Complexity Theory

Part Two

Recap from Last Time

The Complexity Class **P**

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

  P = { L | 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:

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

- 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:

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

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

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

$$\mathbf{R} = \{ L \mid \text{there is a } \sout{\text{polynomial-time}} \text{ decider for } L \}$$

$$\mathbf{RE} = \{ L \mid \text{there is a } \sout{\text{polynomial-time}} \text{ verifier for } L \}$$

We know that **R ≠ RE**.

So does that mean **P ≠ NP**?

A Problem

- The **R** and **RE** languages correspond to problems that can be decided and verified, *period*, without any time bounds.
- To reason about what's in **R** and what's in **RE**, we used two key techniques:
  - ***Universality***: TMs can simulate other TMs.
  - ***Self-Reference***: TMs can get their own source code.
- Why can't we just do that for **P** and **NP**?

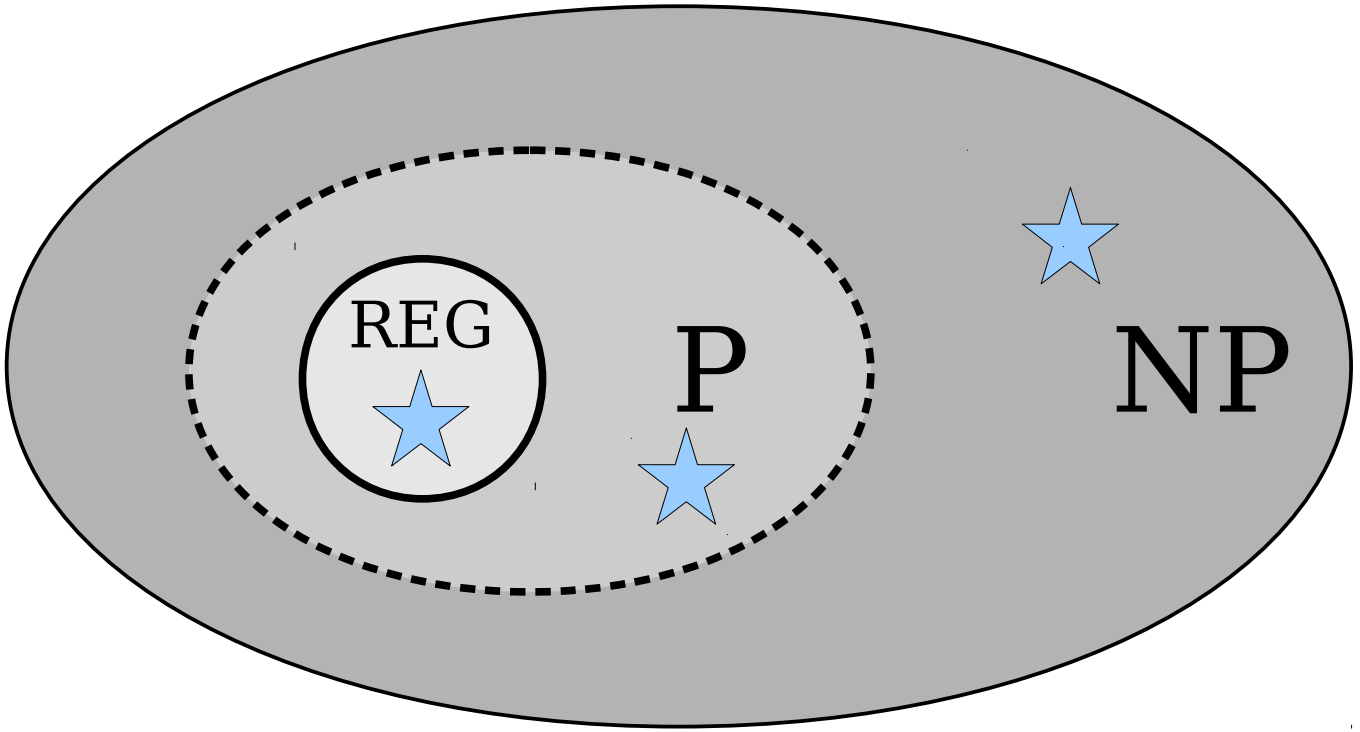
Theorem (Baker-Gill-Solovay): Any proof that purely relies on universality and self-reference cannot resolve $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$.

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?

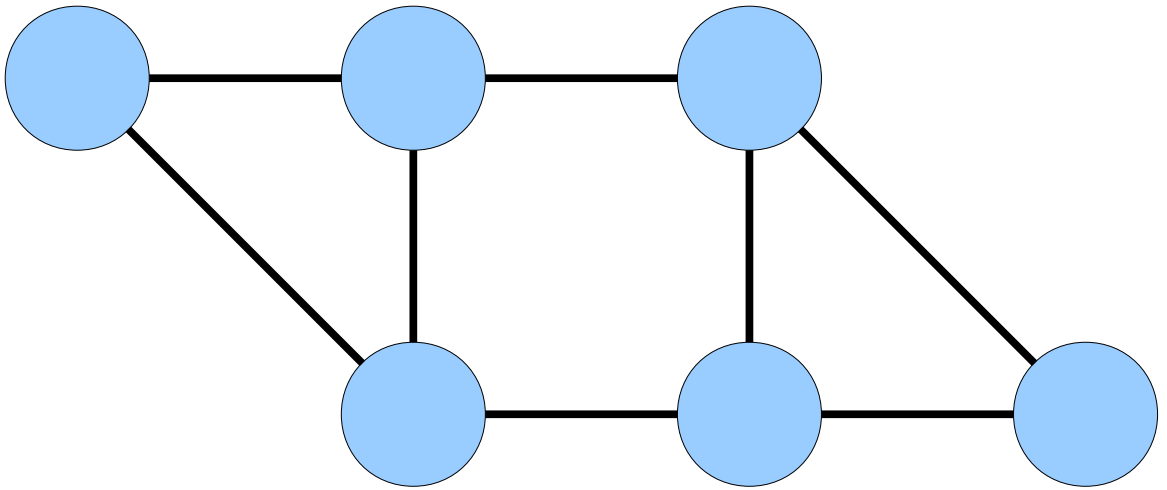
Reducibility

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.

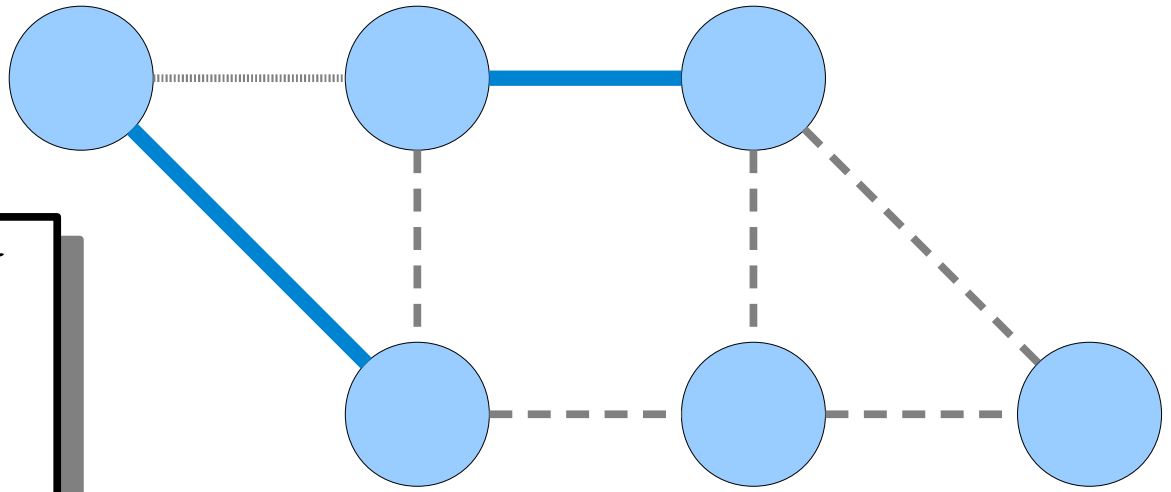
Maximum Matching

- 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

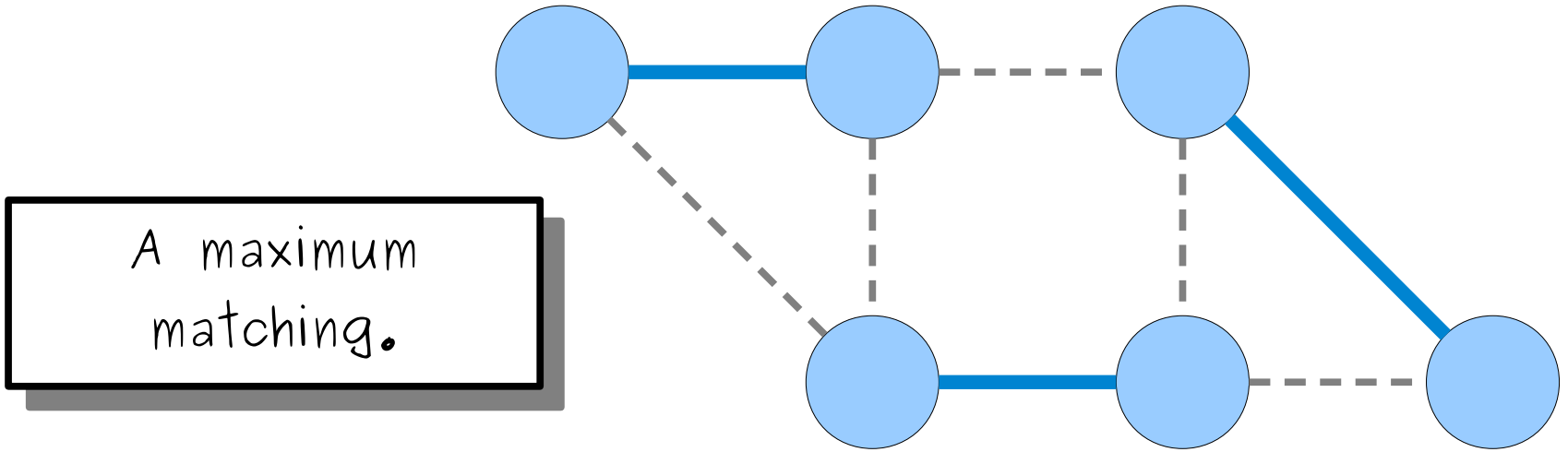
- Given an undirected graph G, a ***matching*** in G is a set of edges such that no two edges share an endpoint.
- A ***maximum matching*** is a matching with the largest number of edges.



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.

