NP-Completeness

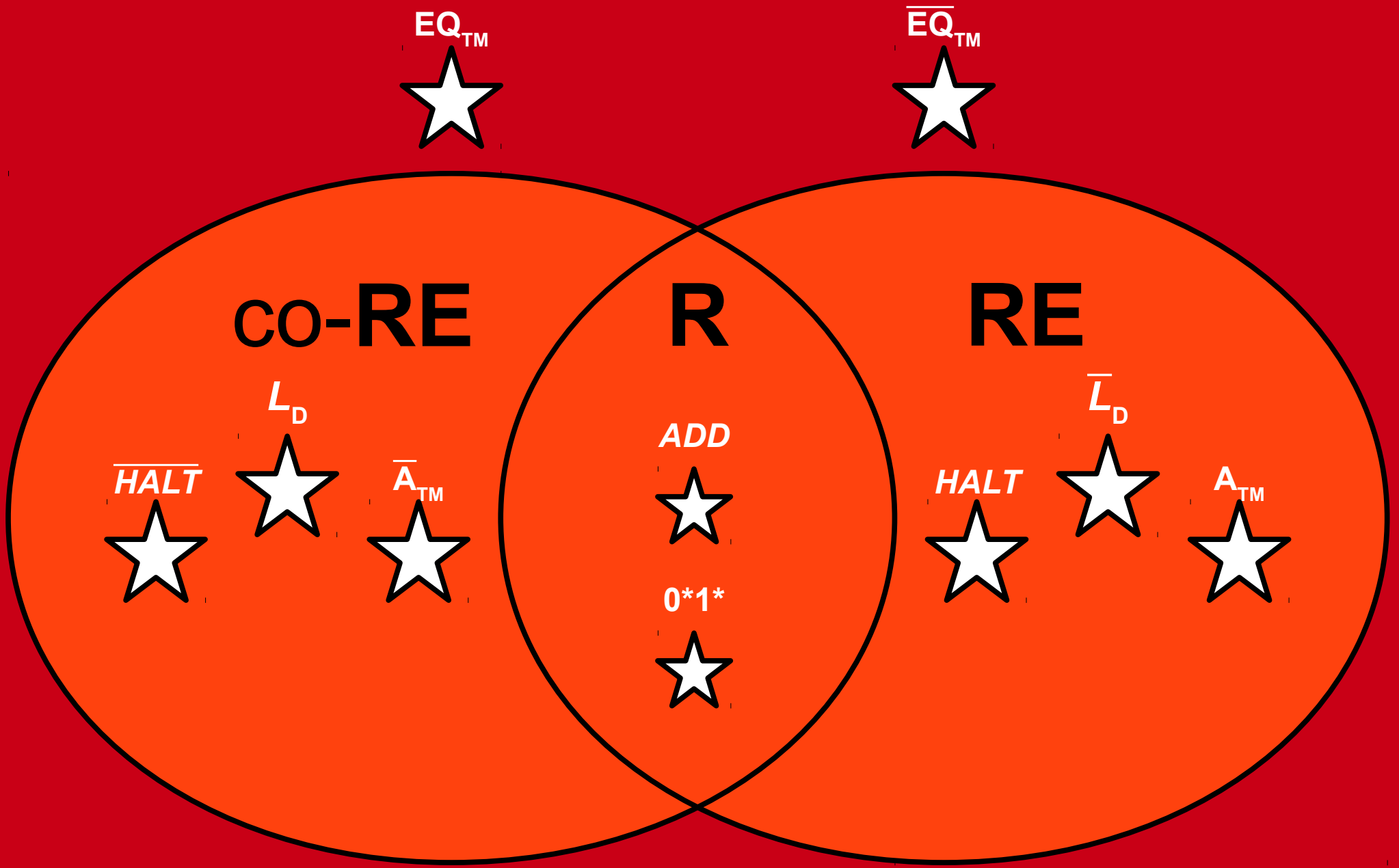
Part I

Outline for Today

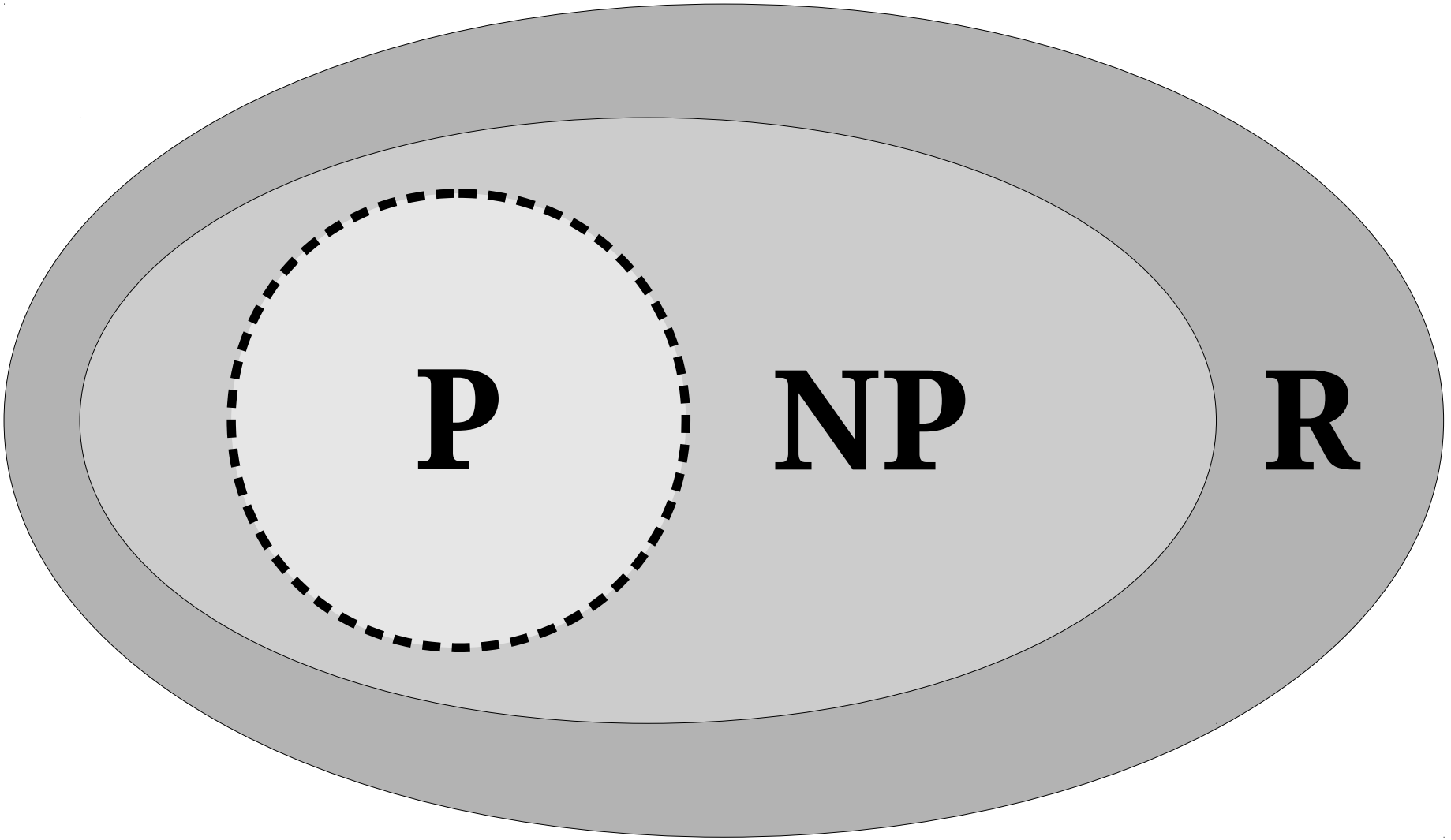
- **Recap from Last Time**
  - Welcome back from break! Let's make sure we're all on the same page here.
- **Polynomial-Time Reducibility**
  - Connecting problems together.
- **NP-Completeness**
  - What are the hardest problems in **NP**?
- **The Cook-Levin Theorem**
  - A concrete **NP**-complete problem.

Recap from Last Time

The Limits of Computability



The Limits of Efficient Computation



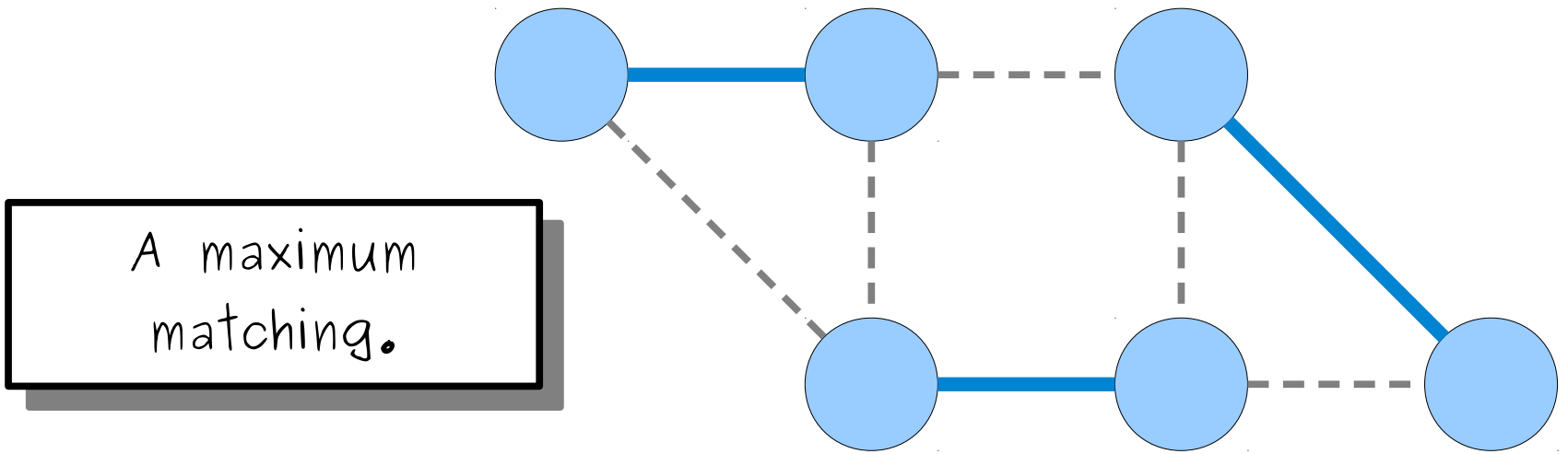
P and **NP** Refresher

- The class **P** consists of all problems solvable in deterministic *polynomial* time.
- The class **NP** consists of all problems solvable in *nondeterministic* polynomial time.
- Equivalently, **NP** consists of all problems for which there is a deterministic, polynomial-time verifier for the problem.

