Complexity Theory

Part Two

Recap from Last Time

The Complexity Class **P**

- The complexity class **P** (***polynomial time***) is defined as

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

- Intuitively, **P** contains all decision problems that can be solved efficiently.
- This is like class **R**, except with "efficiently" tacked onto the end.

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* }

- Intuitively, **NP** is the set of problems where "yes" answers can be checked efficiently.
- This is like the class **RE**, but with "efficiently" tacked on to the definition.

The Biggest Unsolved Problem in Theoretical Computer Science:

$$\mathbf{P} \stackrel{?}{=} \mathbf{NP}$$

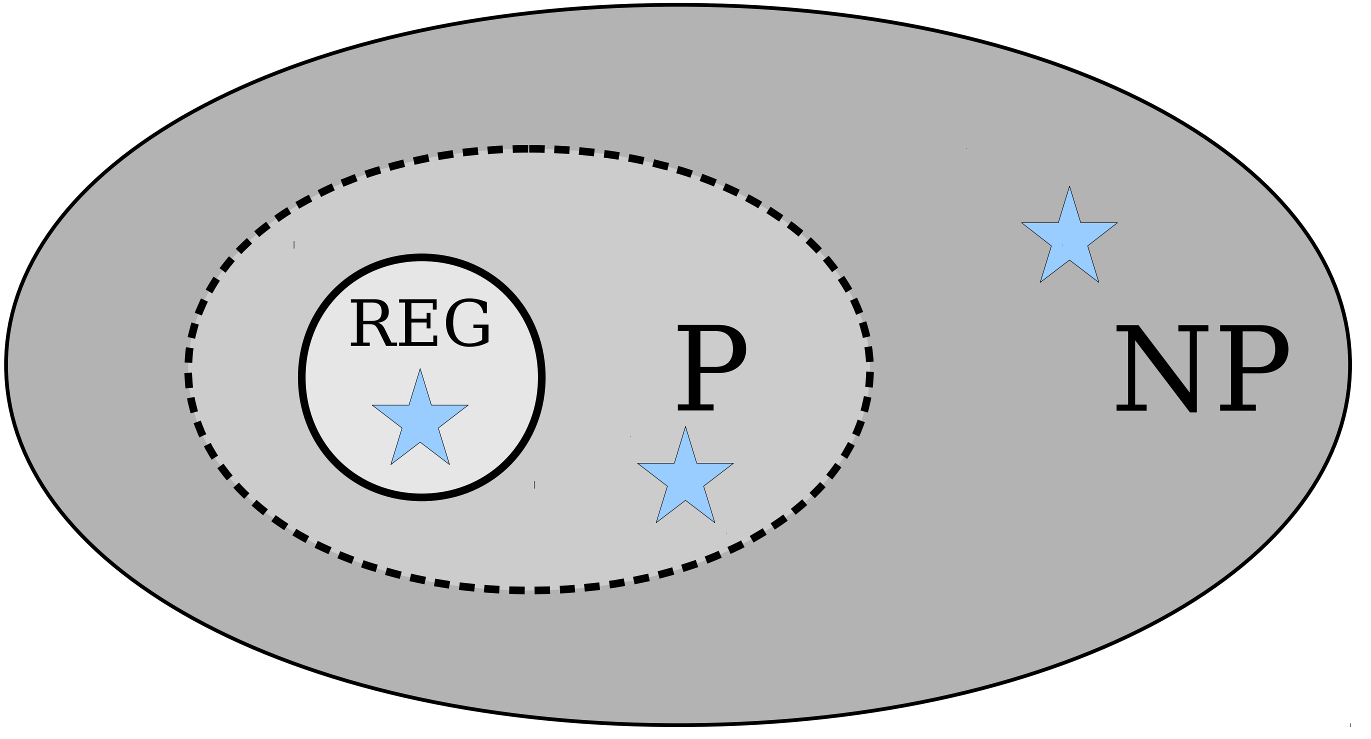
Theorem (Baker-Gill-Solovay): Any proof that purely relies on universality and self-reference cannot resolve $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$.

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?

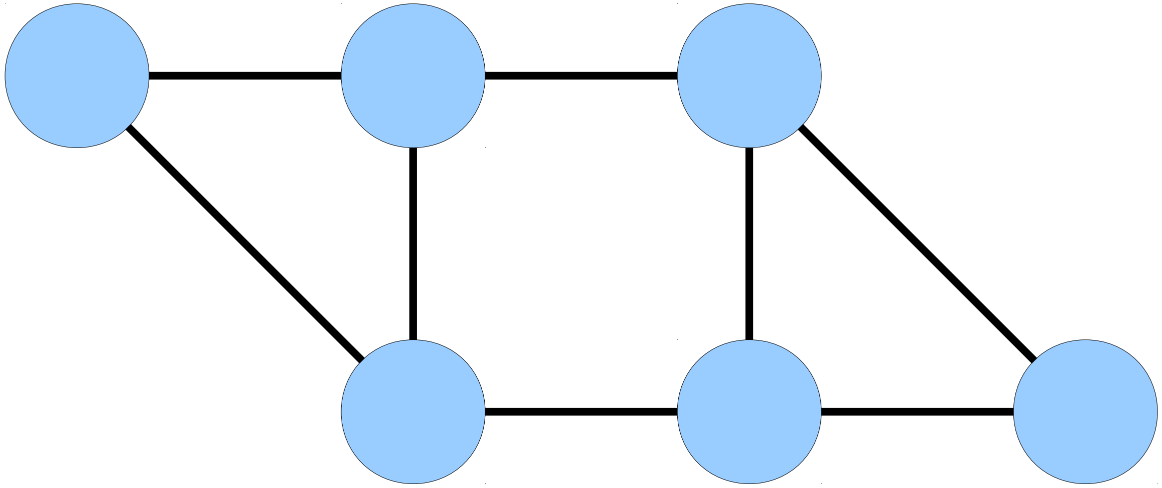
Reducibility

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.

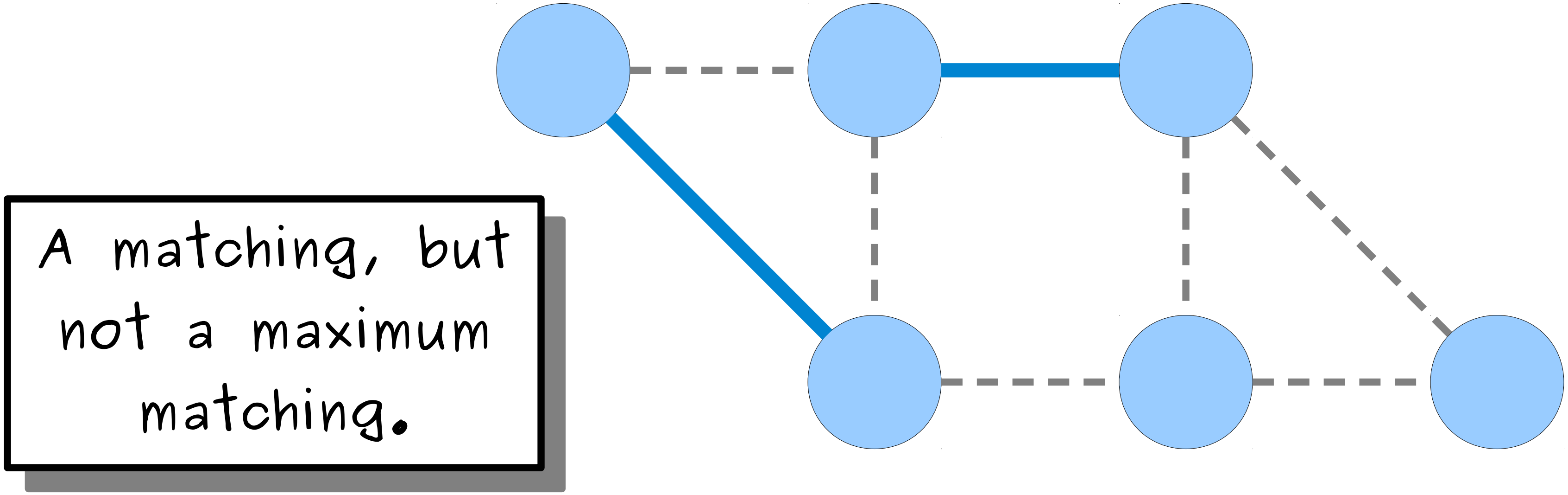
Maximum Matching

- 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.



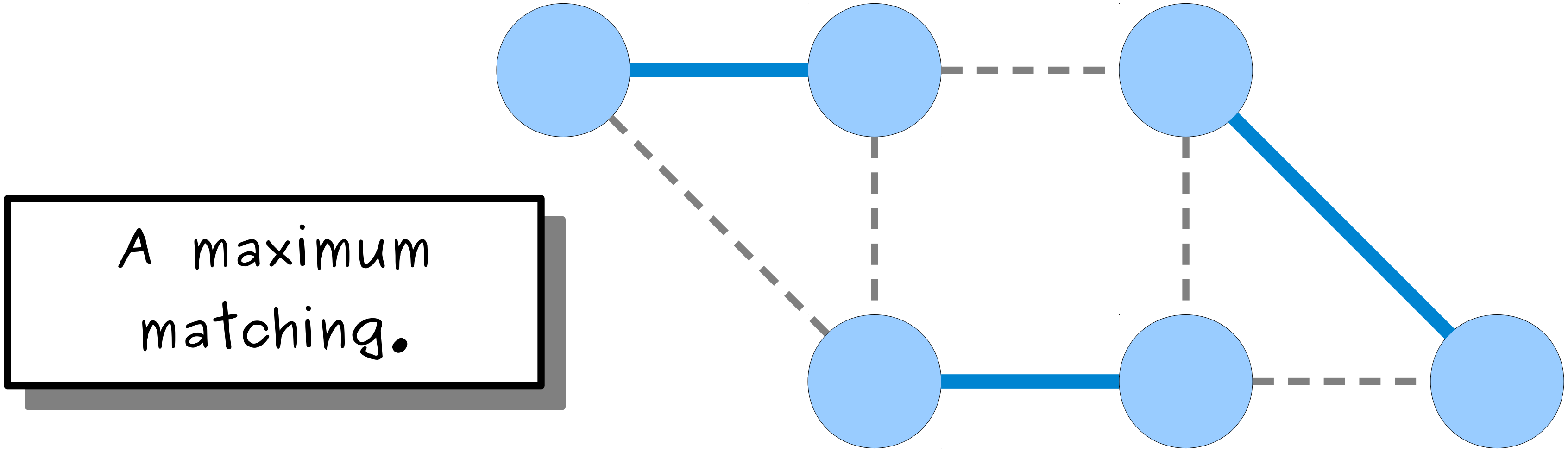
Maximum Matching

- 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.



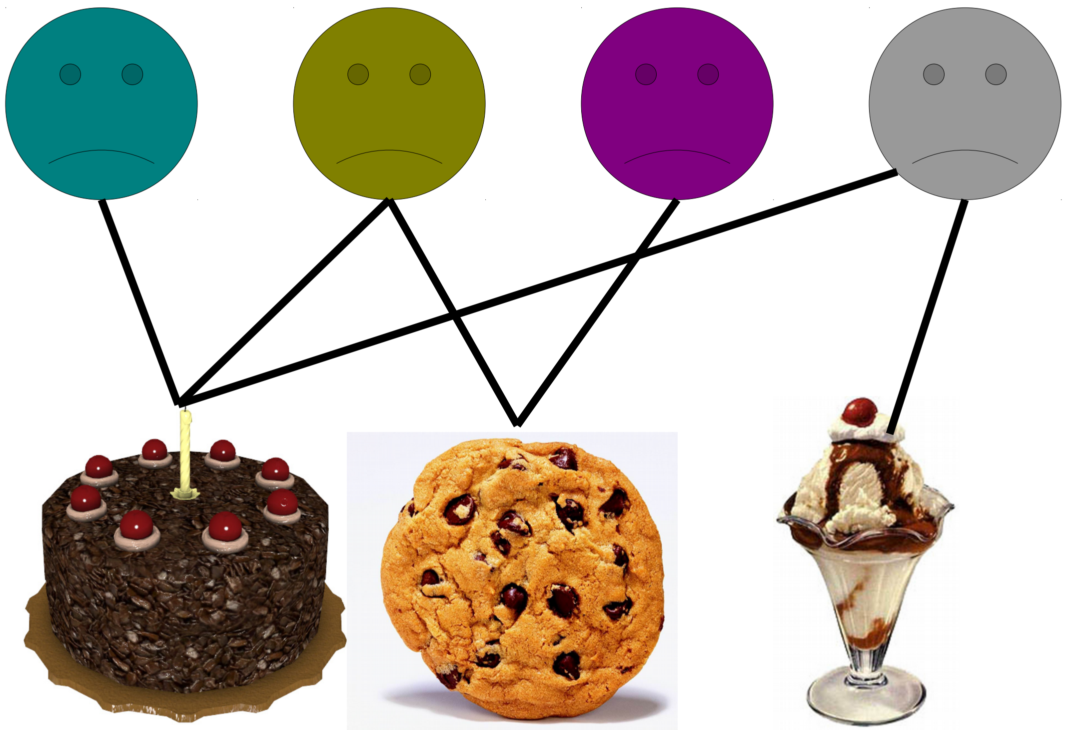
Maximum Matching

- 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.



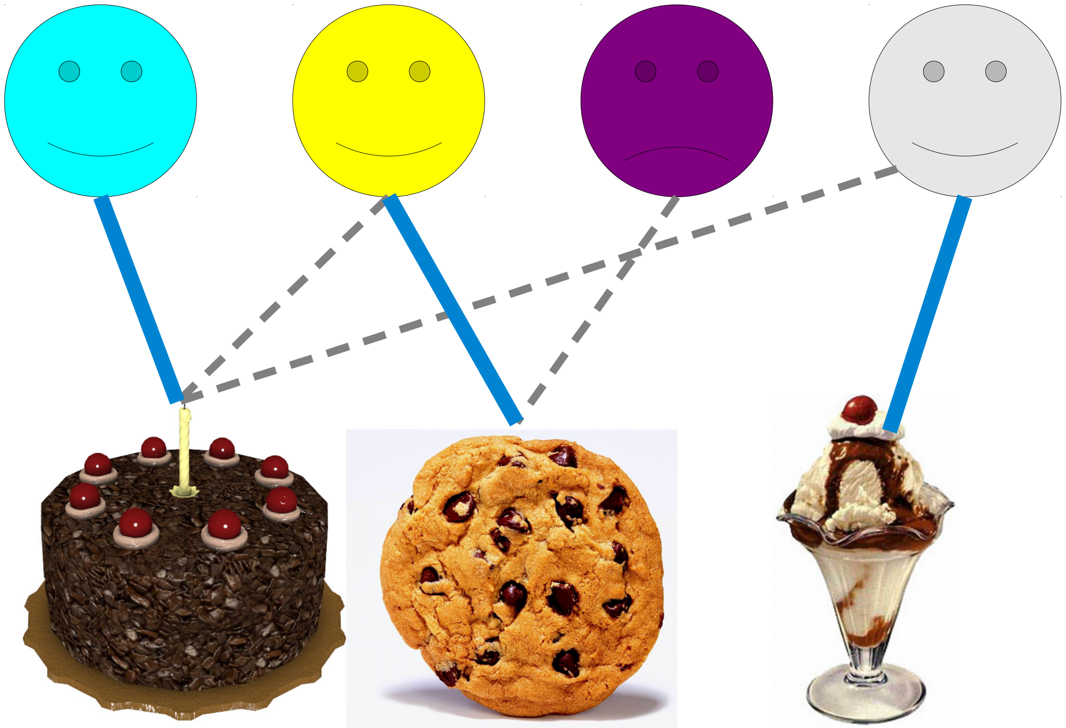
Maximum Matching

- 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.