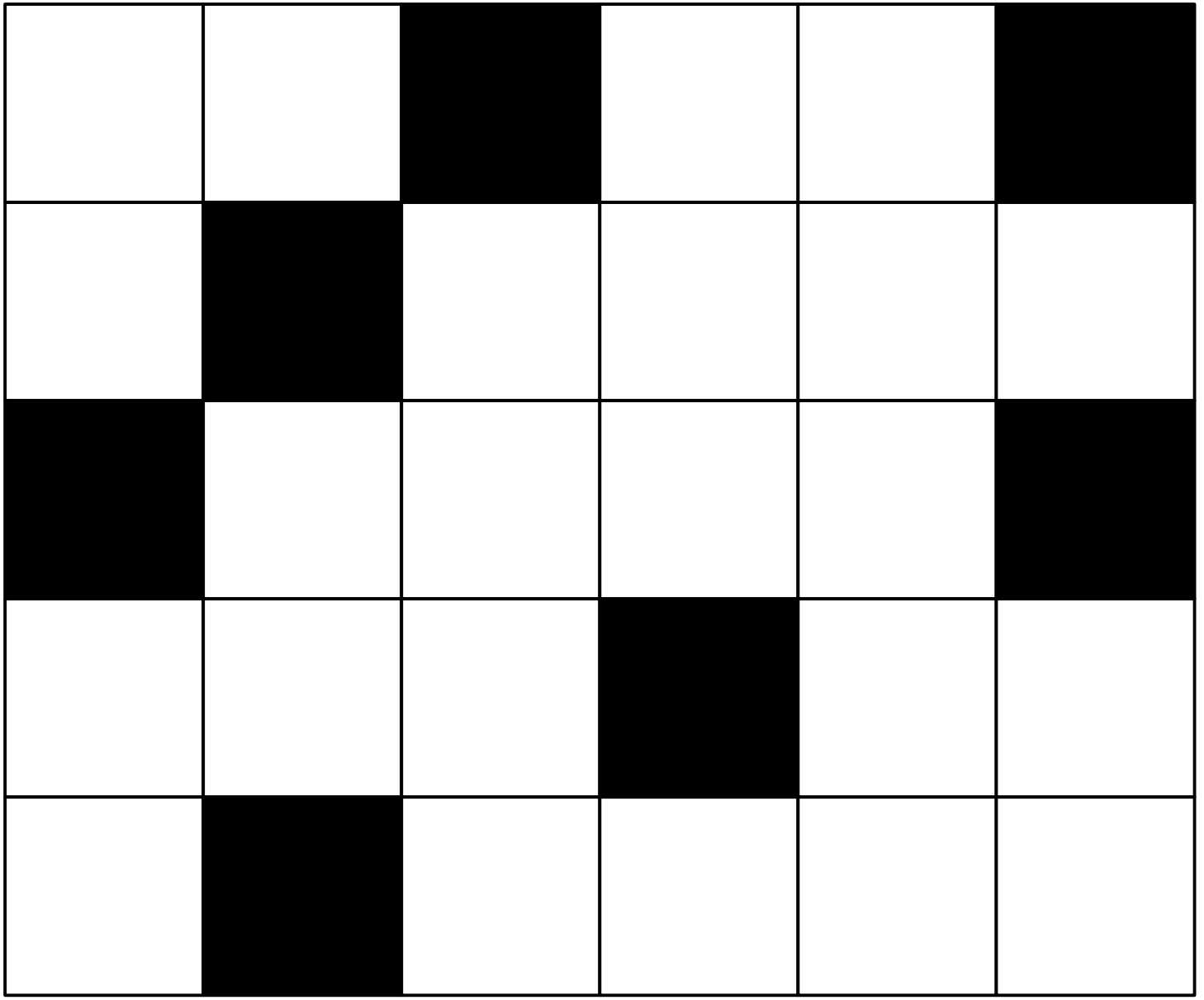


Maximum Matching

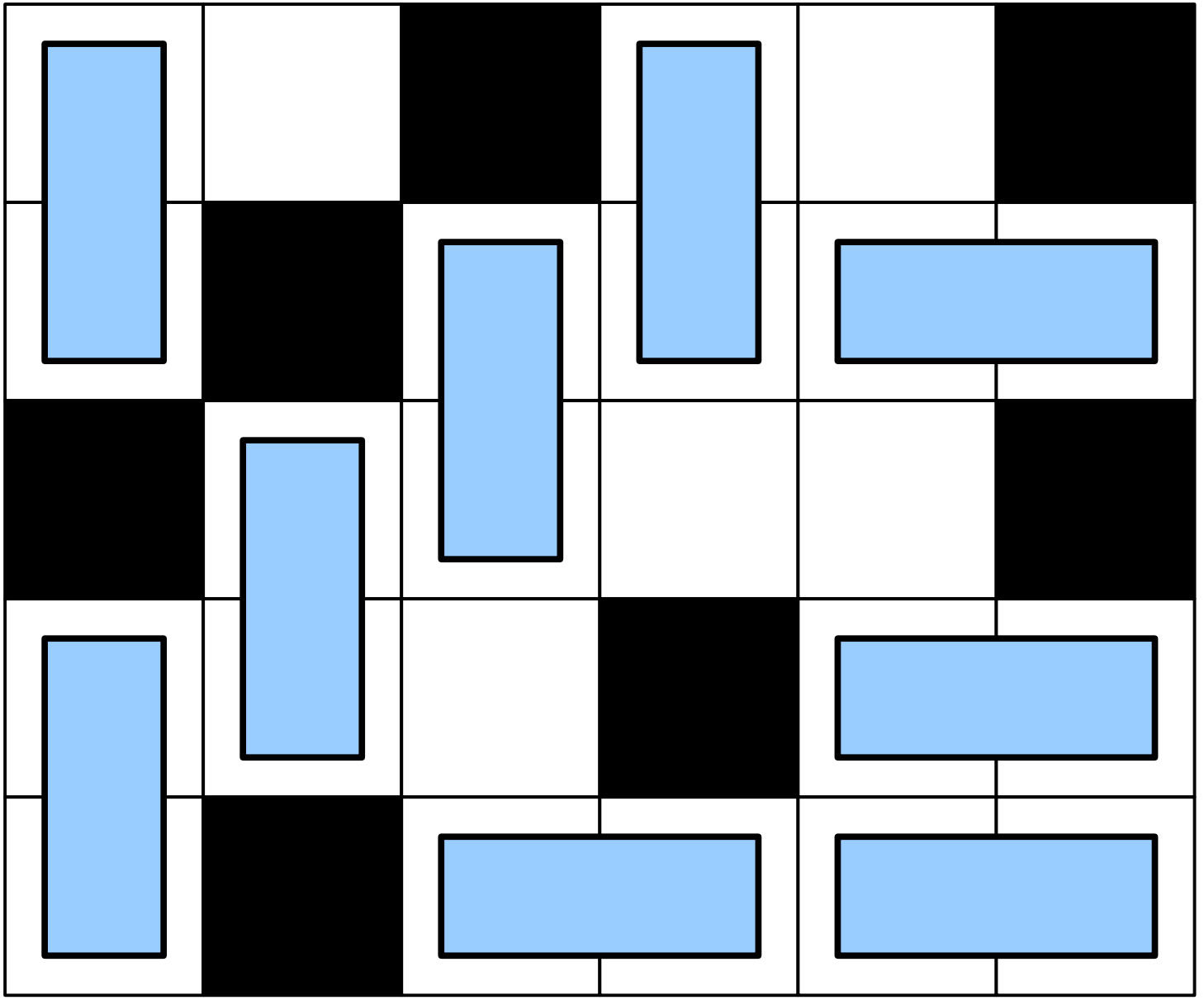
- Jack Edmonds' paper “Paths, Trees, and Flowers” gives a polynomial-time algorithm for finding maximum matchings.

- Using this fact, what other problems can we solve?