Reducibility

Maximum Matching

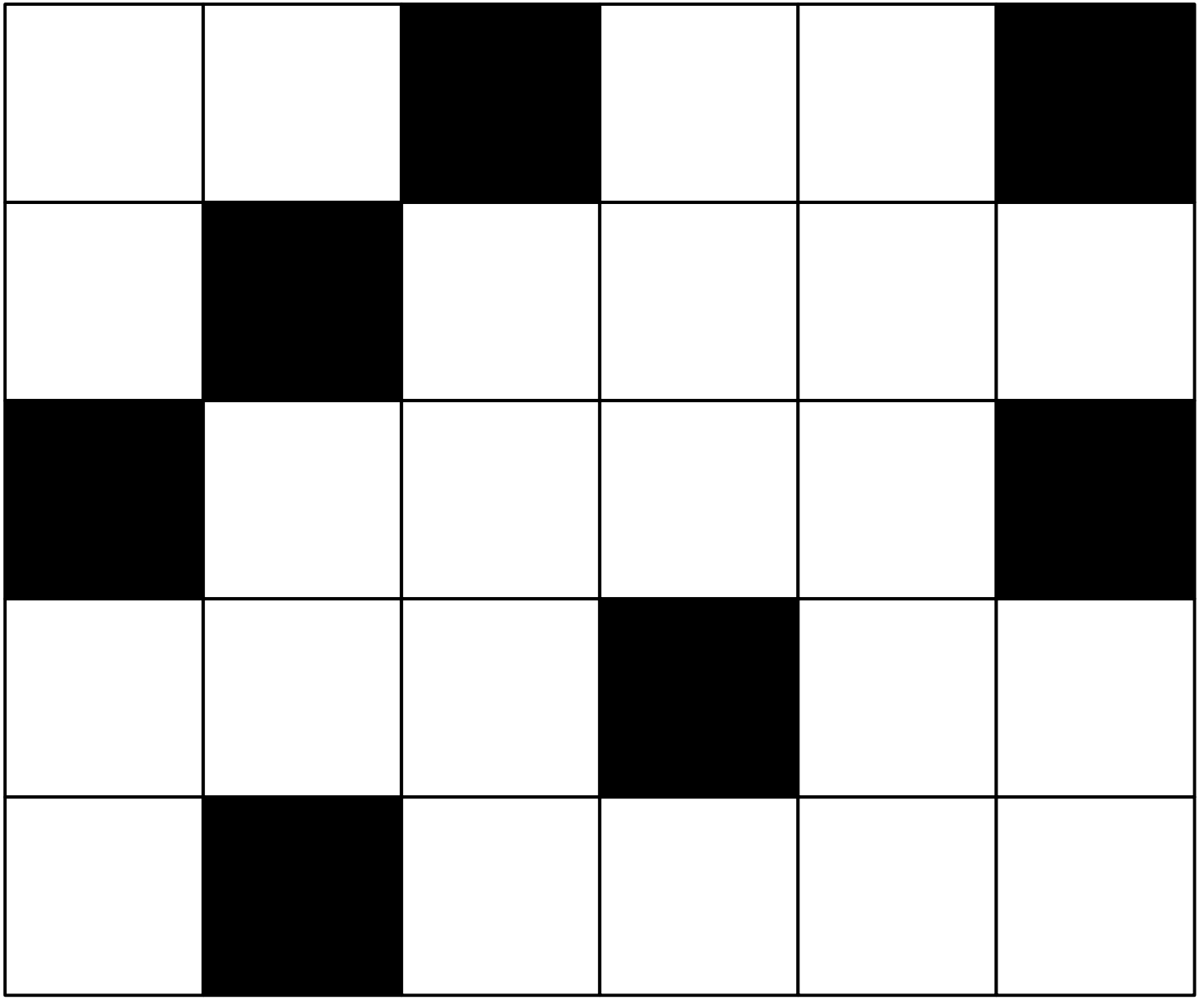
- Given an undirected graph G, a ***matching*** in G is a set of edges such that no two edges share an endpoint.
- A ***maximum matching*** is a matching with the largest number of edges.



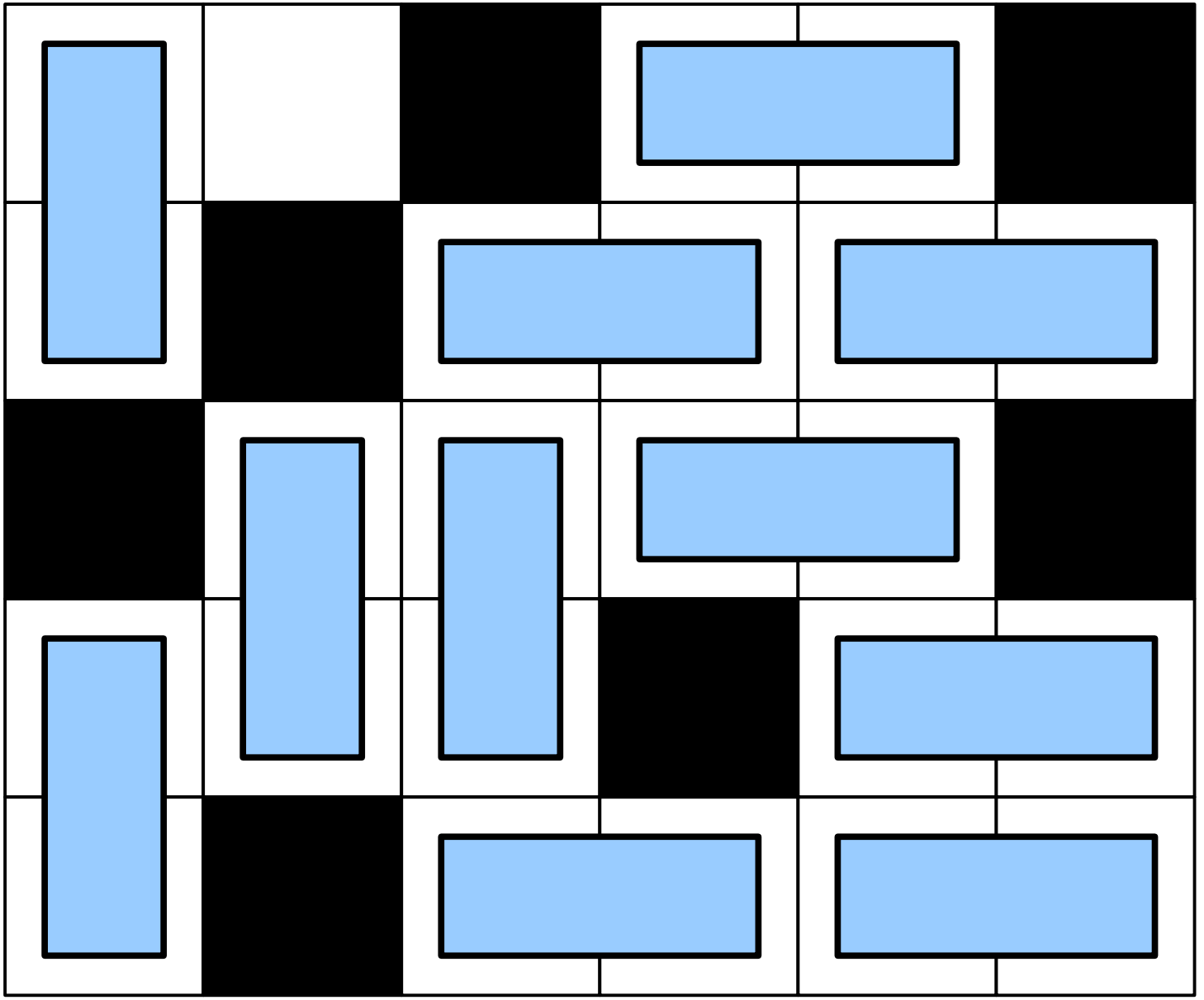
Maximum Matching

- 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?

