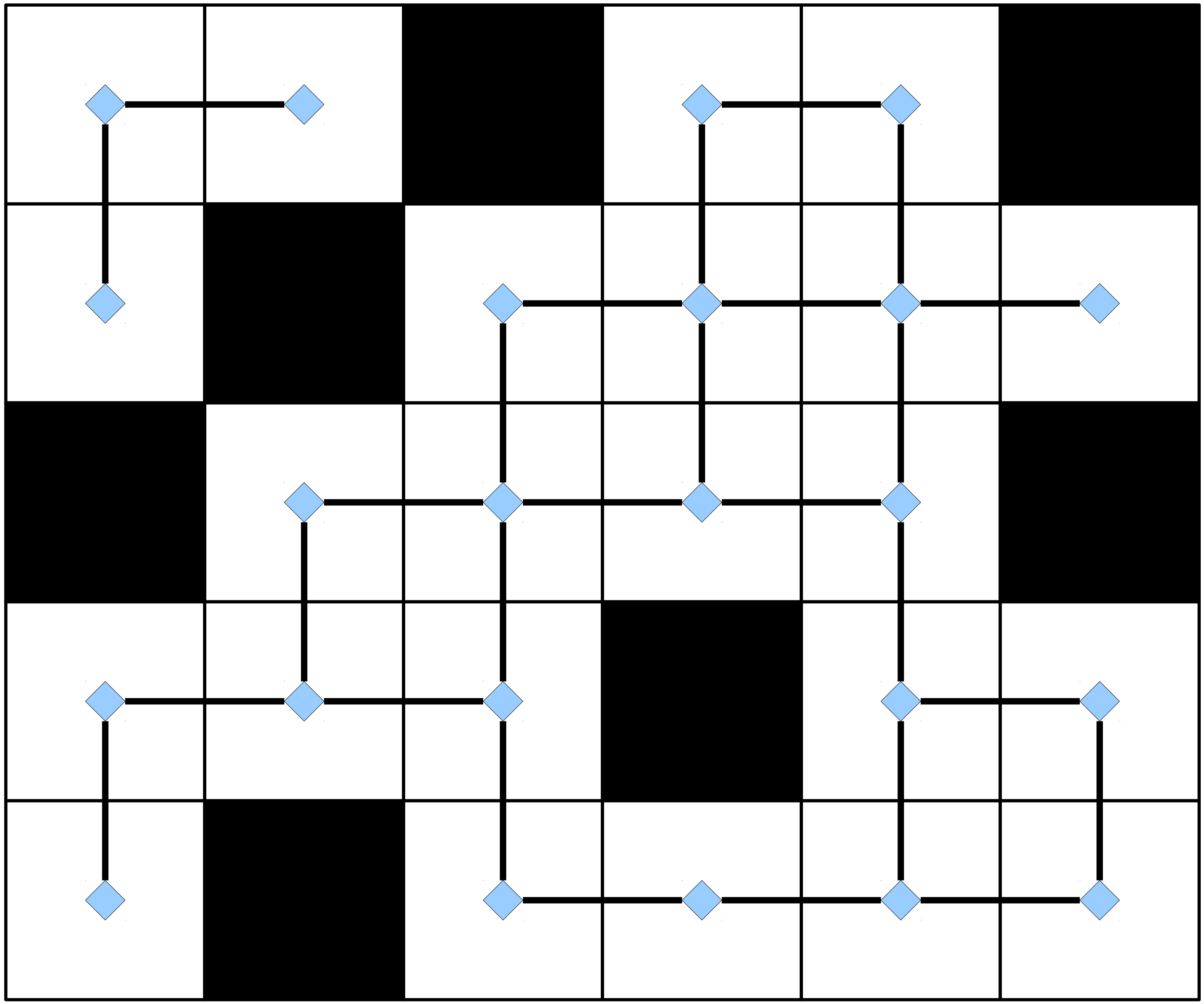
Solving Domino Tiling



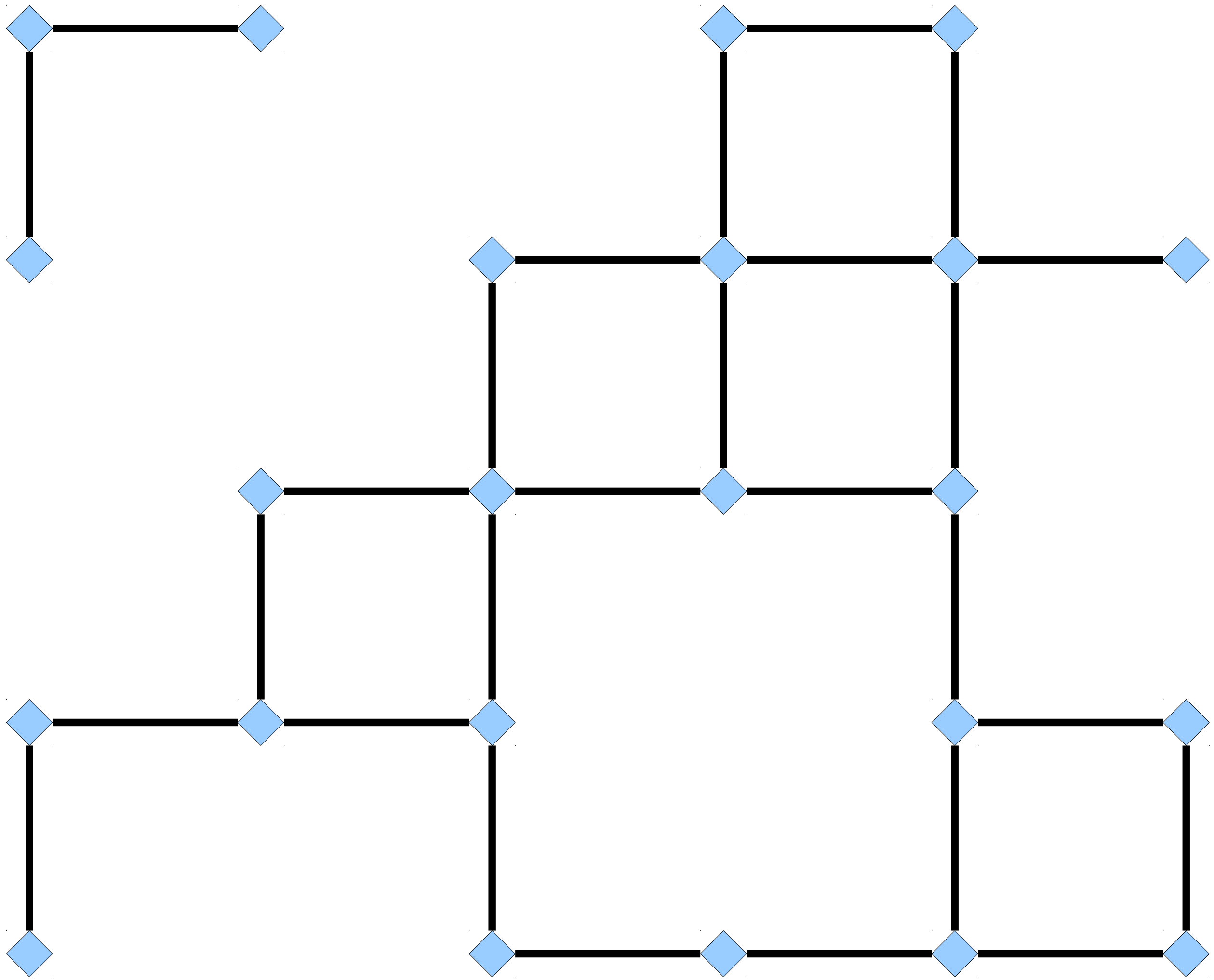
Solving Domino Tiling



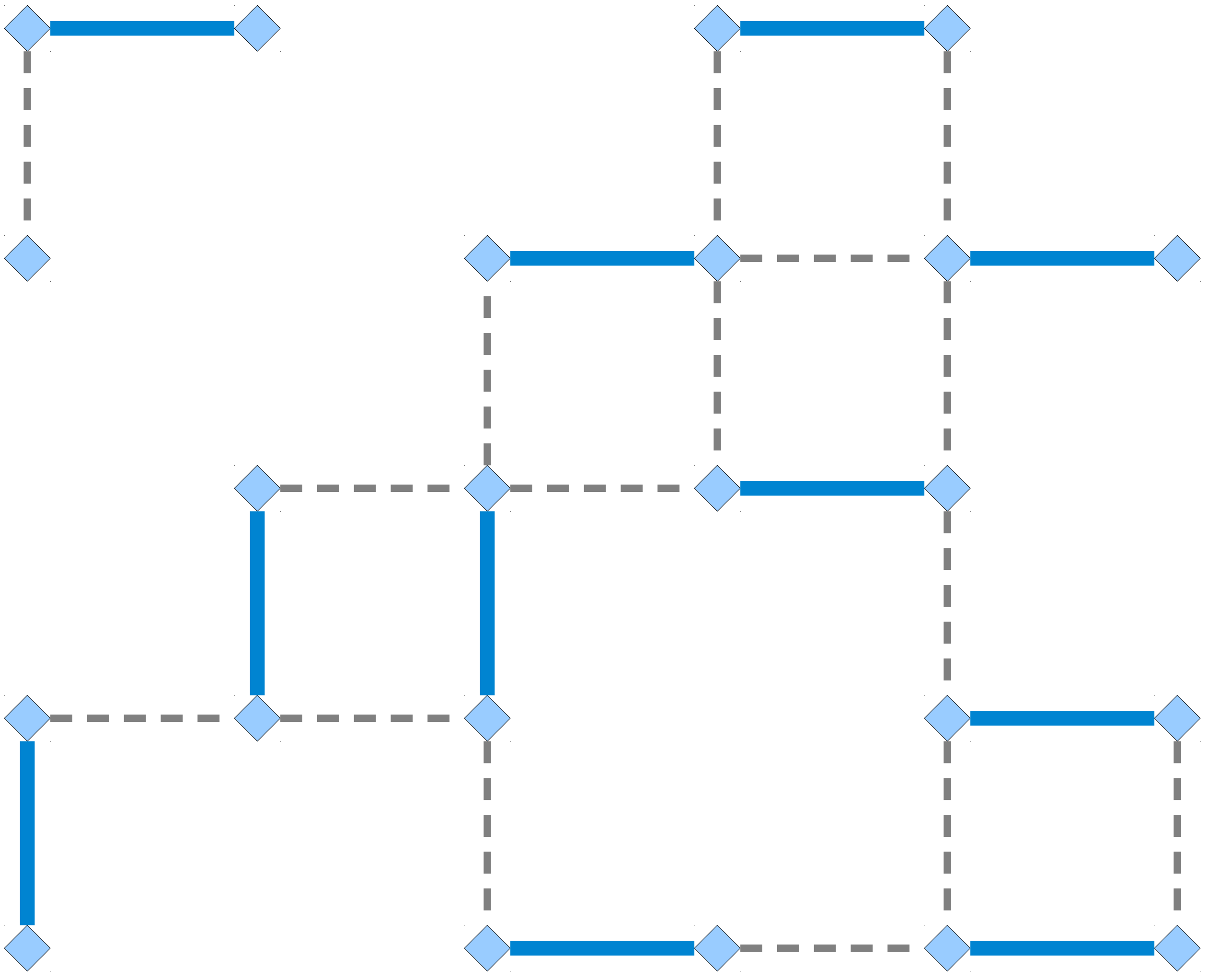
Solving Domino Tiling



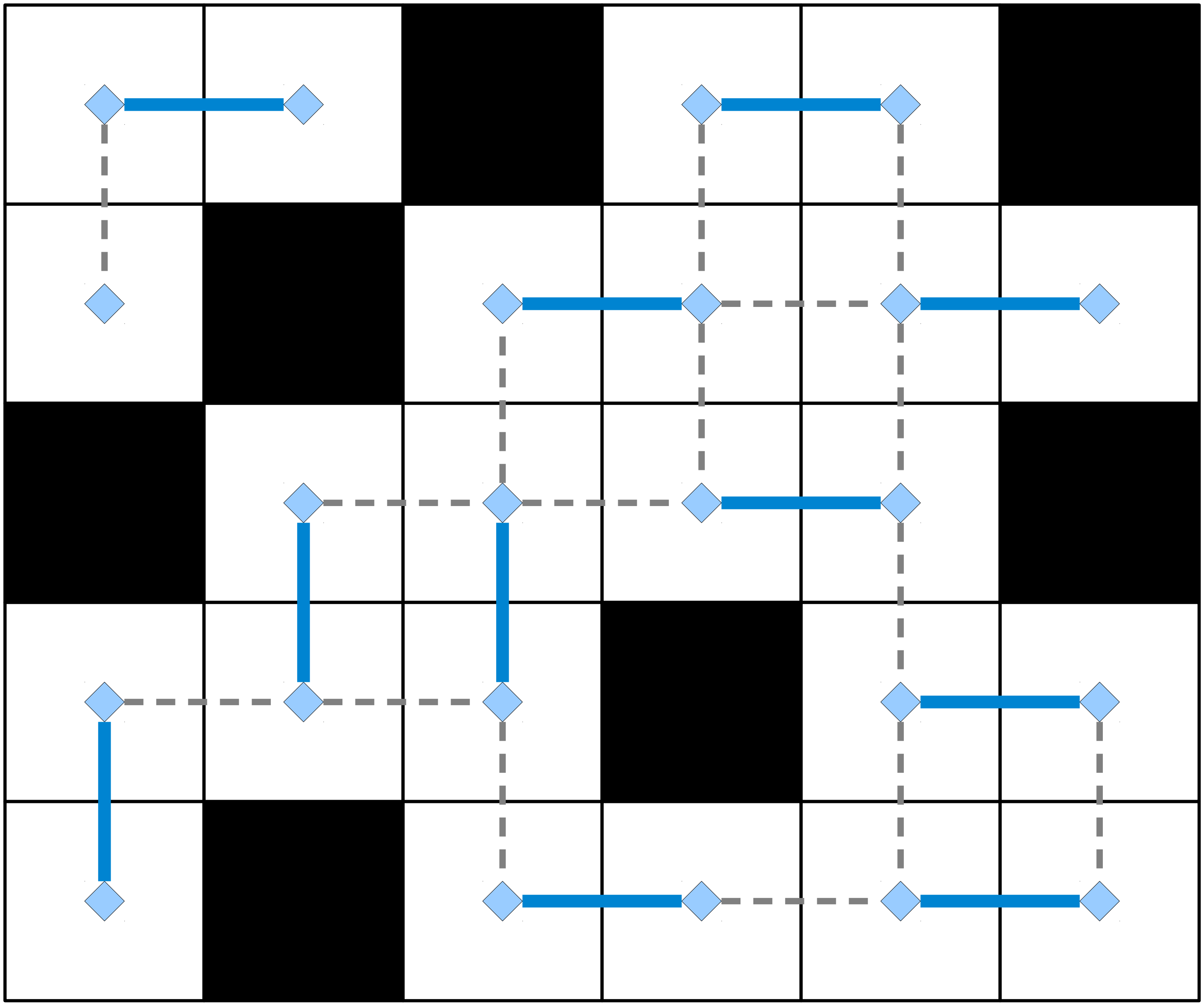
Solving Domino Tiling



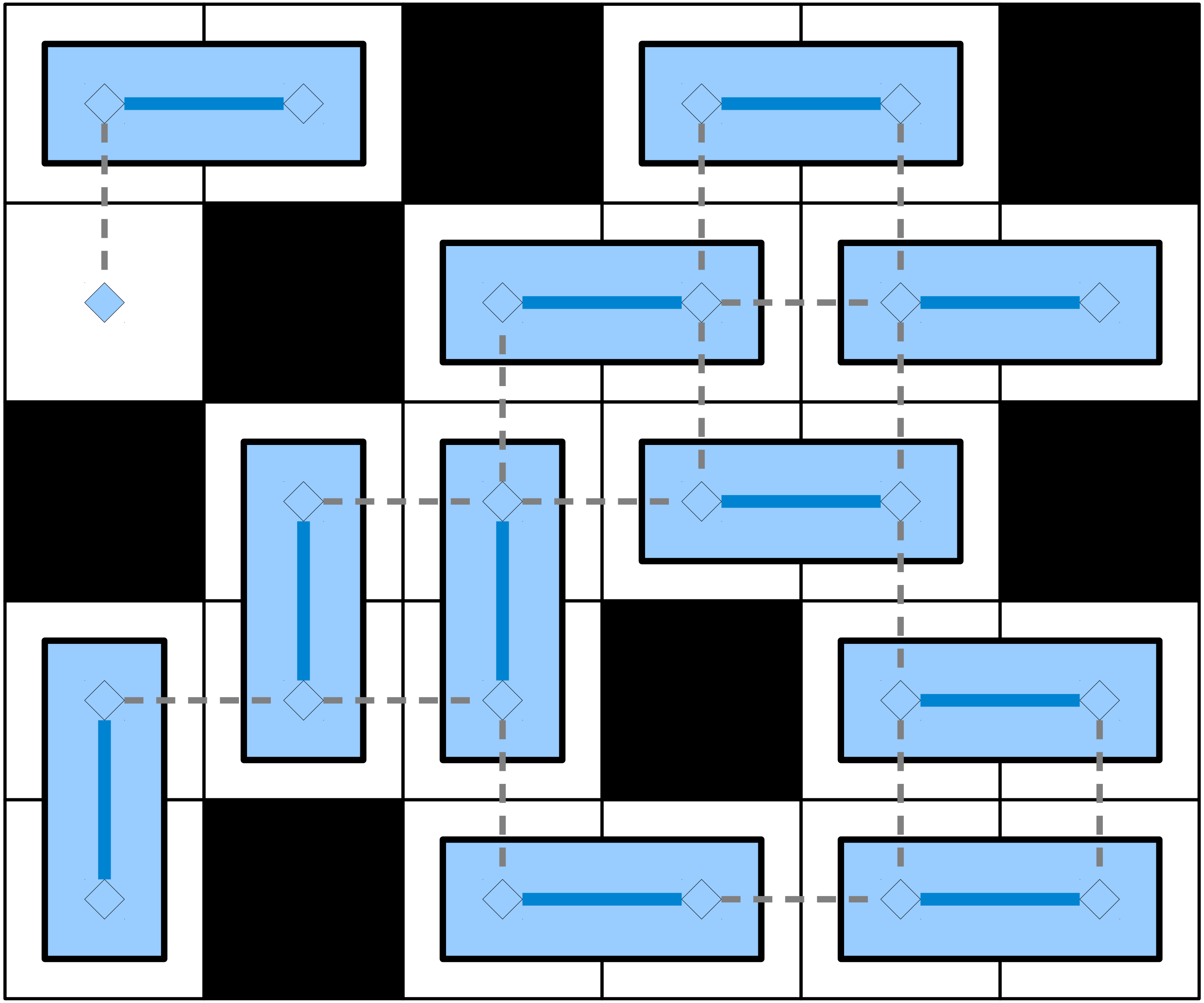
