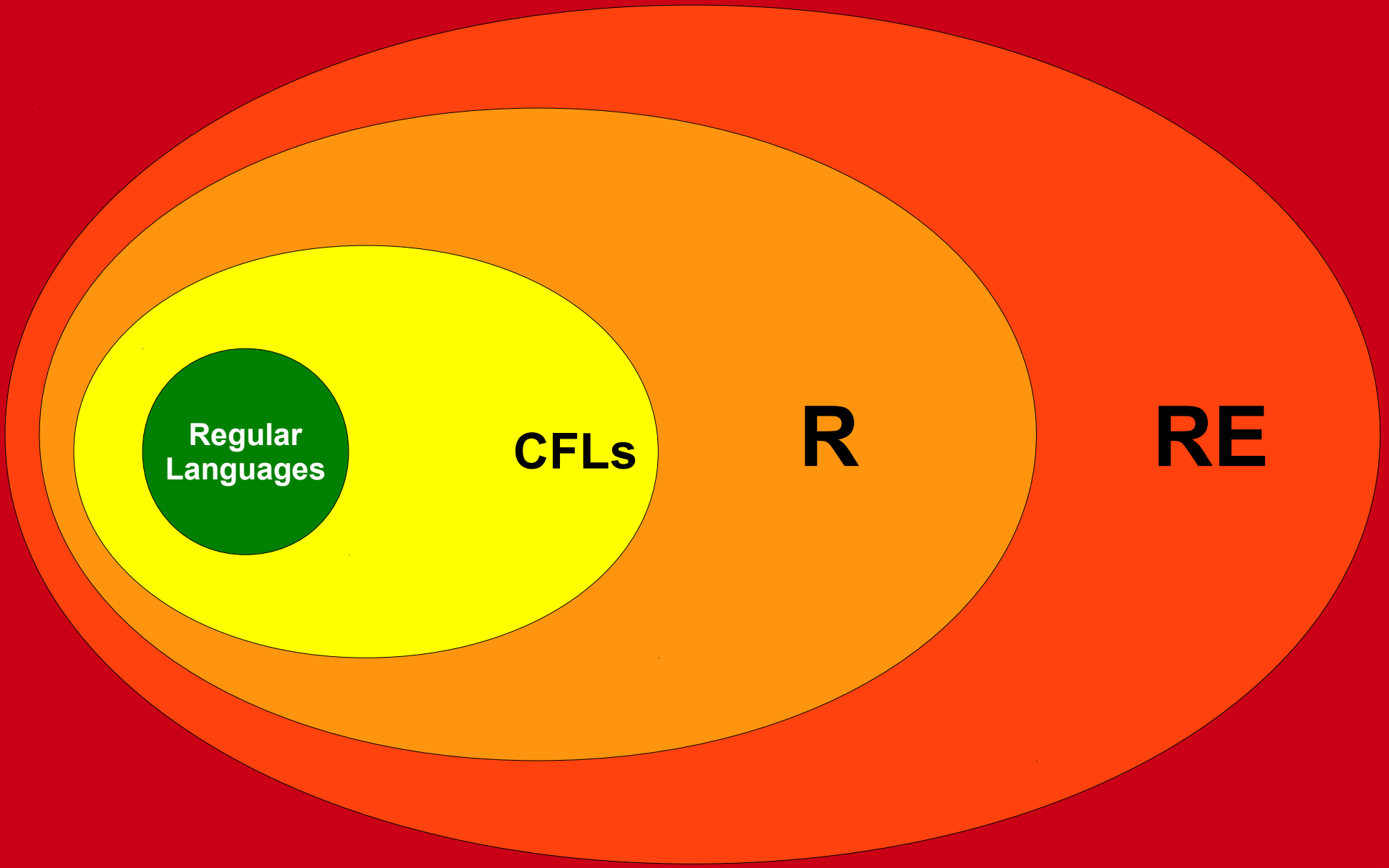