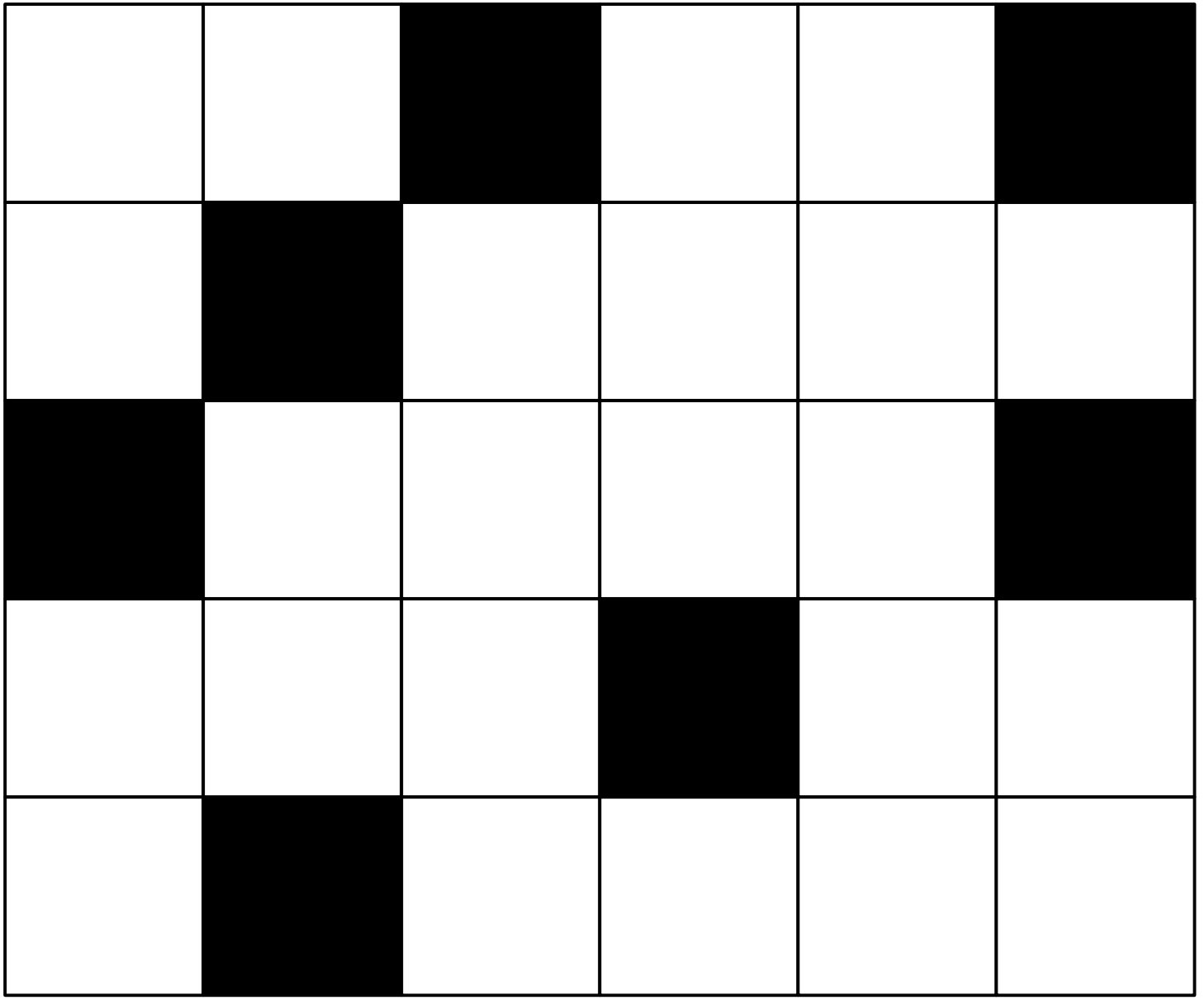
Domino Tiling



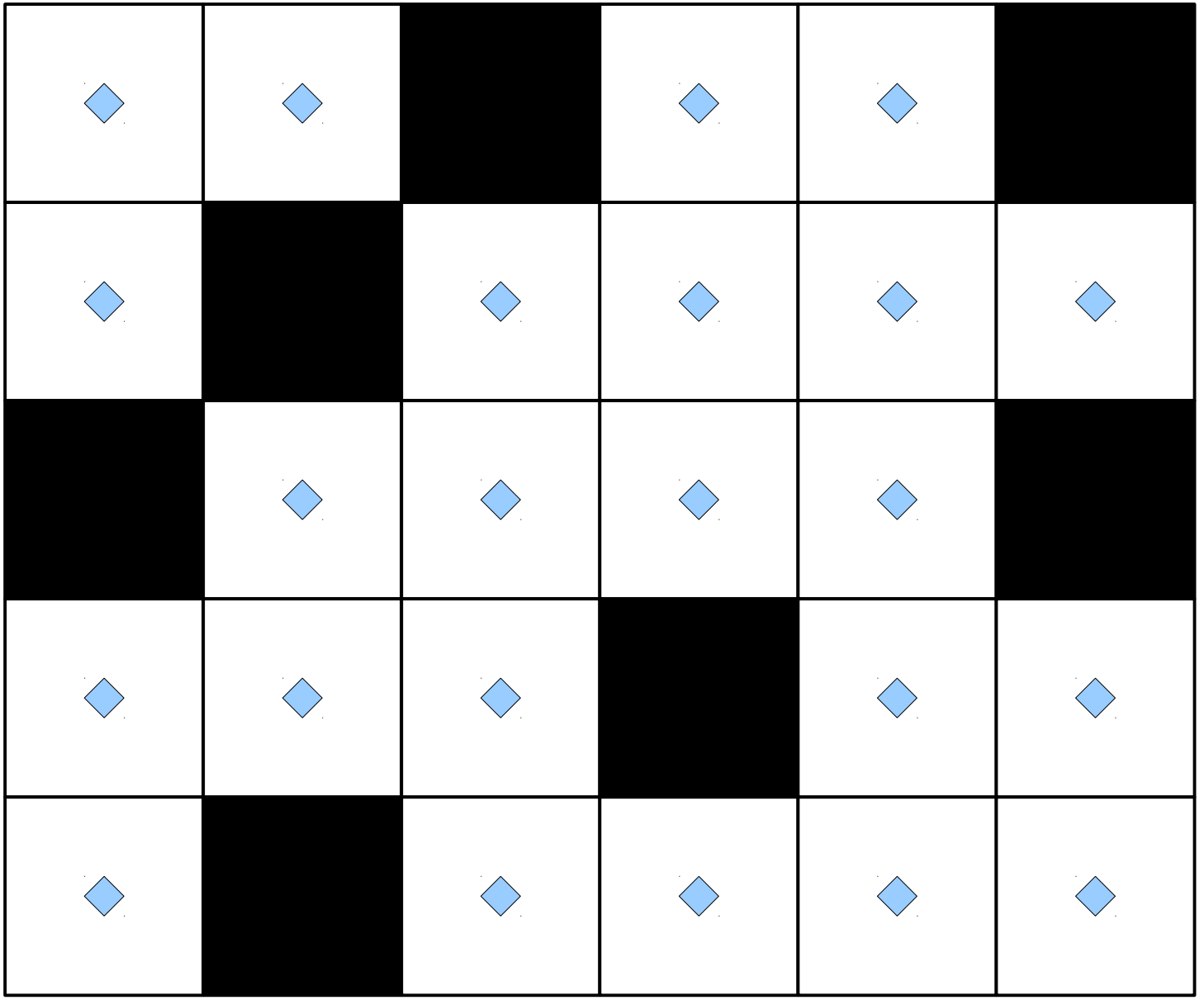
Domino Tiling



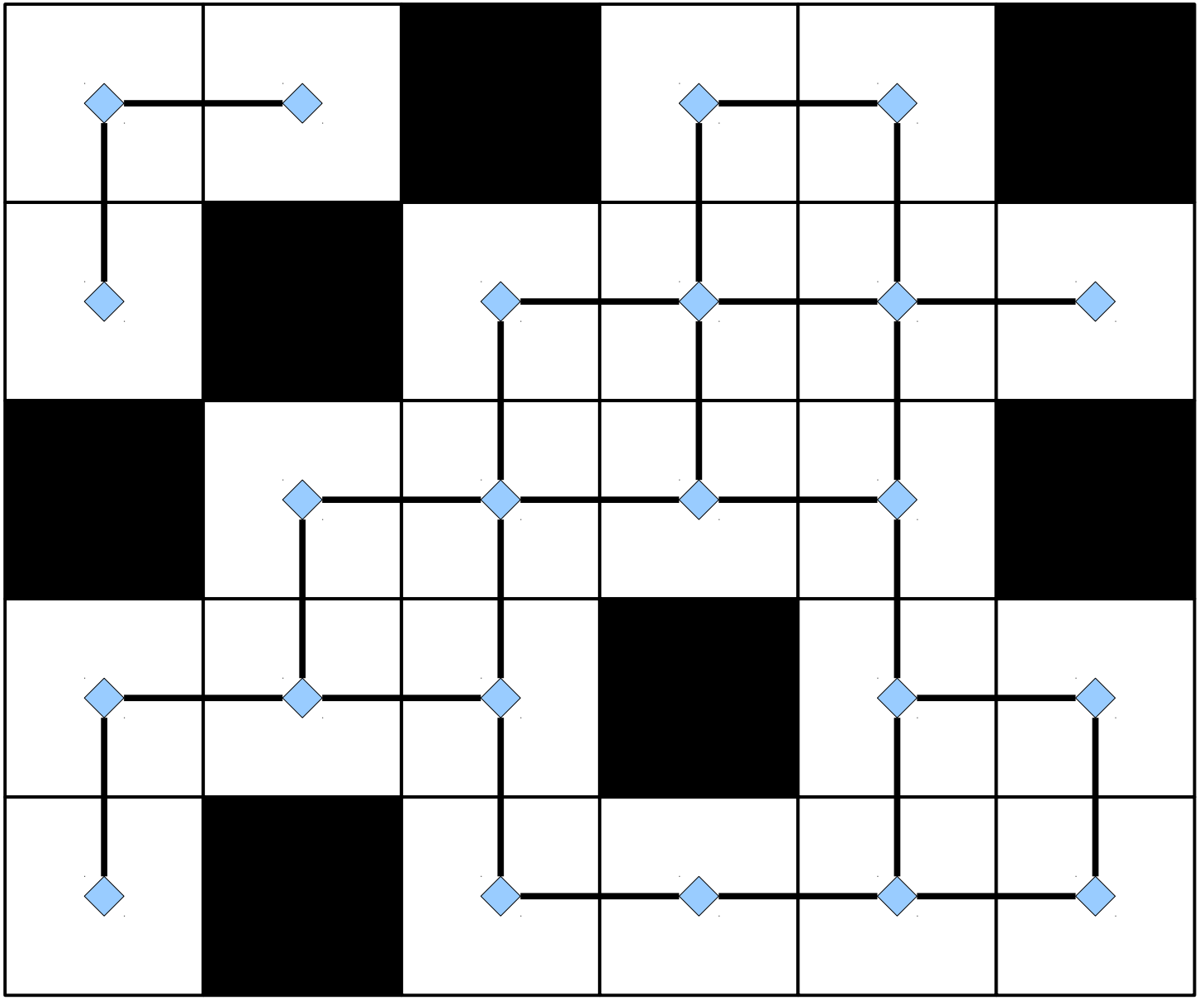
Solving Domino Tiling



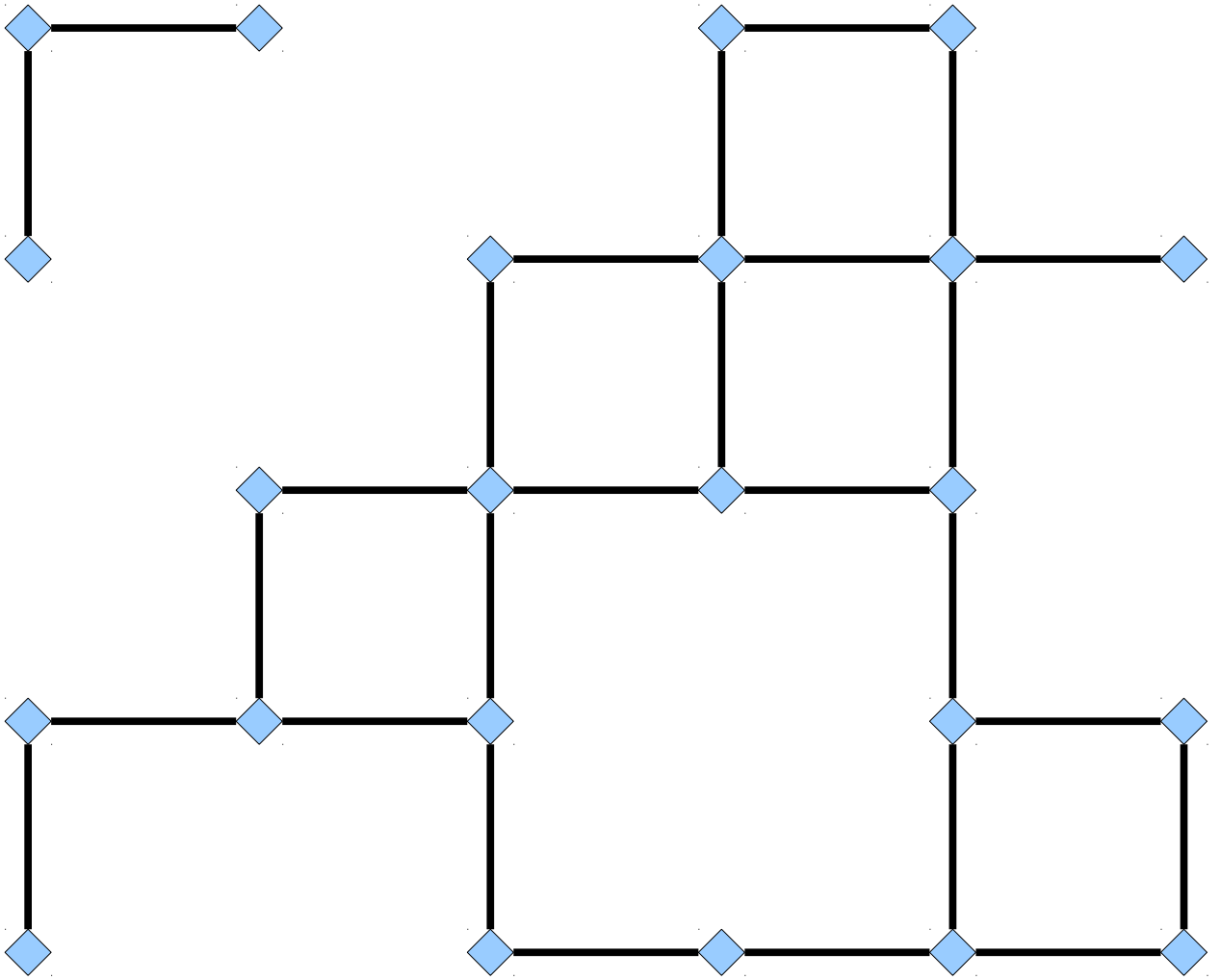
Solving Domino Tiling



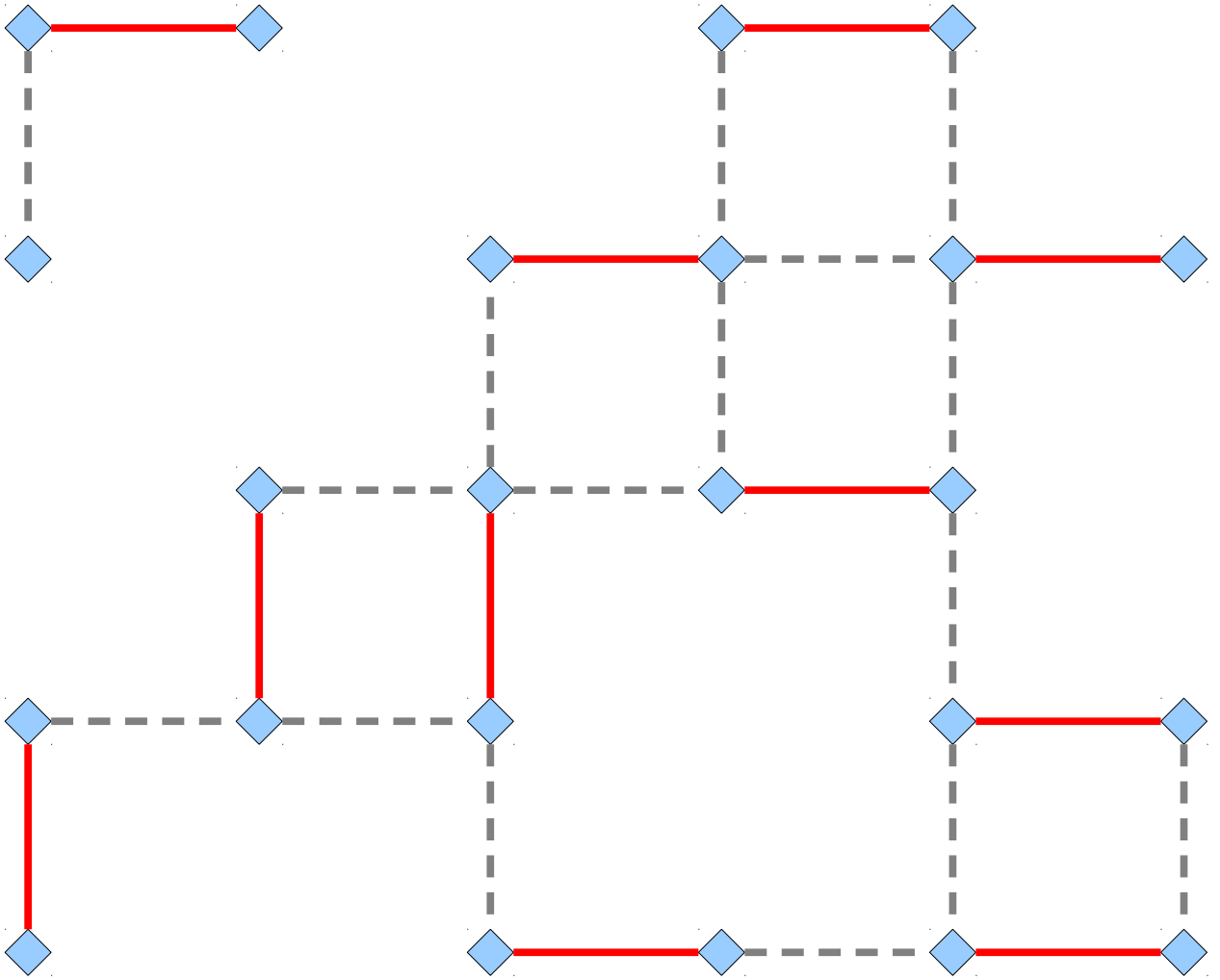
Solving Domino Tiling



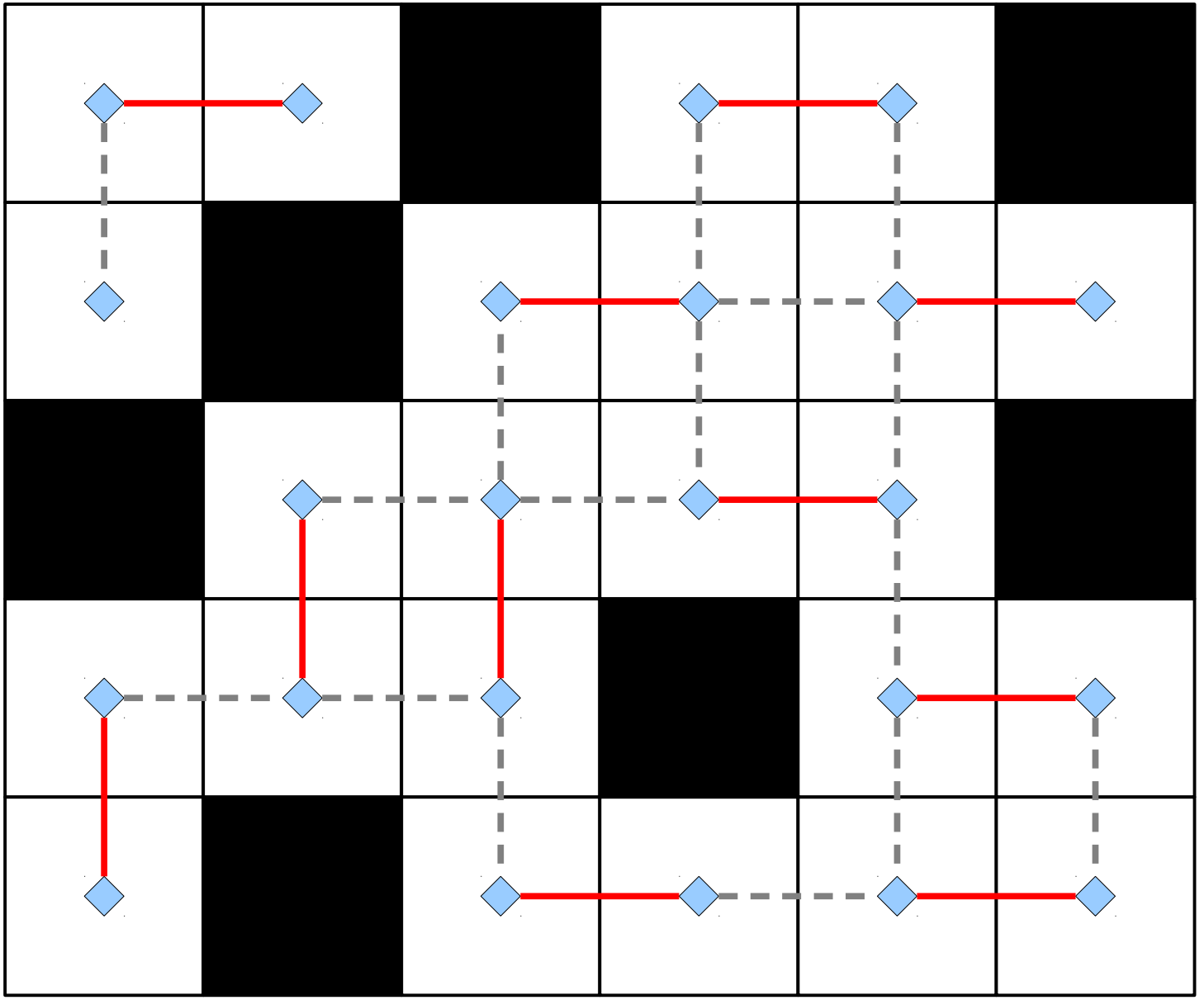
Solving Domino Tiling



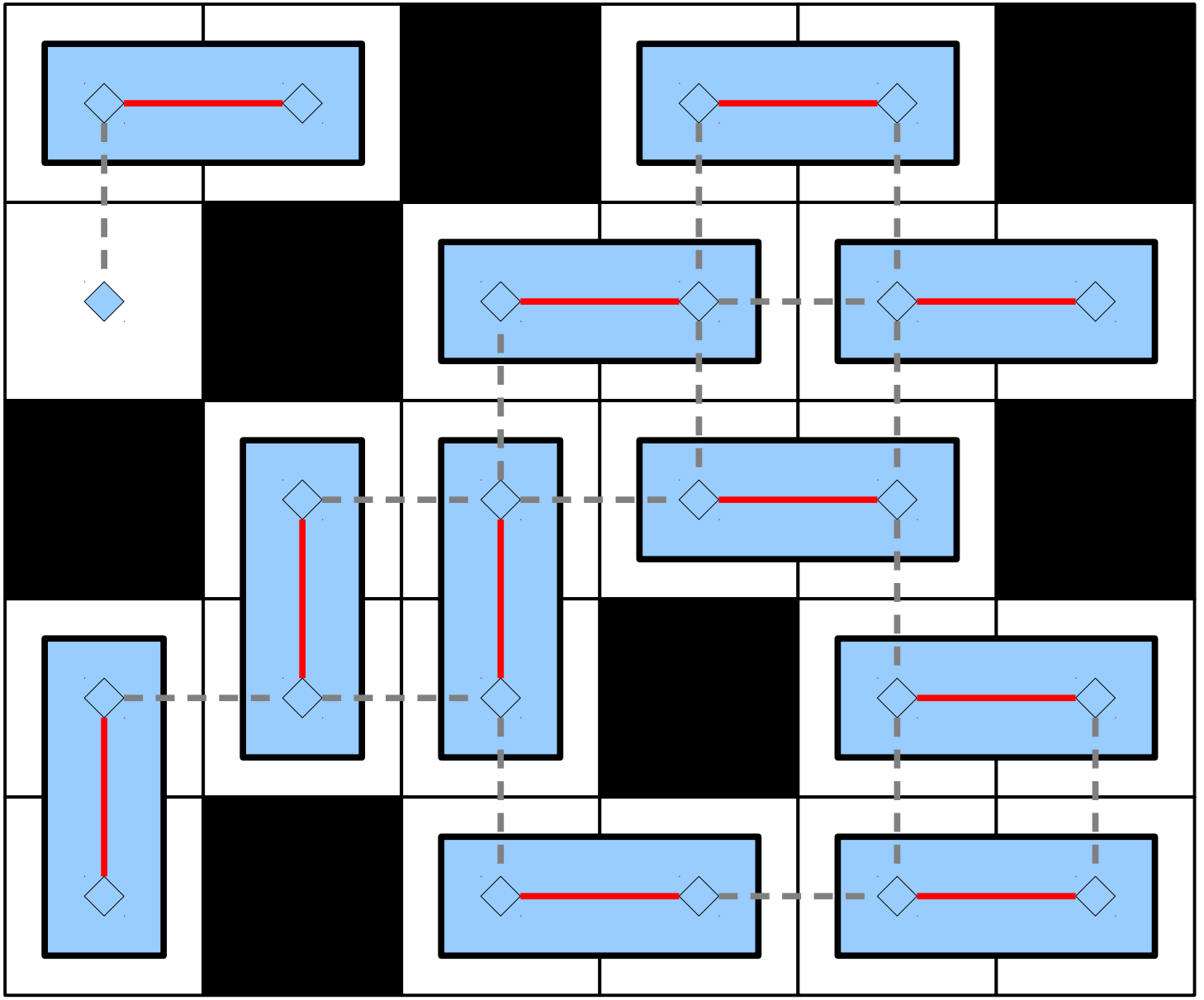
Solving Domino Tiling



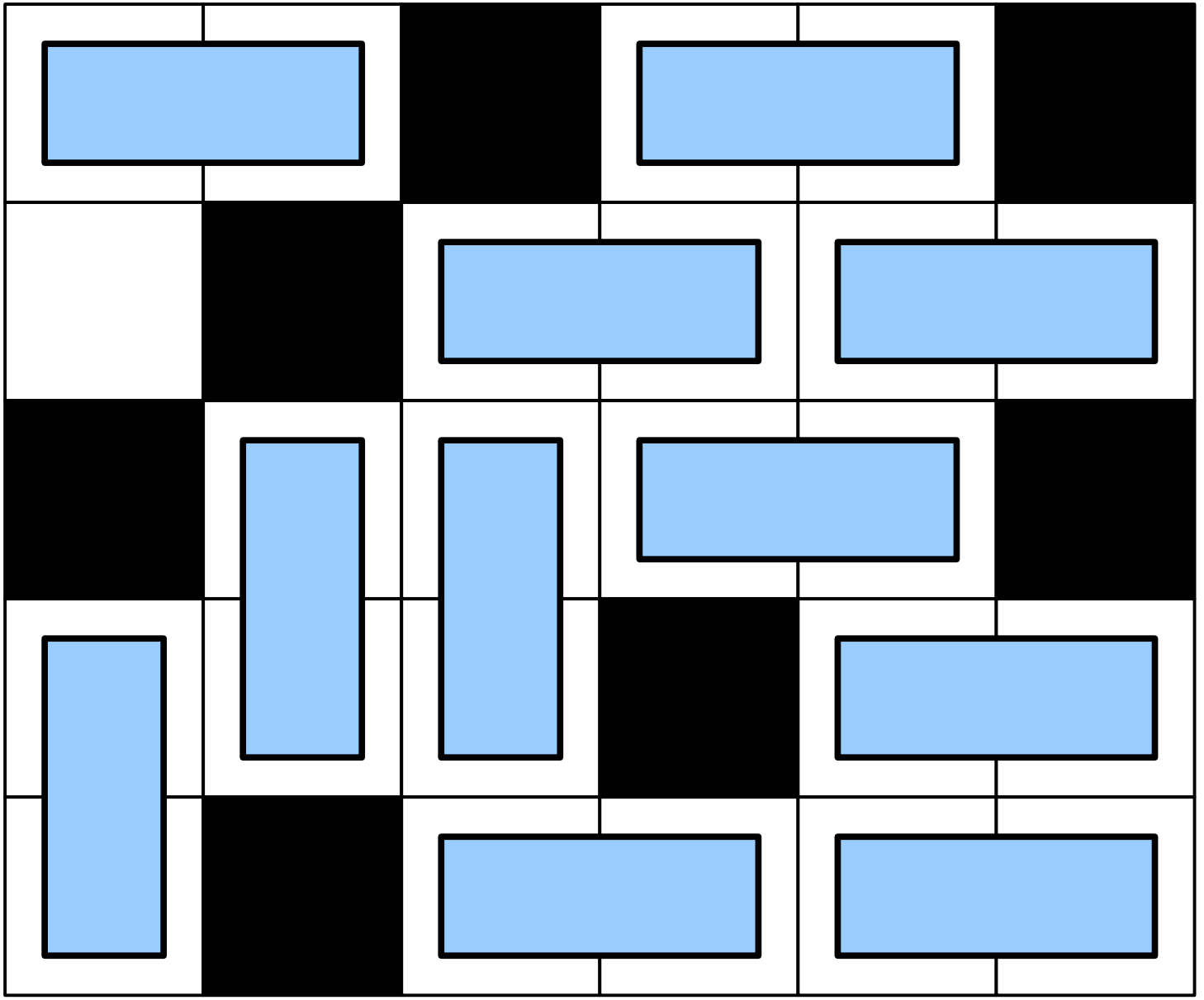
Solving Domino Tiling



Solving Domino Tiling



Solving Domino Tiling



The Setup

- To determine whether you can place at least k dominoes on a crossword grid, do the following:
  - Convert the grid into a graph: each empty cell is a node, and any two adjacent empty cells have an edge between them.
  - Ask whether that graph has a matching of size k or greater.
  - Return whatever answer you get.
- **Claim:** This runs in polynomial time.

In Pseudocode

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

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*?

- As a formal language:

  ***REACHABILITY* =**
  { ⟨*G*, *s*, *t*⟩ | *G* is a directed graph, *s* and *t* are nodes in *G*, and there's a path from *s* to *t* }

- ***Theorem:*** *REACHABILITY* ∈ **P**.

- Given that we can solve the reachability problem in polynomial time, what other problems can we solve in polynomial time?

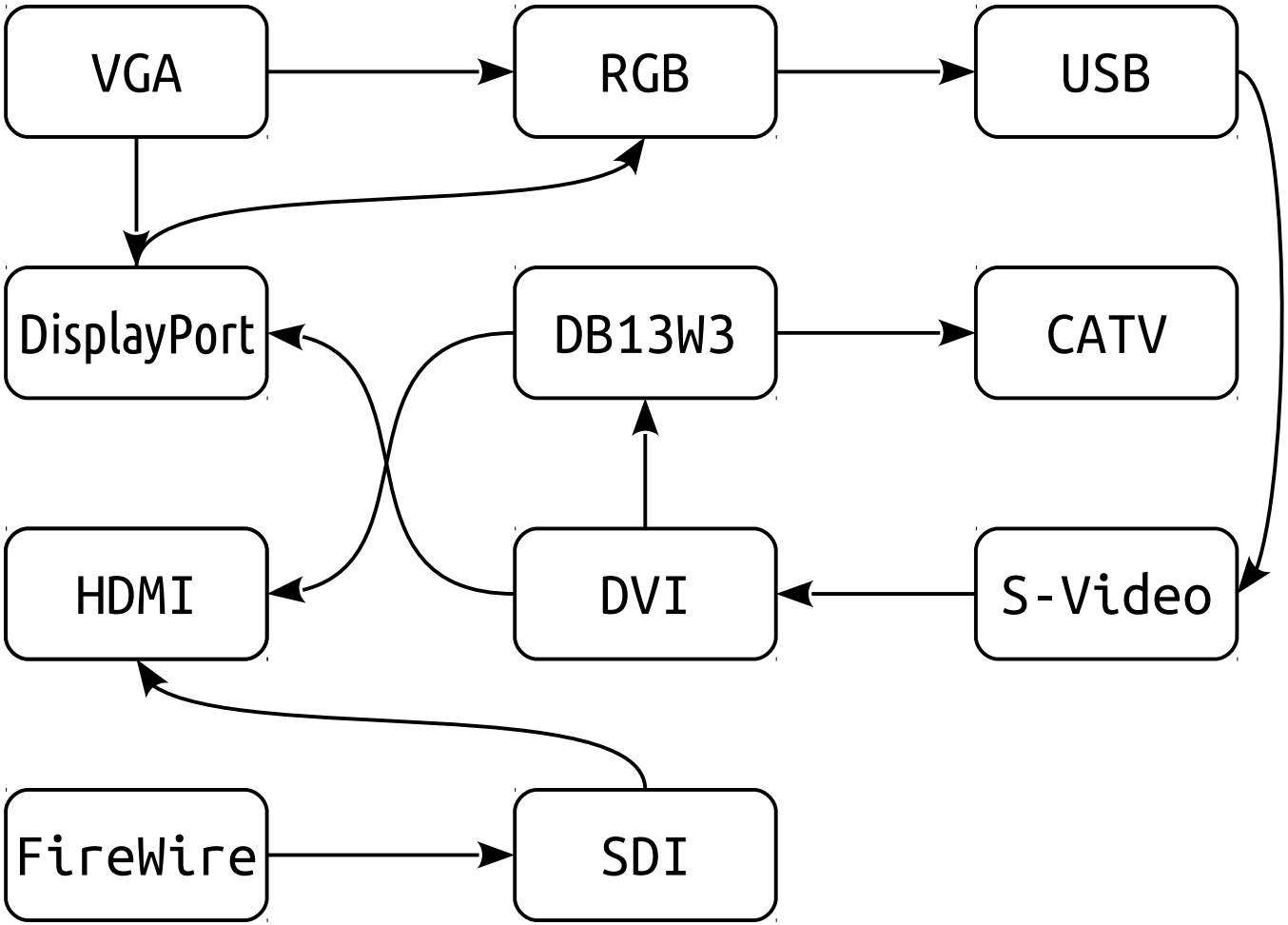
Converter Conundrums

- 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

- Given a a list of plug converters, here's a algorithm for determining whether you can plug your computer into the projector:
  - Create a graph with one node per connector type and an edge from one type of connector to another if there's a converter from the first type to the second.
  - Use the reachability algorithm to see whether you can get from VGA to HDMI.
  - Return whatever the result of that algorithm is.
- **Claim:** This runs in polynomial time.