Solving Domino Tiling



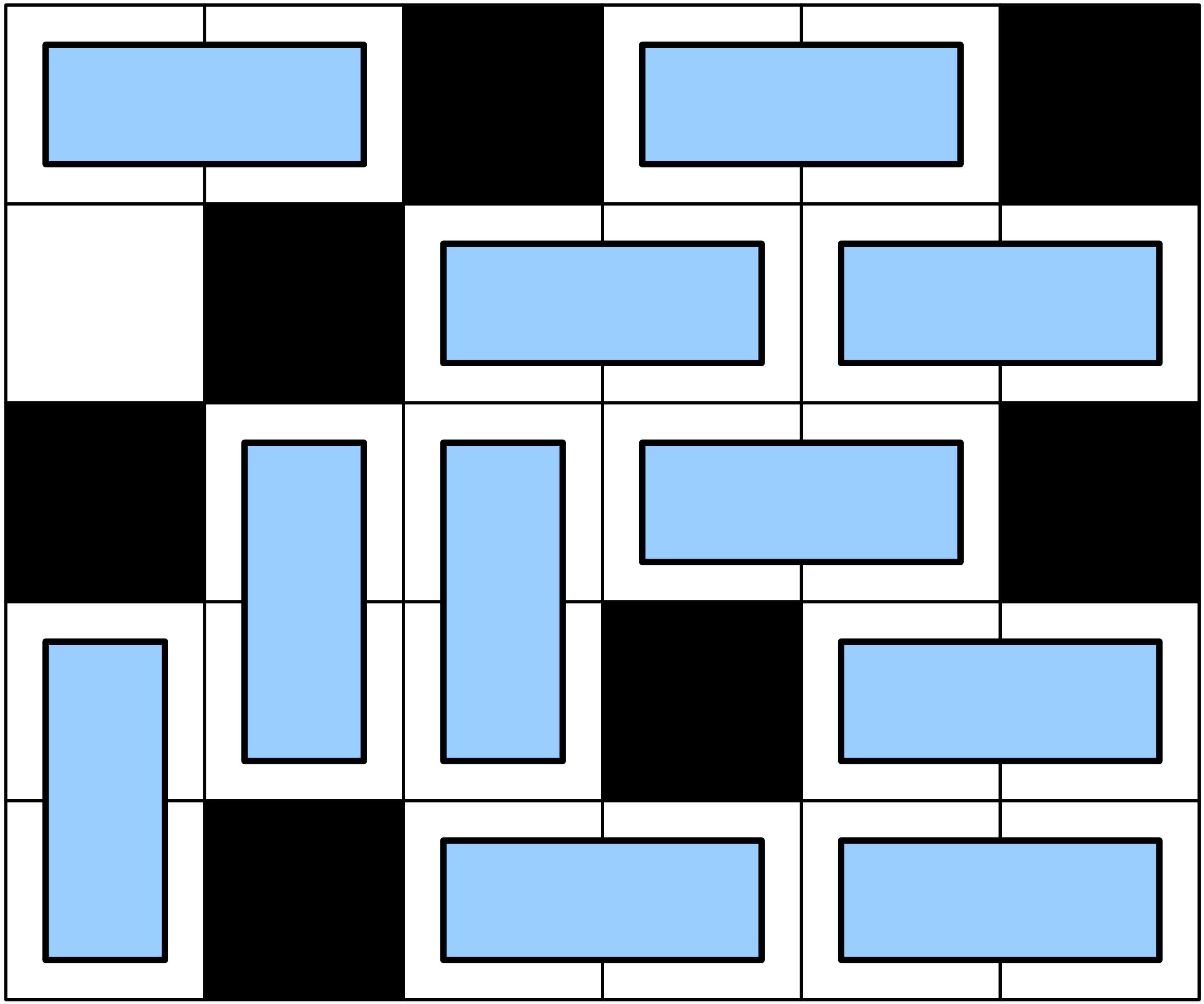
Solving Domino Tiling



Solving Domino Tiling



Solving Domino Tiling



In Pseudocode

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

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!

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?