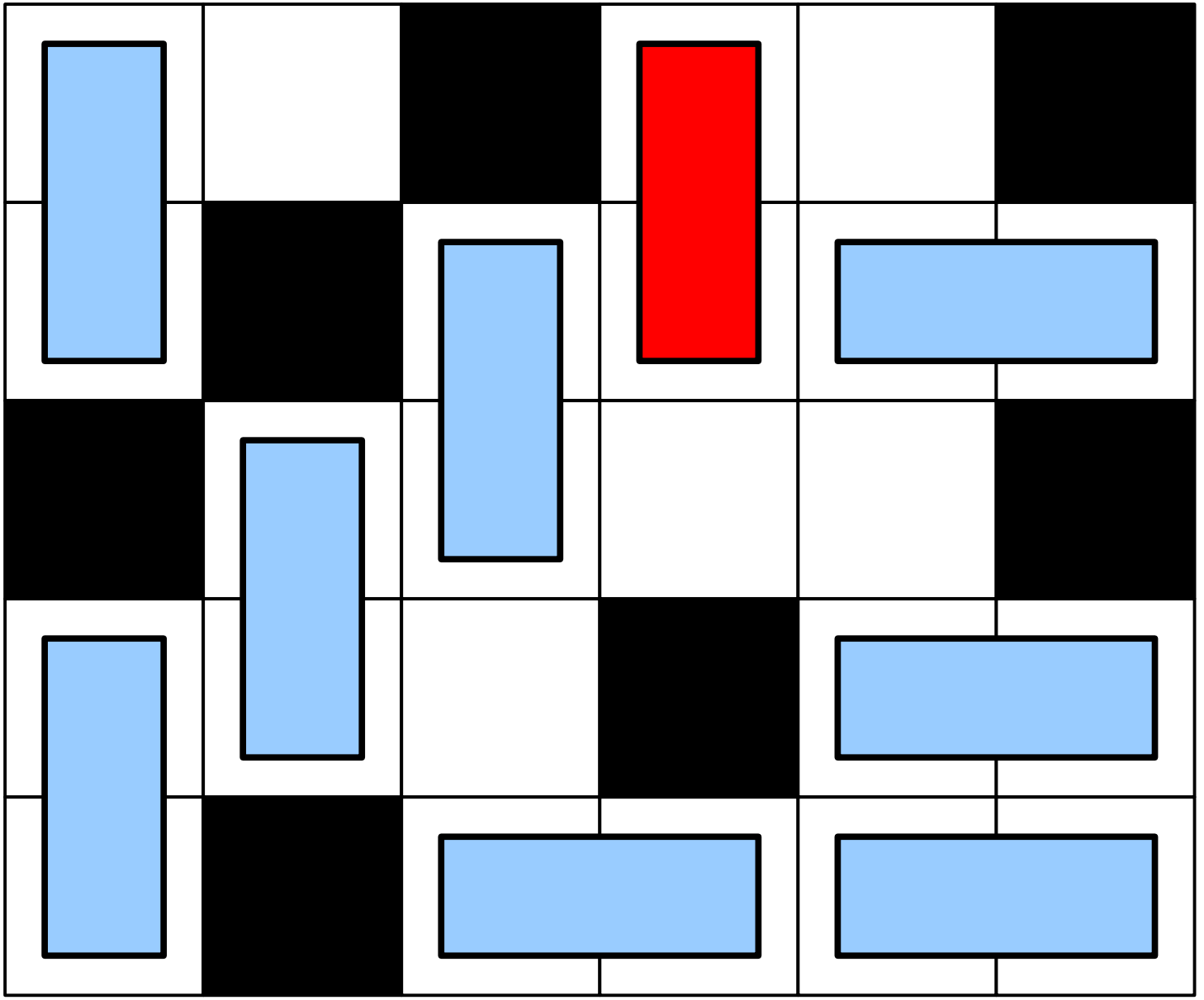
Domino Tiling



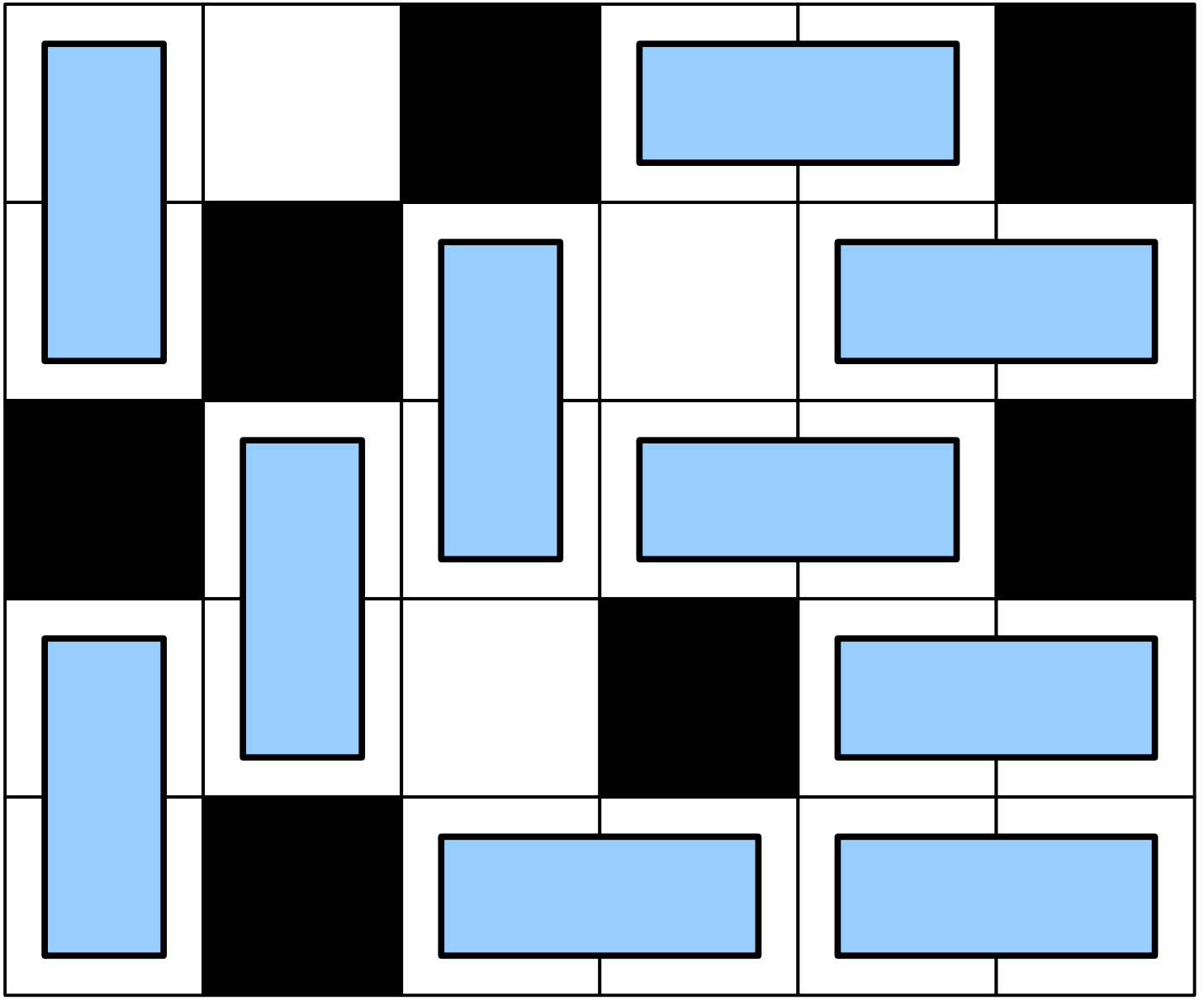
Domino Tiling



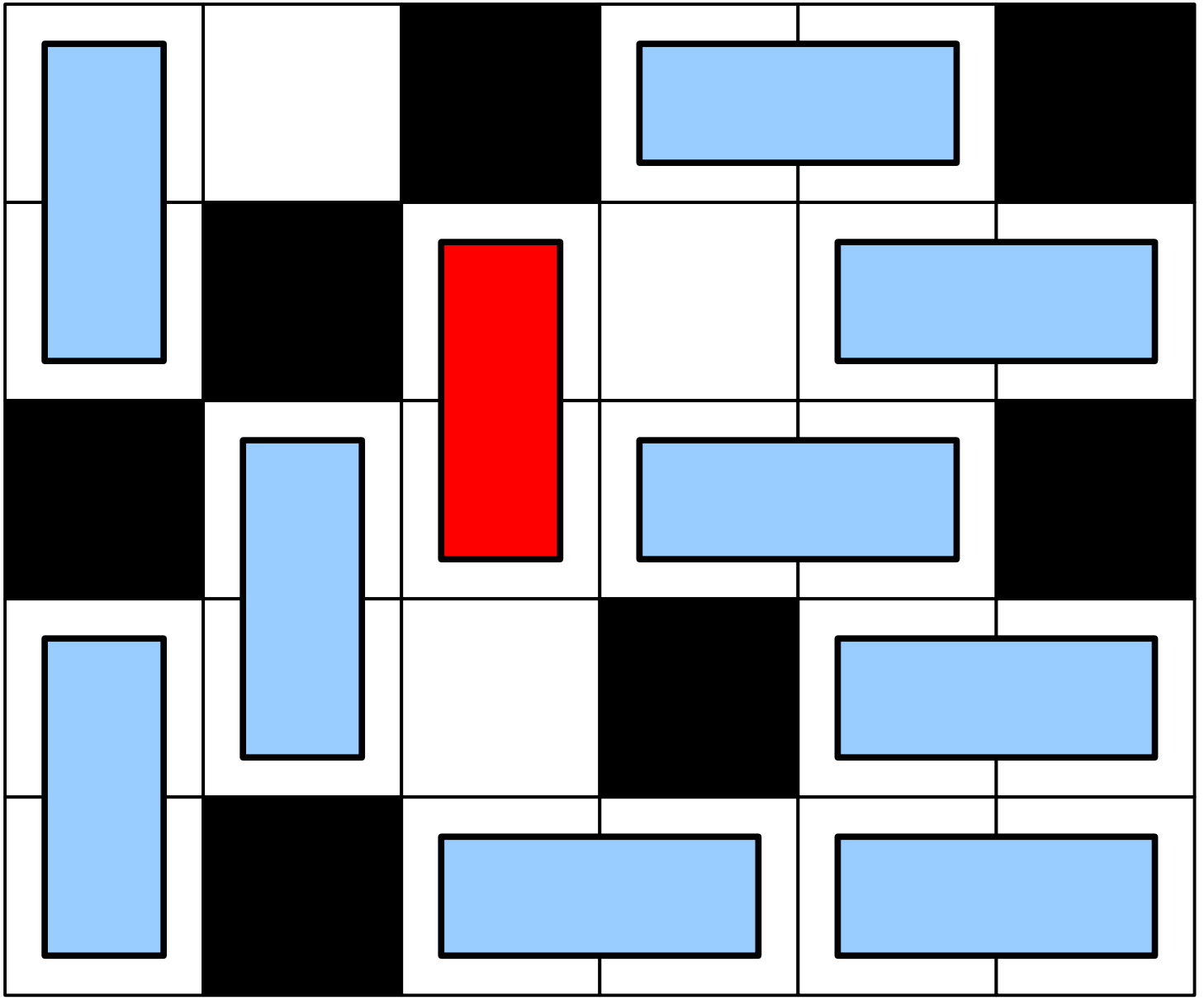
Domino Tiling



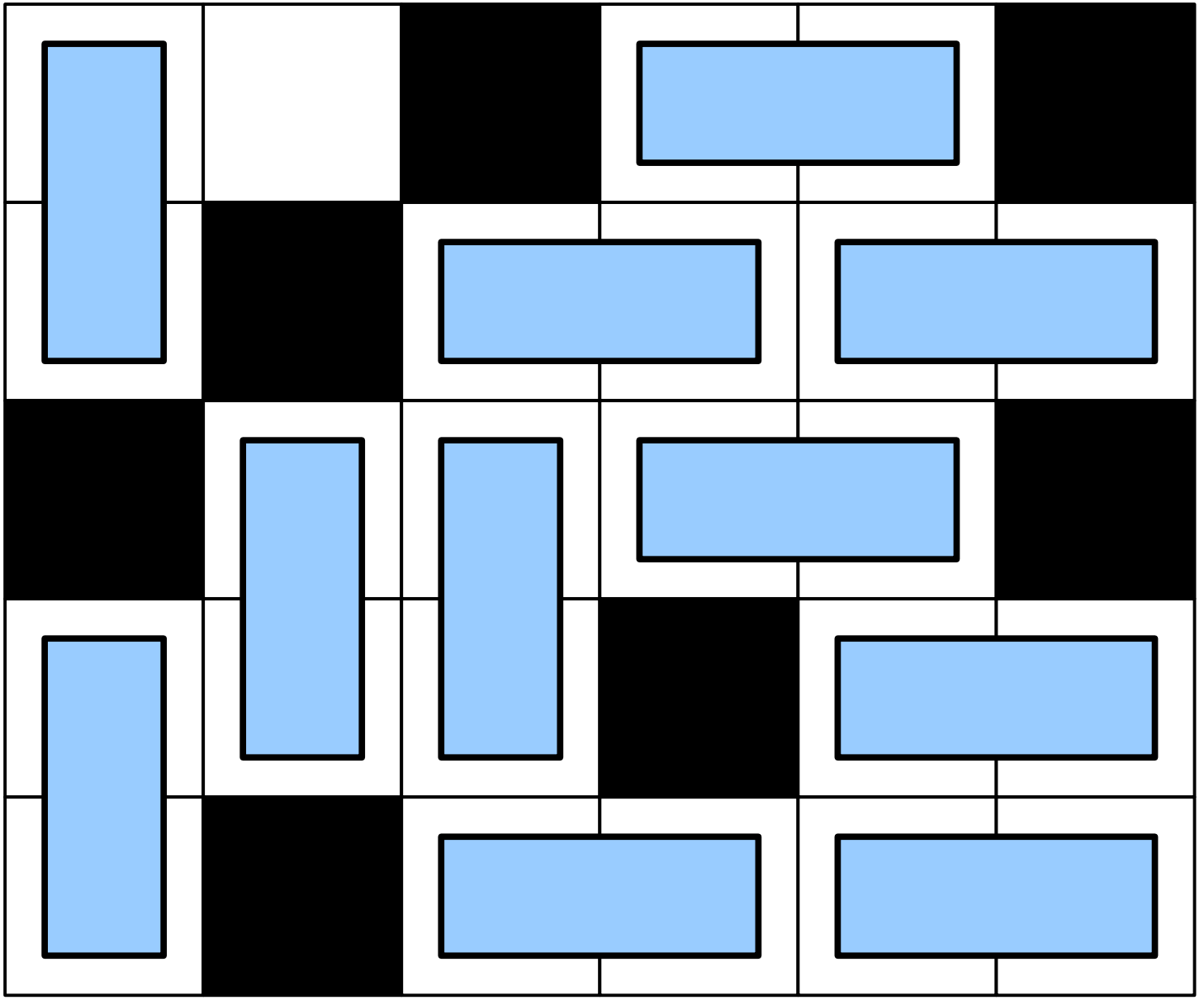
Domino Tiling



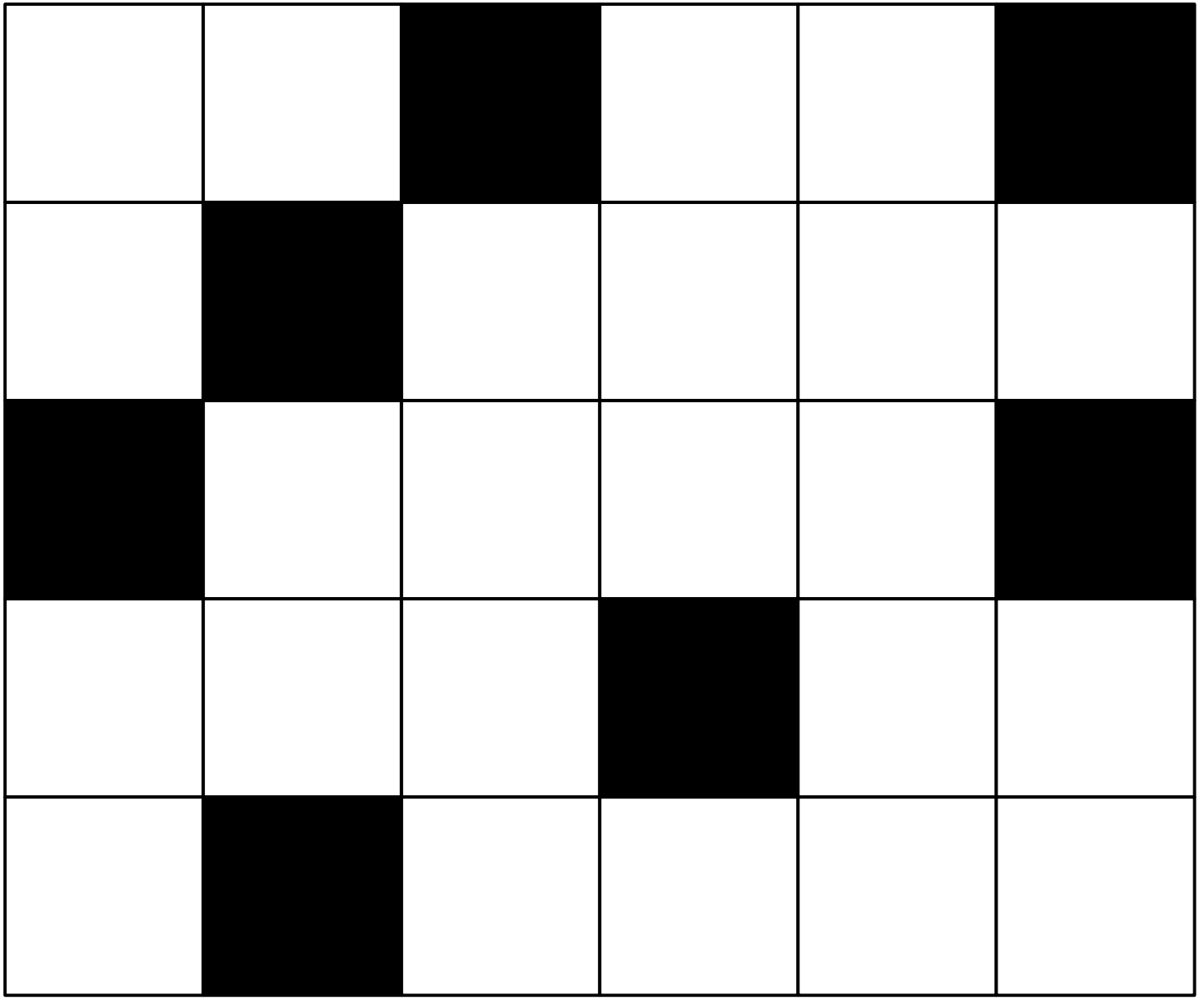
Domino Tiling