Maximum Matching

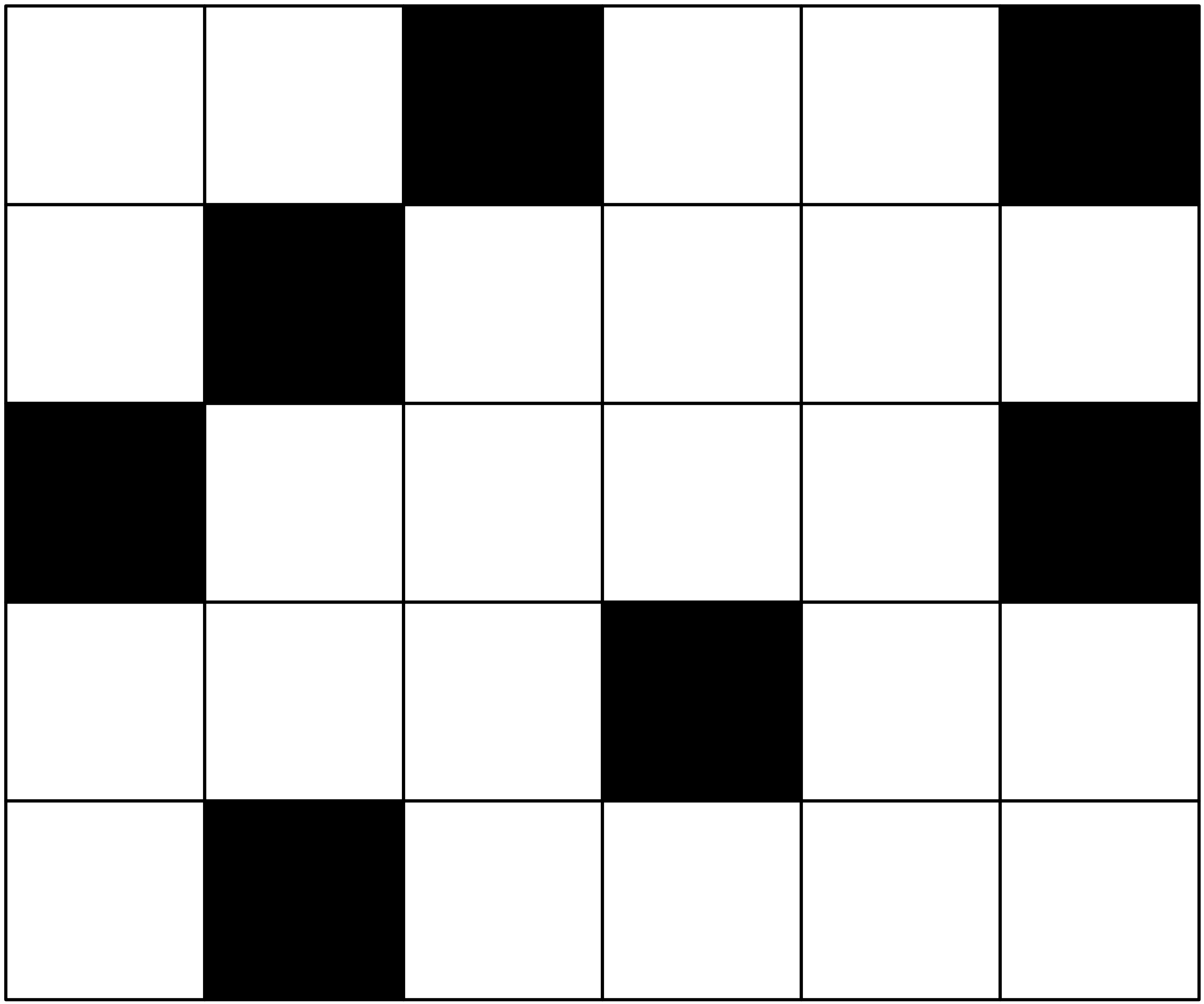
- Given an undirected graph G, a ***matching*** in G is a set of edges such that no two edges share an endpoint.
- A ***maximum matching*** is a matching with the largest number of edges.



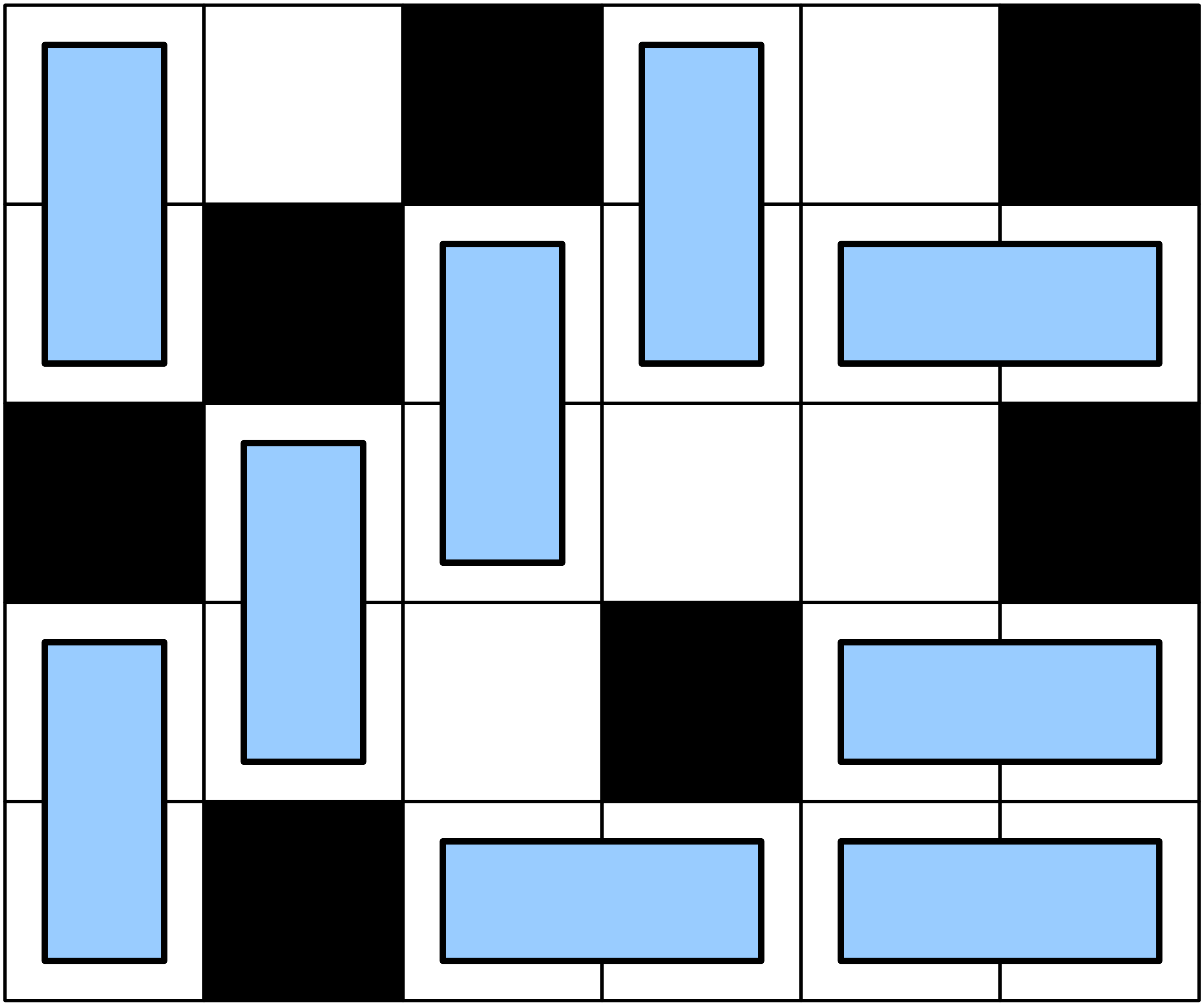
Maximum Matching

- Jack Edmonds' paper “Paths, Trees, and Flowers” gives a polynomial-time algorithm for finding maximum matchings.
  - He’s the guy from last time with the quote about “better than decidable.”
- Using this fact, what other problems can we solve?