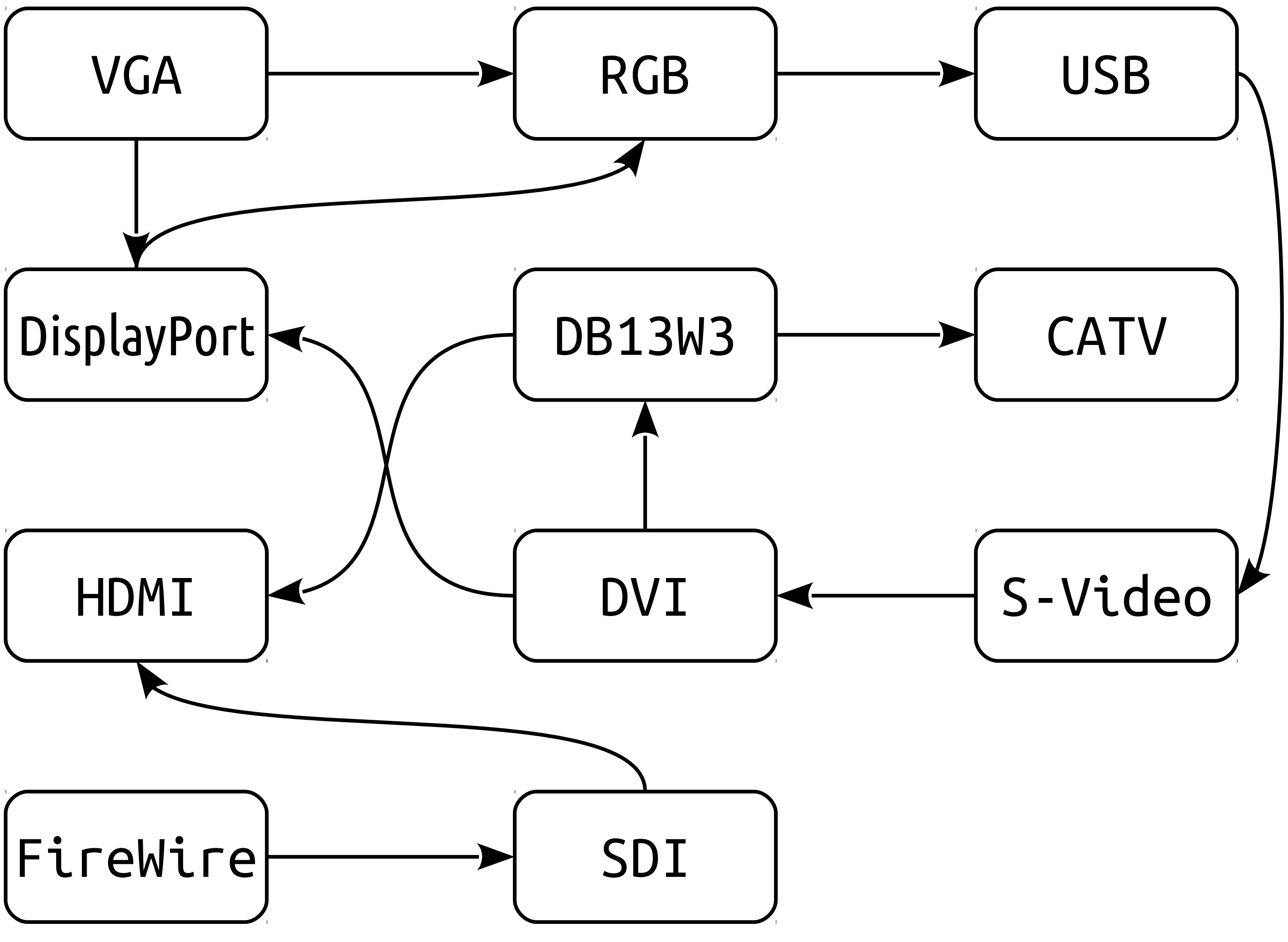
Converter Conundrums

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

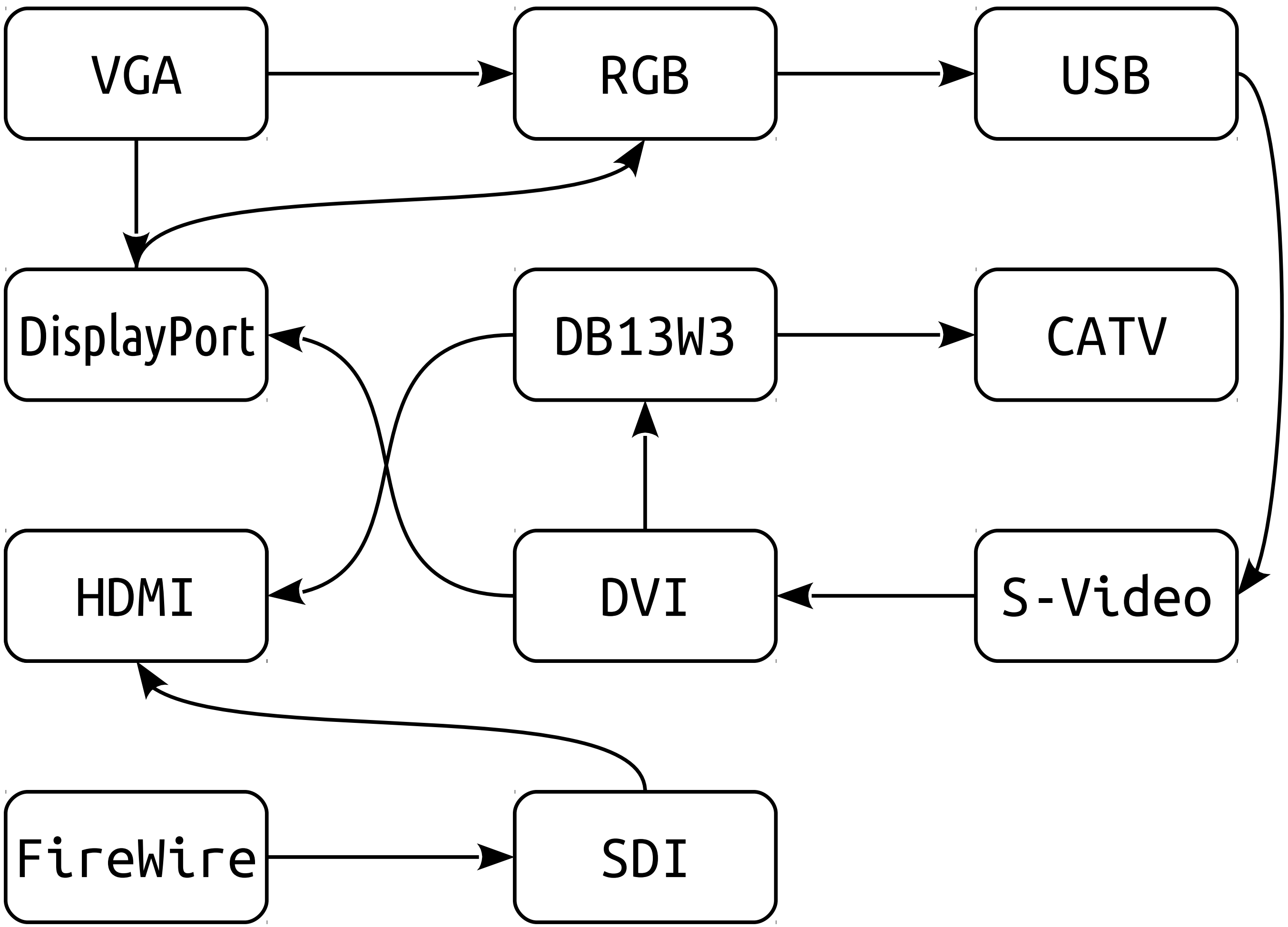
Converter Conundrums

Connectors

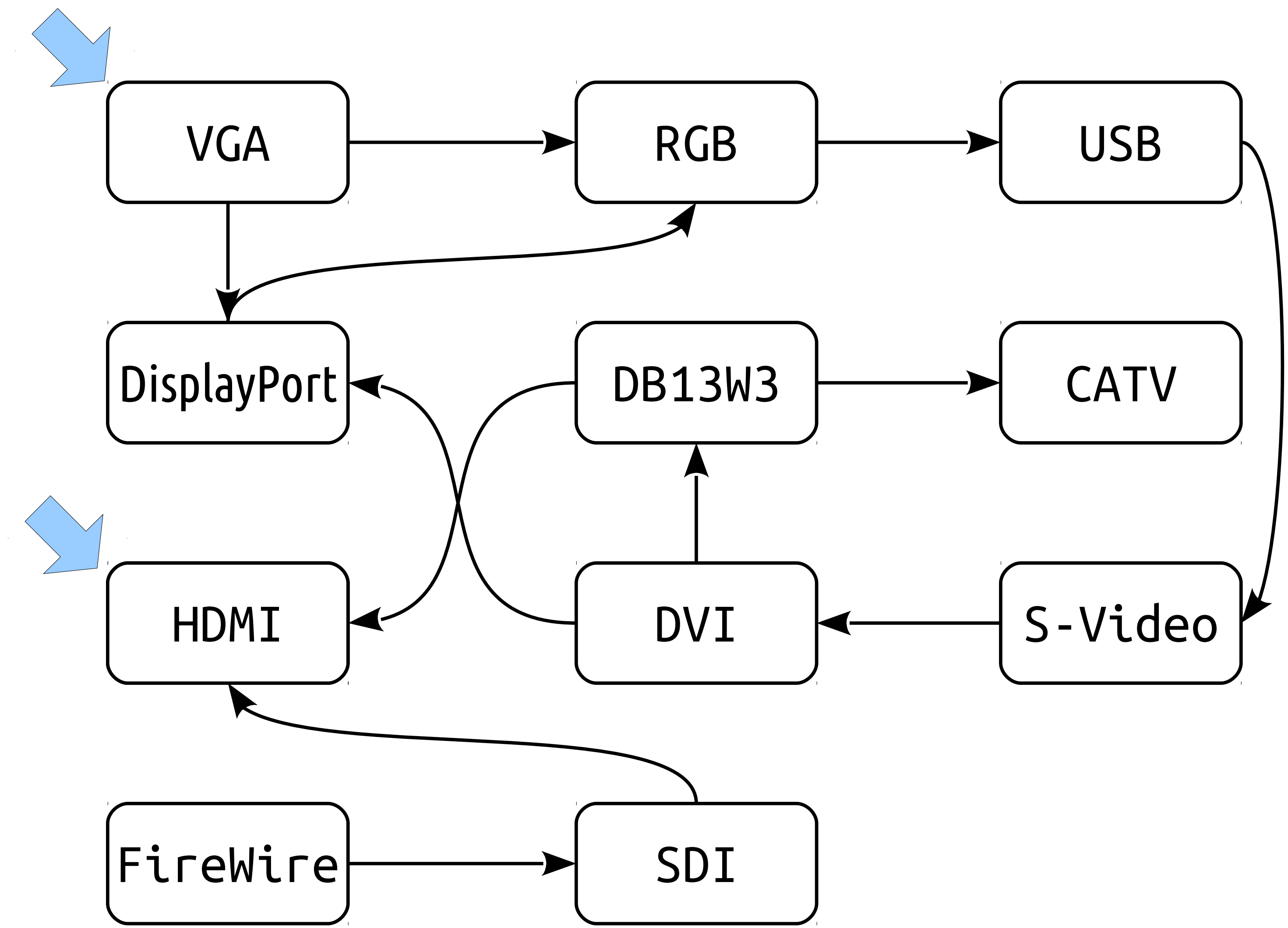
RGB to USB
VGA to DisplayPort
DB13W3 to CATV
DisplayPort to RGB
DB13W3 to HDMI
DVI to DB13W3
S-Video to DVI
FireWire to SDI
VGA to RGB
DVI to DisplayPort
USB to S-Video
SDI to HDMI