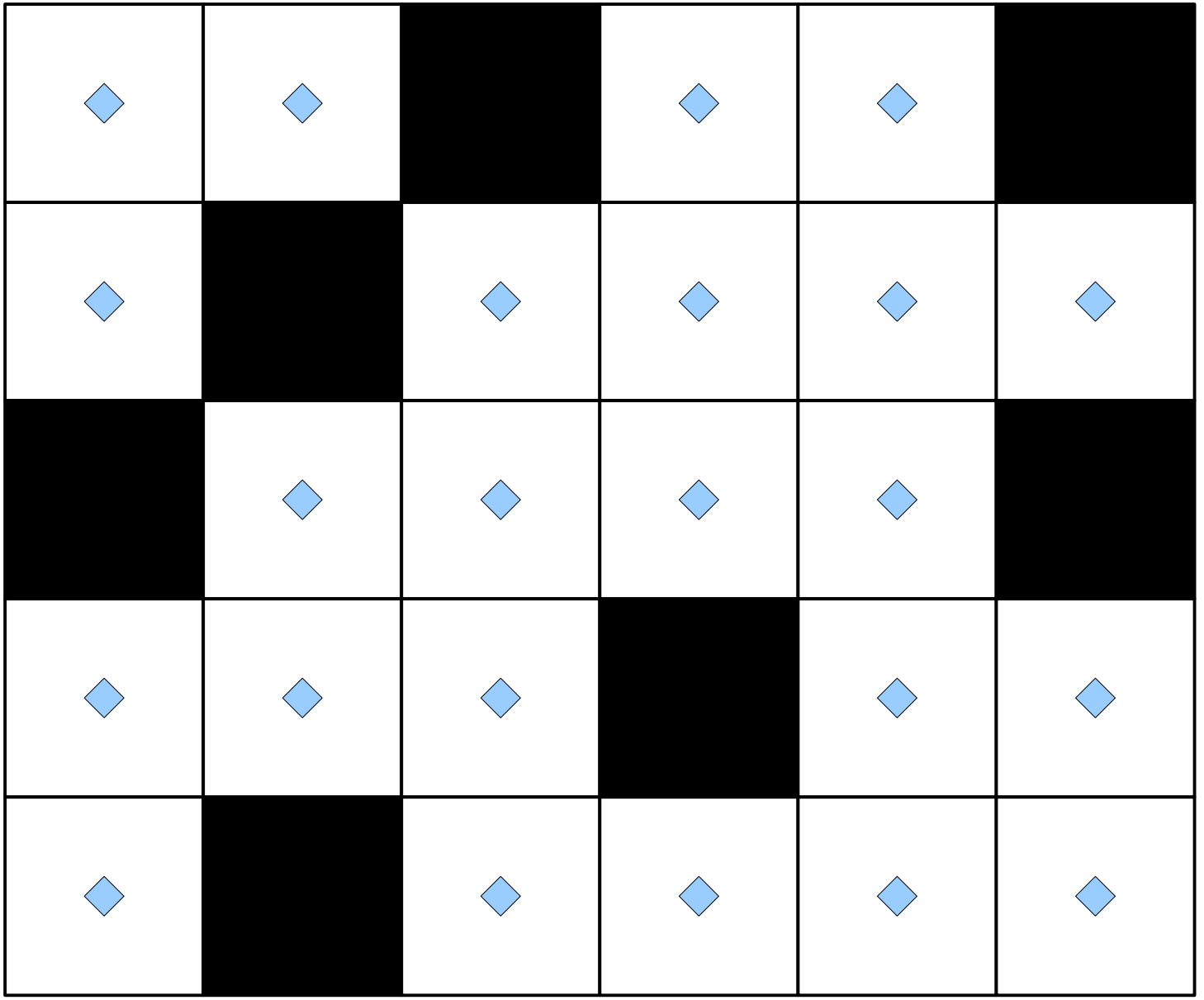
Domino Tiling



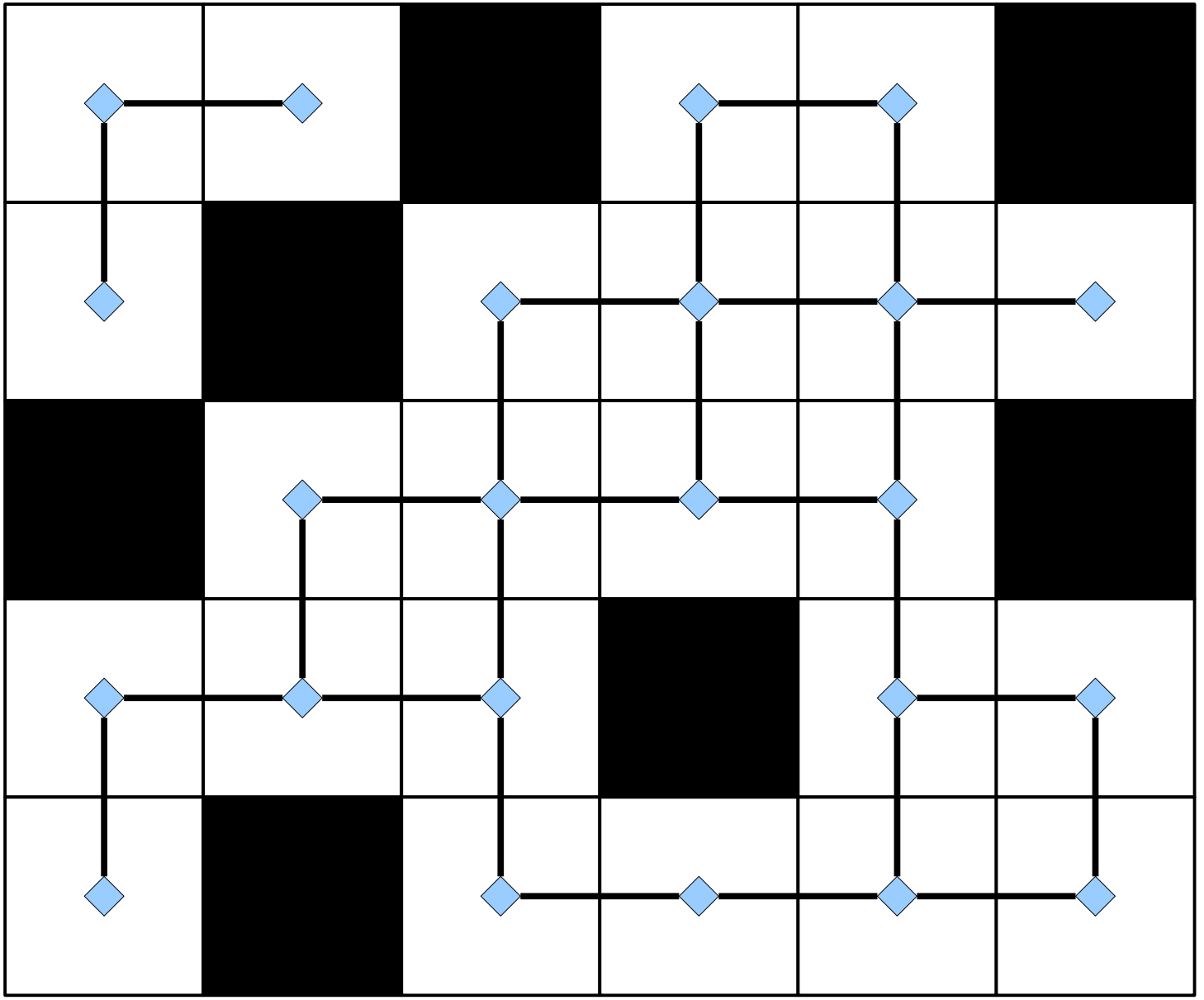
Solving Domino Tiling



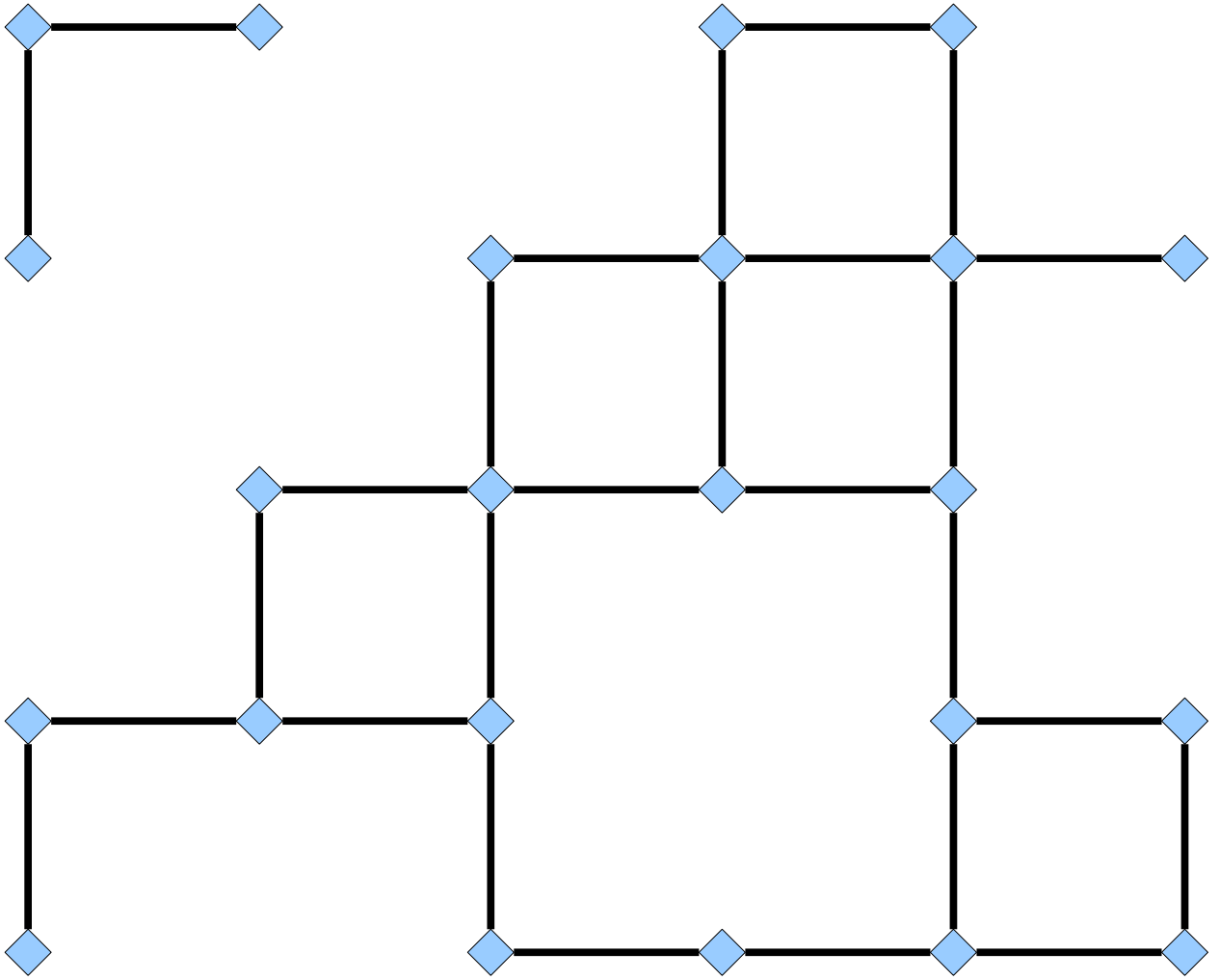
Solving Domino Tiling



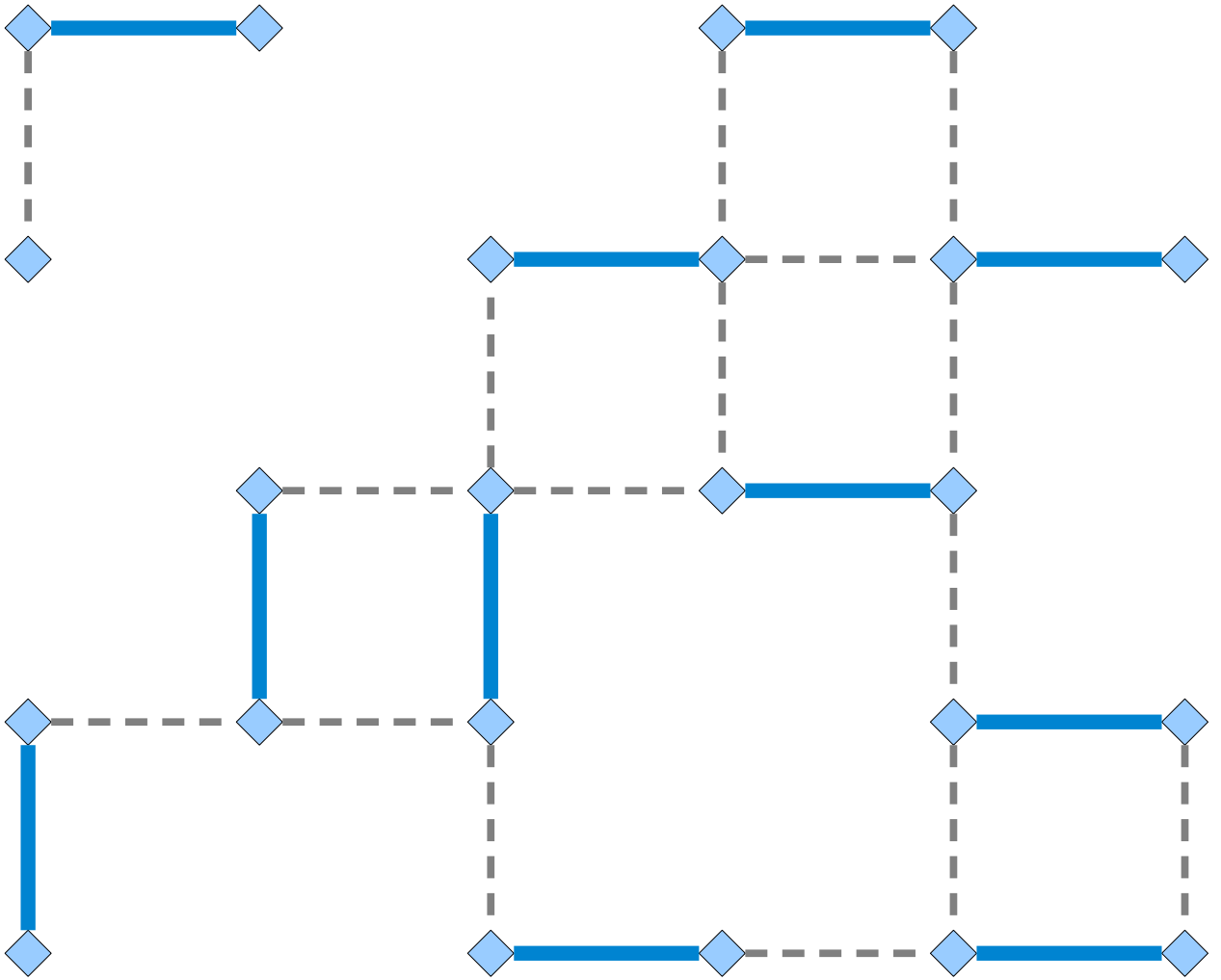
Solving Domino Tiling



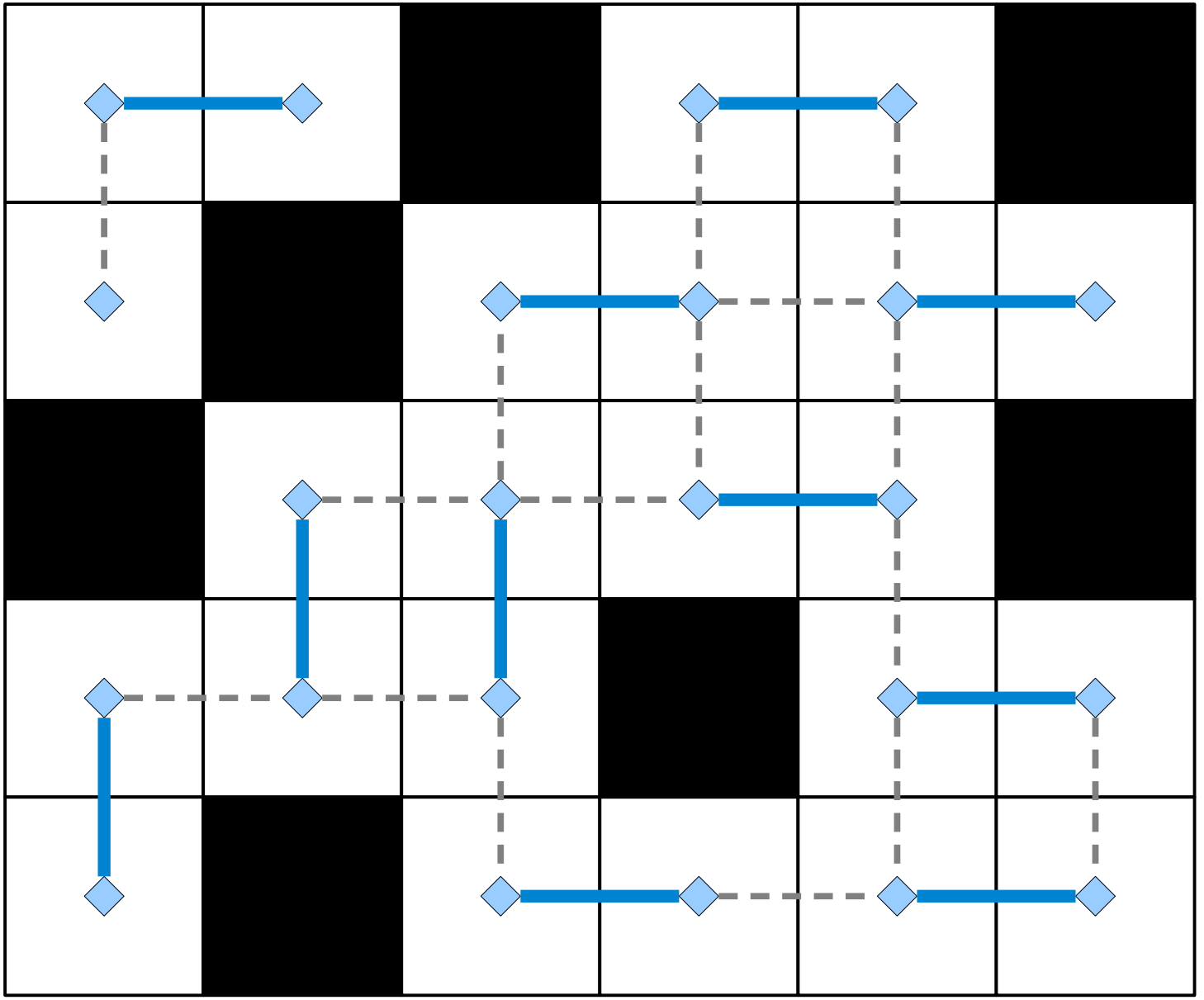