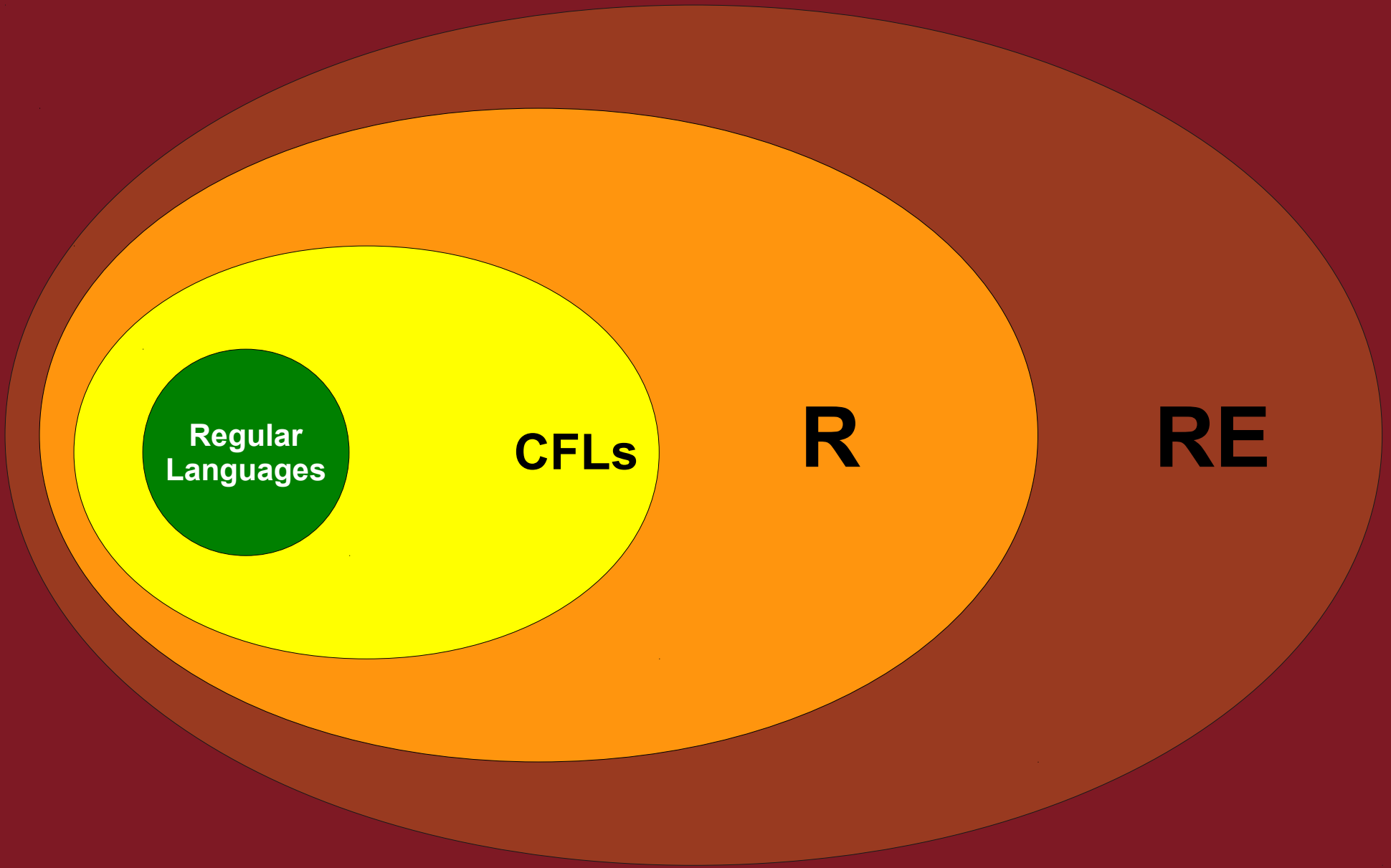