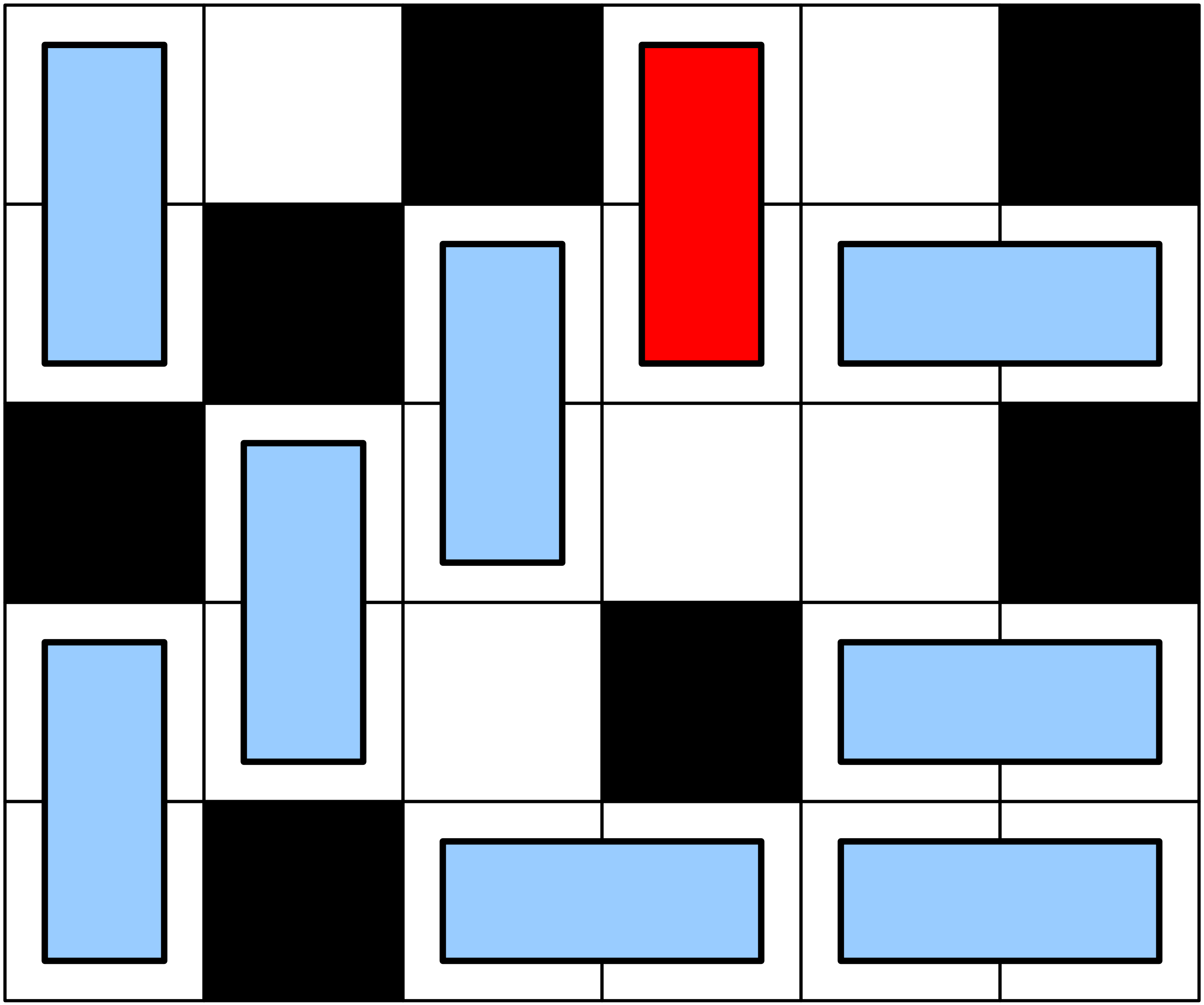
Domino Tiling



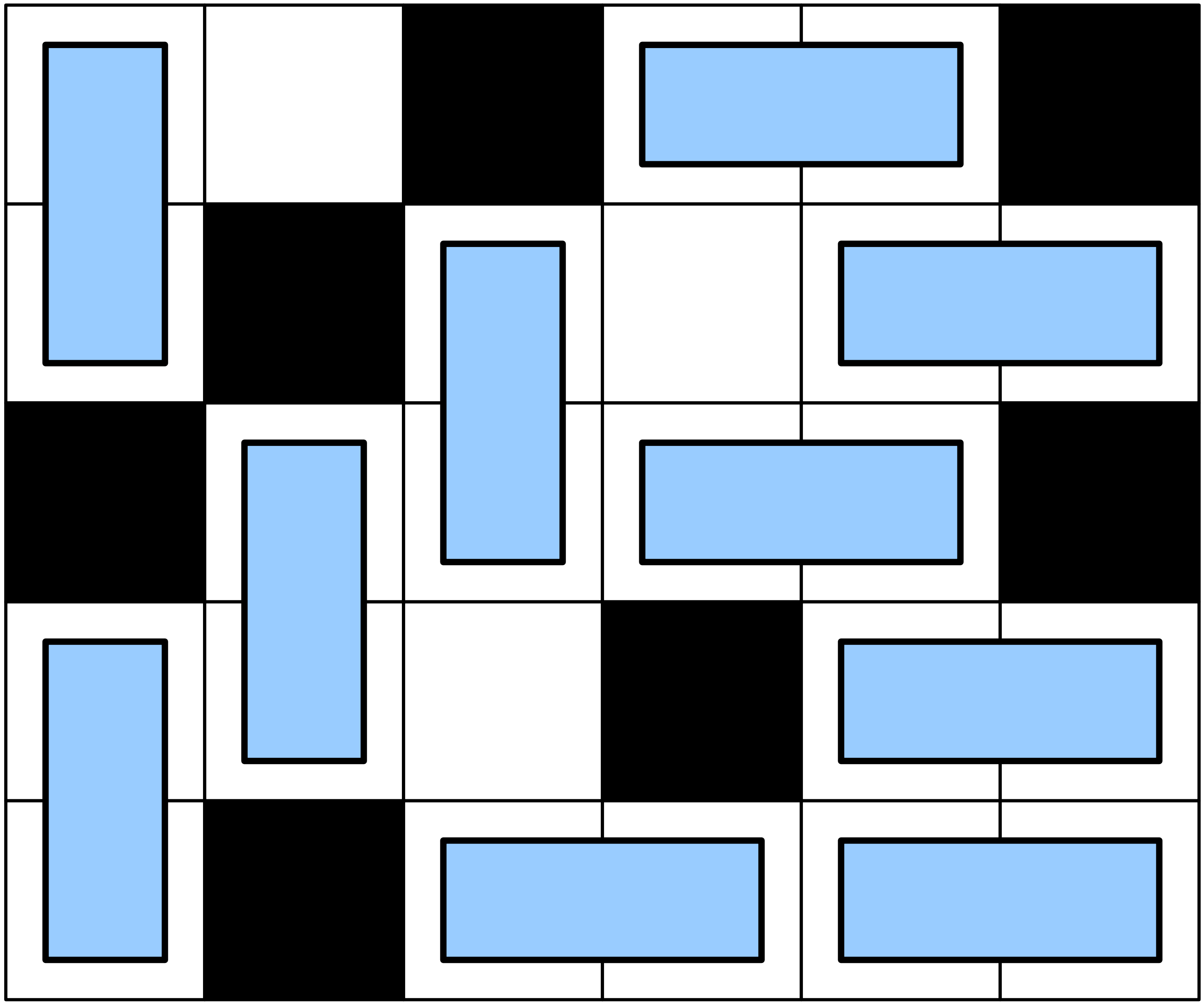
Domino Tiling



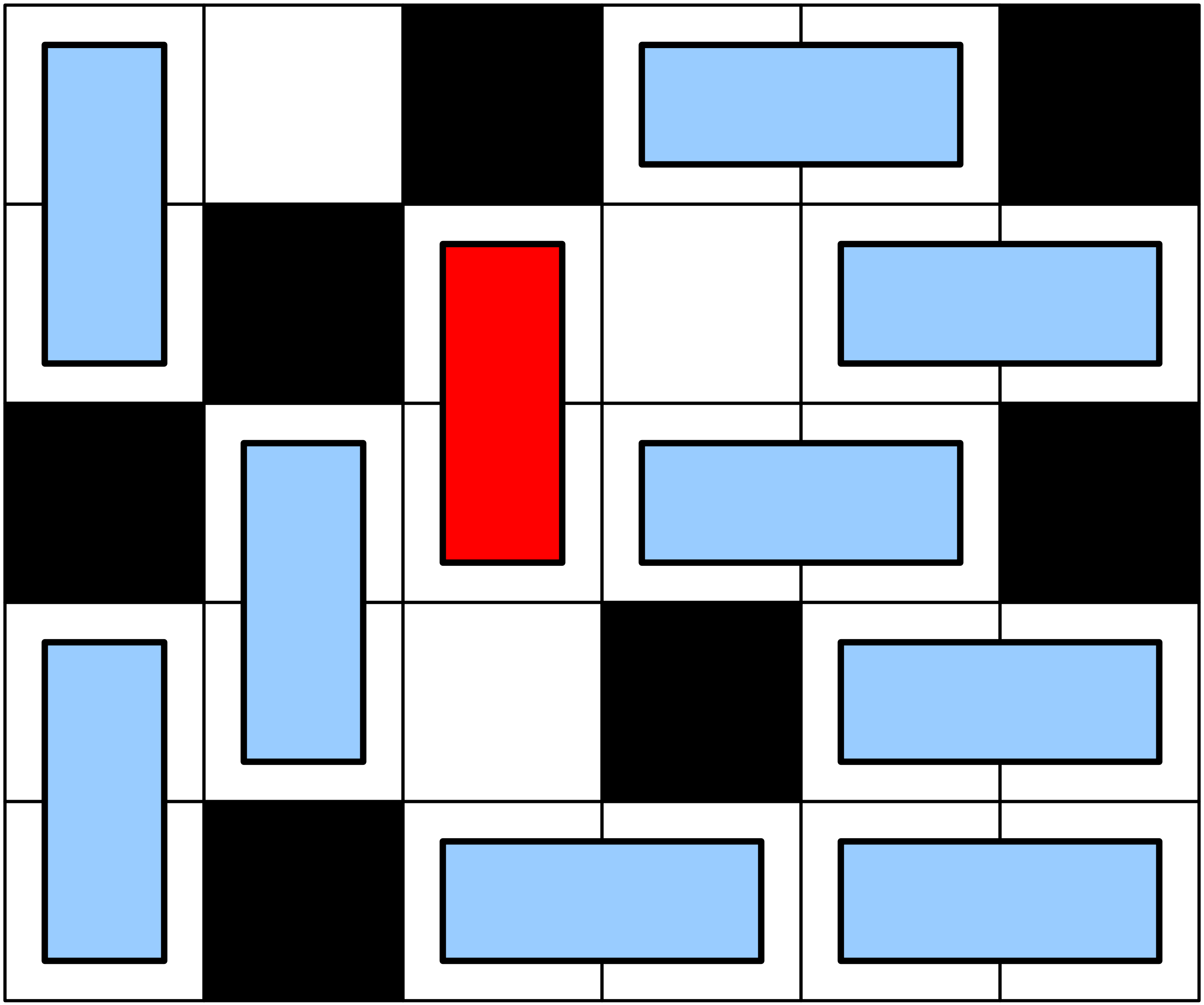
Domino Tiling



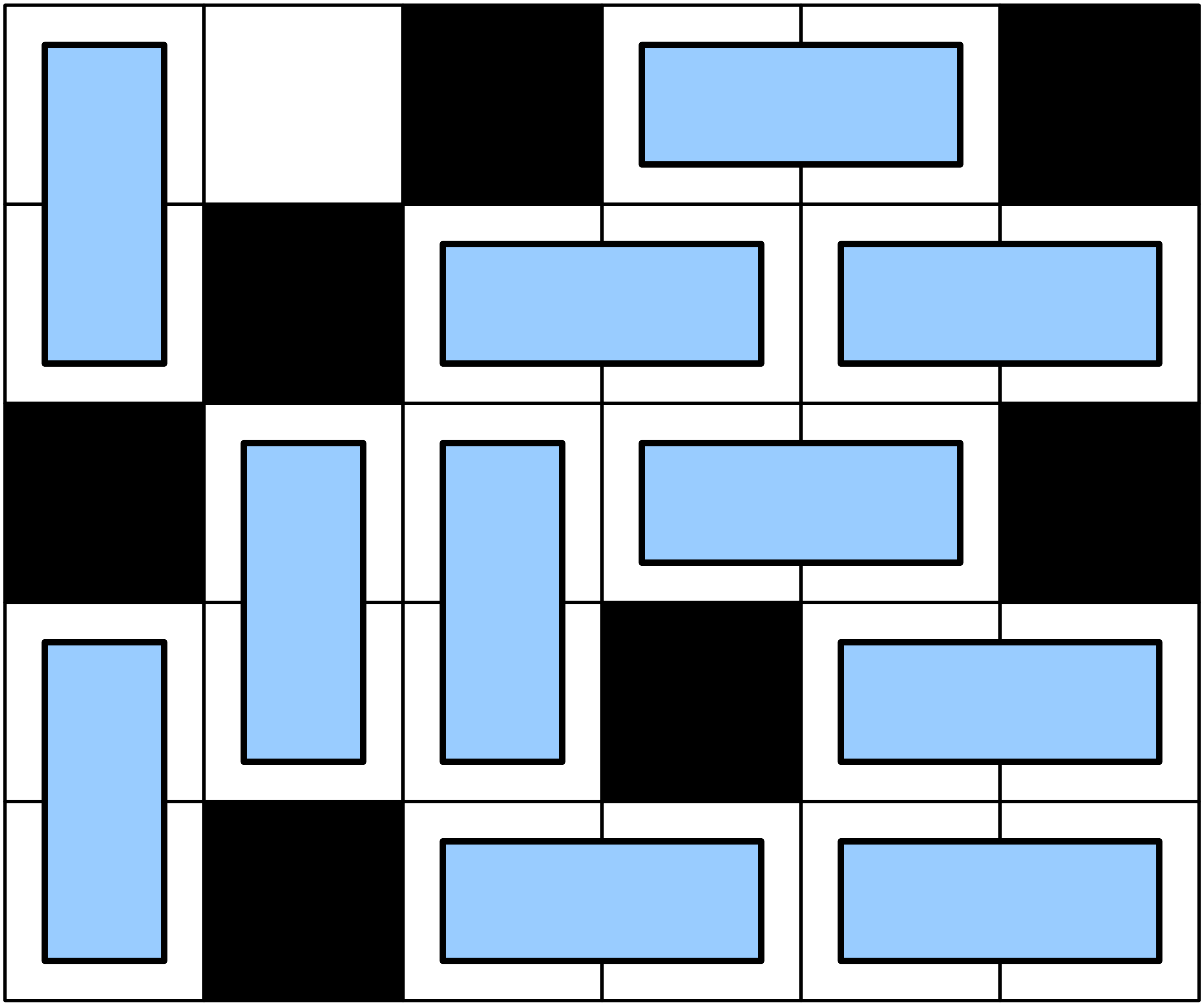
Domino Tiling



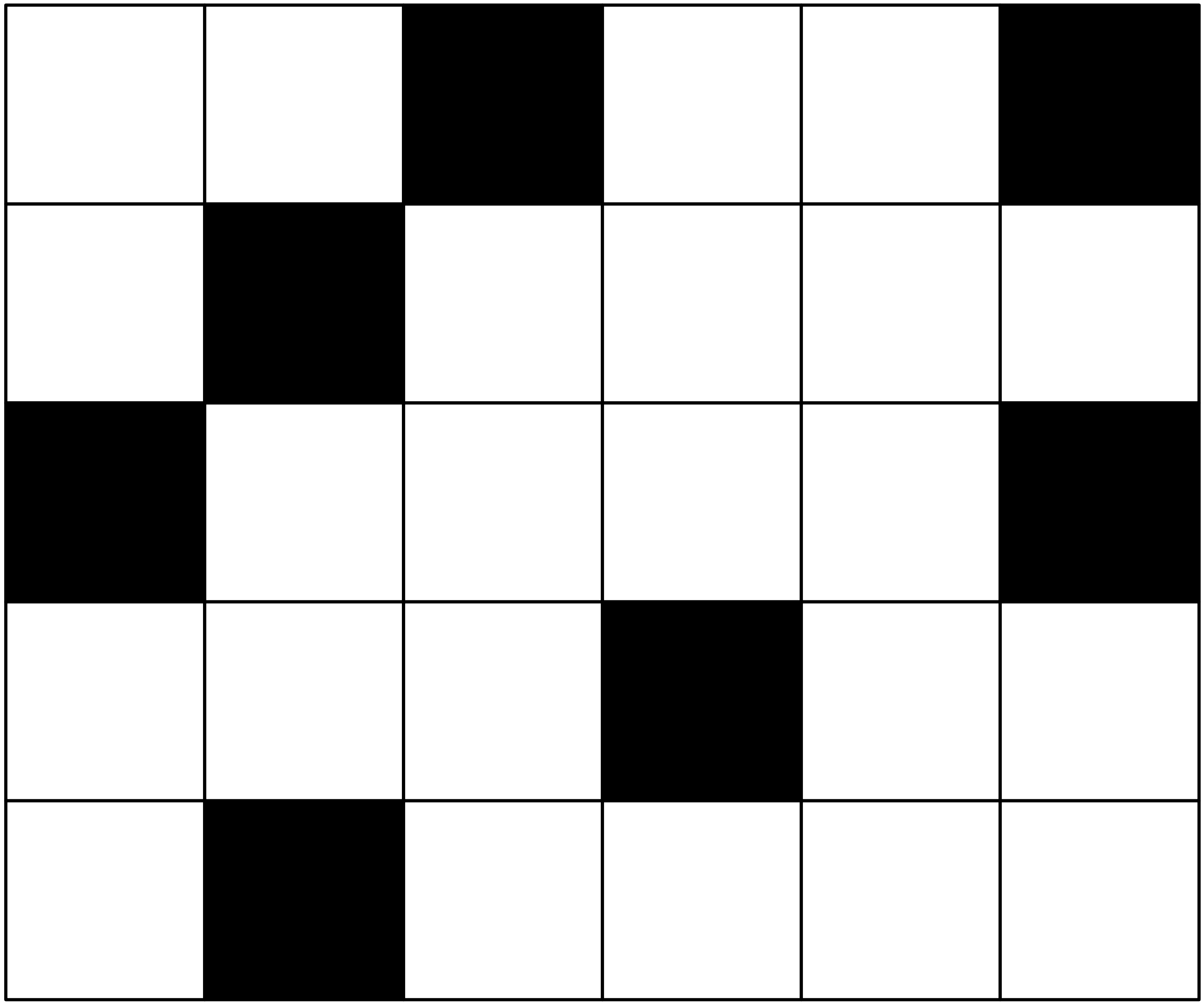
Domino Tiling



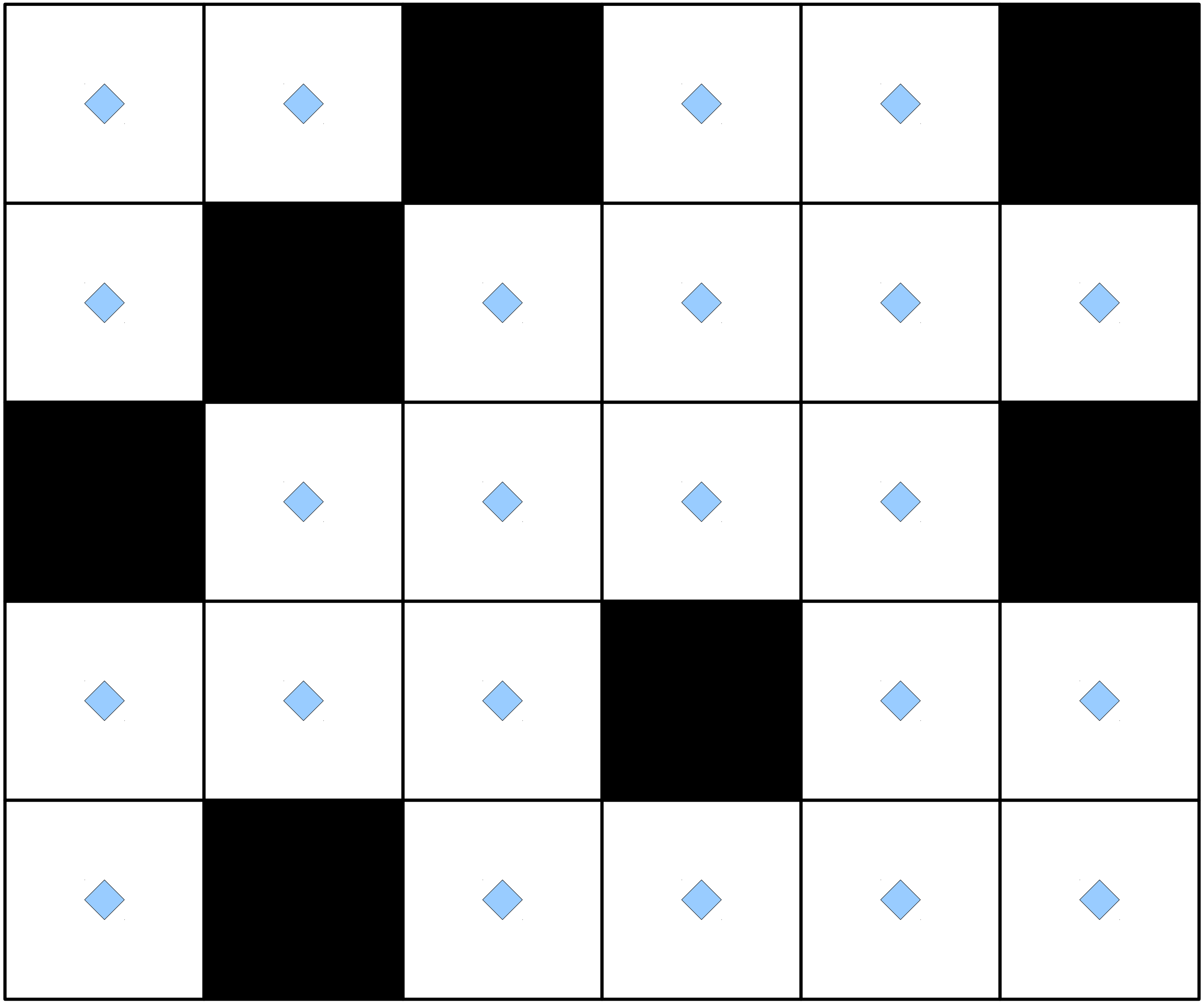
Domino Tiling



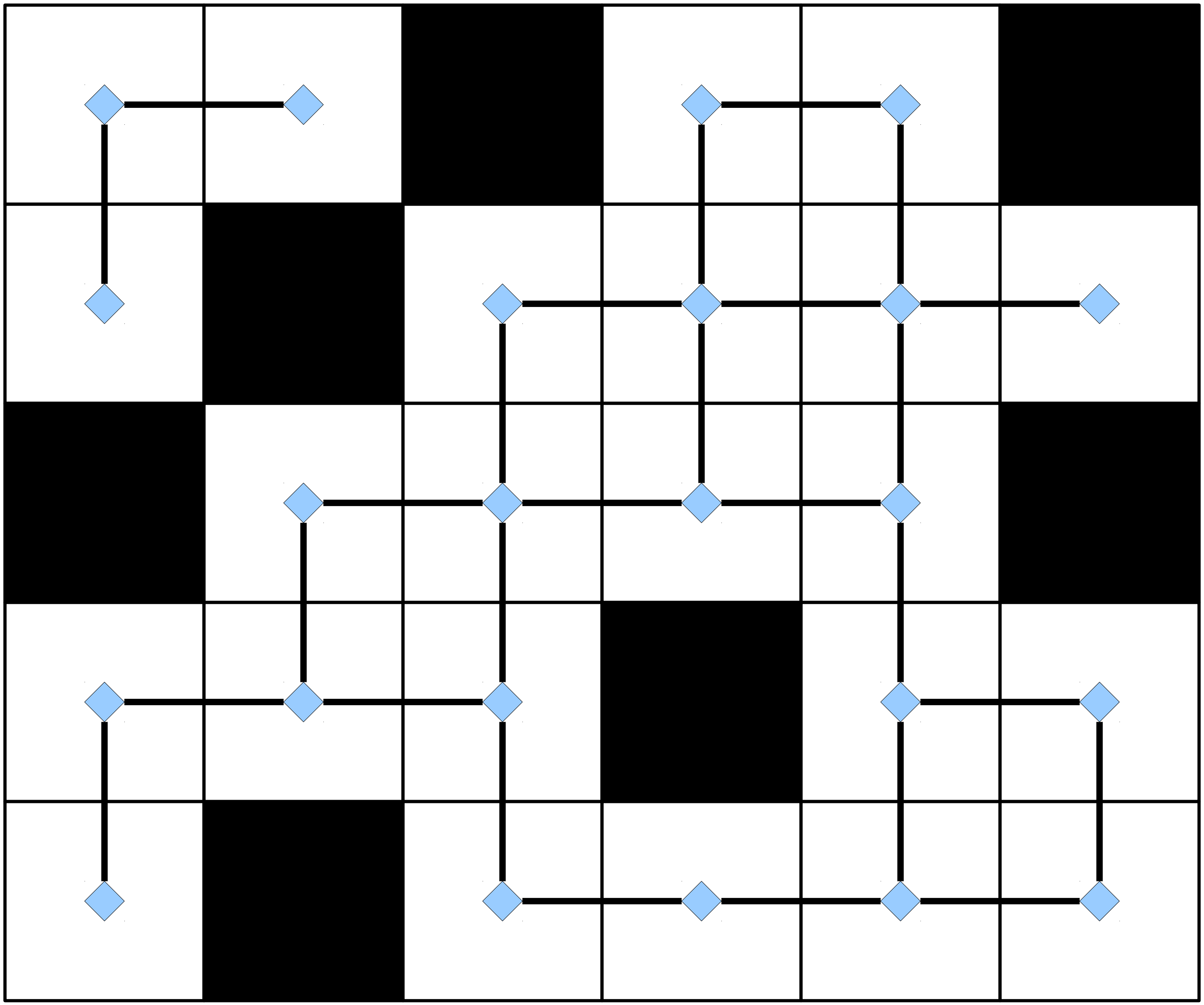
Solving Domino Tiling



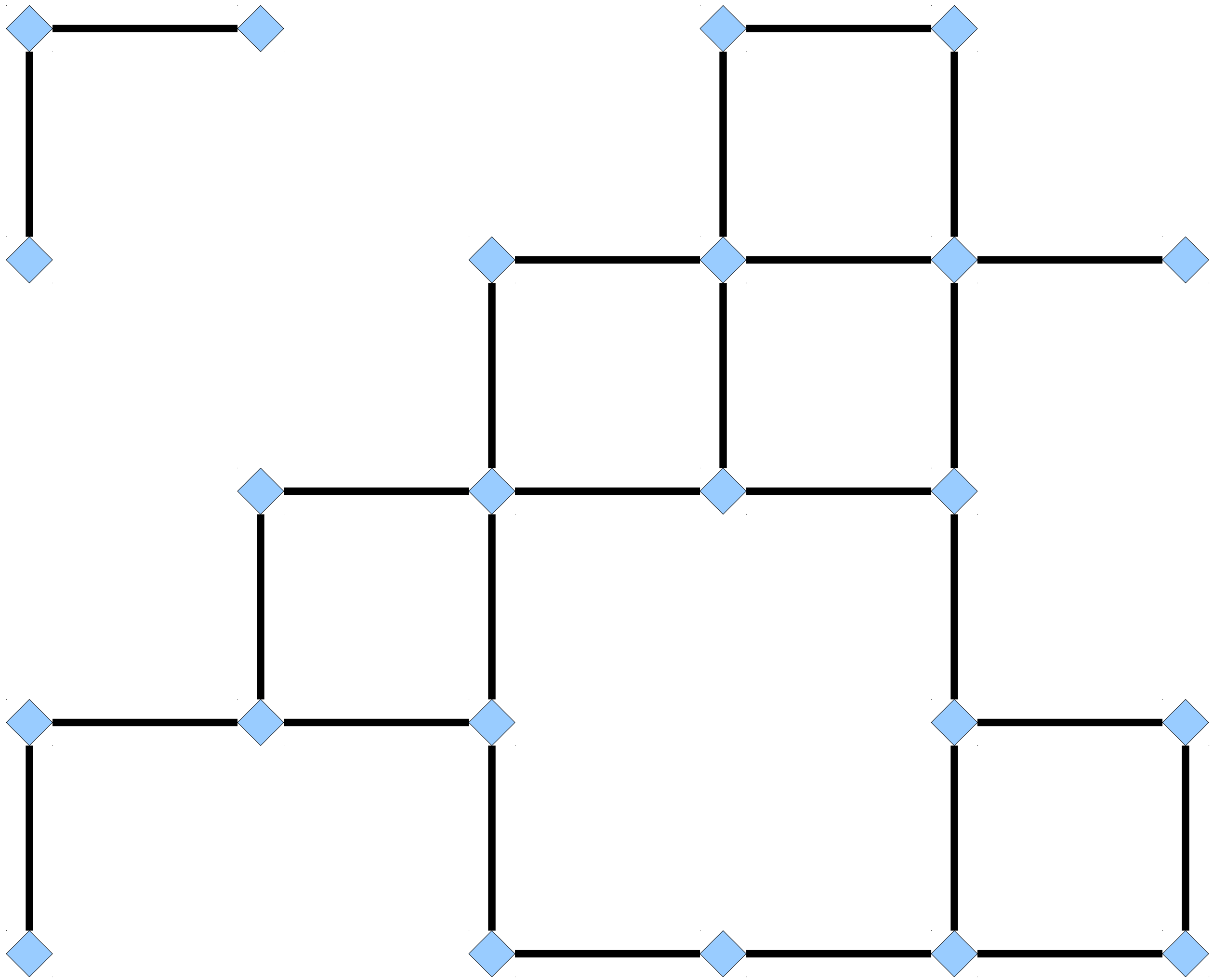