Solving Domino Tiling



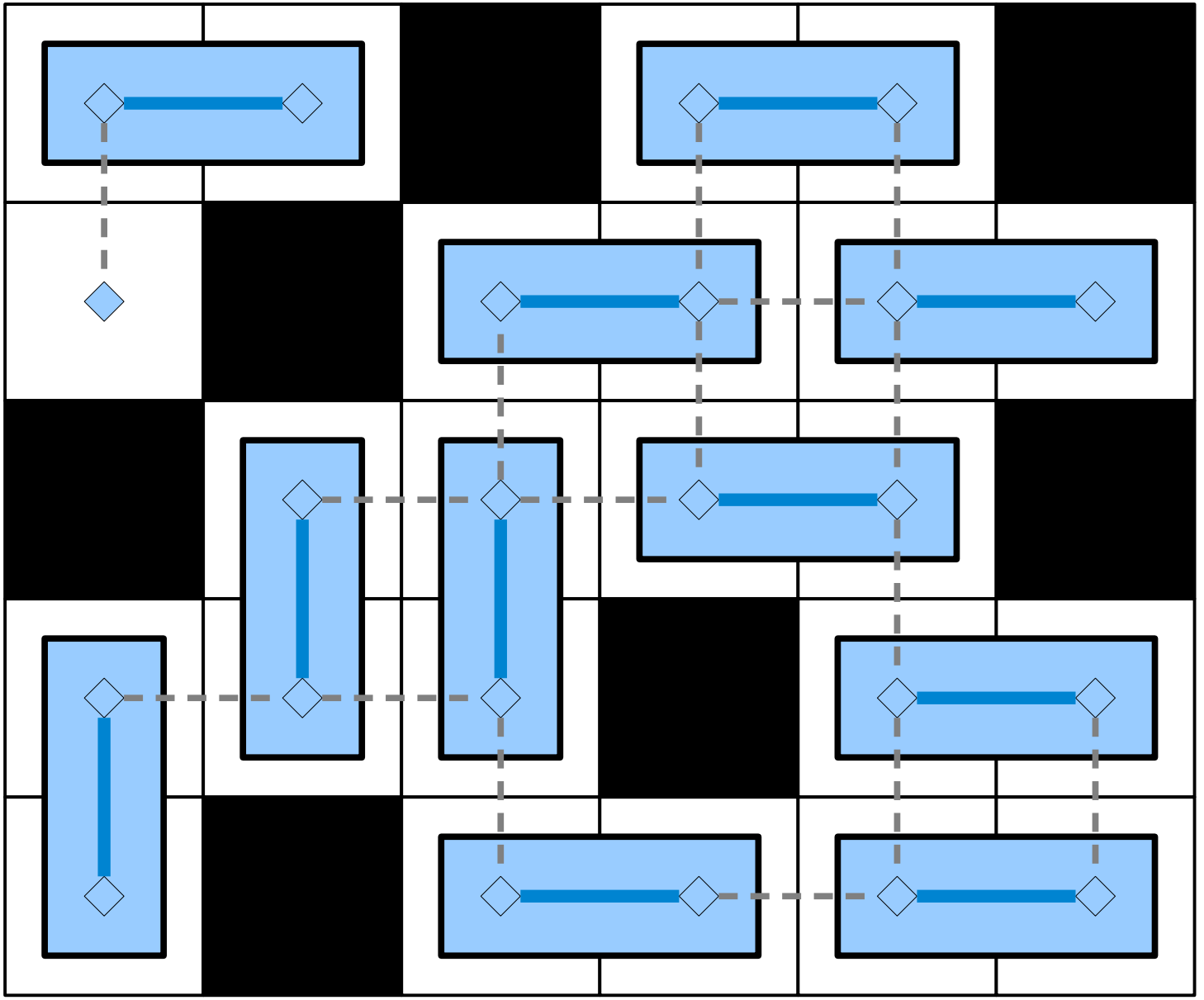
Solving Domino Tiling



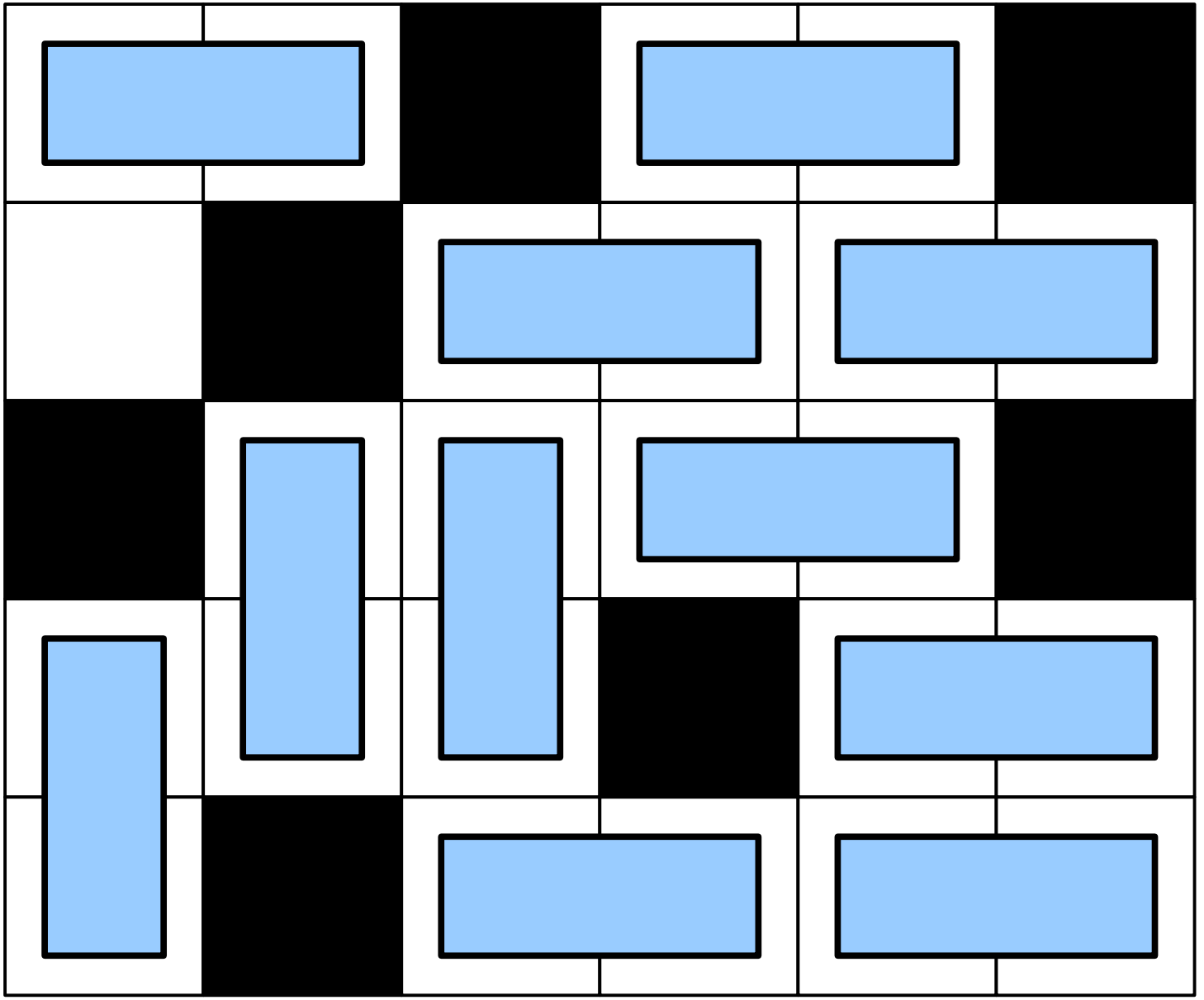
Solving Domino Tiling



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 on a 2D grid can't be "harder" than solving maximum matching.