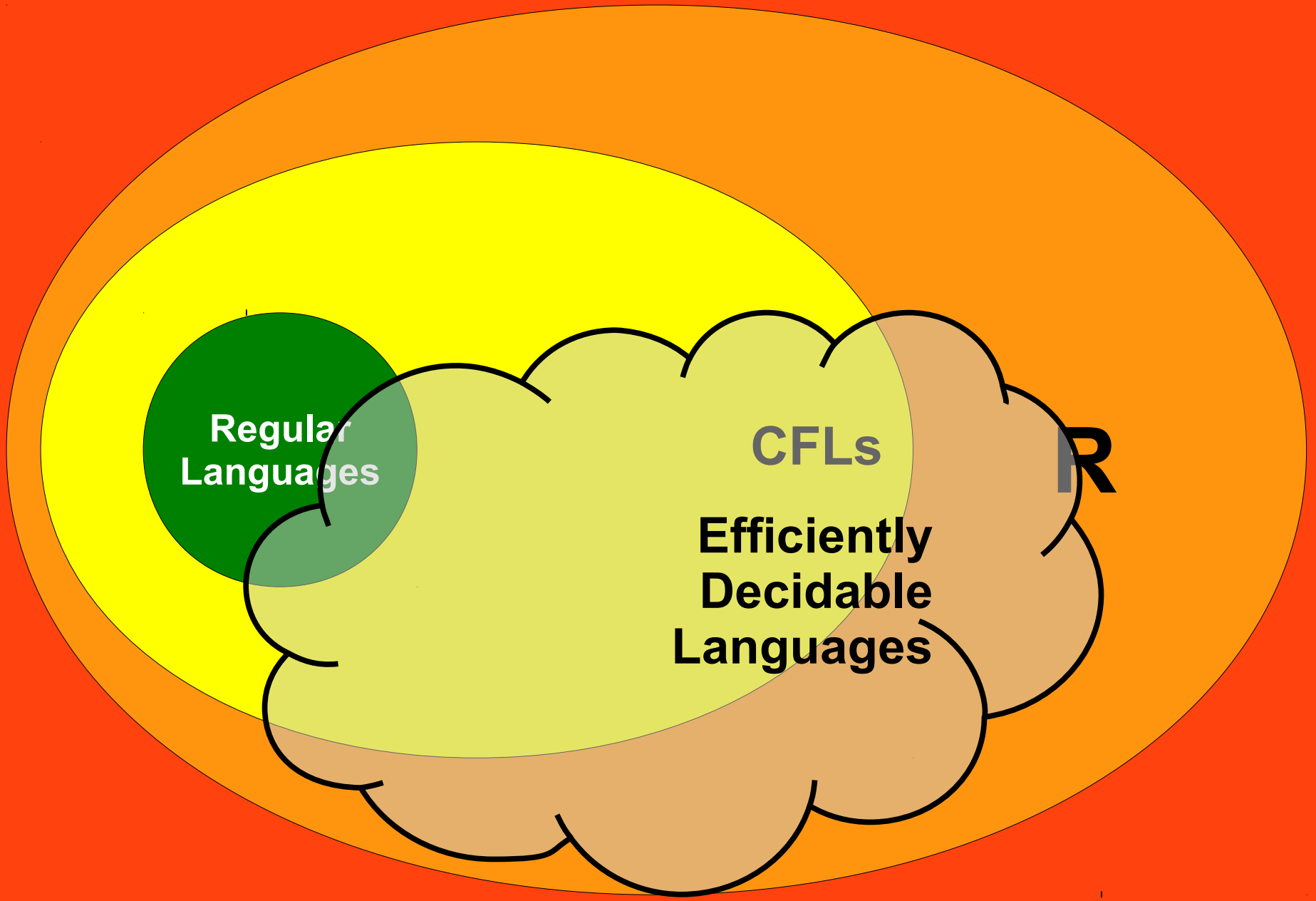
NP-Completeness