Solving Domino Tiling



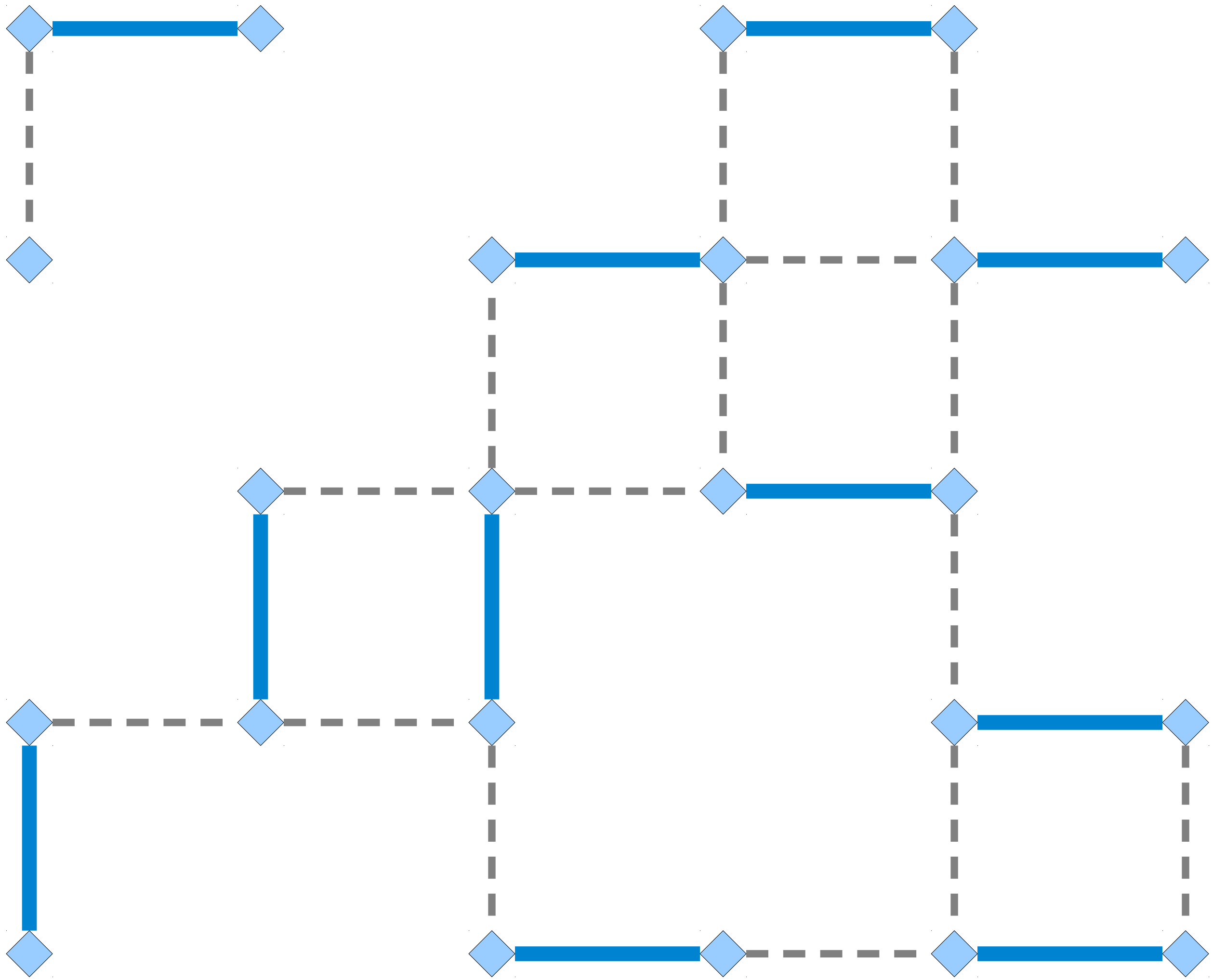
Solving Domino Tiling



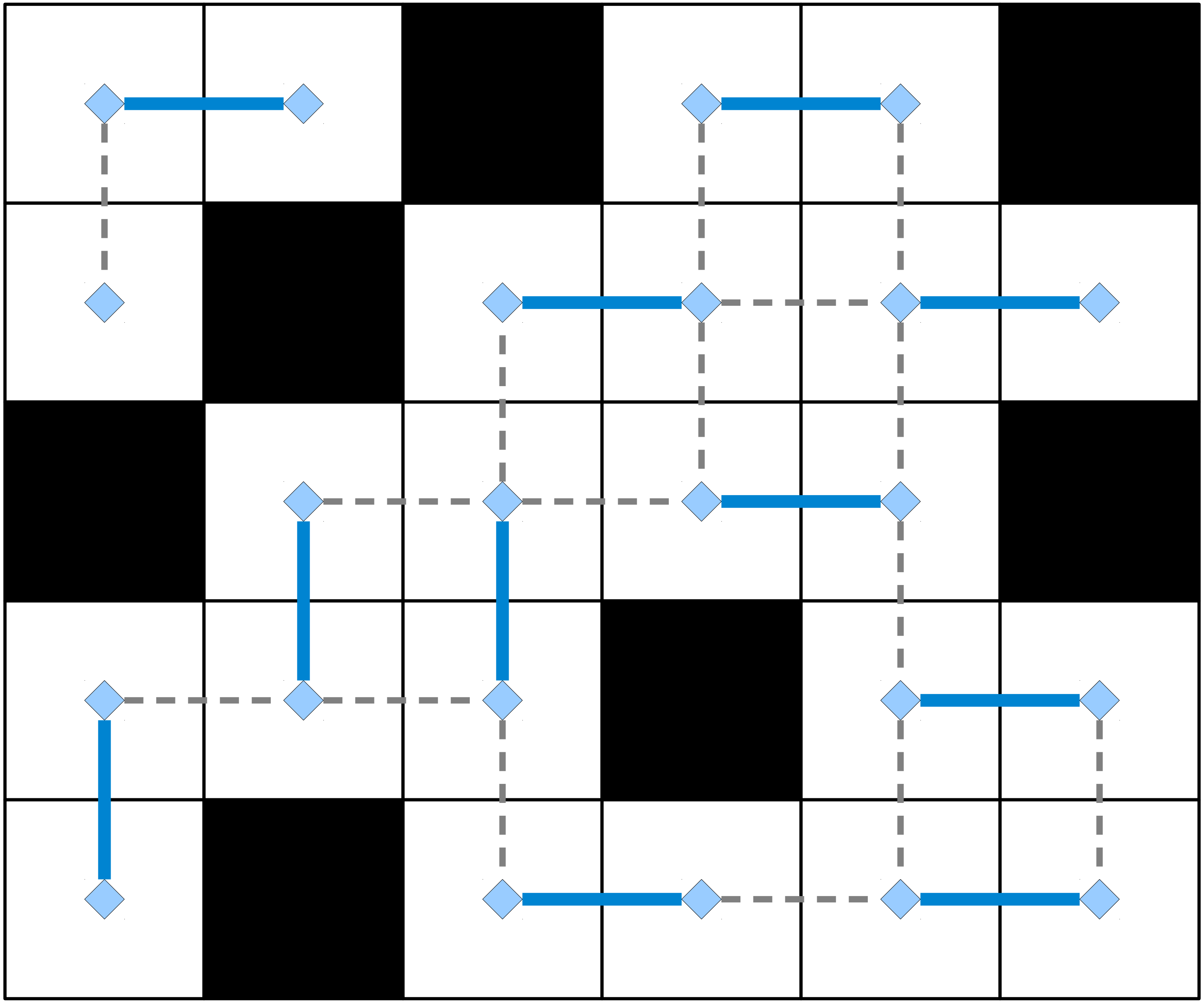
Solving Domino Tiling



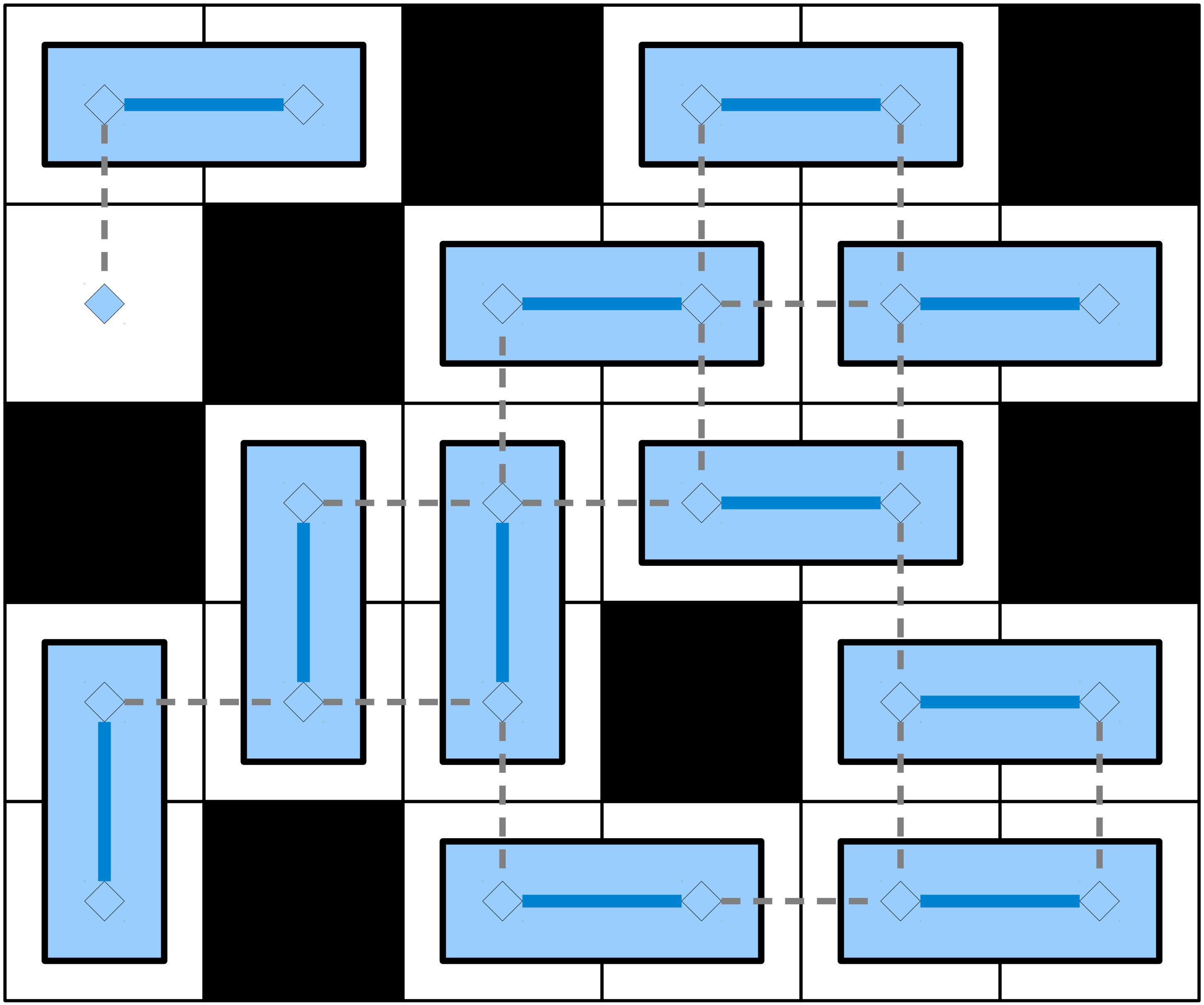
Solving Domino Tiling



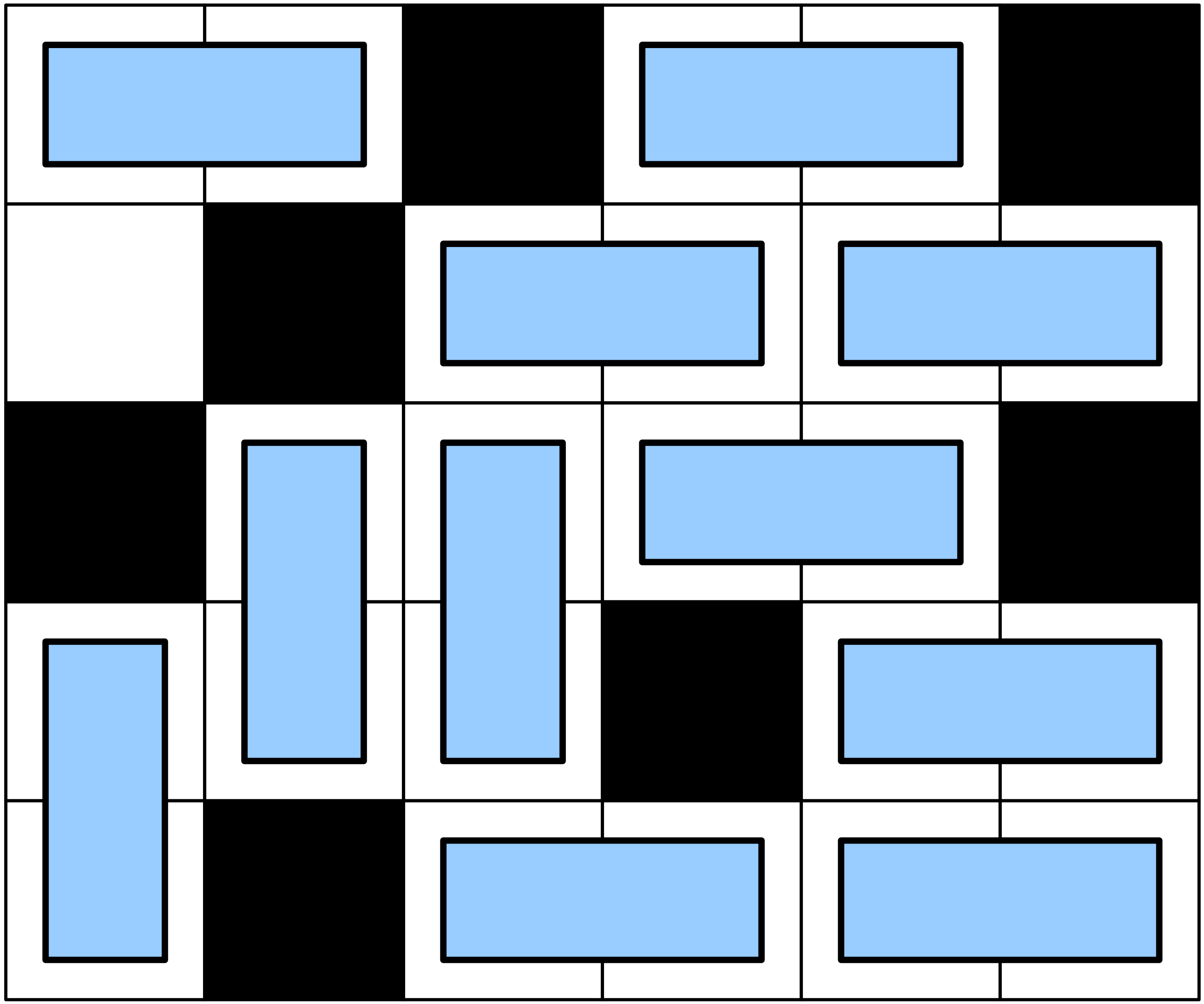
Solving Domino Tiling



Solving Domino Tiling



Solving Domino Tiling



```
bool canPlaceDominoes(Grid G, int k) {
    return hasMatching(gridToGraph(G), k);
}
```

Which of the following is the most reasonable conclusion to draw, given the existence of the above function?

A. Solving domino tiling can't be "harder" than solving maximum matching.

B. Solving maximum matching can't be "harder" than solving domino tiling.

C. Both A and B.

Answer at
https://pollev.com/cs103

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*?

- This problem can be solved in polynomial time (use DFS or BFS).