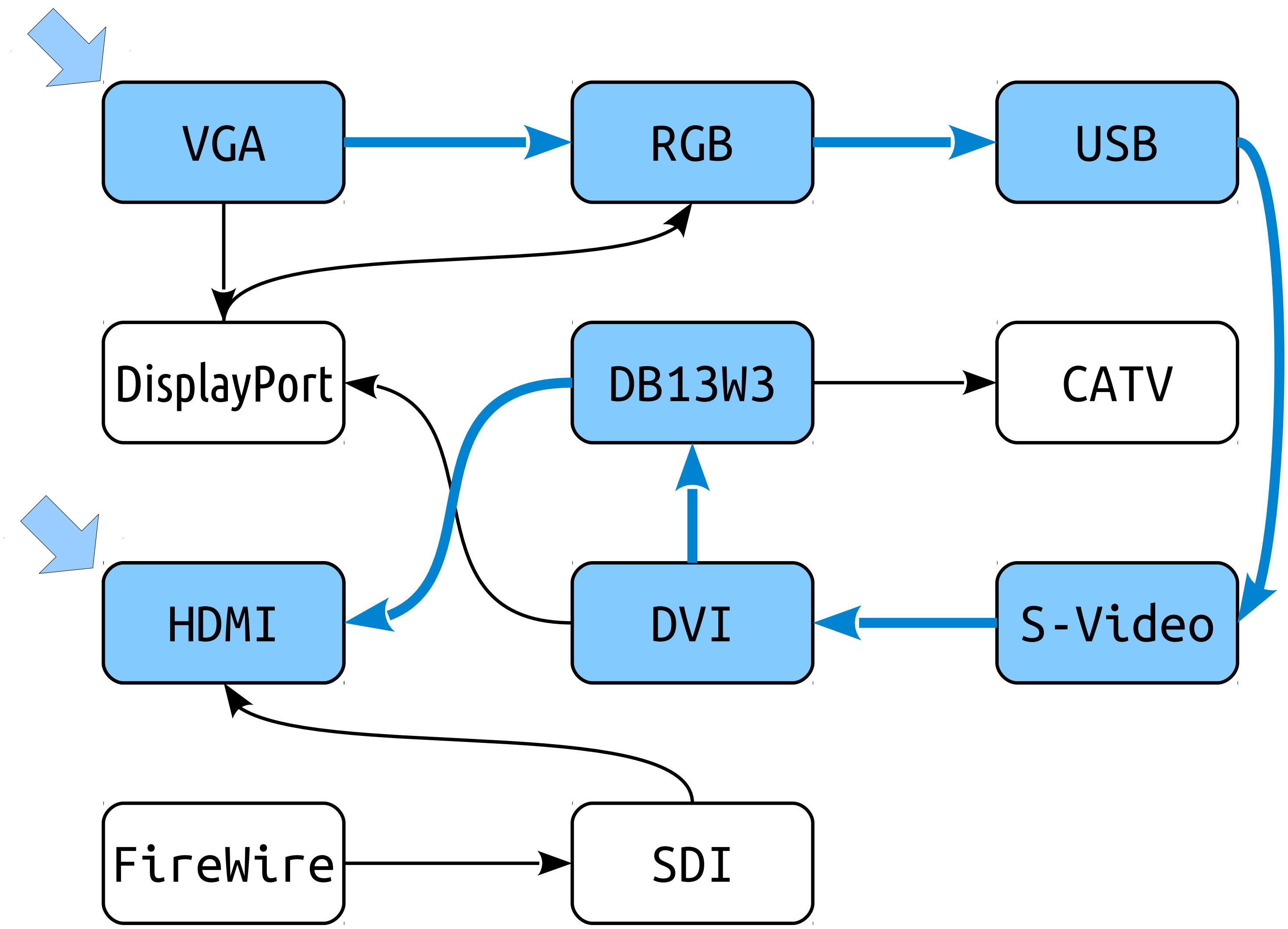
Converter Conundrums



Converter Conundrums

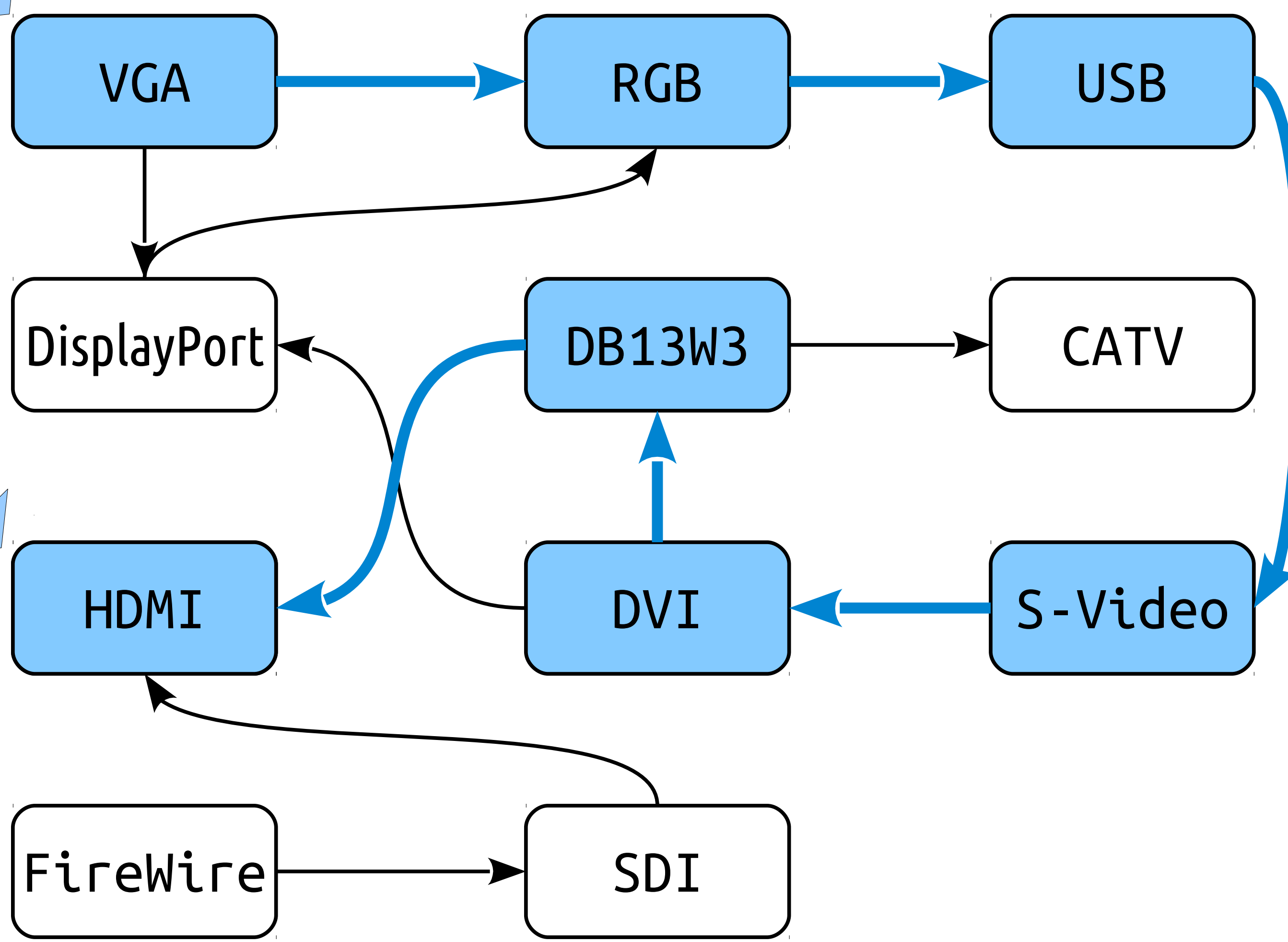


Converter Conundrums



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

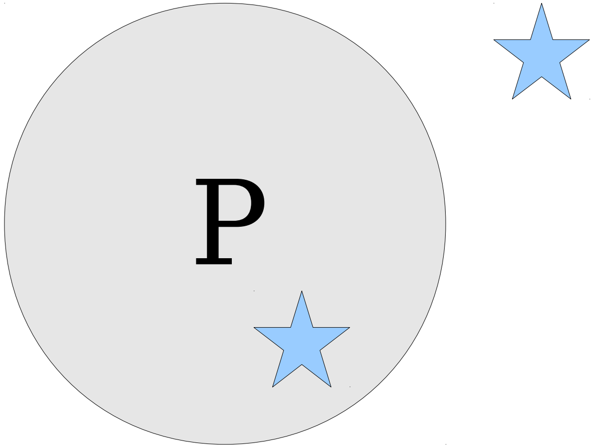
$$A \leq_{\mathrm{p}} B.$$

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

* Assuming that `transform` runs in polynomial time.