Part One

Recap from Last Time


Regular Languages
CFLs
R
RE
All Languages


Regular Languages
CFLs
R
RE
All Languages


Regular Languages
CFLs
R
Efficiently Decidable Languages
Undecidable Languages

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:

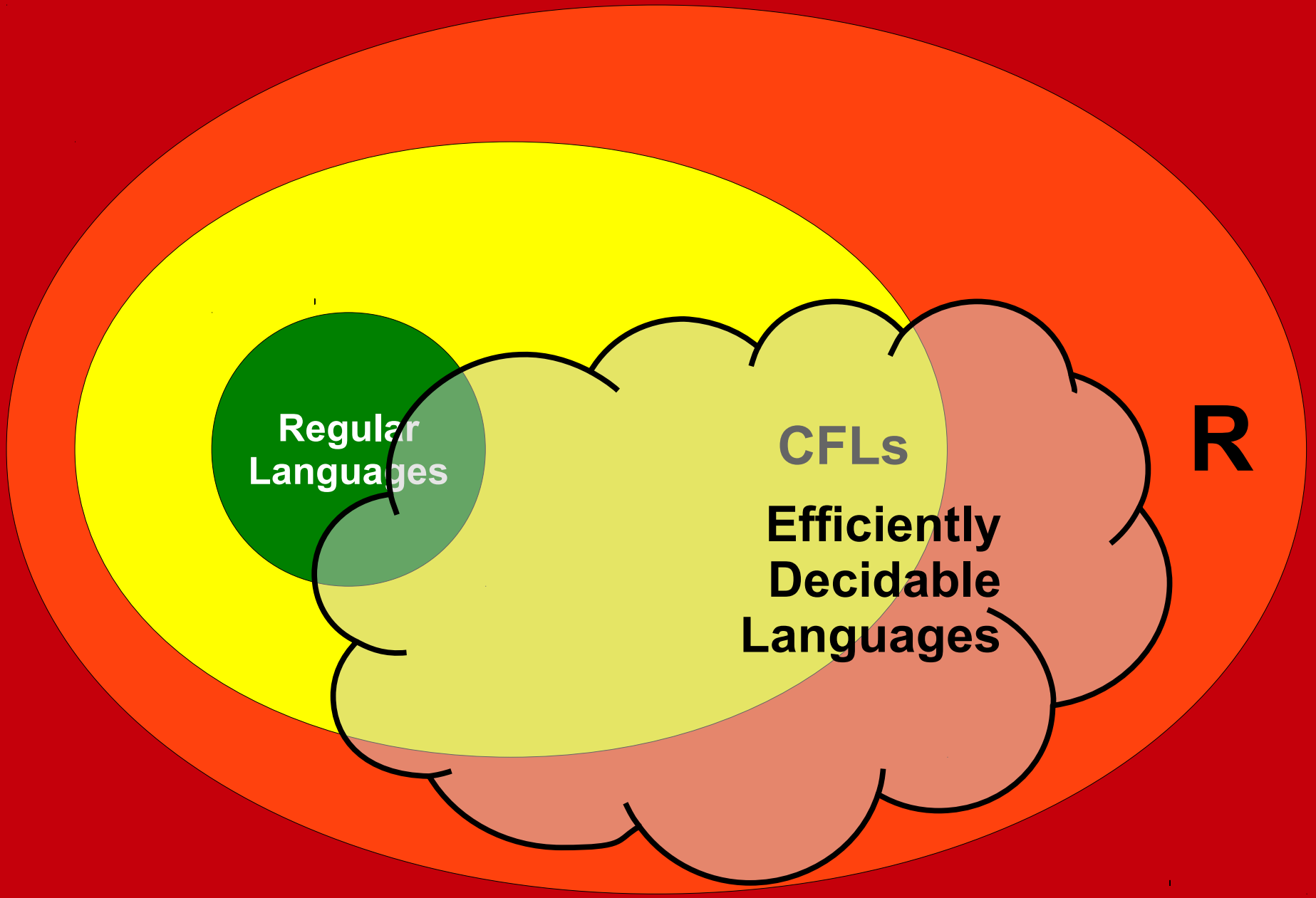
$$\mathbf{P} = \{ L \mid \text{There is a polynomial-time decider for } L \}$$