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

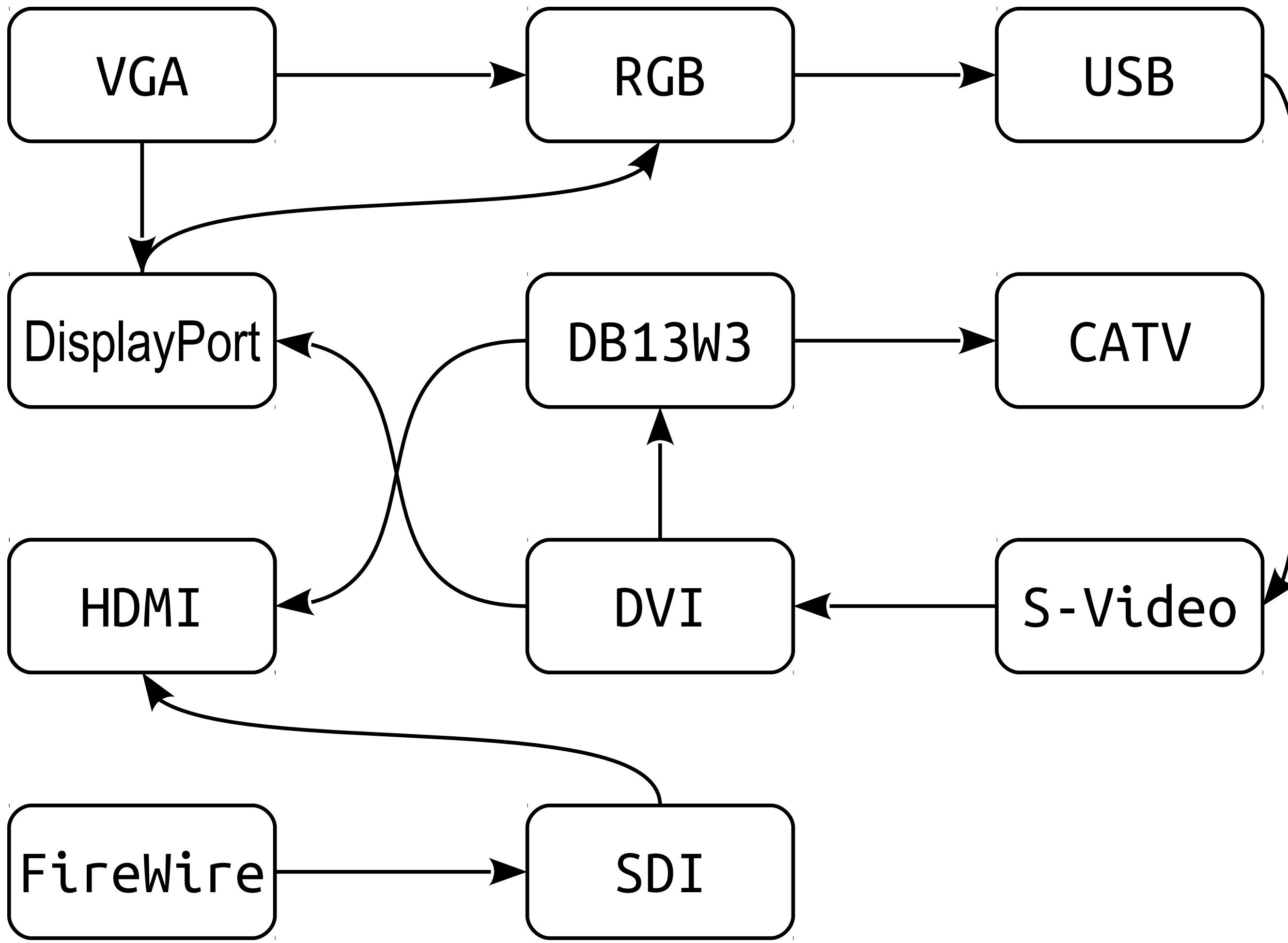
Connectors

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

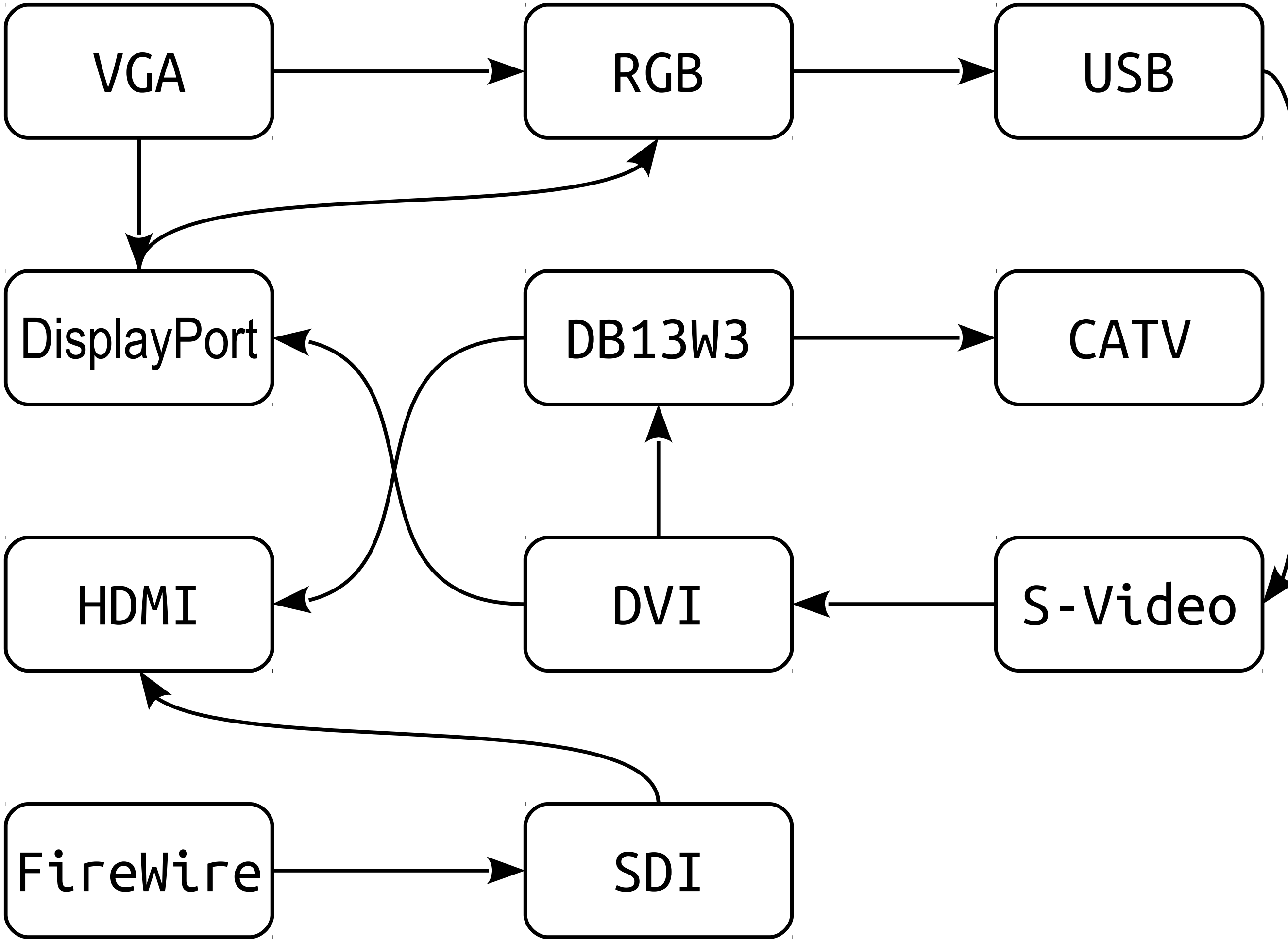
Converter Conundrums

Connectors

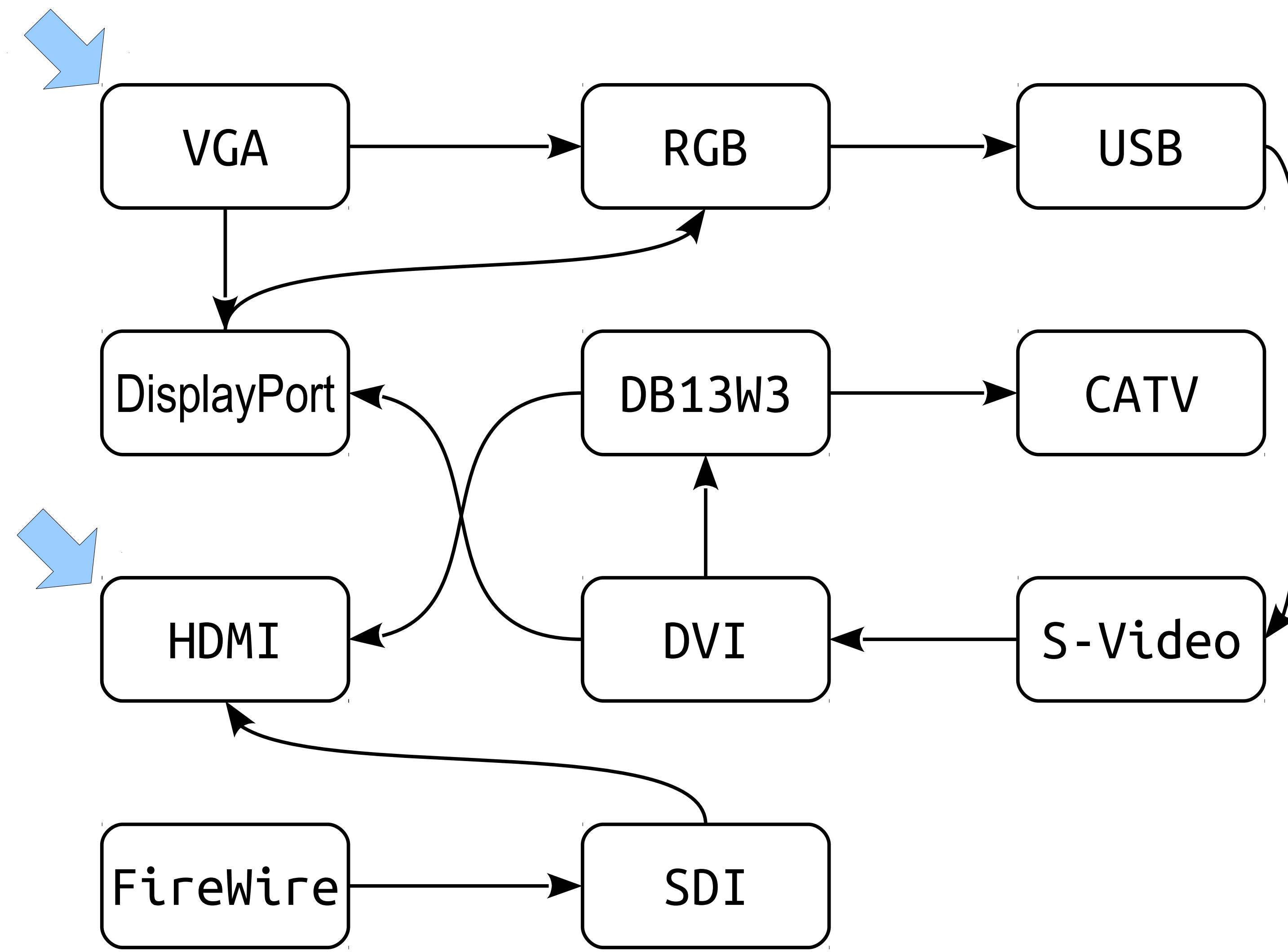
RGB to USB
VGA to DisplayPort
DB13W3 to CATV
DisplayPort to RGB
DB13W3 to HDMI
DVI to DB13W3
S-Video to DVI
FireWire to SDI
VGA to RGB
DVI to DisplayPort
USB to S-Video
SDI to HDMI