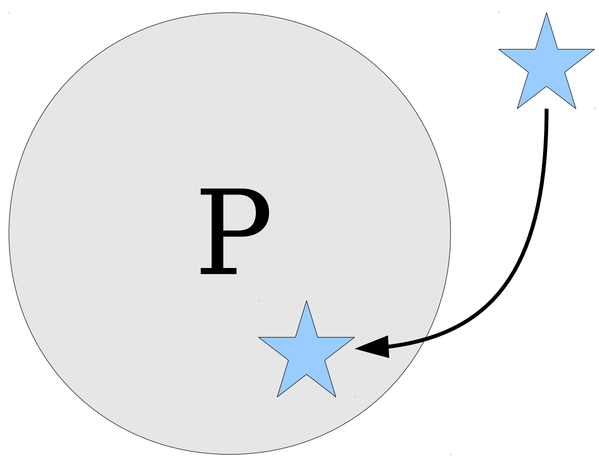
Polynomial-Time Reductions

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



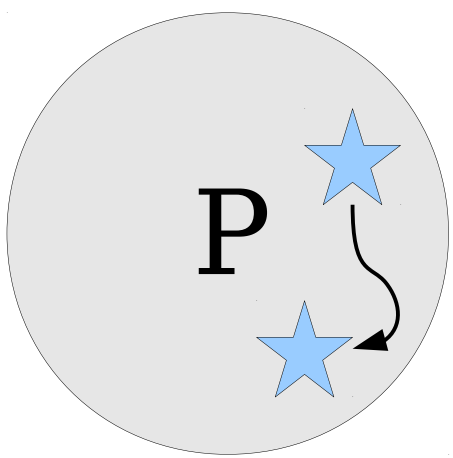
Polynomial-Time Reductions

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



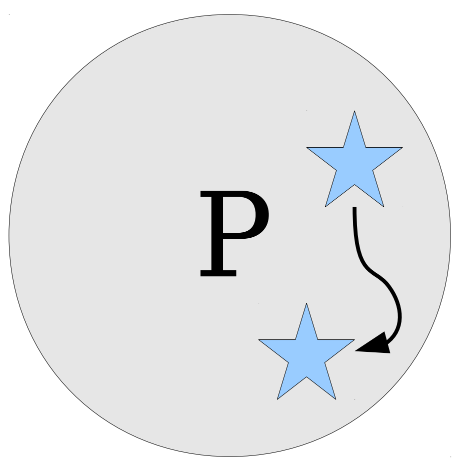
Polynomial-Time Reductions

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



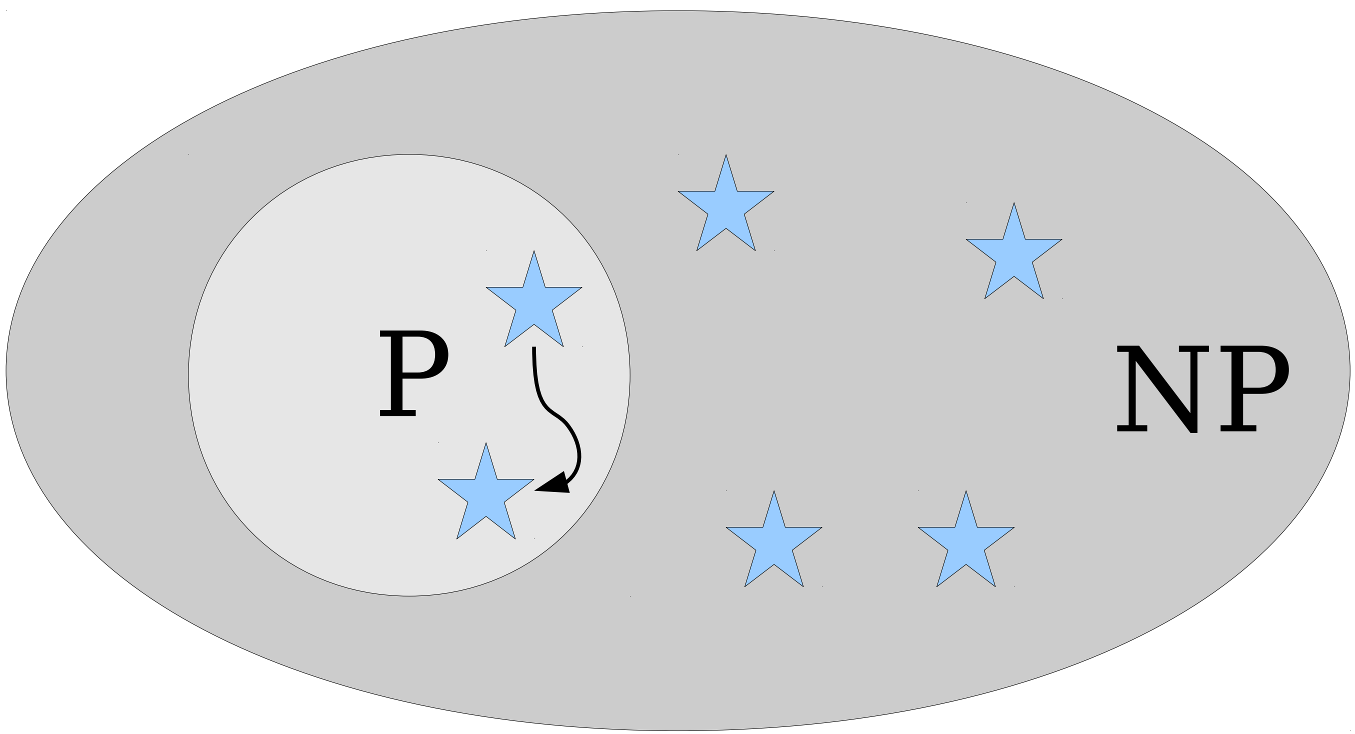
Polynomial-Time Reductions

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



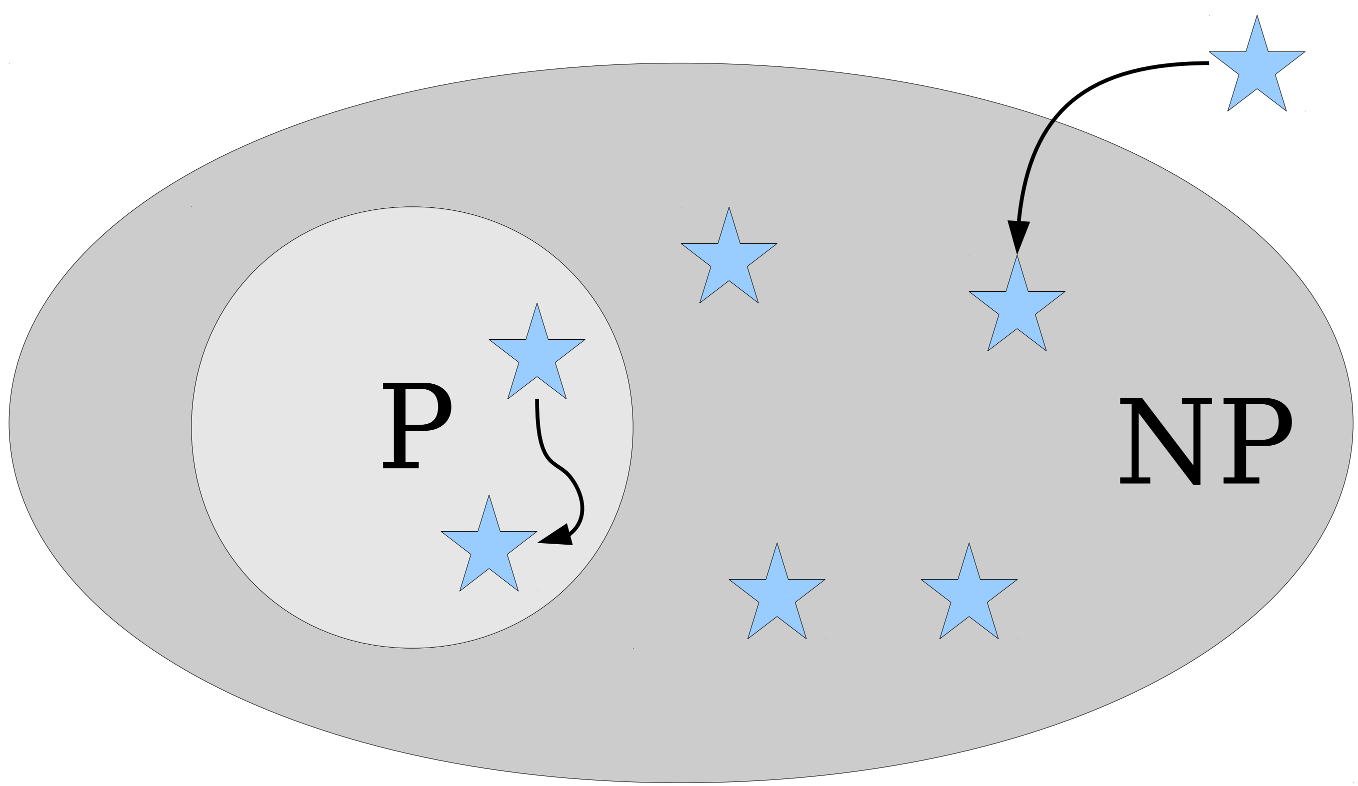
Polynomial-Time Reductions

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



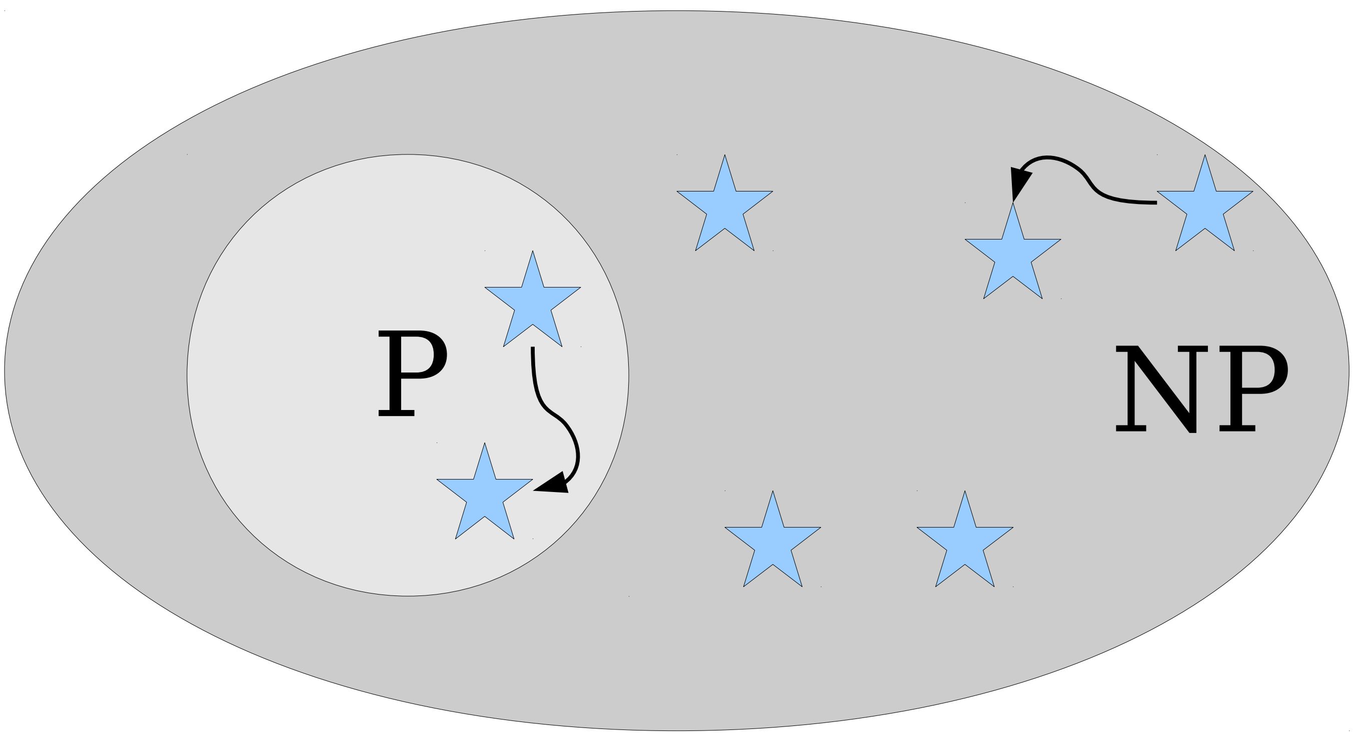
Polynomial-Time Reductions

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



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

What else can we do with it?

Time-Out for Announcements!

Please evaluate this course on Axess.

Your feedback makes a difference.

Problem Set Nine

- Problem Set Nine is due this ***Wednesday*** at the start of class.
  - ***No late submissions can be accepted***. This is university policy – sorry!
  - This problem set is much shorter than the other ones we've given out so far this quarter.
- Check the course website for the final office hours timetable for the week.

Final Exam Logistics

- Our final exam is this Friday from 3:30PM – 6:30PM. Rooms are divvied up by last (family) name:
  - `Abb – Kan`: Go to Bishop Auditorium.
  - `Kar – Zuc`: Go to Cemex Auditorium.
- Exam is cumulative and all topics from the lectures and problem sets are fair game.
- The exam focus is roughly 50/50 between discrete math topics (PS1 – PS5) and computability/complexity topics (PS6 – PS9).
- As with the midterms, the exam is closed-book, closed-computer, and limited-note. You can bring a single, double-sided, 8.5” × 11” sheet of notes with you to the exam.

Preparing for the Exam

- Up on the course website, you'll find
  - three sets of extra practice problems (EPP9 – EPP11), with solutions, and
  - four practice final exams, with solutions.
- Feel free to ask questions about them on Piazza or in office hours.
  - Thursday office hours are a ***great*** place to ask questions!
- Need more practice on a particular topic? Let us know!
  - Folks on Piazza have asked for more practice with Myhill-Nerode and CFG design, and we're working on putting together some more practice problems along those lines. Stay tuned!

Your Questions

“Regarding CS tracks, are there any references out there that help describe the CS tracks in greater detail and how each apply to industry today?”

This is a great question and I don't have a good answer for you. I'll ask around and see if I can come up with anything. If so, I'll share it on Wednesday. If not, I'll ask around about making one!

“Keith! as someone struggles in CS/STEM, your constant encouragement and belief in us makes a world of difference. Any advice for the future when things get hard”

On entry to this quarter, I've had 7,265 total students in the courses I've taught. I've seen a lot of people struggle. I've met people for whom the material didn't click as quickly as they'd like. I've chatted with folks who were worried about whether they could keep up. But I have never, not once, met someone I legitimately thought could not learn this material.

Learning is uncomfortable – it requires you to face the harsh reality that there are things that you don't understand as well as you think you do. It requires repetition and practice and can be frustrating. But the rewards are immense. Take a look back on what you've accomplished in this quarter. Where did you start? And where are you now? It's easy to lose sight of that in the daily/weekly/monthly grind, but it's so refreshing when you see it.

And you're always welcome to come talk to me even if you aren't in CS103! I love hearing what people are up to!

“How do you think the mindset of computer science thinking extend to life? (How CS affects one's outlook of life compared to non-STEM disciplines?)”

I'll take this one in class because I was silly and didn't budget enough time to write up an answer before coming to class. ☺

Back to CS103!

NP-Hardness and **NP**-Completeness

Question: What makes a problem hard to solve?

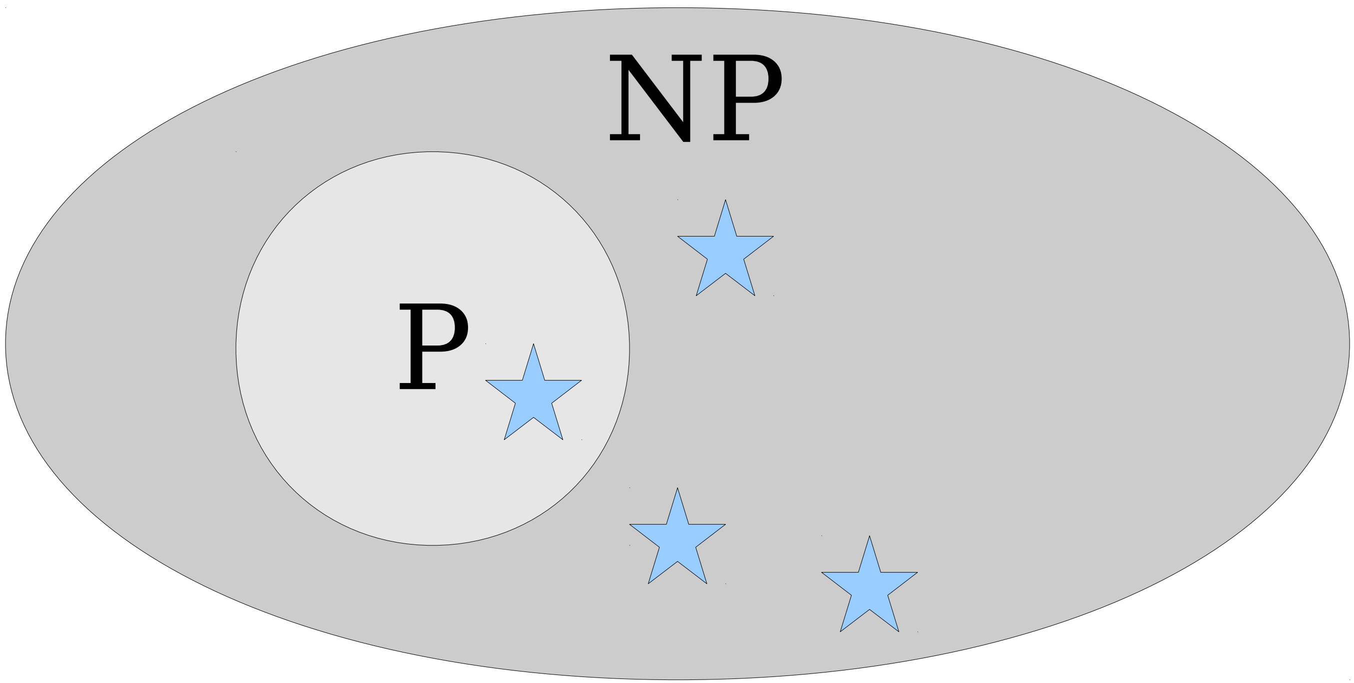
Intuition: If $A \leq_p B$, then problem B is at least as hard* as problem A.