In Pseudocode

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

A Commonality

- Both of the solutions to our previous problems had the following form:

```
boolean solveProblemA(input) {
   return solveProblemB(transform(input));
}
```

- **Important observation:** Assuming that we already have a solver for problem *B*, the only work done here is transforming the input to problem *A* into an input to problem *B*.
- All the "hard" work is done by the solver for *B*; we just turn one input into another.

Mathematically Modeling this Idea

Polynomial-Time Reductions

- Let A and B be languages.
- A ***polynomial-time reduction*** from A to B is a function $f : \Sigma^* \to \Sigma^*$ such that
  - For any $w \in \Sigma^*$, $w \in A$ iff $f(w) \in B$.
  - The function f can be computed in polynomial time.
- What does this mean?

Polynomial-Time Reductions

- If f is a polynomial-time reduction from A to B, then

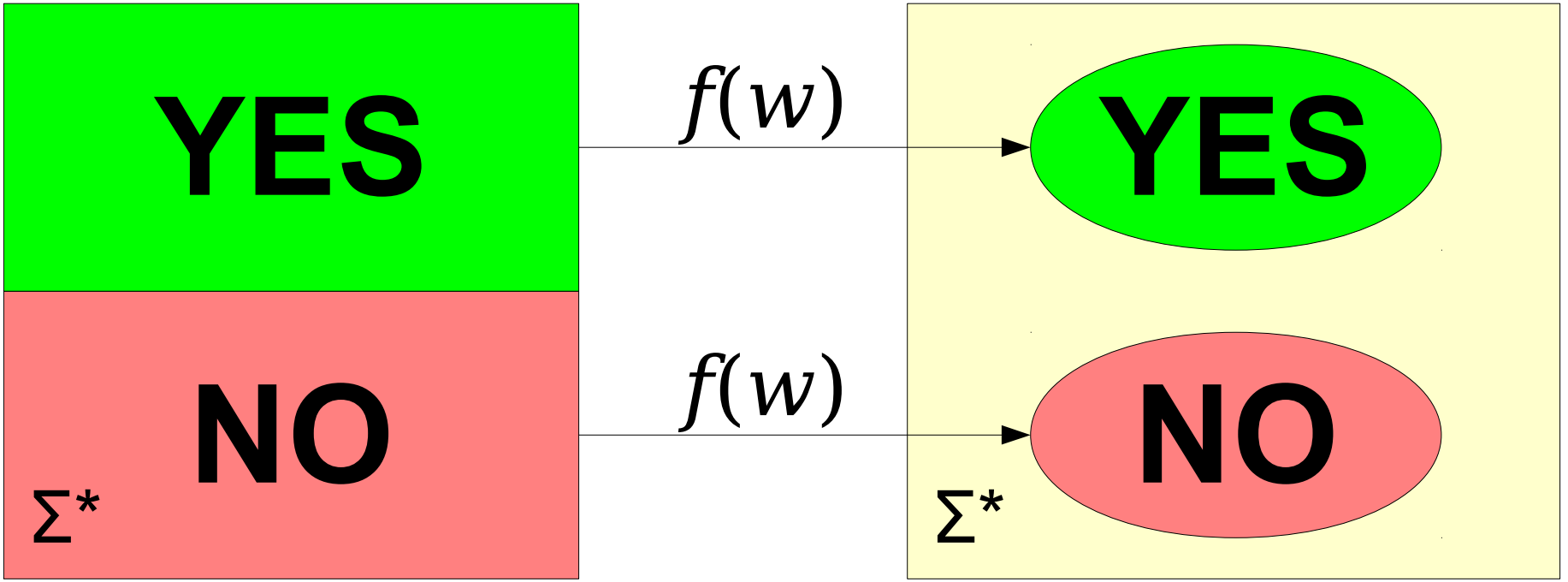
$$\forall w \in \Sigma^*. (w \in A \leftrightarrow f(w) \in B)$$

- ***If you want to know whether $w \in A$, you can instead ask whether $f(w) \in B$.***
  - Every $w \in A$ maps to some $f(w) \in B$.
  - Every $w \notin A$ maps to some $f(w) \notin B$.

Polynomial-Time Reductions

- If f is a polynomial-time reduction from A to B, then

$$\forall w \in \Sigma^*. (w \in A \leftrightarrow f(w) \in B)$$



Reductions, Programmatically

- Suppose we have a solver for problem B that's defined in terms of problem A in this specific way:

```
boolean solveProblemA(input) {
   return solveProblemB(transform(input));
}
```

- The reduction from A to B is the function `transform` in the above setup: it maps "yes" answers to A to "yes" answers to B and "no" answers to A to "no" answers to B.

Reducibility among Problems

- Suppose that A and B are languages where there's a polynomial-time reduction from A to B.
- We'll denote this by writing

$$\boldsymbol{A \leq_p B}$$

- You can read this aloud as "A polynomial-time reduces to B" or "A poly-time reduces to B."

Reductions and **P**

- ***Theorem:*** If $A \leq_p B$ and $B \in \mathbf{P}$, then $A \in \mathbf{P}$.
- ***Proof sketch:*** Show that this pseudocode runs in polynomial time:

```
boolean solveProblemA(input) {
   return solveProblemB(transform(input));
}
```

Calling `transform` only takes polynomial time. Since it runs in polynomial time, the `transform` function can only produce an output that's polynomially larger than its input. Feeding that into a polynomial-time function `solveProblemB` then only takes polynomial time. Overall, this is a polynomial-time algorithm for A, so $A \in \mathbf{P}$. ■

Time-Out for Announcements!

Please evaluate this course in Axess.

Your feedback does make a difference.

Problem Set 9

- Problem Set 9 goes out now and is due at the start of Friday's lecture.
  - Explore **P** and **NP**, **NP**-completeness, and The Big Picture.
  - This problem set is downweighted relative to the other problem sets.
- No late submissions accepted for this problem set (university policy.)
- Problem Set 8 was due at the start of today's lecture; due Wednesday with a late period.

Practice Final Exam

- We are working on scheduling a practice final exam at the start of finals week.
- Details will be released once we have a room reserved.

Alternate Final Exams

- Unfortunately, we've had to revise our policy with respect to alternate exams.
- We will only offer alternate exams for OAE accommodations, medical/family emergencies, or hard conflicts with other final exams.
- I'm sorry about this – it was a hard decision to make.

Your Questions!

“How do you get better at teaching?”

“Are you going to remember to answer all the questions you said you'll answer at the end of the quarter? If so, how much time do you anticipate Friday's timeout will take?”

“I heard you like to cook - can you share a Thanksgiving recipe with us?”

Roasted Maple-Glazed Carrots with Thyme and Walnuts

This recipe comes from a cooking class at Sacramento's Food Cooperative. The thyme is the sleeper hit here – it really makes everything come together!

Ingredients

- 2 pounds baby bunch carrots, peeled and sliced from top to bottom.
- 2 tbsp extra virgin olive oil
- Sea salt and freshly ground pepper to taste
- 2 tbsp maple syrup
- 2/3 tbsp melted butter
- 1/4 cup walnuts, toasted and chopped
- 1 tbsp chopped fresh thyme

Directions

1. Preheat oven to 400°F.
2. Ensure that the carrots are dry. Then, toss with olive oil and season generously with salt and pepper.
3. Spread the carrots out onto a parchment-paper-lined baking sheet, giving the carrots plenty of space. Bake for 20 minutes until slightly brown and remove from oven. (You may want to turn the carrots halfway through the baking to ensure that they brown uniformly.)
4. Combine the maple syrup and melted butter. Pour over the carrots and stir them around to coat well. Increase the heat in the oven to 475°F, then bake for up to 10 minutes to get the glaze to stick to the carrots and turn glossy. Be careful - this can burn if you don't keep an eye on it.
5. Sprinkle with walnuts and fresh thyme and toss to combine. Season with salt to taste. Serve warm or at room temperature.

“Have you every come across a Turing Machine where you thought, 'ok, this thing is definitely sentient?'”

Back to CS103!

Reductions and **NP**

```
boolean solveProblemA(input) {
  return solveProblemB(transform(input));
}
```

What happens if `transform` runs in ***deterministic*** polynomial time, but `solveProblemB` runs in ***nondeterministic*** polynomial time?

Reductions and **NP**

- We can reduce problems in **NP** to one another using polynomial-time reductions.
- The reduction itself must be computable in ***deterministic*** polynomial time.
- The output of that reduction is then fed in as input to a ***nondeterministic, polynomial-time*** algorithm.
- Remember – *the goal of the reduction is to transform the problem, not solve it!*

A Sample Reduction

$$\{\ 137, 42, 271, 103, 154, 16, 3\ \}$$

$$k = 452$$

Given a set $S \subseteq \mathbb{N}$ and a natural number k, the ***subset sum problem*** is to find a subset of S whose sum is exactly k.

SUBSET-SUM ∈ **NP**

- Here's a nondeterministic, polynomial-time algorithm for *SUBSET-SUM*:
- N = "On input $\langle S, k \rangle$:
  - ***Nondeterministically*** guess a subset $I \subseteq S$.
  - ***Deterministically*** verify whether the sum of the elements of I is equal to k.
  - If so, accept; otherwise reject."