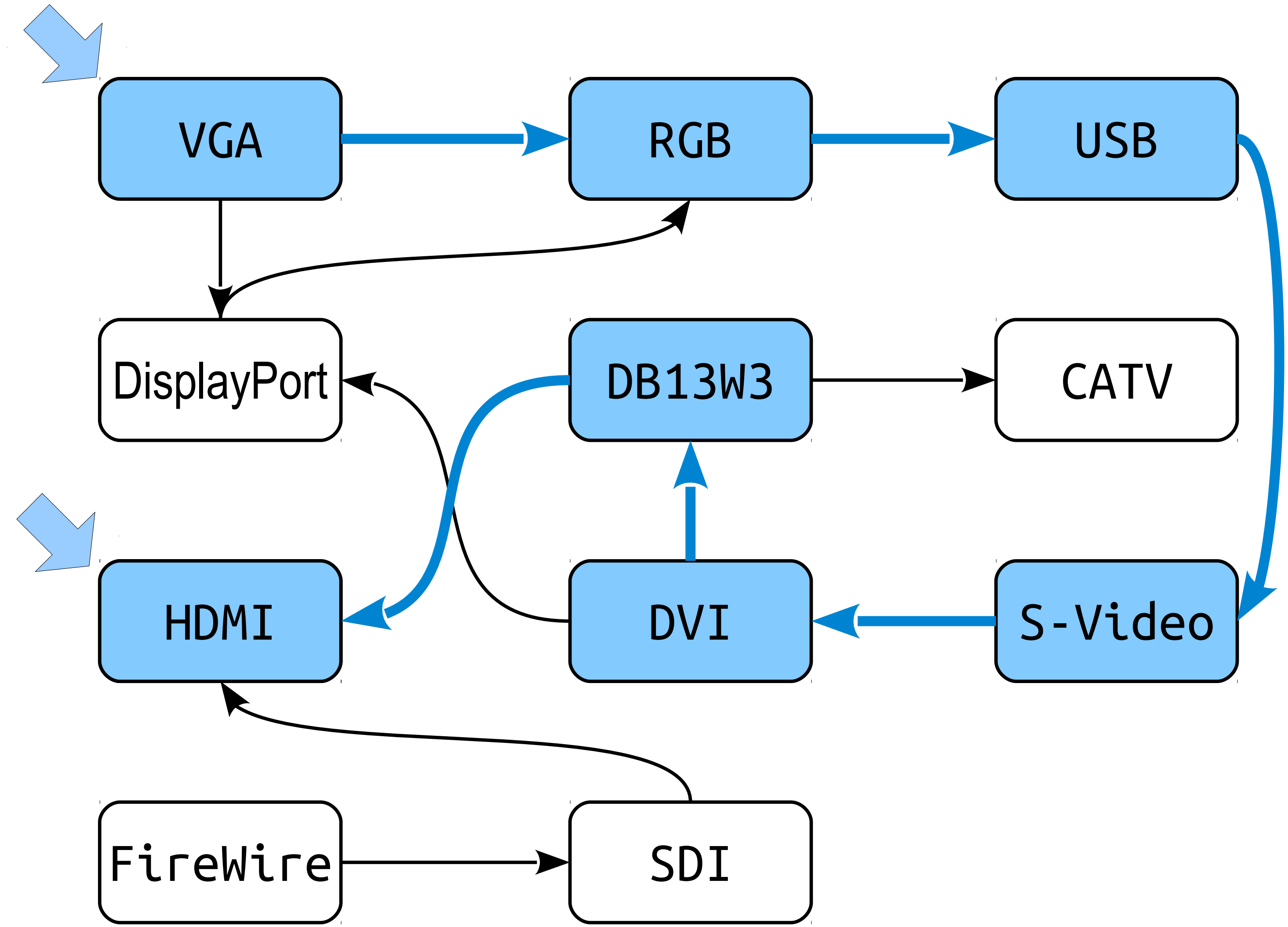
Converter Conundrums



Converter Conundrums



Converter Conundrums



In Pseudocode

```
bool canPlugIn(vector<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(translate(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(translate(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***.

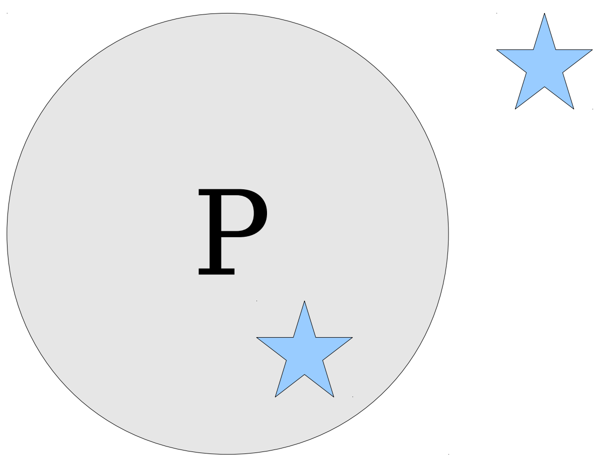
* Assuming that `translate` runs in polynomial time.

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

- This is a powerful general problem-solving technique. You'll see it a lot in CS161.

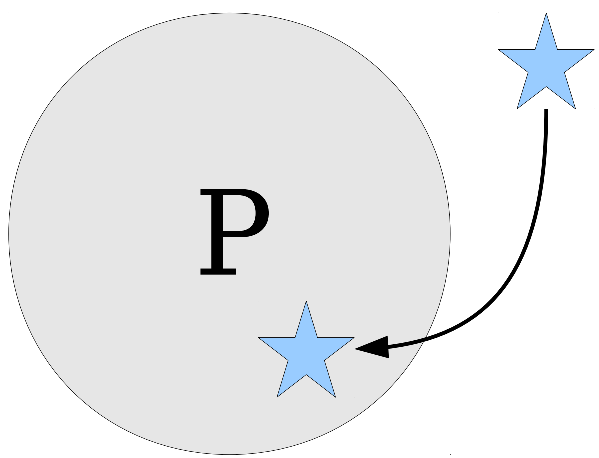
Polynomial-Time Reductions

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



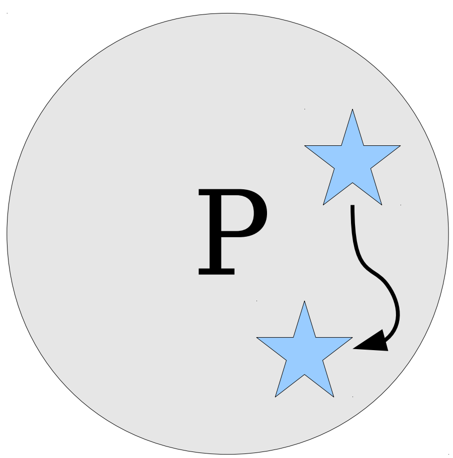
Polynomial-Time Reductions

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



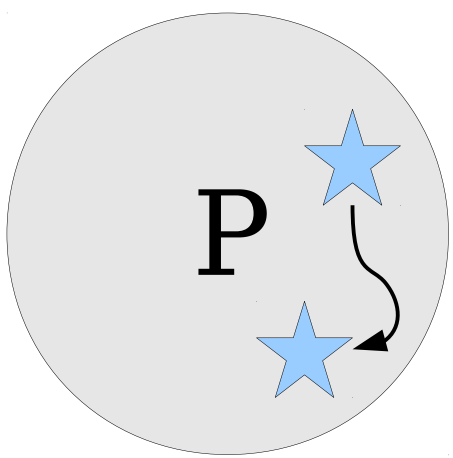
Polynomial-Time Reductions

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



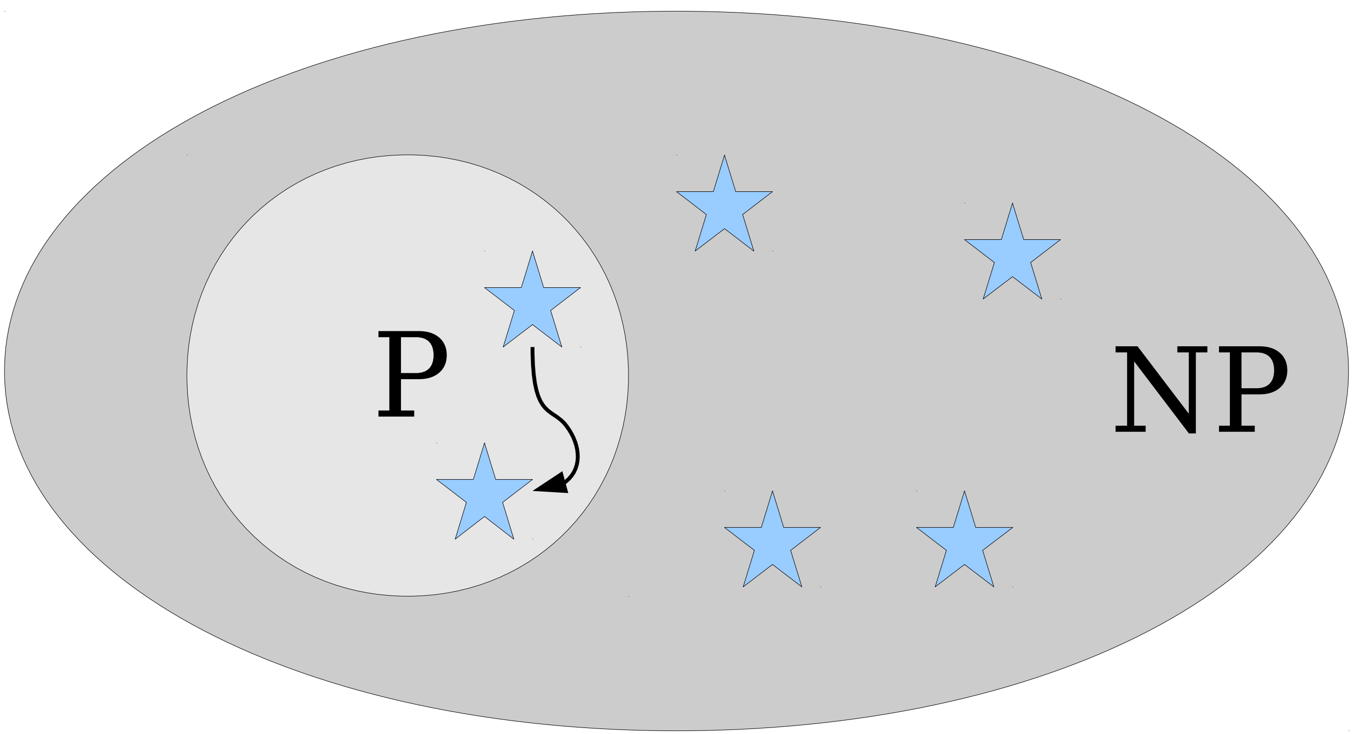
Polynomial-Time Reductions

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



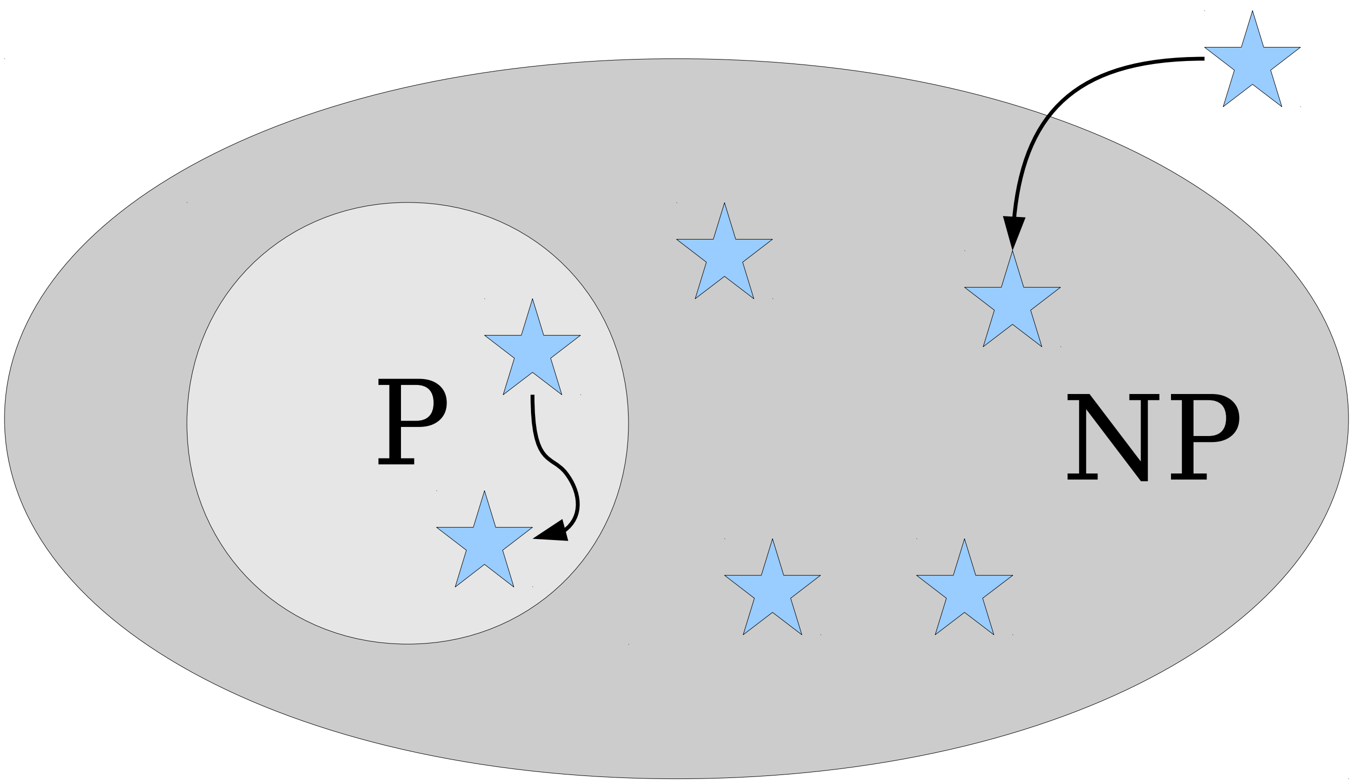
Polynomial-Time Reductions

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



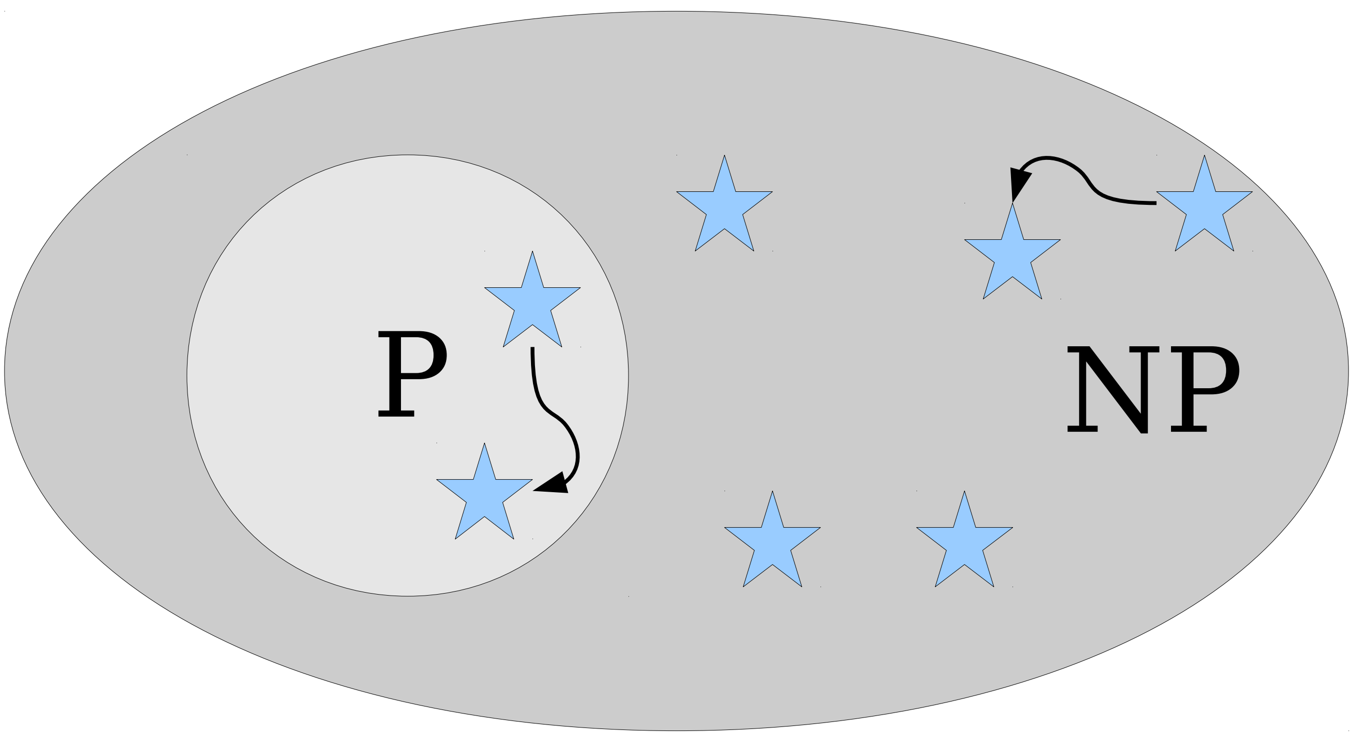
Polynomial-Time Reductions

- 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.

Final Exam Logistics

- Our final exam is on ***Monday, March 18th*** from ***7:00PM – 10:00PM***. Locations are the same as the first exam and are divvied up by last (family) name:
  - `A – P`: Go to Hewlett 200.
  - `Q – Z`: Go to Hewlett 201.
- The final exam is cumulative and covers topics from PS1 – PS9 and L00 – L24. Coverage will focus on material from the second half of the course which you haven't been tested on yet.
  - Noteworthy exceptions: Material from week 10 lectures will not be tested. Material found purely in the optional written portion of PS9 also will not be tested.
- Like the midterms, it's closed-book, closed-computer, and limited-note. You can bring one double-sided 8.5" × 11" notes sheet with you.
- Students with OAE accommodations: you should have heard from us regarding your final exam time and location. Please double check this and reach out to us ASAP if anything looks incorrect.

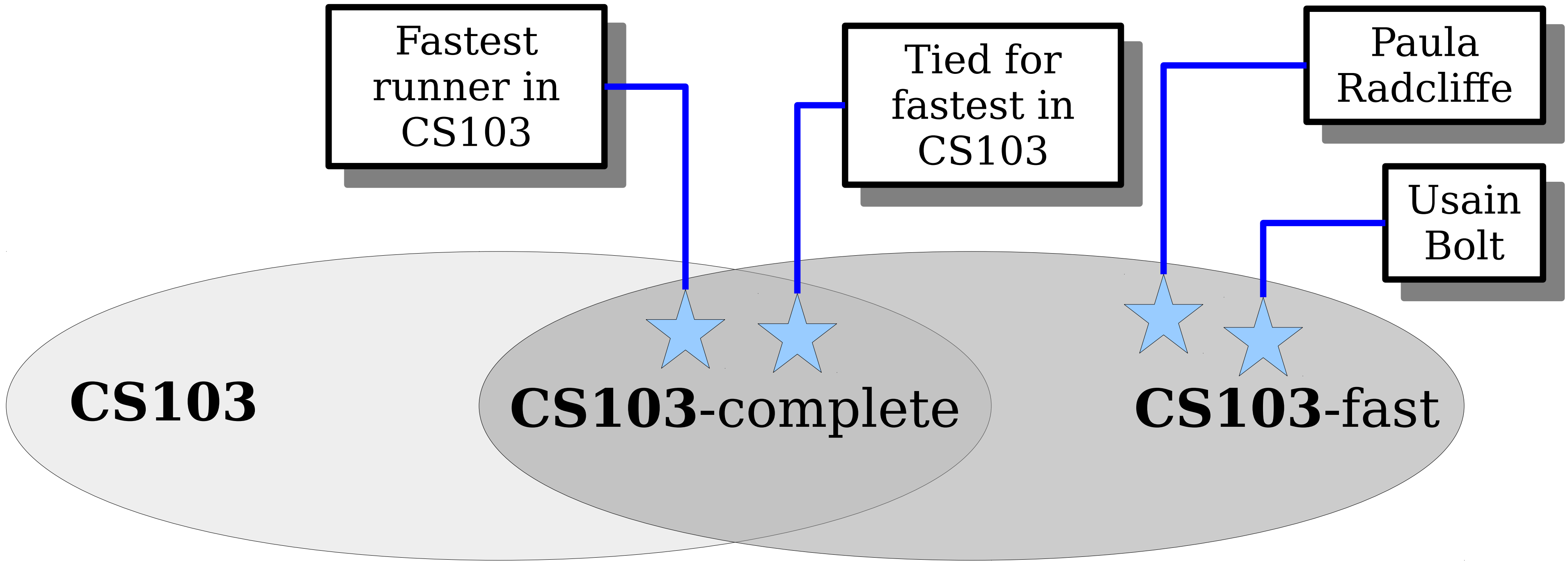
Preparing for the Exam

- Benson will be holding a final exam review session tomorrow, ***Thursday, March 14th*** at ***5:30PM - 6:30PM*** over Zoom (note the updated time!).
  - It will be recorded and the video made available on Canvas.
- We also have two practice finals and a giant compendium of practice problems on the course website.
- As always, ***keep the TAs in the loop when studying***! That's what we're here for.

Back to CS103!

NP-Hardness and **NP**-Completeness

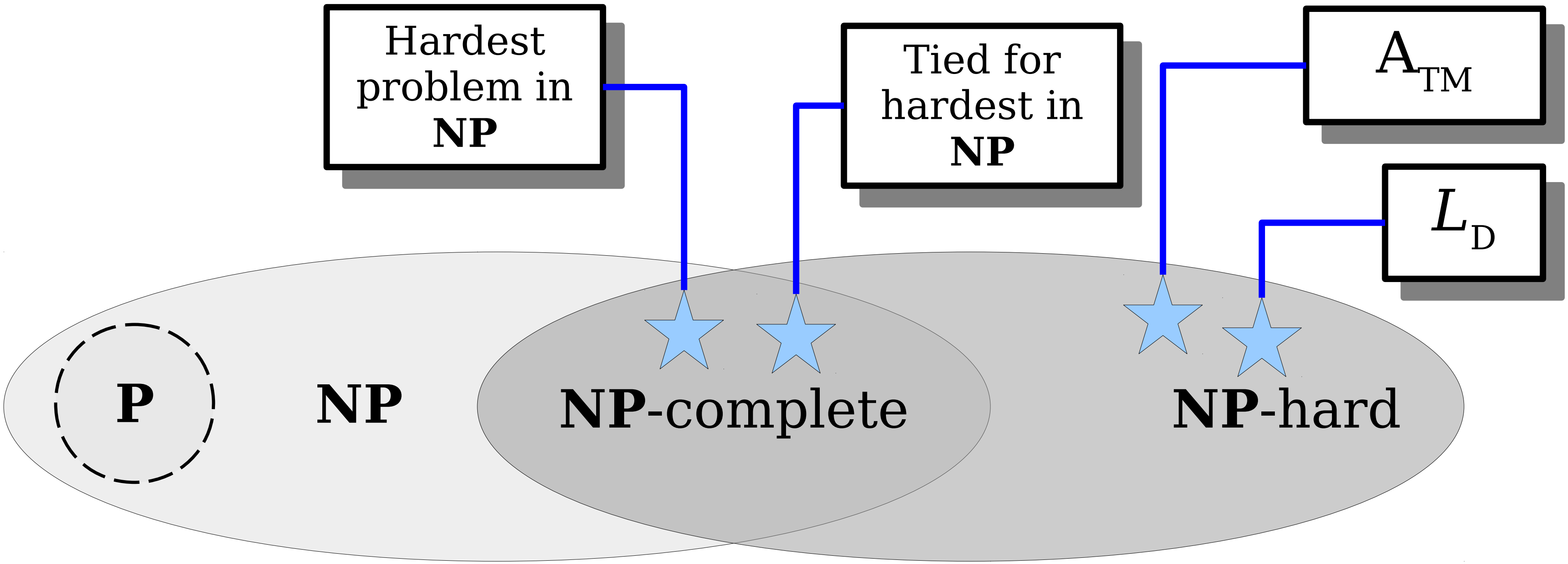
An Analogy: Running Really Fast



For people A and B, we say $\boldsymbol{A \leq_r B}$ if
A's top running speed is at most B's top speed.
(Intuitively: B can run at least as fast as A.)

We say that person P is ***CS103-fast*** if
$\forall A \in \mathbf{CS103}.\ A \leq_r P.$
(How fast are you if you're CS103-fast?)

We say that person P is ***CS103-complete*** if
$P \in$ **CS103** and P is **CS103**-fast.