B. Solving maximum matching can't be "harder" than solving domino tiling on a 2D grid.

C. Both A and B.

Go to PollEv.com/cs103spr25

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.

```
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

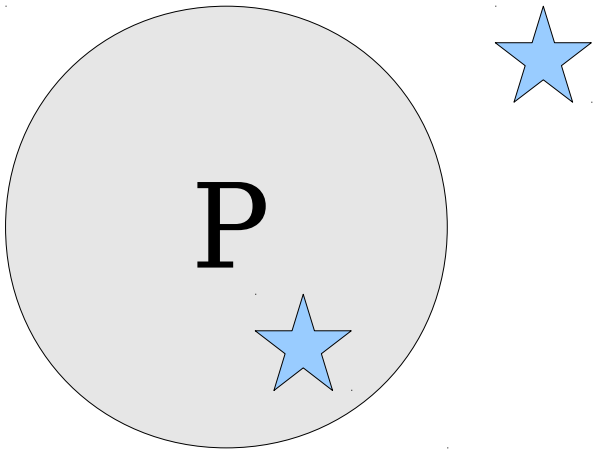
$$A \leq_{\mathrm{p}} B.$$

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

* Assuming that `translate` runs in polynomial time.

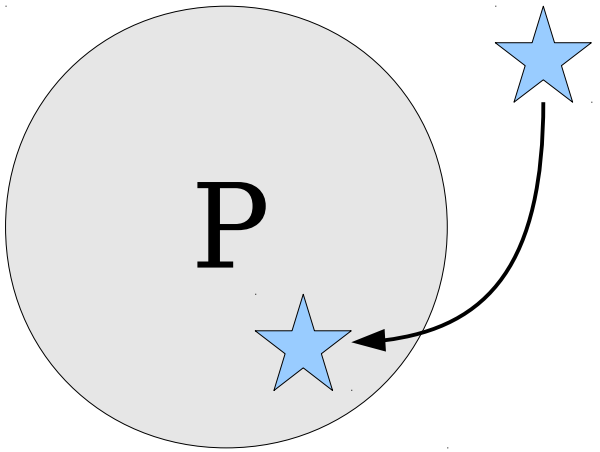
Polynomial-Time Reductions

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



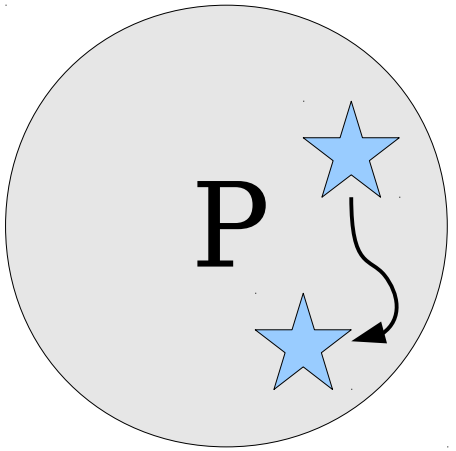
Polynomial-Time Reductions

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



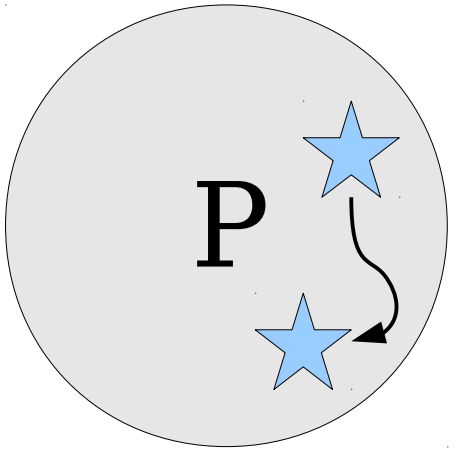
Polynomial-Time Reductions

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



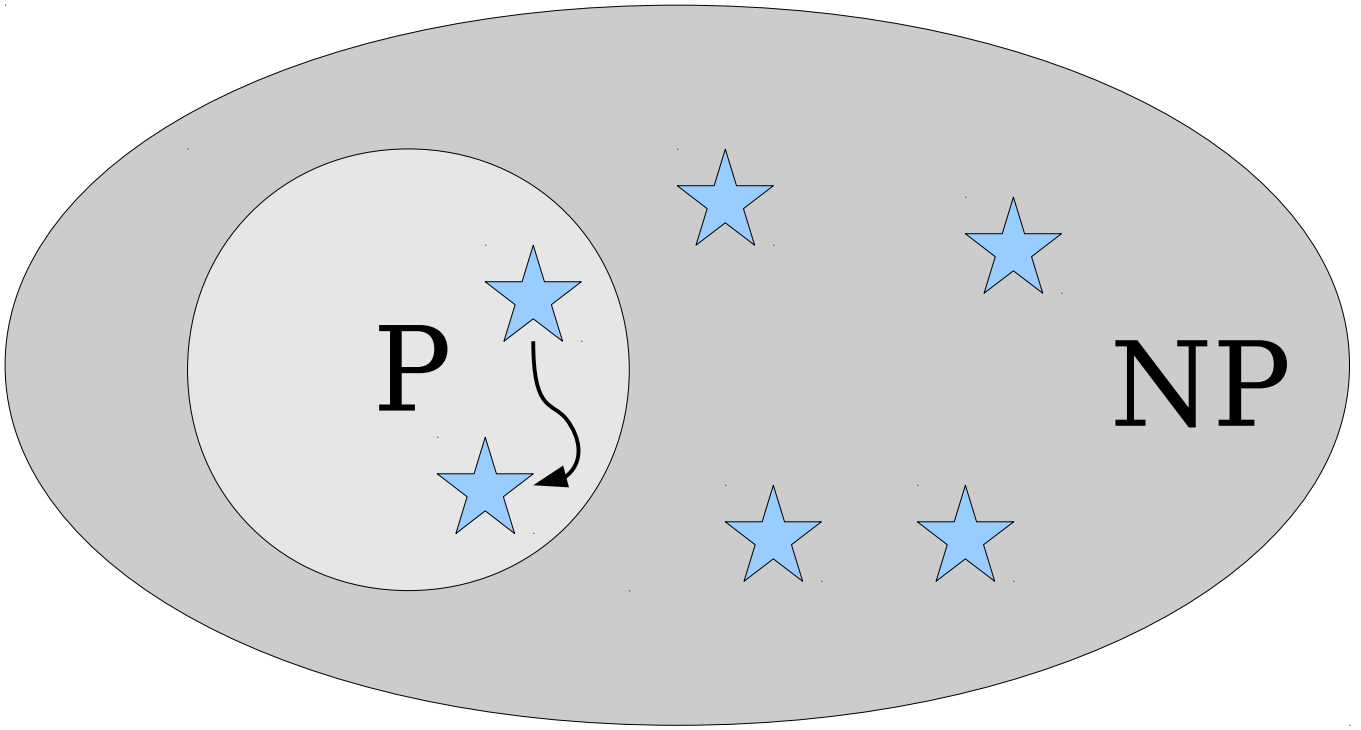
Polynomial-Time Reductions

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



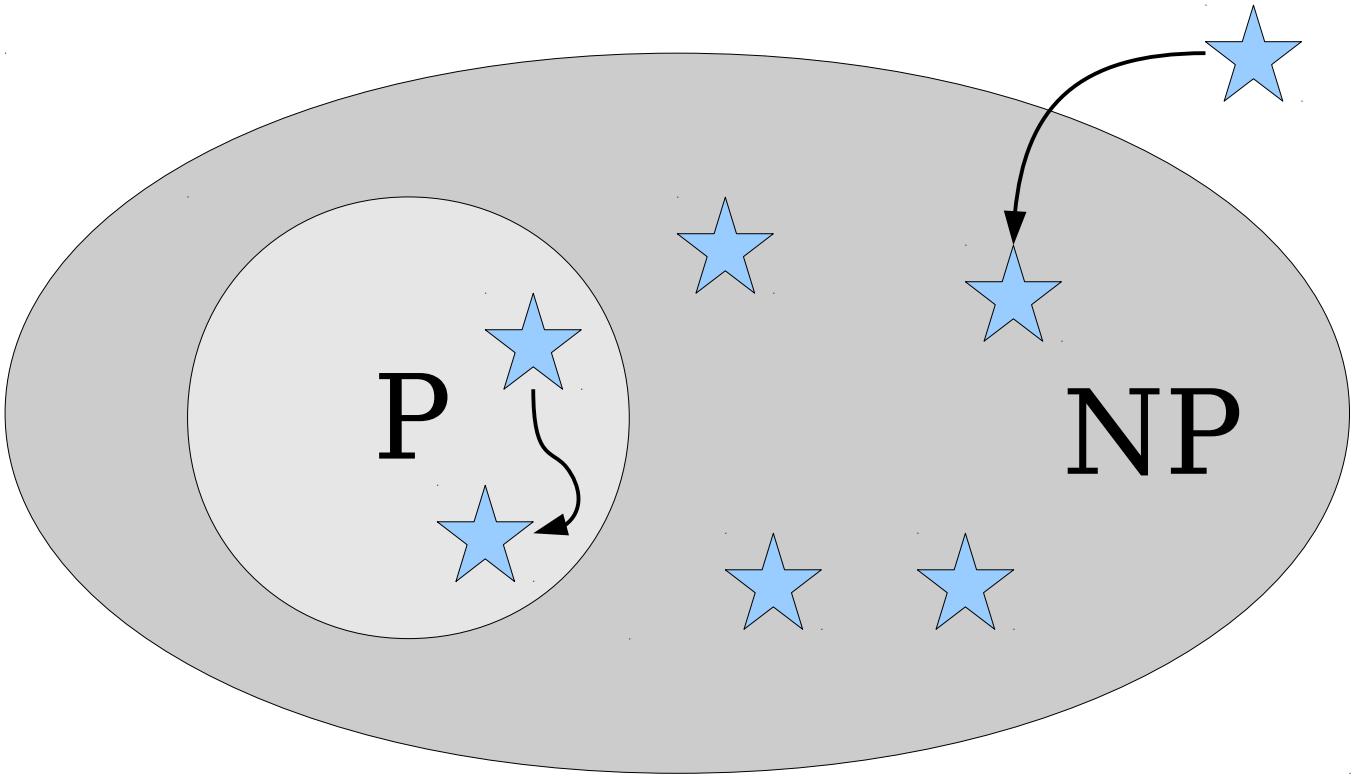
Polynomial-Time Reductions

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



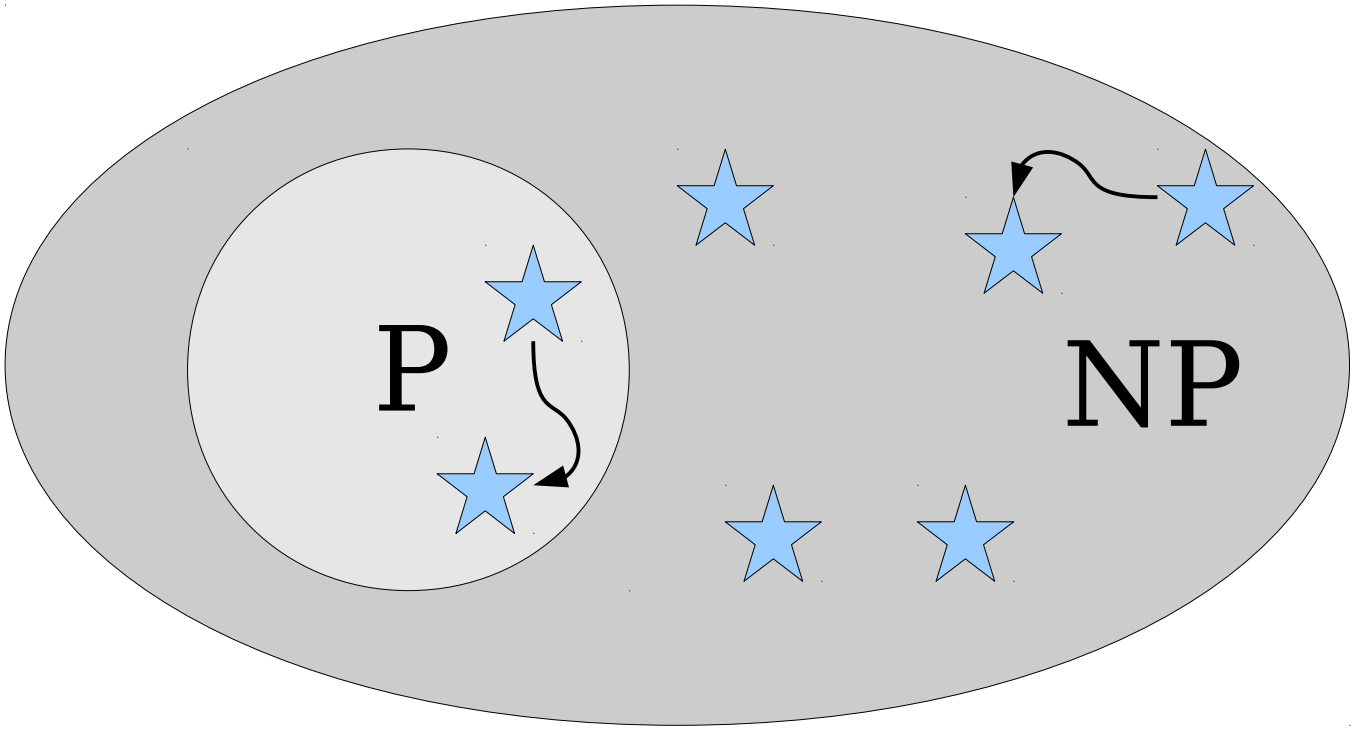
Polynomial-Time Reductions

- 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?

Another Example

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?

$$p \wedge q$$

$$p \wedge \neg p$$

$$p \rightarrow (q \wedge \neg q)$$

- 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:

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

- Finding good algorithms for SAT is important in practice and lot of effort has been devoted to developing tools for SAT.
- But SAT also played a starring role in the formulation of P vs. NP.

An Idea

- We will show how to encode a computation of a polynomial time Turing Machine M on input x as a propositional logic formula F

M(x) accepts iff F is satisfiable

Components

- We must show how to
  - Encode M's tape
  - Encode the tape head
  - Encode which instruction is being executed
  - Encode each instruction
  - Encode accepting/rejecting

- We are just going to sketch this out ...

Insight

- Let's say TM M runs for at most n steps
  - Could be any polynomial in n, but let's keep things simple
- Then M can access at most n tape cells
- We can represent the history of M's entire computation as an n x n array
  - One row for each of n time steps
  - n cells in a row for the tape contents at each step
  - Some of the cells in the array may not be used, but that's OK, what's important is that we can give a (polynomial in n) upper bound on the number of cells needed

The Tape

- Assume the tape alphabet is {true, false}
- Imagine a 2D grid of boolean variables
  - The *i*th row is the tape at computation step *i*
  - *bij* is the value of the *j*th tape cell at step *i*

b00	b01	b02	b03	...
b10	b11	b12	b13	...
b20	b21	b22	b23	...
...	...	...	...	...

The Tape Head

- Represent the tape head's position at all points in time as another n x n array of boolean variables
  - The tape head can only be at one position on the tape at each point in time
  - So only one boolean in each row can be true
  - Add boolean formulas to enforce this for every cell, e.g.:
    - $h20 \rightarrow (\neg h21 \wedge \neg h22 \wedge \neg h23 \wedge ...)$
    - $h21 \rightarrow (\neg h20 \wedge \neg h22 \wedge \neg h23 \wedge ...)$
    - ...

h00	h01	h02	h03	...
h10	h11	h12	h13	...
h20	h21	h22	h23	...
...	...	...	...	...

The Program Statement

- At each step, the TM executes one program statement
- If there are k statements in the program, we can use a k x n array of boolean variables *sij* to represent the statement being executed at each step
  - Only one statement can be the focus at each point in time
  - The construction is similar to the tape head

The Initial State

- Write a formula that defines the starting state of the machine:
  - If the input is *i0 i1 i2 i3,* then
    - b00 ↔ i0 ∧ b01 ↔ i1 ∧ b02 ↔ i2 ∧ b03 ↔ i3
    - All other tape locations constrained to be 'false'
    - b04 ↔ false ∧ b05 ↔ false ∧ ...
  - Execution starts with the first instruction:
    - *s00* ↔ *true*
  - The tape head starts at the first input cell:
    - *h00* ↔ *true*

Executing a Statement

- Let the *w*th statement be `write true'
- Assume we are at time step *i*
- Write true at the current head position, copy tape contents from previous step at other positions:

  $(siw \wedge hi0) \rightarrow bi0$

  $(siw \wedge \neg hi0) \rightarrow (bi0 \leftrightarrow b(i-1)0)$

  $(siw \wedge hi1) \rightarrow bi1$

  $(siw \wedge \neg hi1) \rightarrow (bi1 \leftrightarrow b(i-1)1)$

  ...
- In the next step we execute the next program statement:

  $siw \rightarrow s(i+1)(w+1)$
- *The tape head does not move:*

  $(siw \wedge hi0) \rightarrow h(i+1)0$

  $(siw \wedge hi1) \rightarrow h(i+1)1$

  ...

Accepting and Rejecting

- One boolean variable a to representing accepting or rejecting

- If instruction w is accept, then add
  - $siw \rightarrow a$
- If instruction w is reject, then add
  - $siw \rightarrow \neg a$

Encoding P Using SAT

- We can construct a boolean formula (of polynomial size) that is satisfiable if and only if the polynomial time decider M accepts its input x.
- Conjunction of:
  - Formulas for the tape contents
  - Formulas for the head position
  - Formulas for program statements

Encoding NP Using SAT

- To encode polynomial time verifiers, we change only the input!
  - If the input is *i0 i1 i2 i3,* then
    - b00 ↔ i0 ∧ b01 ↔ i1 ∧ b02 ↔ i2 ∧ b03 ↔ i3
  - After the input, we leave a number of boolean variables equal to the size of the certificate unconstrained in step 0
    - Since these variables are unconstrained, the SAT algorithm can fill them in to make the formula satisfiable if possible – i.e., SAT can guess the certificate!
  - All remaining tape locations are constrained to be 'false' in step 0