What other problems can we solve with a solver for *SUBSET-SUM*?

$$U = \{1, 2, 3, 4, 5, 6\}$$

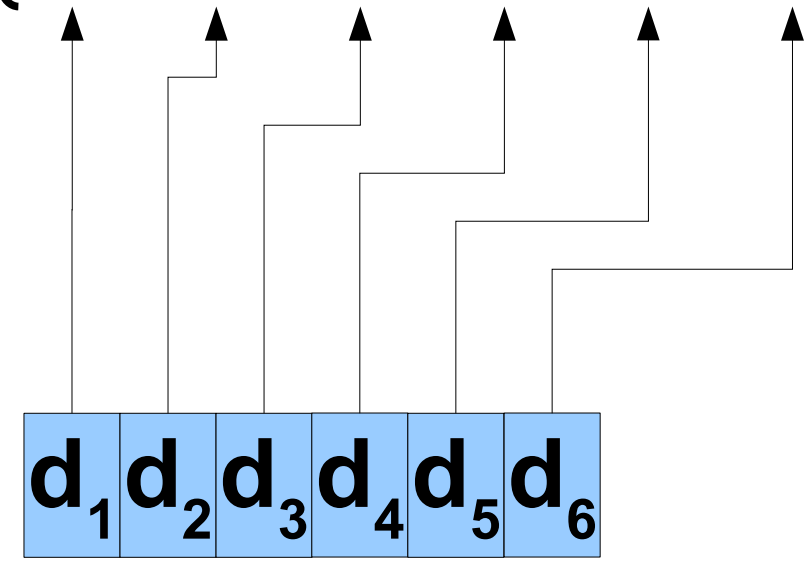
$$S = \{\{1, 2, 5\}, \{2, 5\}, \{1, 3, 6\}, \{2, 3, 4\}, \{4\}, \{1, 5, 6\}\}$$

Let U be a set of elements (the ***universe***) and $S \subseteq \wp(U)$. An ***exact covering*** of U is a collection of sets $I \subseteq S$ such that every element of U belongs to exactly one set in I.

Solving *EXACT-COVER* with *SUBSET-SUM*

$$S = \left\{ \begin{array}{ccc} \{1, 2, 5\}, & \{2, 5\}, & \{1, 3, 6\}, \\ \{2, 3, 4\}, & \{\ 4\ \}, & \{1, 5, 6\} \end{array} \right\}$$

$$U = \{1, 2, 3, 4, 5, 6\}$$



$$S = \left\{ \begin{array}{ccc} \{1, 2, 5\}, & \{2, 5\}, & \{1, 3, 6\}, \\ \{2, 3, 4\}, & \{\ 4\ \}, & \{1, 5, 6\} \end{array} \right\}$$

$$U = \{1, 2, 3, 4, 5, 6\}$$

$$S' = \left\{ \begin{array}{ccc} 110010 & , 010010 & , 101001 \\ 011100 & , 000100 & , 100011 \end{array} \right\}$$

$$k = 111111$$

$$S = \left\{ \begin{matrix} \{1, 2, 5\}, \{2, 5\}, \{1, 3, 6\}, \\ \{2, 3, 4\}, \{4\}, \{1, 5, 6\} \end{matrix} \right\}$$

$$U = \{1, 2, 3, 4, 5, 6\}$$

$$S' = \left\{ \begin{matrix} 110010, 010010, 101001 \\ 011100, 000100, 100011 \end{matrix} \right\}$$

$$k = 111111$$

The Basic Intuition

- Suppose there are n elements in the universe and k different sets.
- Replace each set S with a number that is 1 in its ith position if $i \in S$ and has a 0 in its ith position otherwise.
- Set k to a number that is n digits long, where each digit is a 1.
  - To avoid "overflowing" one column into another, assume the numbers are written in base $k+1$.

In Pseudocode

```
boolean hasExactCover(List<Set> sets,
                      Set<Universe>) {
  return hasSubsetSum(setsToNumbers(sets),
                      lotsOfOnes(sets));
}
```

Reductions and **NP**

- ***Theorem:*** If $A \leq_p B$ and $B \in \mathbf{NP}$, then $A \in \mathbf{NP}$.
- ***Proof sketch:*** Show that this pseudocode runs in ***nondeterministic*** polynomial time:

```
boolean solveProblemA(input) {
   return solveProblemB(transform(input));
}
```

Calling `transform` only takes polynomial time. The `transform` function can only produce an output that's polynomially larger than its input. Feeding that into a polynomial-time function `solveProblemB` then only takes ***nondeterministic*** polynomial time. Overall, this is a ***nondeterministic*** polynomial-time algorithm for A, so $A \in \mathbf{NP}$. ■

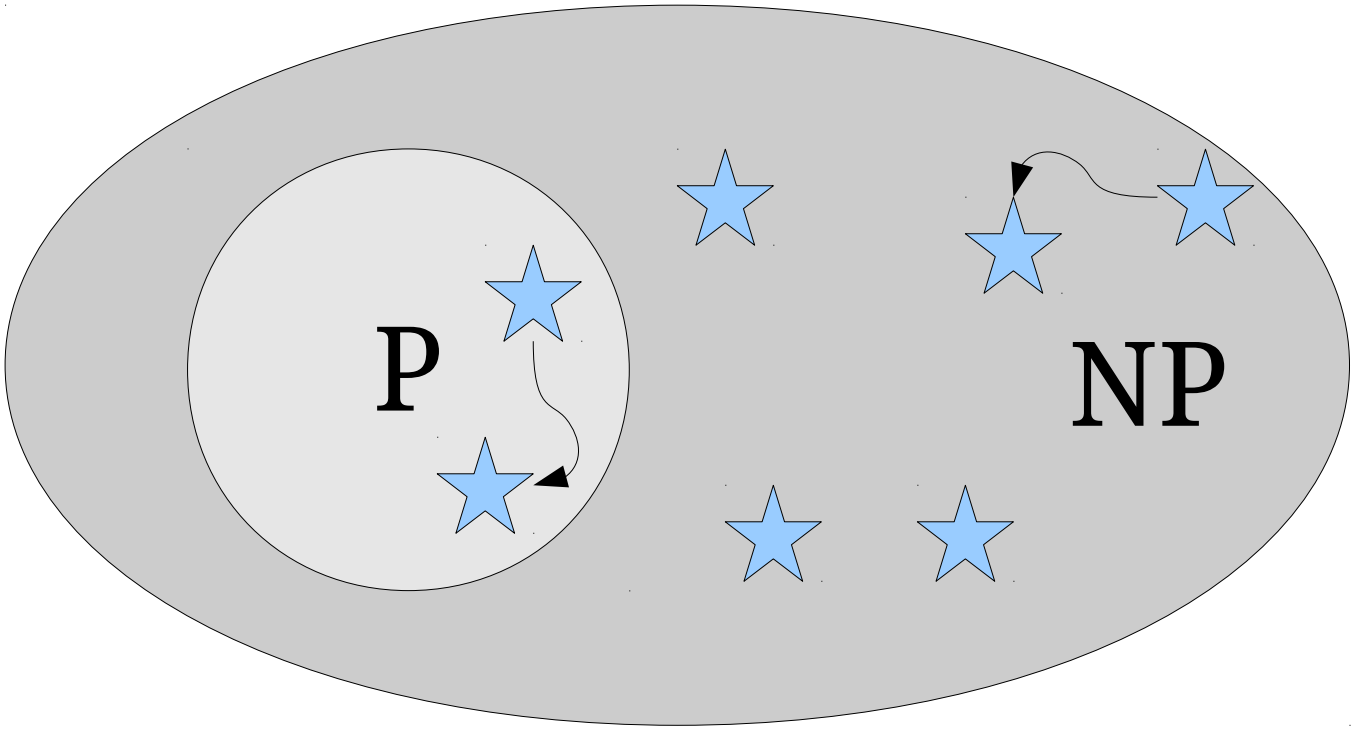
Reducibility, Summarized

- A polynomial-time reduction is a way of transforming inputs to problem A into inputs to problem B that preserves the correct answer.
- If there is a polynomial-time reduction from A to B, we denote this by writing $A \leq_p B$.
- Two major theorems:
  - ***Theorem:*** If $B \in \mathbf{P}$ and $A \leq_p B$, then $A \in \mathbf{P}$.
  - ***Theorem:*** If $B \in \mathbf{NP}$ and $A \leq_p B$, then $A \in \mathbf{NP}$.

NP-Hardness and **NP**-Completeness

Polynomial-Time Reductions

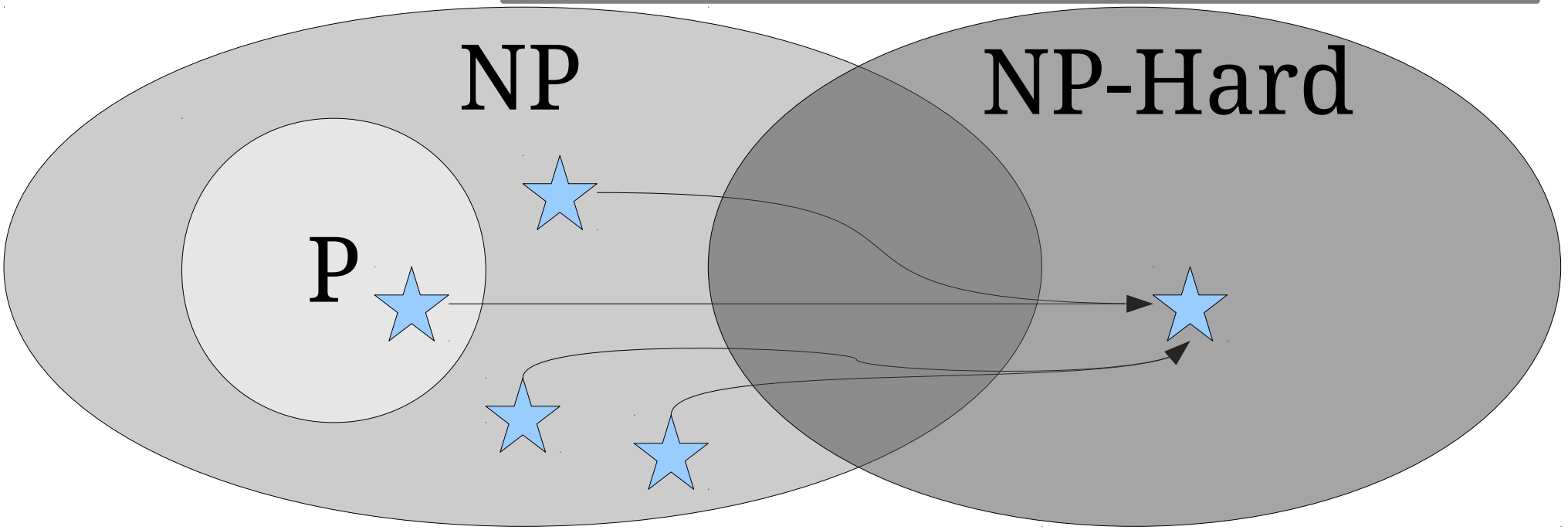
- If $L_1 \leq_P L_2$ and $L_2 \in \mathbf{P}$, then $L_1 \in \mathbf{P}$.
- If $L_1 \leq_P L_2$ and $L_2 \in \mathbf{NP}$, then $L_1 \in \mathbf{NP}$.