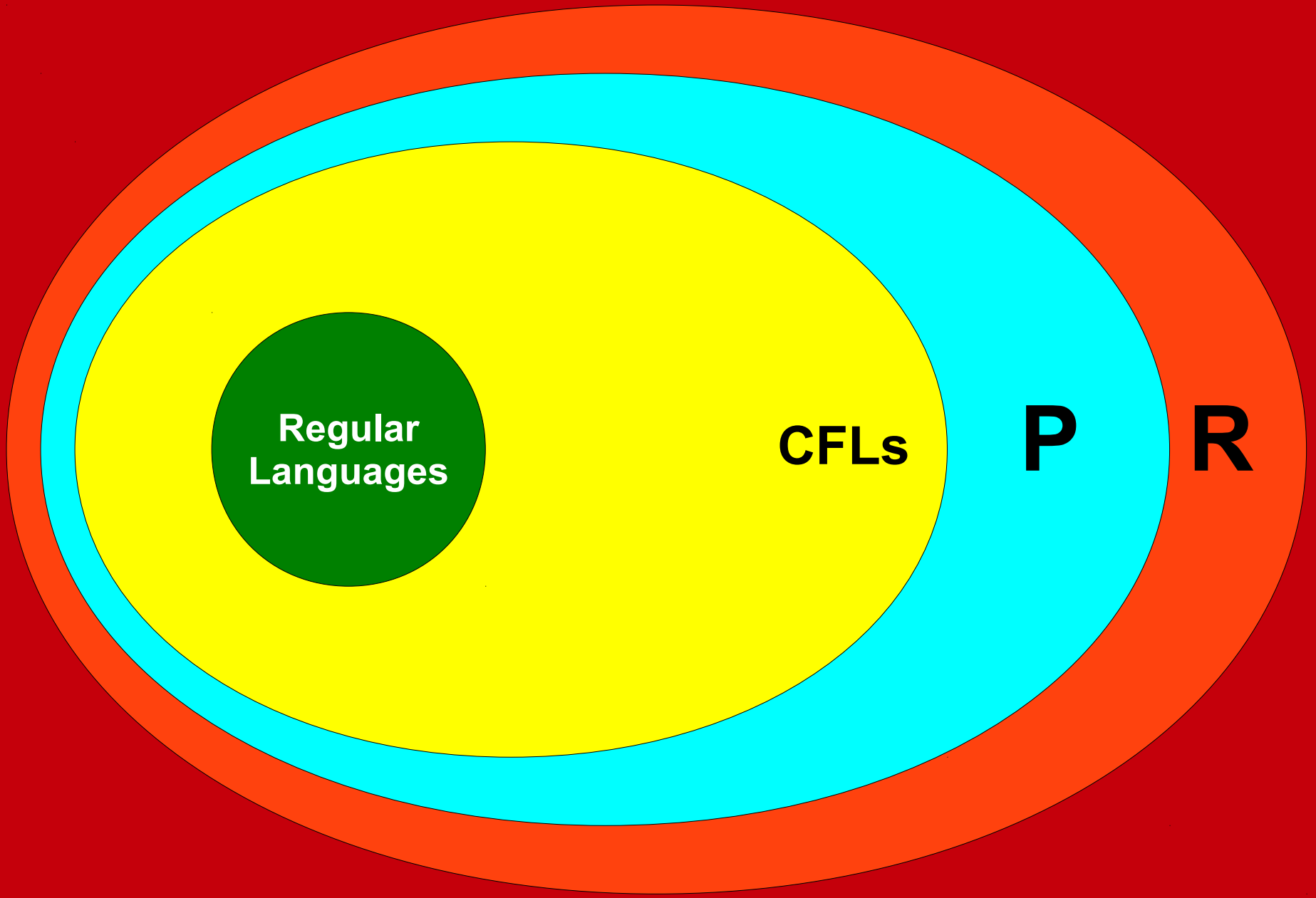
- Assuming the Cobham-Edmonds thesis, a language is in **P** if it can be decided efficiently.

New Stuff!