```
bool PtimeVerifier(Machine M, Input x) {
  return isSatisfiable(ToFormula(M,x));
}
```

Intuition:

Executing a plynomial time verifier can’t be “harder” than solving SAT because if we can solve SAT efficiently, we can solve polynomial time verification efficiently

We've seen that *every* NP problem is reducible to SAT.

So SAT is a least as hard as every problem in NP.

SAT is *NP-hard*

SAT is also in NP

Idea: The certificate for a SAT problem is a satisfying assignment, which can be checked in linear time.

Since SAT is in NP and also at least as hard as every other problem in NP, we say SAT is *NP-complete*

Time-Out for Announcements!

Please evaluate this course on Axess.

Your feedback makes a difference.

Final Exam Logistics

- Our final exam is on ***Saturday, Jun 7th*** from ***8:30AM – 11:30 AM***.
  - Seating assignments will be online soon; we'll make an announcement when they're ready.
- The final exam is cumulative
- 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.

Review Sessiom

- ***Thursday, 2-3 PM*** in STLC115

Back to CS103!

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 **P** ≟ **NP** question.

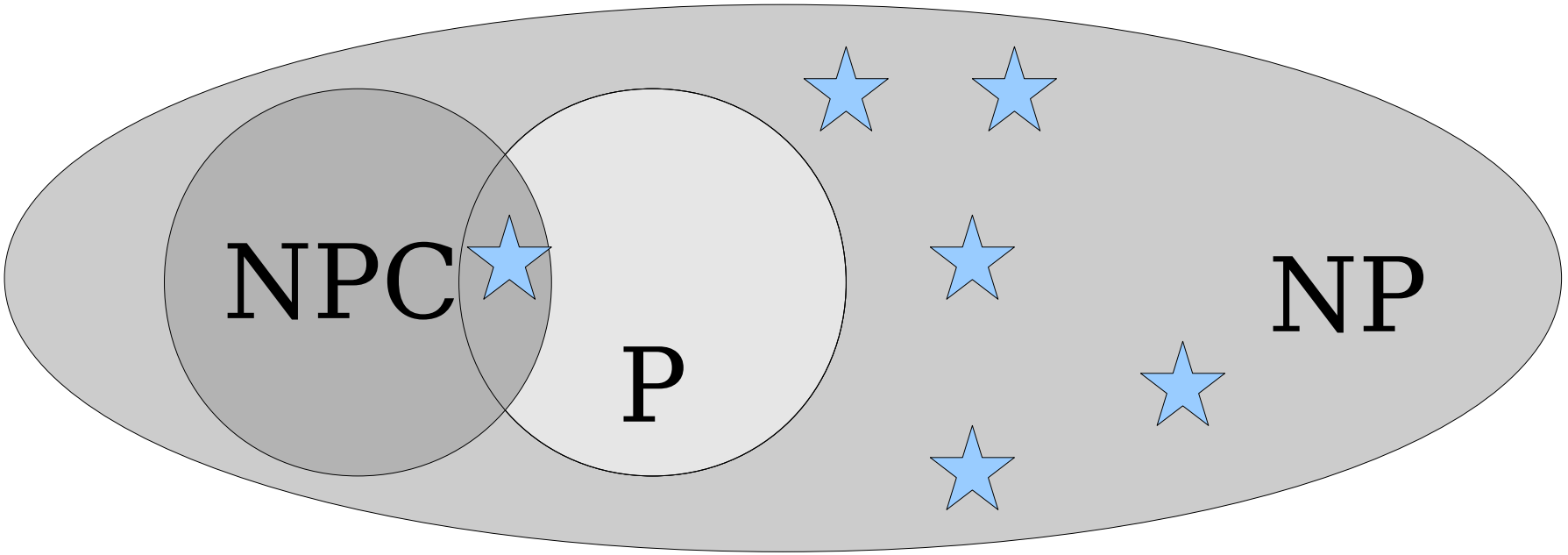
The Tantalizing Truth

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.

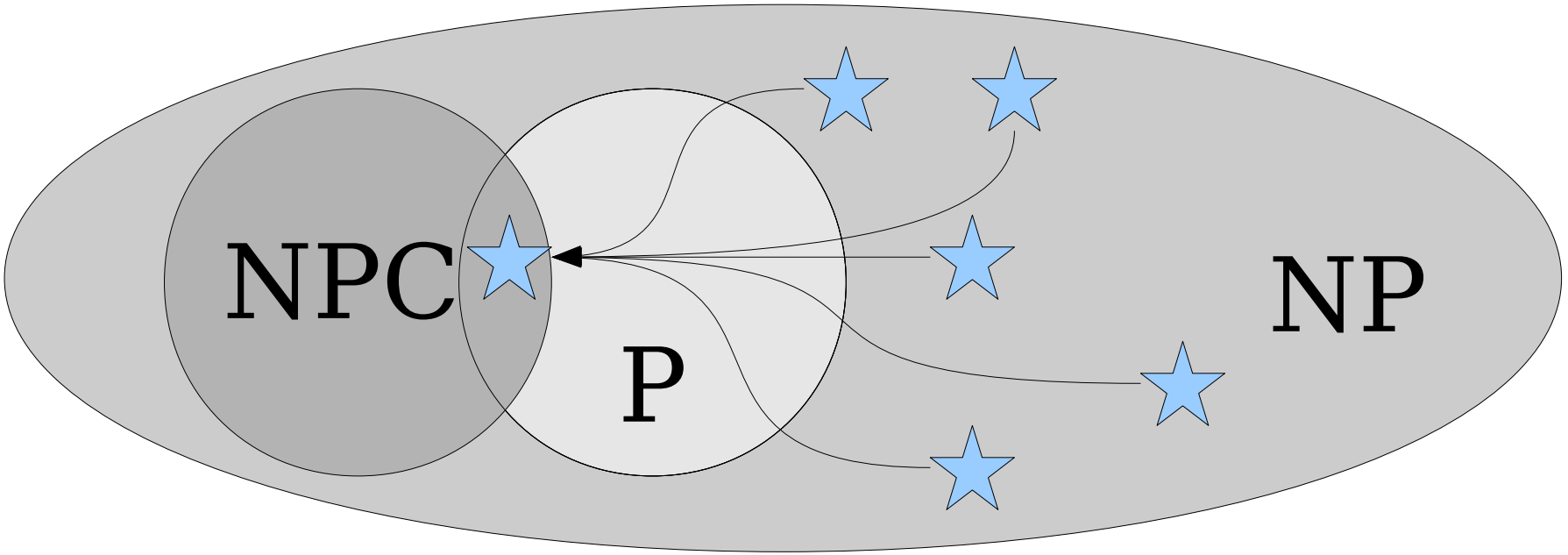
The Tantalizing Truth

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



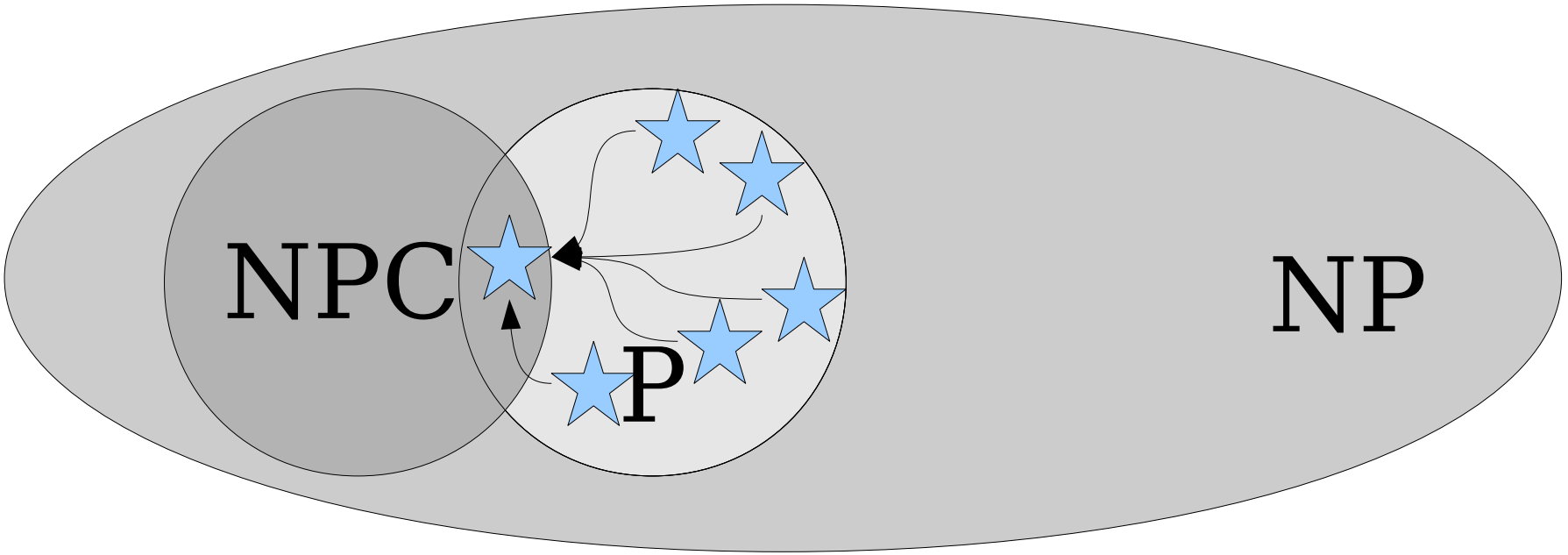
The Tantalizing Truth

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



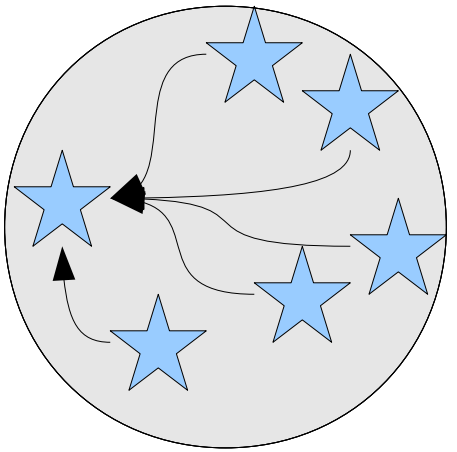
The Tantalizing Truth

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**.

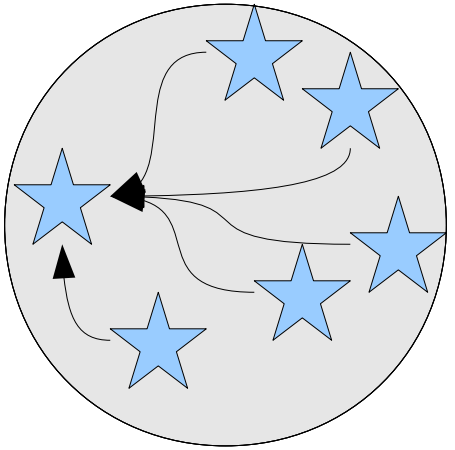


P = NP

The Tantalizing Truth

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

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

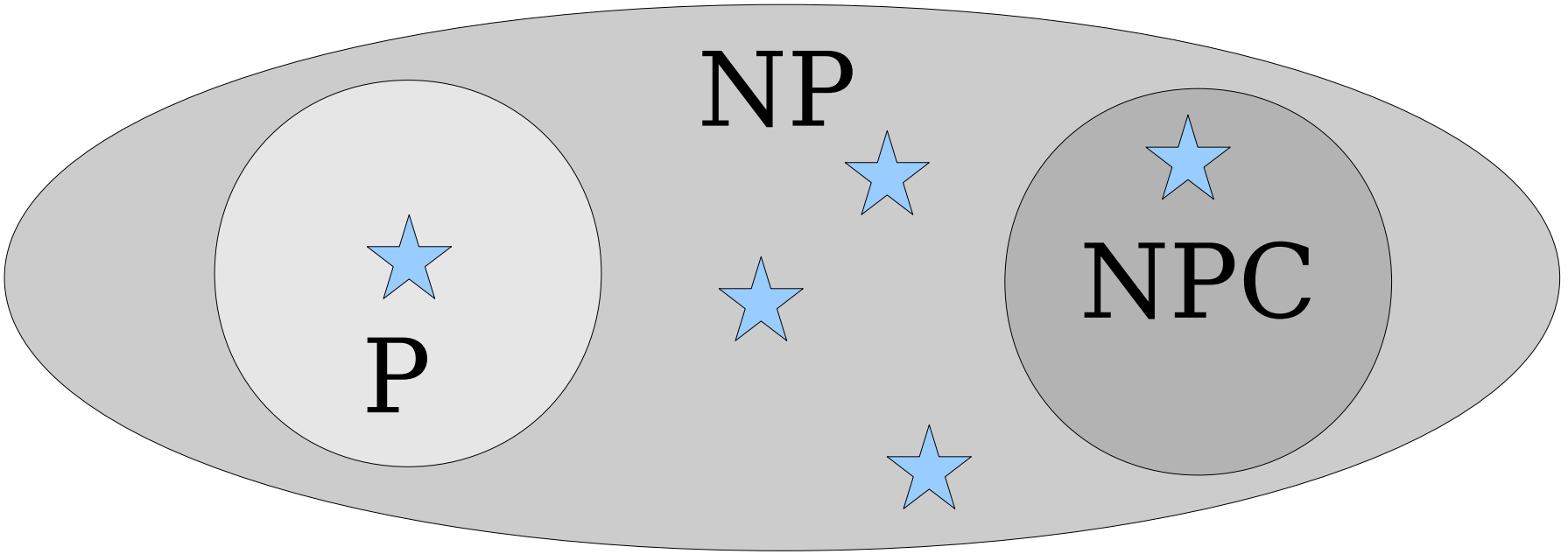
Theorem: If *any* **NP**-complete language is not in **P**, then **P ≠ NP**.

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**. ■



Why All This Matters

- Resolving **P** ≟ **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 an 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.

The Big Picture

Take a minute to reflect on your journey.

Set Theory

Power Sets

Cantor's Theorem

Direct Proofs

Parity

Proof by Contrapositive

Proof by Contradiction

Modular Congruence

Propositional Logic

First-Order Logic

Logic Translations

Logical Negations

Propositional Completeness

Vacuous Truths

Perfect Squares

Triangular Numbers

Tournaments

Functions

Injections

Surjections

Involutions

Monotone Functions

Minkowski Sums

Bijections

Graphs

Connectivity

Independent Sets

Vertex Covers

Trees

Bipartite Graphs

The Pigeonhole Principle

Ramsey Theory

Mathematical Induction

Complete Induction

The Spanning Tree Protocol

Formal Languages

DFAs

Regular Languages

Closure Properties

NFAs

Subset Construction

Kleene Closures

Error-Correcting Codes

Regular Expressions

State Elimination

Monoids

Distinguishability

Myhill-Nerode Theorem

Nonregular Languages

Context-Free Grammars

Merkle-Damgård Construction

Fixed Point Theorems

Turing Machines

Church-Turing Thesis

TM Encodings

Universal Turing Machines

Self-Reference

Decidability

Recognizability

Self-Defeating Objects

Undecidable Problems

The Halting Problem

Verifiers

Diagonalization Language

R and **RE**

co-**RE**

Complexity Class **P**

Complexity Class **NP**

P ≟ **NP** Problem

Polynomial-Time Reducibility

NP-Completeness

You’ve given yourself the foundation to tackle problems from all over computer science.

PRPs and PRFs

From CS255
Intro to Cryptography

- Pseudo Random Function (**PRF**) defined over (K,X,Y):

$$F: K \times X \rightarrow Y$$

such that exists "efficient" algorithm to evaluate F(k,x)

- Pseudo Random Permutation (**PRP**)

$$E: K \times X \rightarrow X$$

such that:

1. Exists "efficient" algorithm to evaluate E(k,x)
2. The function E(k, ·) is one-to-one
3. "efficient" inversion algorithm D(k,x)

Functions between sets! K × X is the set of all pairs made from K and X.

Definitions in terms of efficiency!

Injectivity!

From CS124
From Languages to Information

Tokenization in NLTK

Bird, Loper and Klein (2009), *Natural Language Processing with Python*. O'Reilly