* 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.

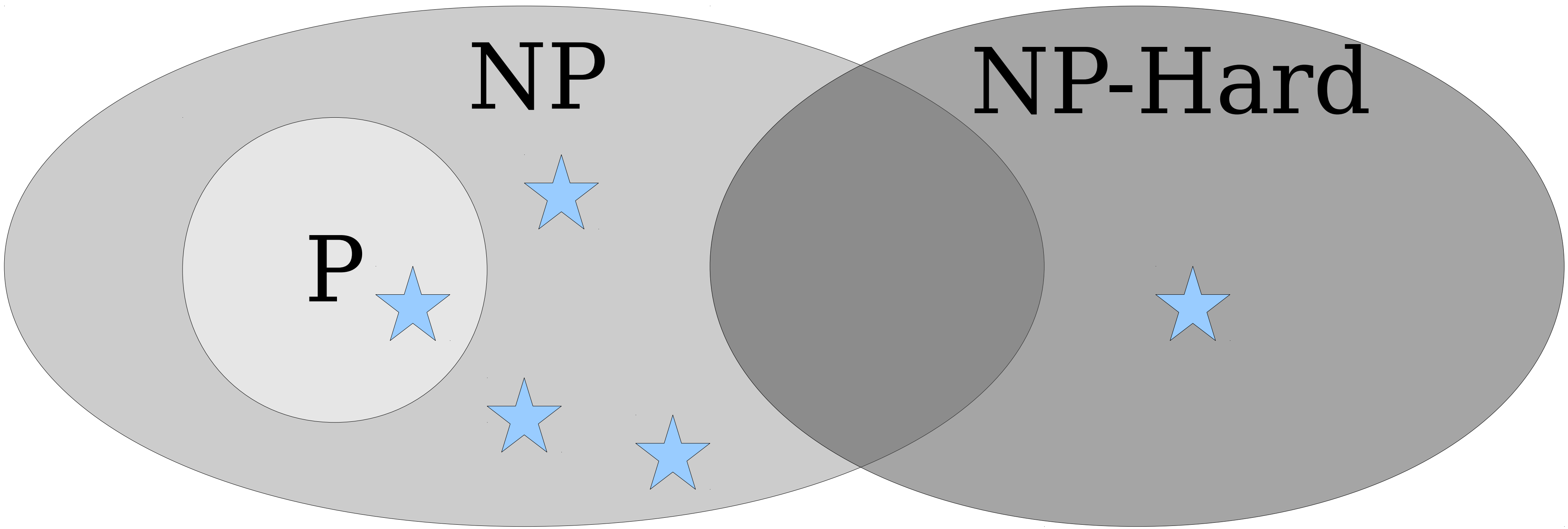
NP-Hardness

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



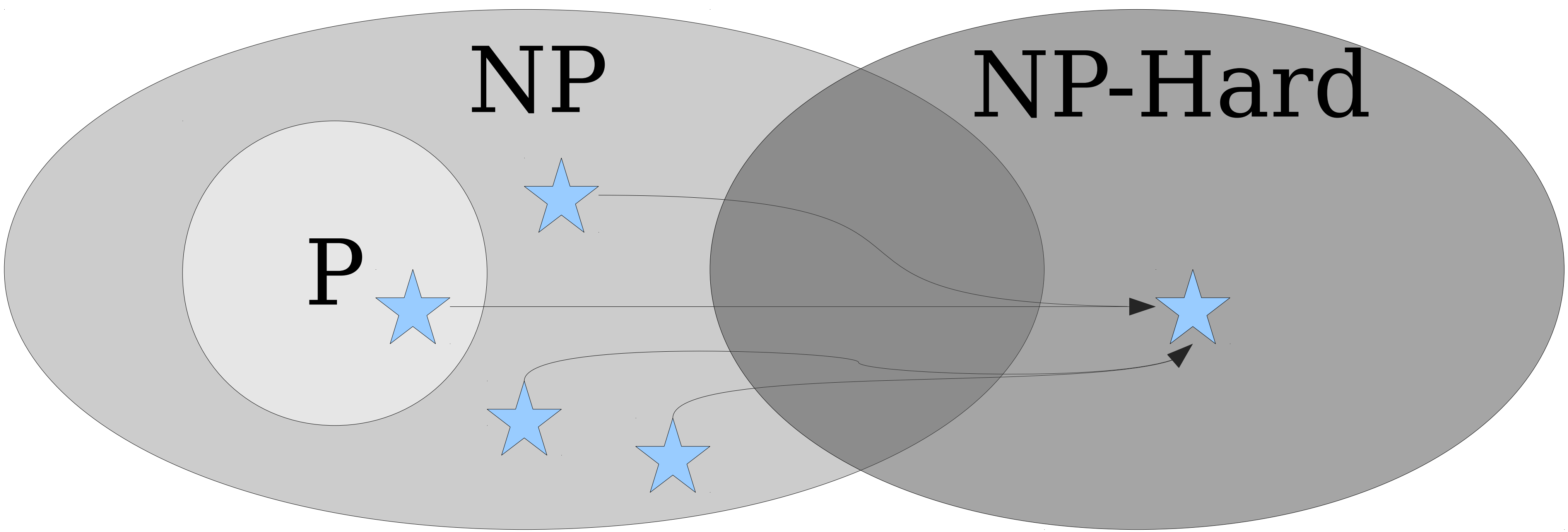
NP-Hardness

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



NP-Hardness

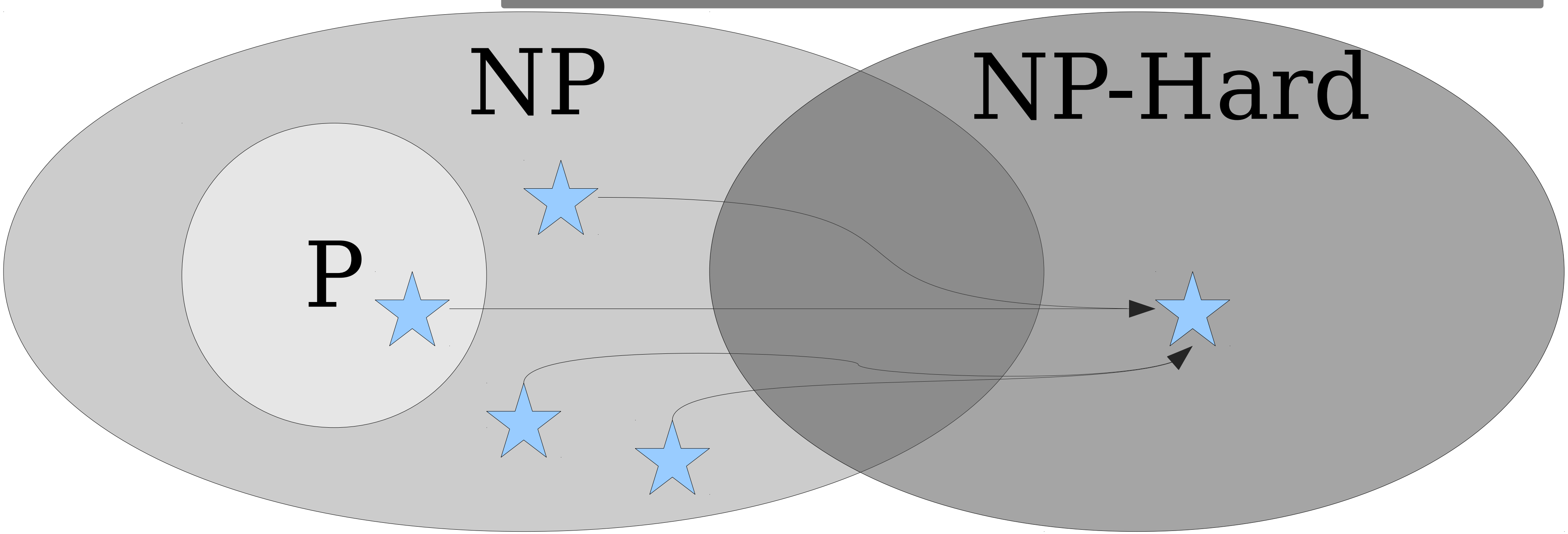
- A language L is called ***NP-hard*** if for *every* $A \in$ **NP**, we have $A \leq_P L$.



NP-Hardness

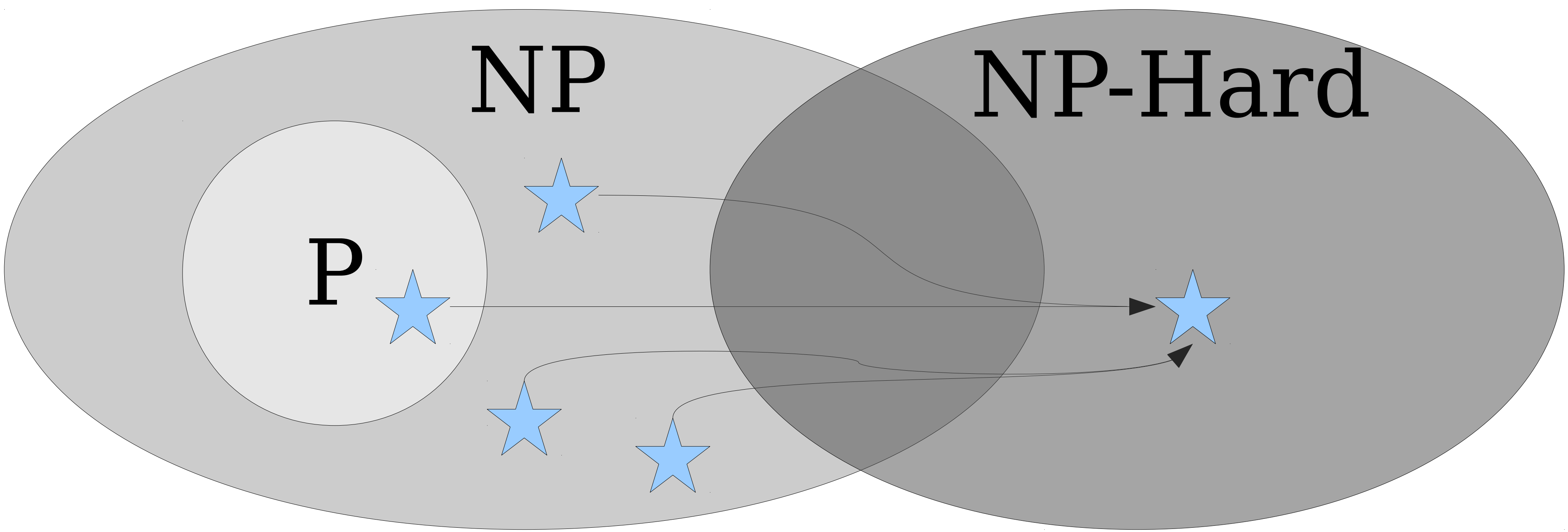
- A language L is called ***NP-hard*** if for *every* $A \in$ **NP**, we have $A \leq_P L$.

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**.