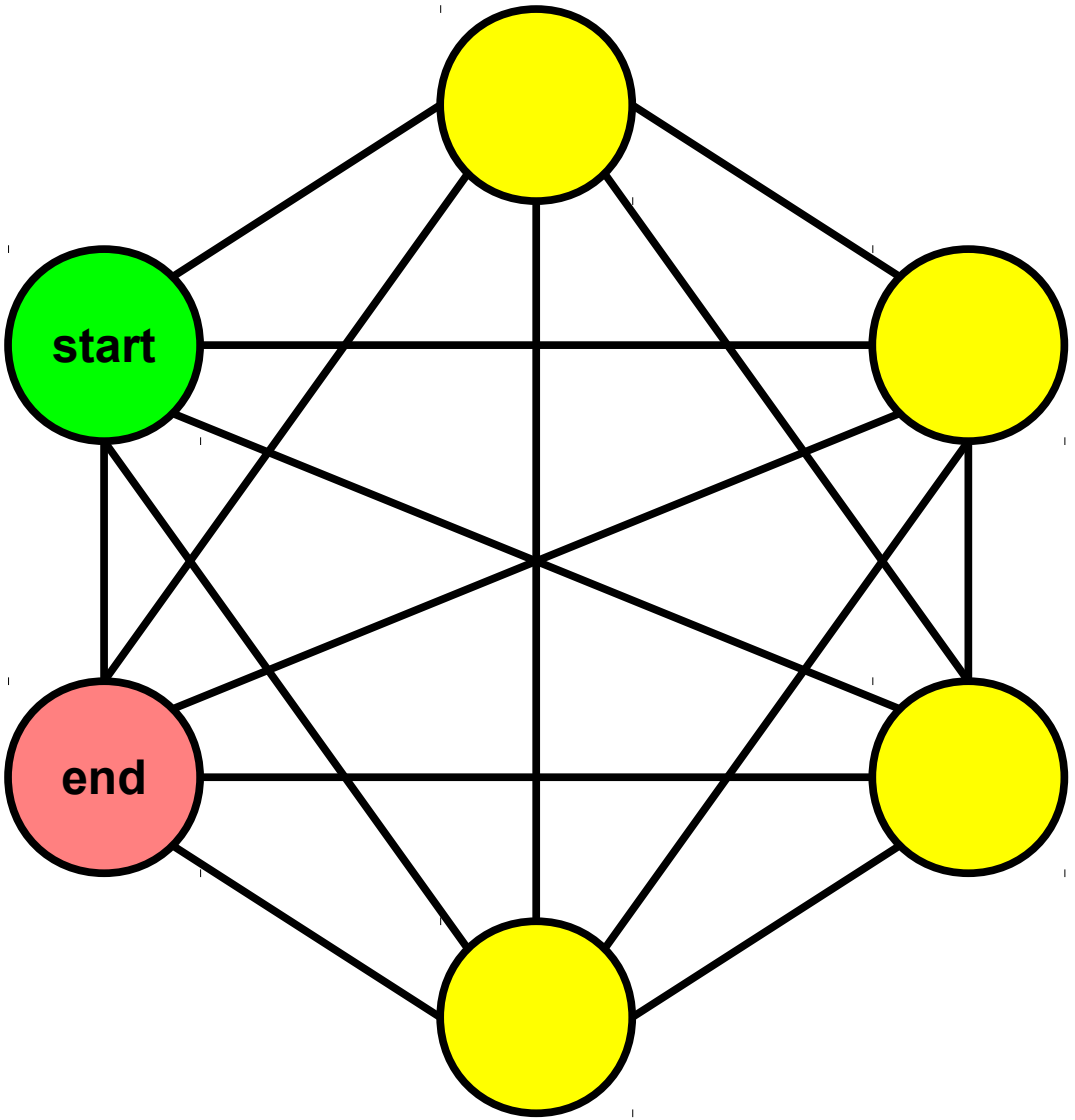
Examples of Problems in **P**

- All regular languages are in **P**.
  - All have linear-time TMs.
- All CFLs are in **P**.
  - Requires a more nuanced argument (the *CYK algorithm* or *Earley's algorithm.*)
- And a *ton* of other problems are in **P** as well.
  - Curious? Take CS161!


Regular Languages
CFLs
Efficiently Decidable Languages
R
Undecidable Languages


Regular Languages
CFLs
P
R
Undecidable Languages

What *can't* you do in polynomial time?



How many simple paths are there from the start node to the end node?

{ 

, 

, 

, 

}

How many subsets of this set are there?

An Interesting Observation

- There are (at least) exponentially many objects of each of the preceding types.
- However, each of those objects is not very large.
  - Each simple path has length no longer than the number of nodes in the graph.
  - Each subset of a set has no more elements than the original set.
- This brings us to our next topic...

What if you need to search a large space for a single object?

Verifiers – Again

		7		6		1		
					3		5	2
3			1		5	9		7
6		5		3		8		9
	1						2	
8		2		1		5		4
1		3	2		7			8
5	7		4					
		4		8		7		

Does this Sudoku problem have a solution?