```
>>> text = 'That U.S.A. poster-print costs $12.40...'
>>> pattern = r'''(?x)    # set flag to allow verbose regexps
...     ([A-Z]\.)+        # abbreviations, e.g. U.S.A.
...   | \w+(-\w+)*        # words with optional internal hyphens
...   | \$?\d+(\.\d+)?%?  # currency and percentages, e.g. $12.40, 82%
...   | \.\.\.            # ellipsis
...   | [][.,;"'?():-_`]  # these are separate tokens; includes ], [
... '''
>>> nltk.regexp_tokenize(text, pattern)
['That', 'U.S.A.', 'poster-print', 'costs', '$12.40', '...']
```

It's a big regex!

Course Recommendations

- ***CS154:*** Introduction to the Theory of Computation
  - The “spiritual sequel” to CS103; does a deep dive into automata, TMs, and computability/complexity theory.
  - If you enjoyed the tail end of this course, highly recommended as a next step.
- ***CS161:*** Design and Analysis of Algorithms
  - A natural next course in CS theory, focusing on the design of efficient algorithms.
  - (Super helpful for job interviews!)
- ***CS143:*** Compilers
  - Use your automata and CFG prowess to translate source code into machine code. Extremely rewarding!
- ***CS257:*** Introduction to Automated Reasoning
  - See how to automate formal proofs, play around with SAT and propositional logic, etc.

Course Recommendations

Theoryland

- CS154 — *Complexity*
- Phil 151 — *Computability*
- Phil 152 — *Computability*
- Math 107 — *Graphs*
- Math 108 — *Graphs*
- Math 113 — *Functions*
- Math 120 — *Functions*
- Math 161 — *Set Theory*
- Math 152 — *Number Theory*

Applications

- CS124 — *Languages / Automata*, *Graphs*, *Functions*
- CS143 — *Languages / Automata*
- CS161 — *Graphs*, *Functions*
- CS224W — *Graphs*, *Functions*
- CS242 — *Graphs*, *Functions*
- CS243 — *Graphs*, *Functions*
- CS246 — *Graphs*, *Functions*
- CS250 — *Graphs*, *Functions*
- CS251 — *Graphs*, *Functions*
- CS255 — *Graphs*, *Functions*

Your Questions

Final Thoughts

Thanks!