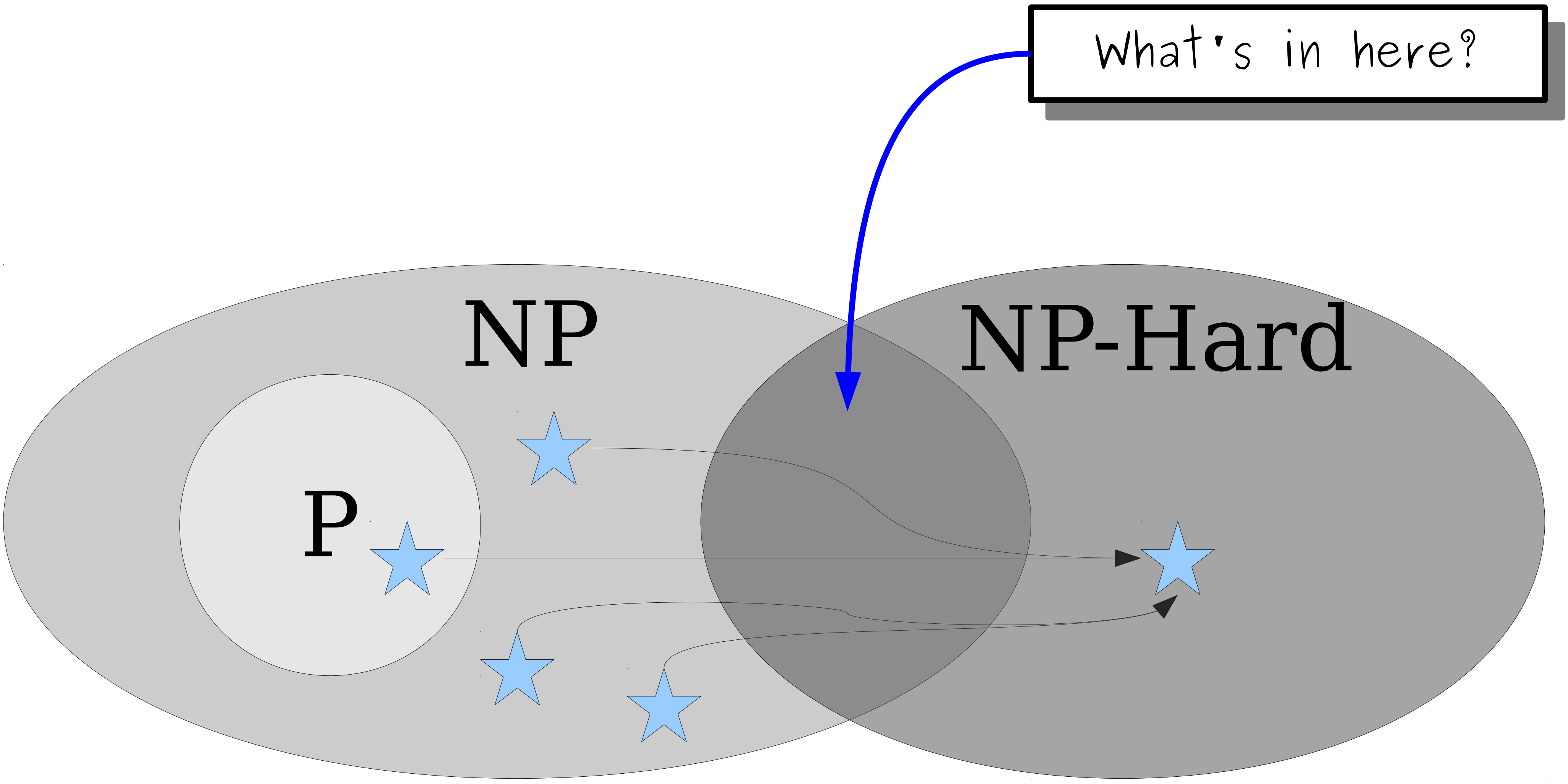
NP-Hardness

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



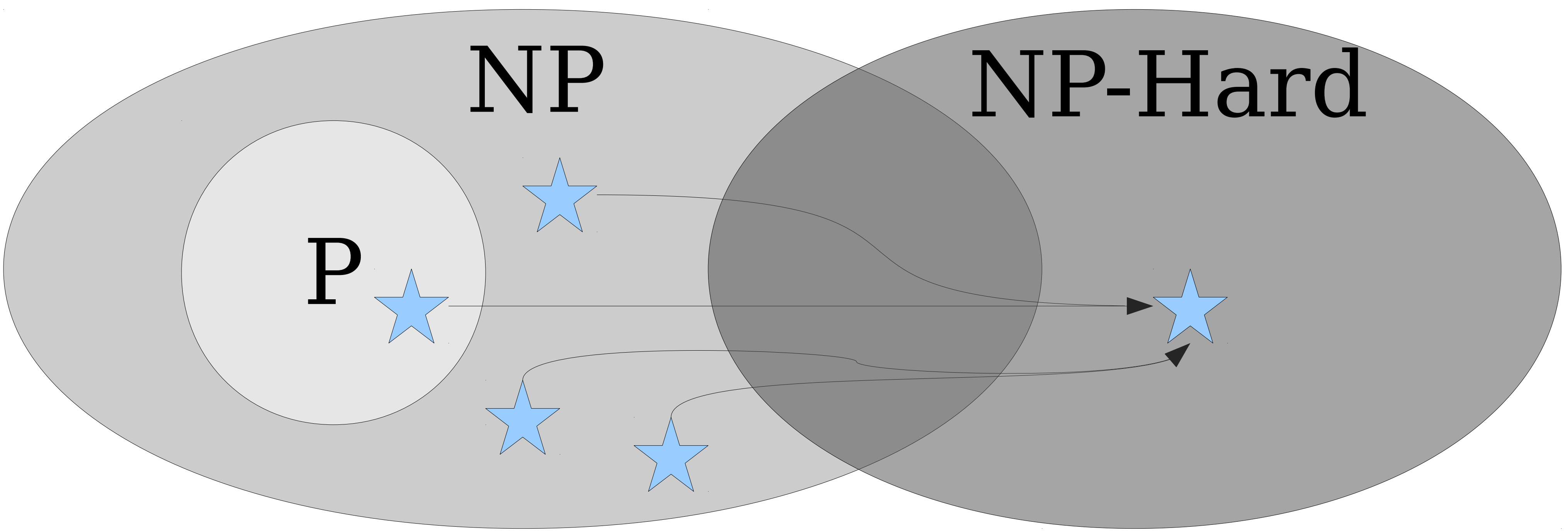
NP-Hardness

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



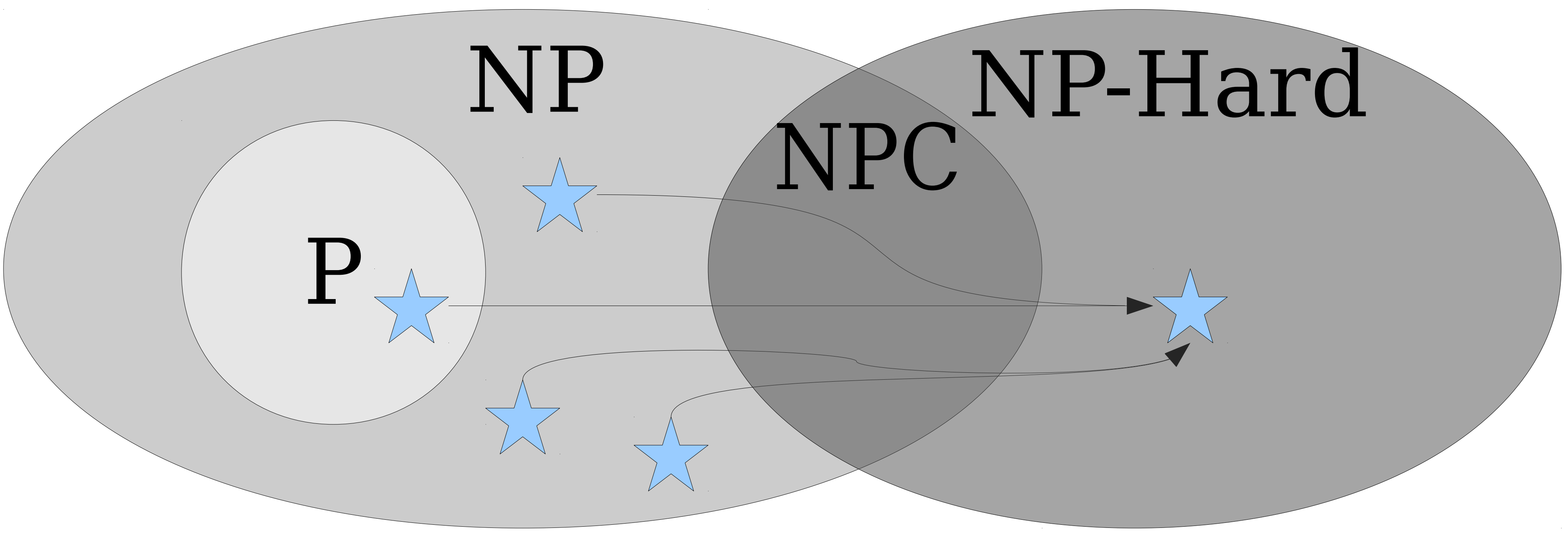
NP-Hardness

- 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 $A \leq_{\mathrm{P}} L$.
- 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.