NP-Hardness

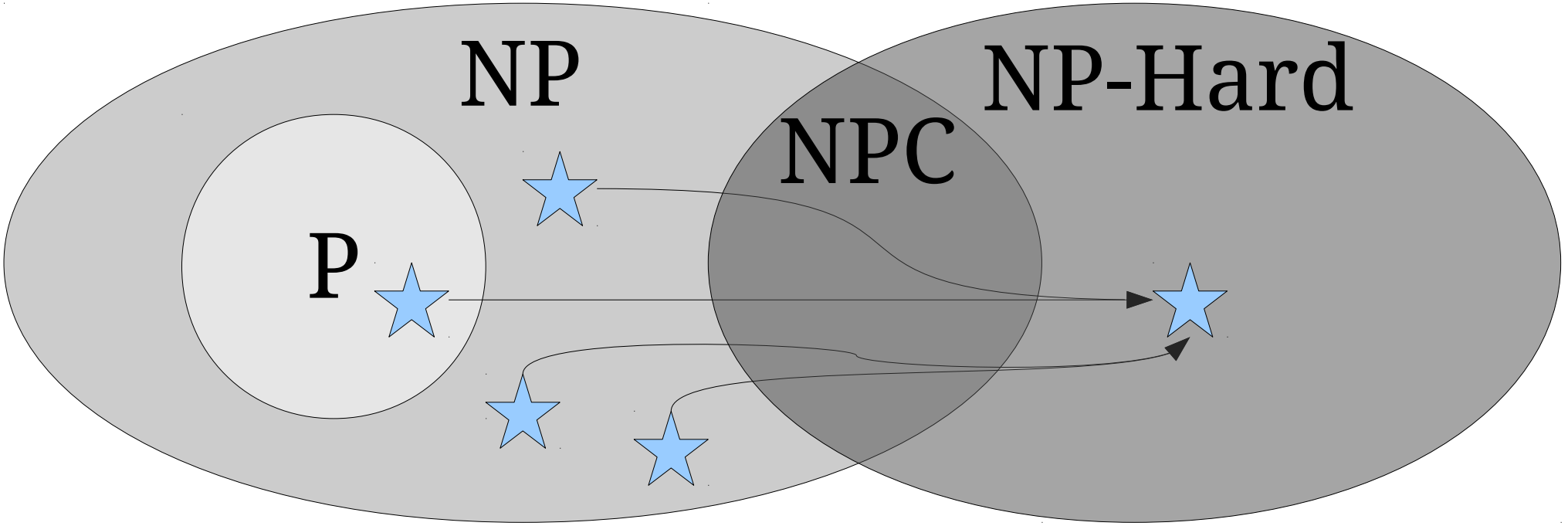
- A language L is called **NP-hard** if for *every* $L' \in \mathbf{NP}$, we have $L' \leq_P L$.

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

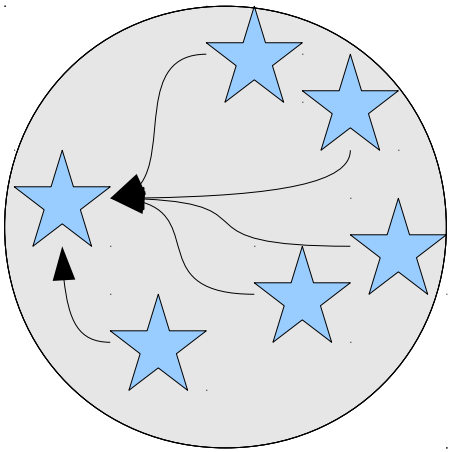
- A language L is called **NP-hard** if for *every* $L' \in \mathbf{NP}$, we have $L' \leq_P L$.
- A language in L is called **NP-complete** if L is **NP**-hard and $L \in \mathbf{NP}$.
- The class **NPC** is the set of **NP**-complete problems.



The Tantalizing Truth

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

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

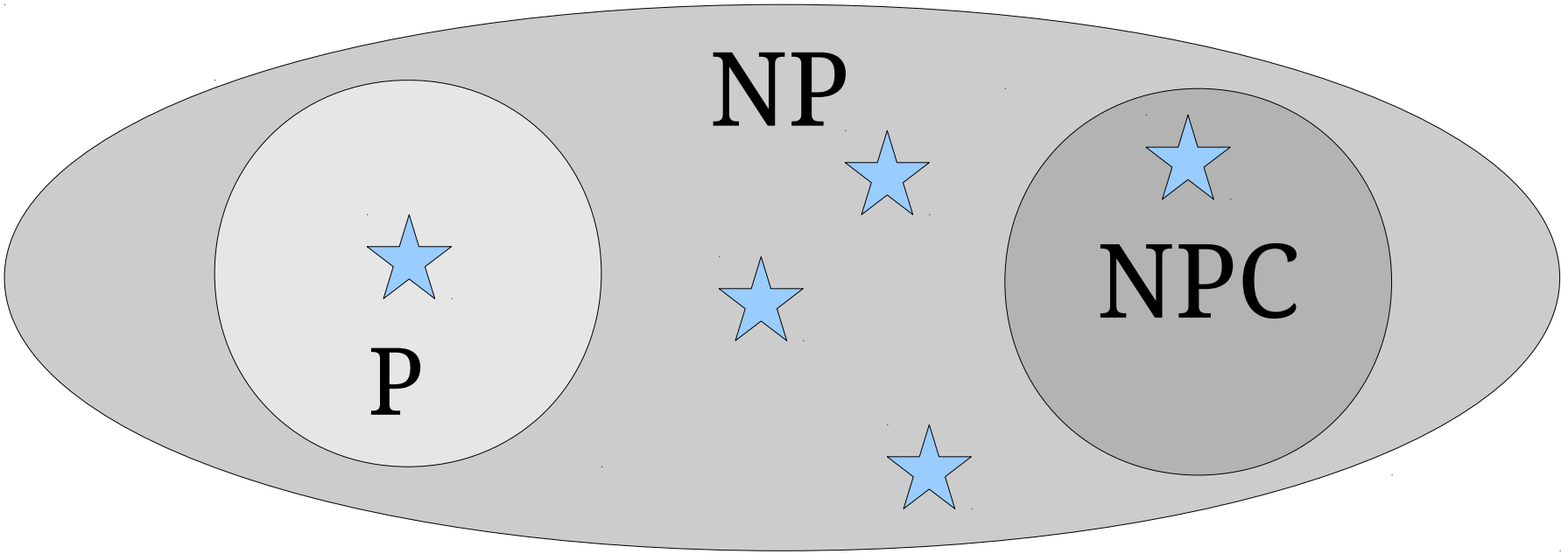


P = NP

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**. ■



A Feel for **NP**-Completeness

- If a problem is **NP**-complete, then under the assumption that **P ≠ NP**, there cannot be an efficient algorithm for it.
- In a sense, **NP**-complete problems are the hardest problems in **NP**.
- All known **NP**-complete problems are enormously hard to solve:
  - All known algorithms for **NP**-complete problems run in worst-case exponential time.
  - Most algorithms for **NP**-complete problems are infeasible for reasonably-sized inputs.

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: Take CS154!

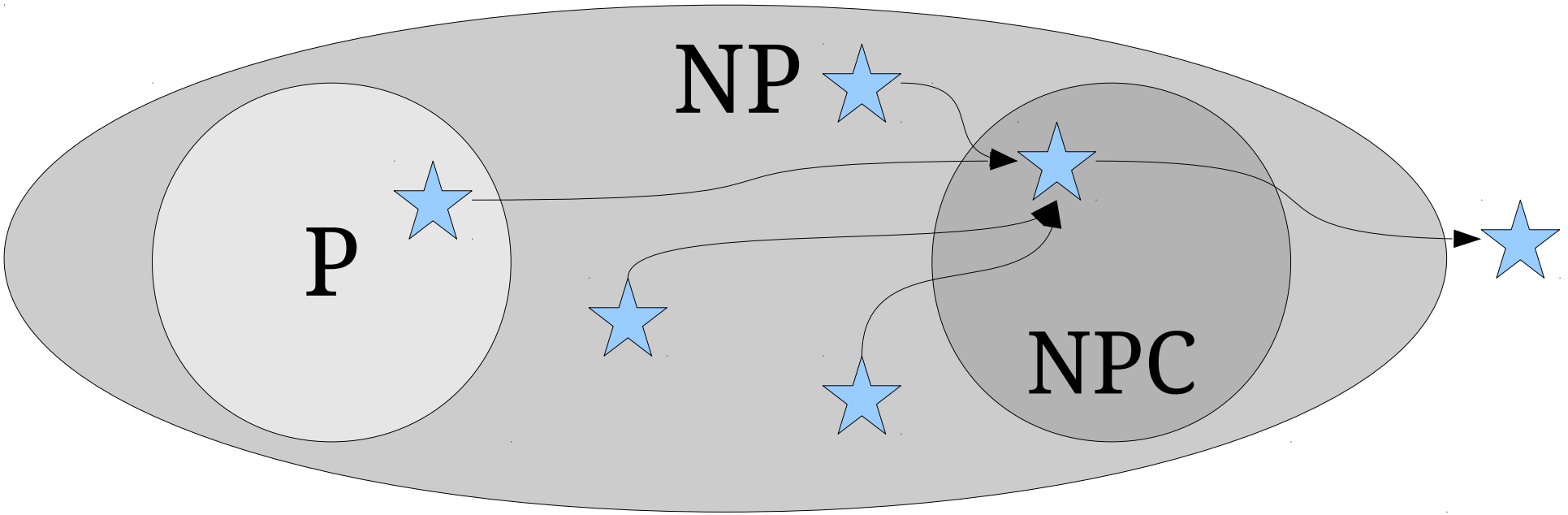
Finding Additional **NP**-Complete Problems

NP-Completeness

Theorem: Let L_1 and L_2 be languages. If $L_1 \leq_P L_2$ and L_1 is **NP**-hard, then L_2 is **NP**-hard.

NP-Completeness

Theorem: Let L_1 and L_2 be languages. If $L_1 \leq_P L_2$ and L_1 is **NP**-hard, then L_2 is **NP**-hard.



NP-Completeness

Theorem: Let L_1 and L_2 be languages. If $L_1 \leq_P L_2$ and L_1 is **NP**-hard, then L_2 is **NP**-hard.

Theorem: Let L_1 and L_2 be languages where $L_1 \in \mathbf{NPC}$ and $L_2 \in \mathbf{NP}$. If $L_1 \leq_P L_2$, then $L_2 \in \mathbf{NPC}$.