(How fast are you if you're CS103-complete?)



For languages A and B, we say $\boldsymbol{A \leq_p B}$ if A reduces to B in polynomial time.

(Intuitively: B is at least as hard as A.)

We say that a language L is ***NP-hard*** if

$\forall A \in \mathbf{NP}.\ A \leq_p L.$

(How hard is a problem that's NP-hard?)

We say that a language L is ***NP-complete*** if

$L \in \mathbf{NP}$ and L is **NP**-hard.

(How hard is a problem that's NP-complete?)

Intuition: The **NP**-complete problems are the hardest problems in **NP**.

If we can determine how hard those problems are, it would tell us a lot about the $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ question.

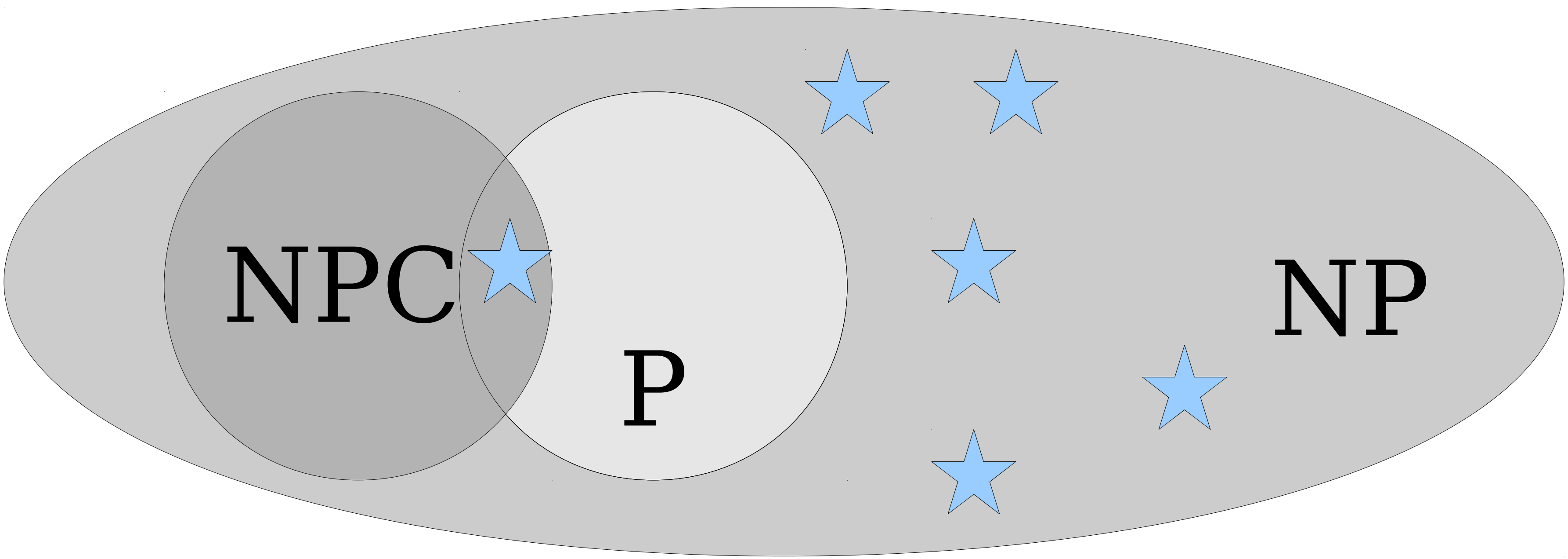
The Tantalizing Truth

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

Intuition: This means the hardest problems in **NP** aren't actually that hard. We can solve them in polynomial time. So that means we can solve all problems in **NP** in polynomial time.

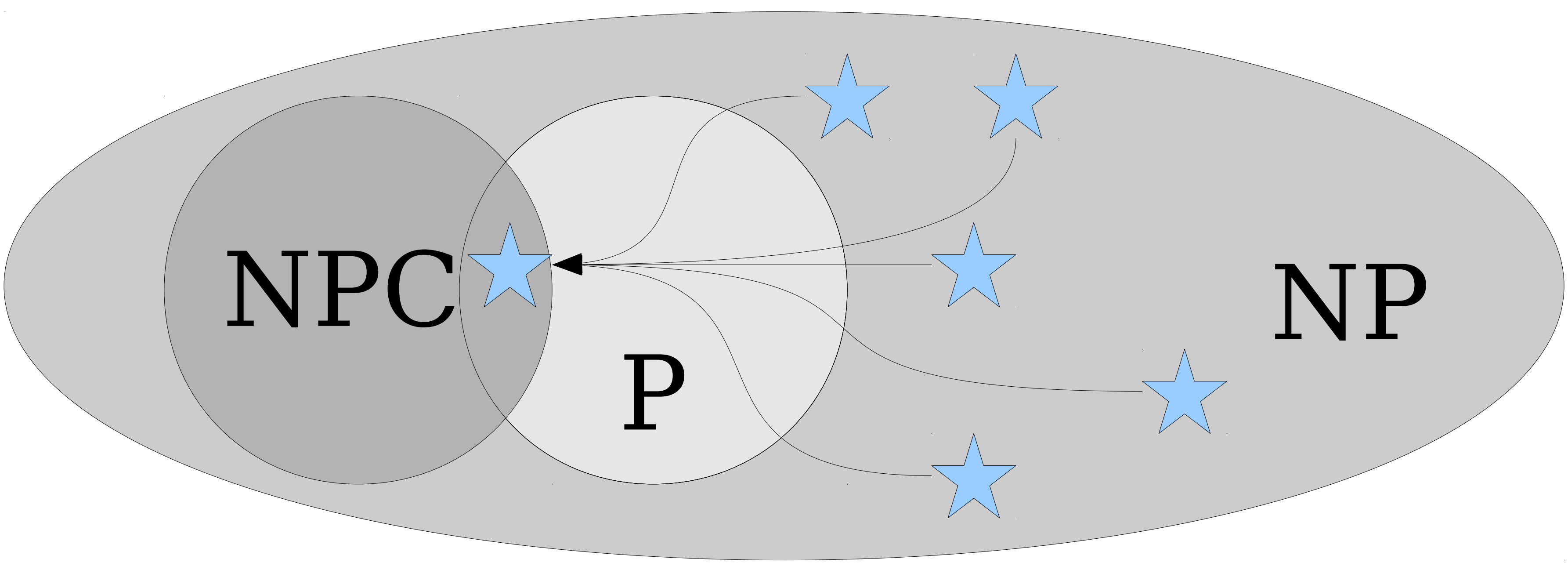
The Tantalizing Truth

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



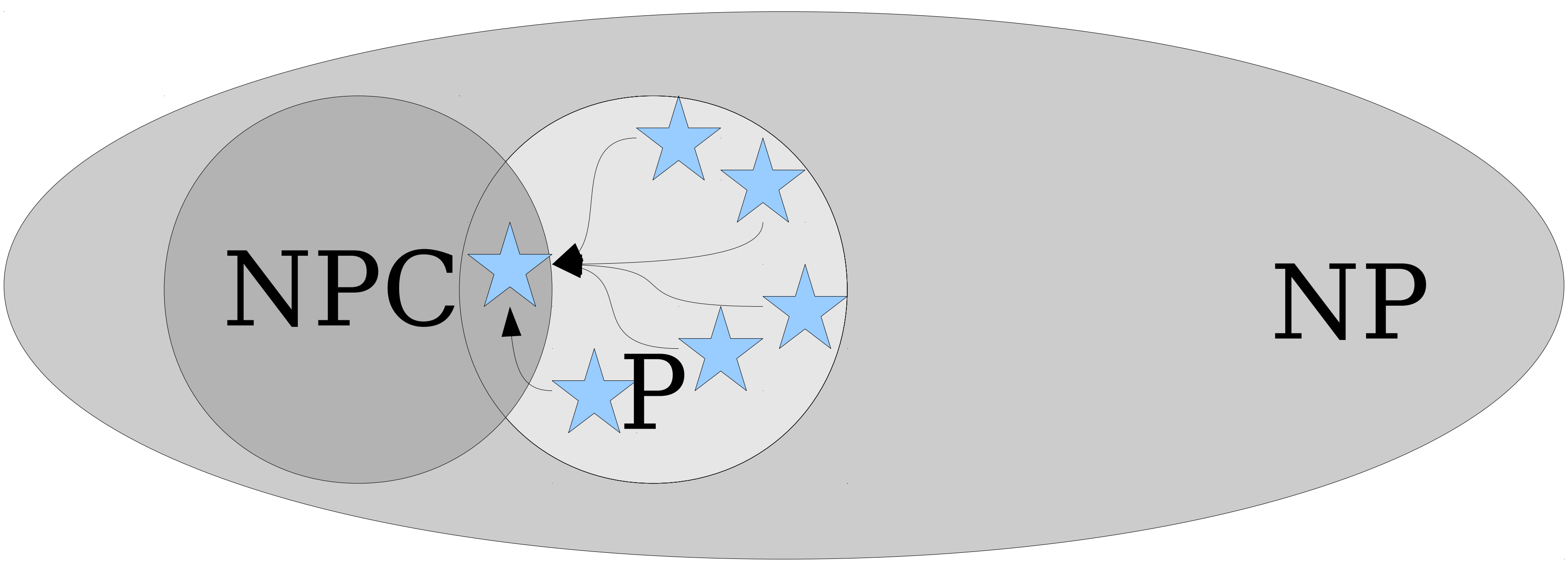
The Tantalizing Truth

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



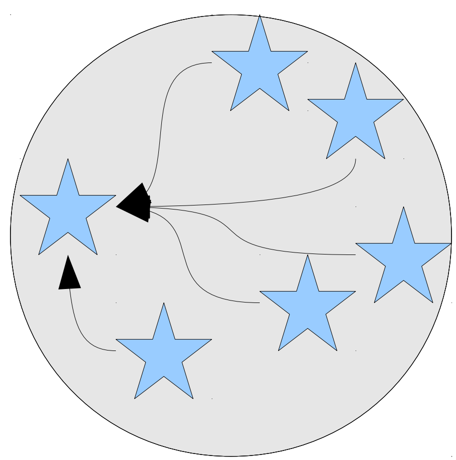
The Tantalizing Truth

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**.

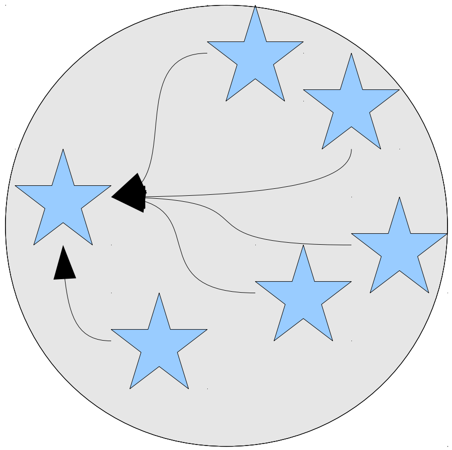


P = NP

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** ⊆ **P**, so **P** = **NP**. ■



P = NP

The Tantalizing Truth

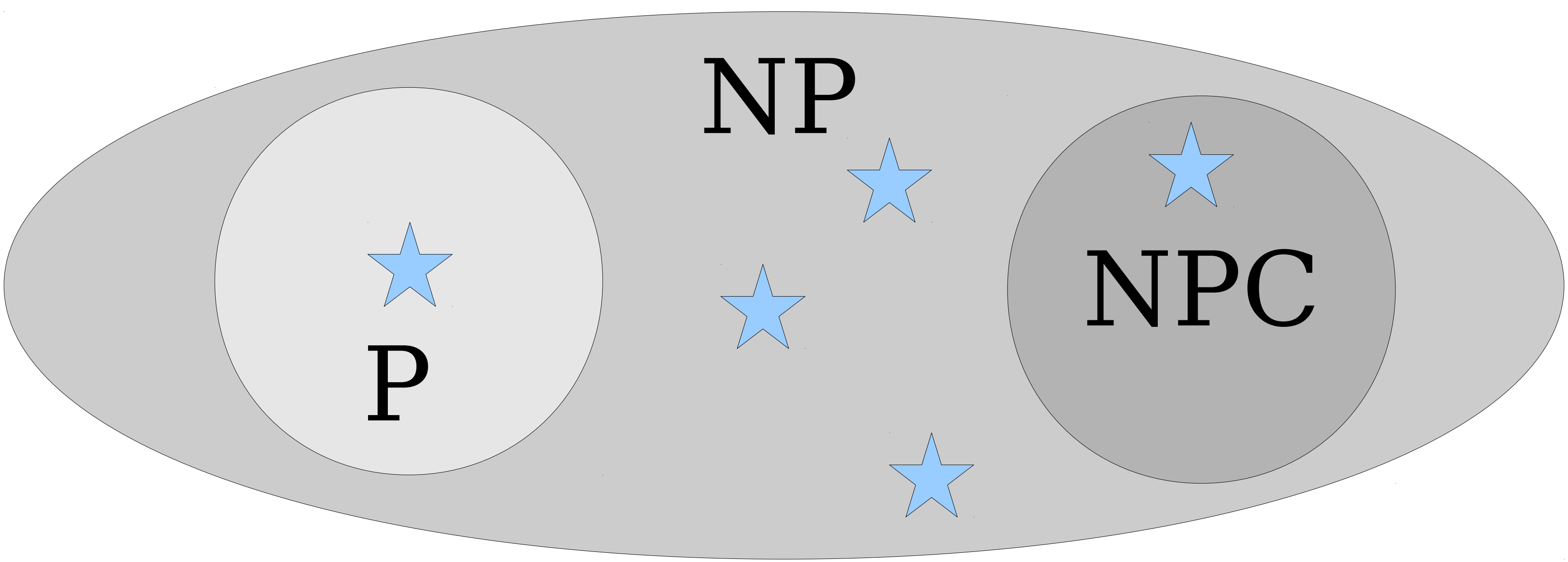
Theorem: If *any* **NP**-complete language is not in **P**, then **P** ≠ **NP**.

Intuition: This means the hardest problems in **NP** are so hard that they can't be solved in polynomial time. So the hardest problems in **NP** aren't in **P**, meaning **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.
- Which of the following formulas are satisfiable?

$$p \land q$$

$$p \land \neg p$$

$$p \to (q \land \neg q)$$

Satisfiability

- A propositional logic formula φ is called ***satisfiable*** if there is some assignment to its variables that makes it evaluate to true.
- Which of the following formulas are satisfiable?

$$p \land q$$

$$p \land \neg p$$

$$p \to (q \land \neg q)$$

- An assignment of true and false to the variables of φ that makes it evaluate to true is called a ***satisfying assignment***.

SAT

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

Given a propositional logic formula φ, is φ satisfiable?

- Formally:

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

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

Proof 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 φ(*w*) that says "there is a certificate *c* where *V* accepts ⟨*w*, *c*⟩." This formula is satisfiable if and only if *w* ∈ *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.

$$\text{SAT} \in \mathbf{P} \quad \leftrightarrow \quad \mathbf{P} = \mathbf{NP}$$

- 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 receive transplants. *(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)*

Next Time

- ***Why All This Matters***
- ***Where to Go from Here***
- ***Ask Me Anything***
- ***Parting Words!***