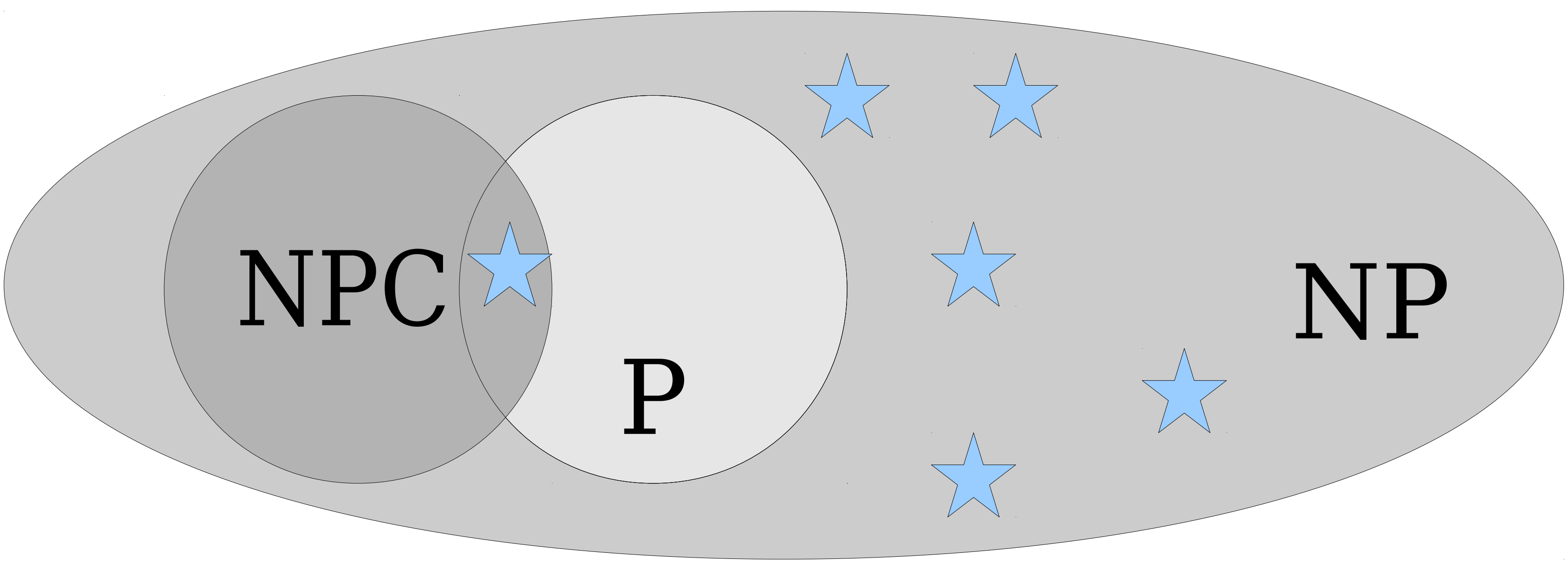


The Tantalizing Truth

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

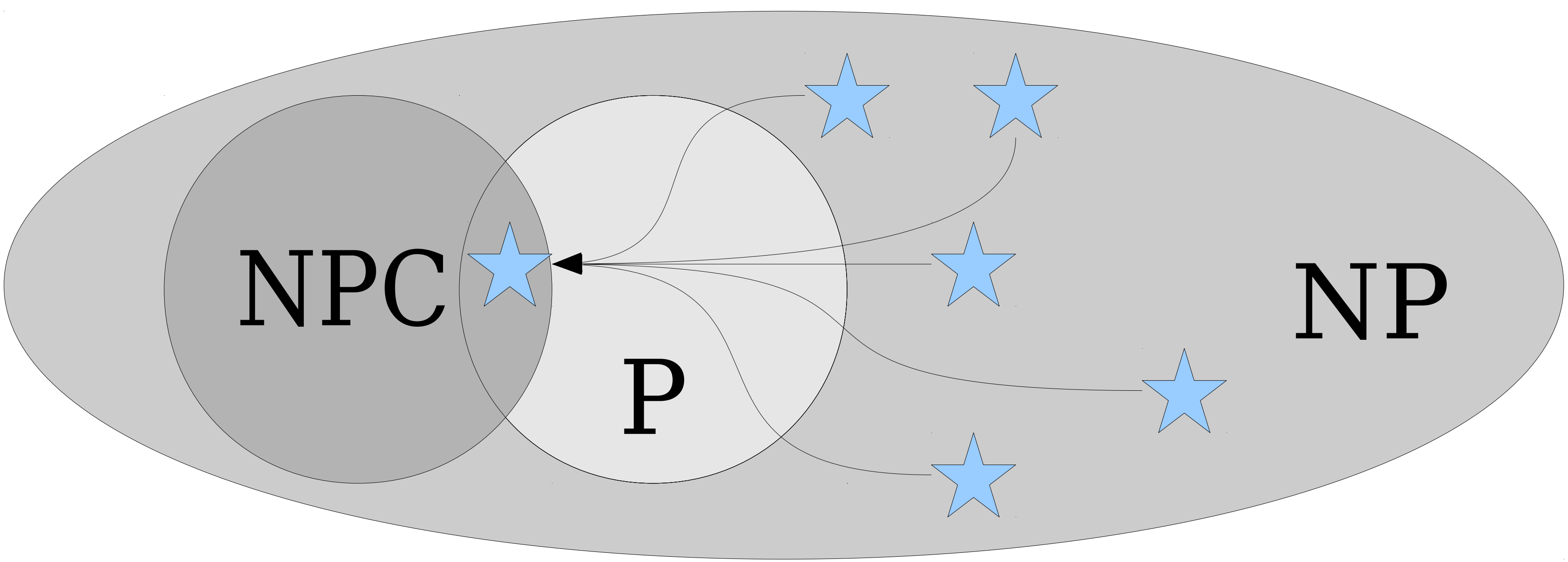
The Tantalizing Truth

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



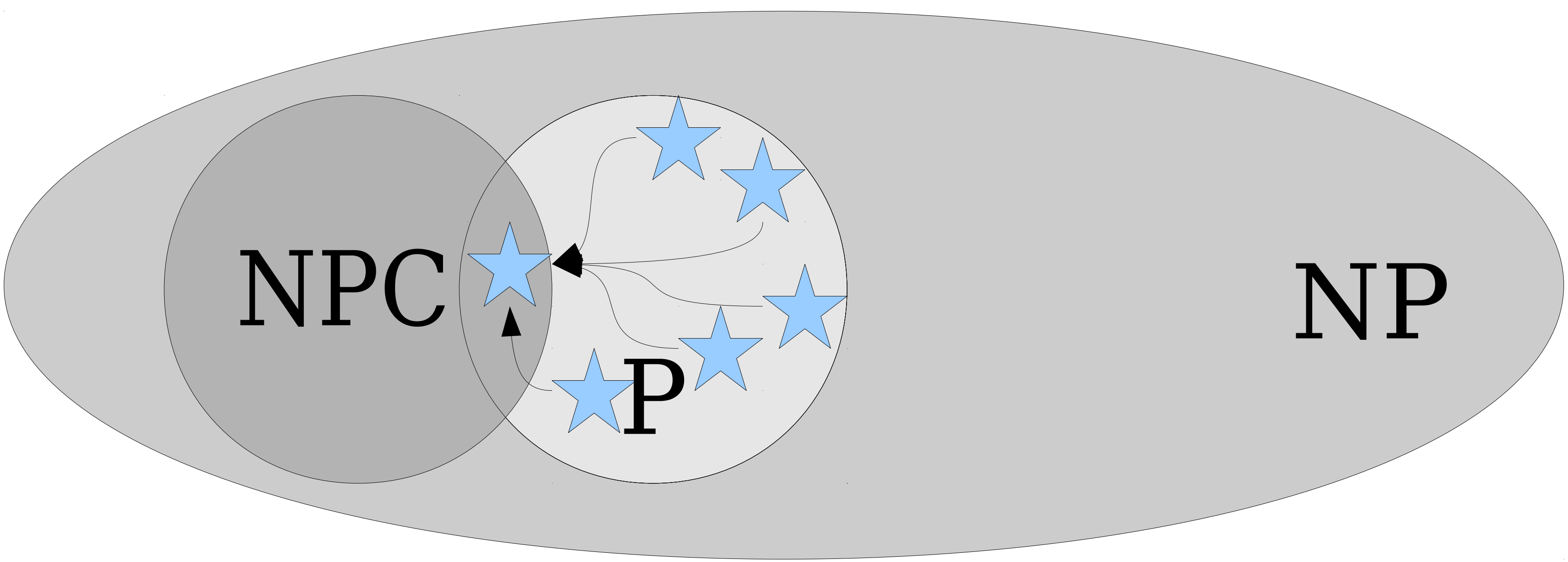
The Tantalizing Truth

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



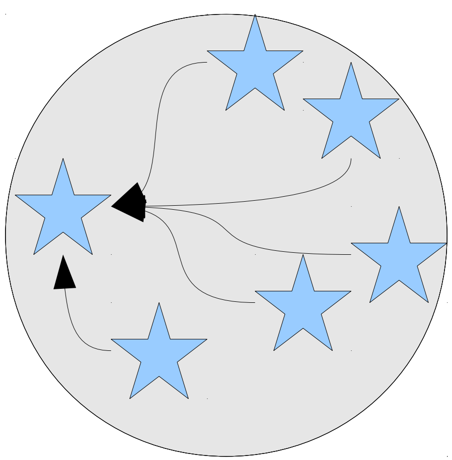
The Tantalizing Truth

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**.

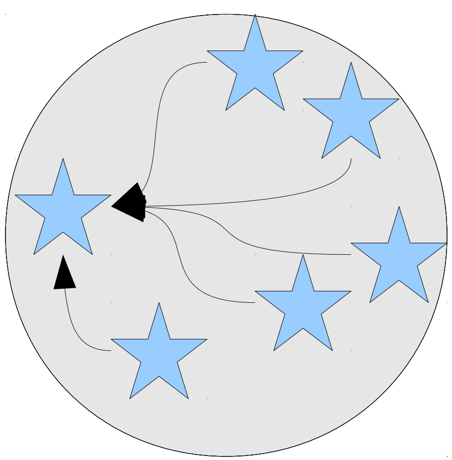


P = NP

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**. ■

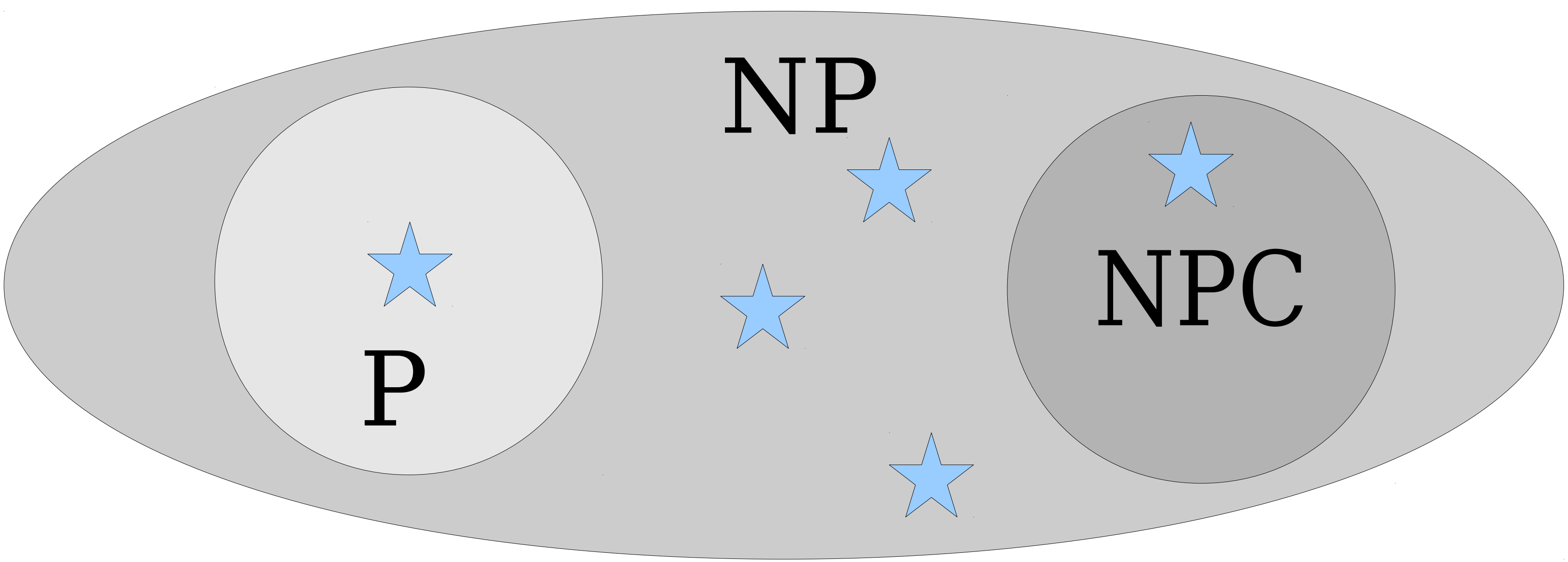


P = NP

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:

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

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

Proof Idea: 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 $\mathbf{P} \neq \mathbf{NP}$.
- 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.

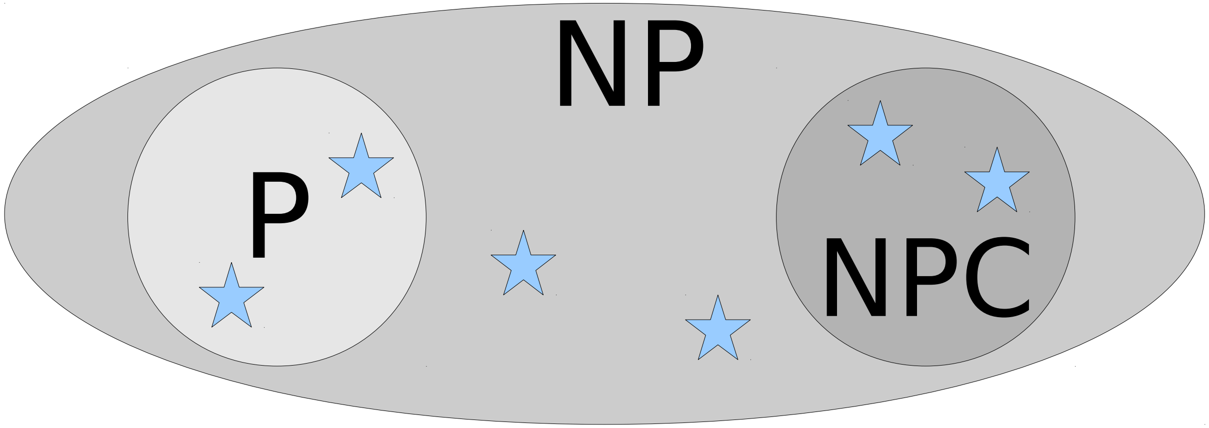
Sample **NP**-Hard Problems

- ***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, 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

Next Time

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