Verifiers – Again

2	5	7	9	6	4	1	8	3
4	9	1	8	7	3	6	5	2
3	8	6	1	2	5	9	4	7
6	4	5	7	3	2	8	1	9
7	1	9	5	4	8	3	2	6
8	3	2	6	1	9	5	7	4
1	6	3	2	5	7	4	9	8
5	7	8	4	9	6	2	3	1
9	2	4	3	8	1	7	6	5

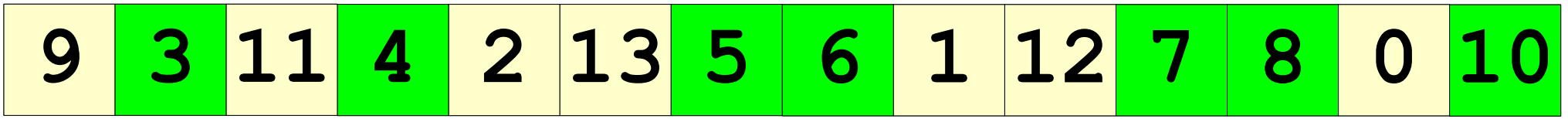
Does this Sudoku problem have a solution?

Verifiers – Again

9	3	11	4	2	13	5	6	1	12	7	8	0	10

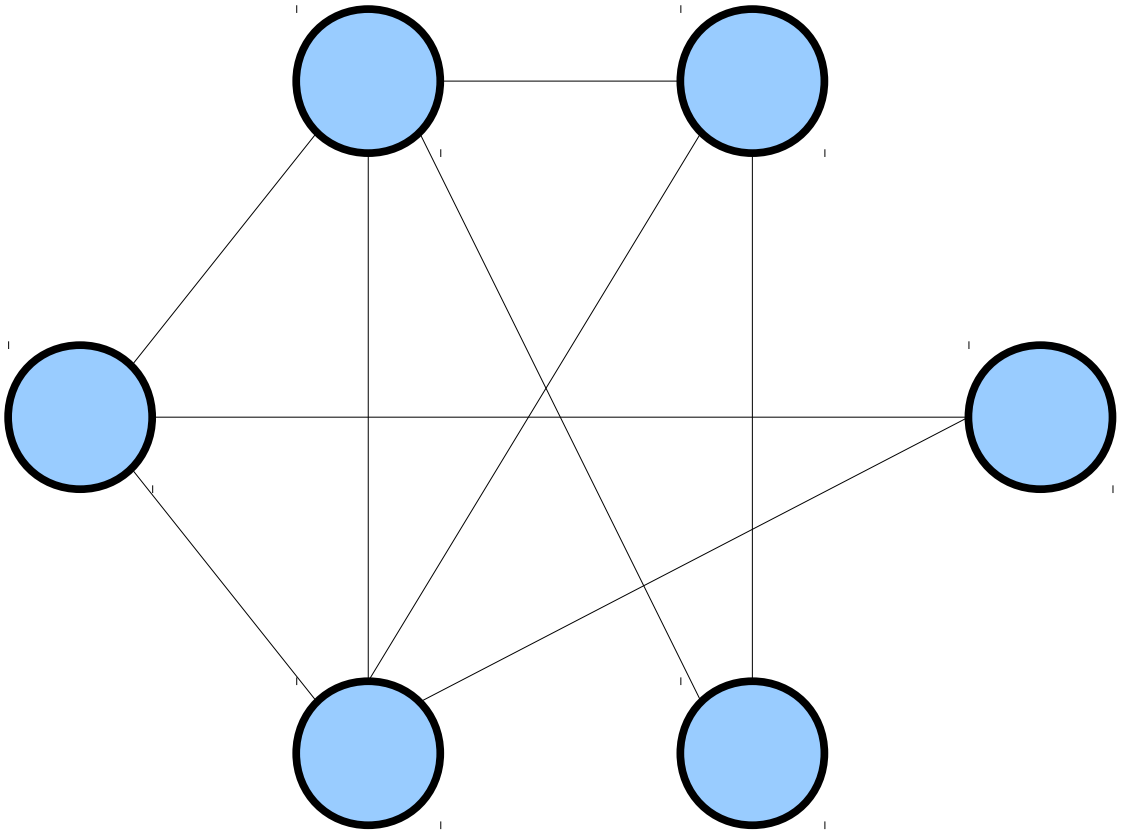
Is there an ascending subsequence of length at least 7?

Verifiers – Again



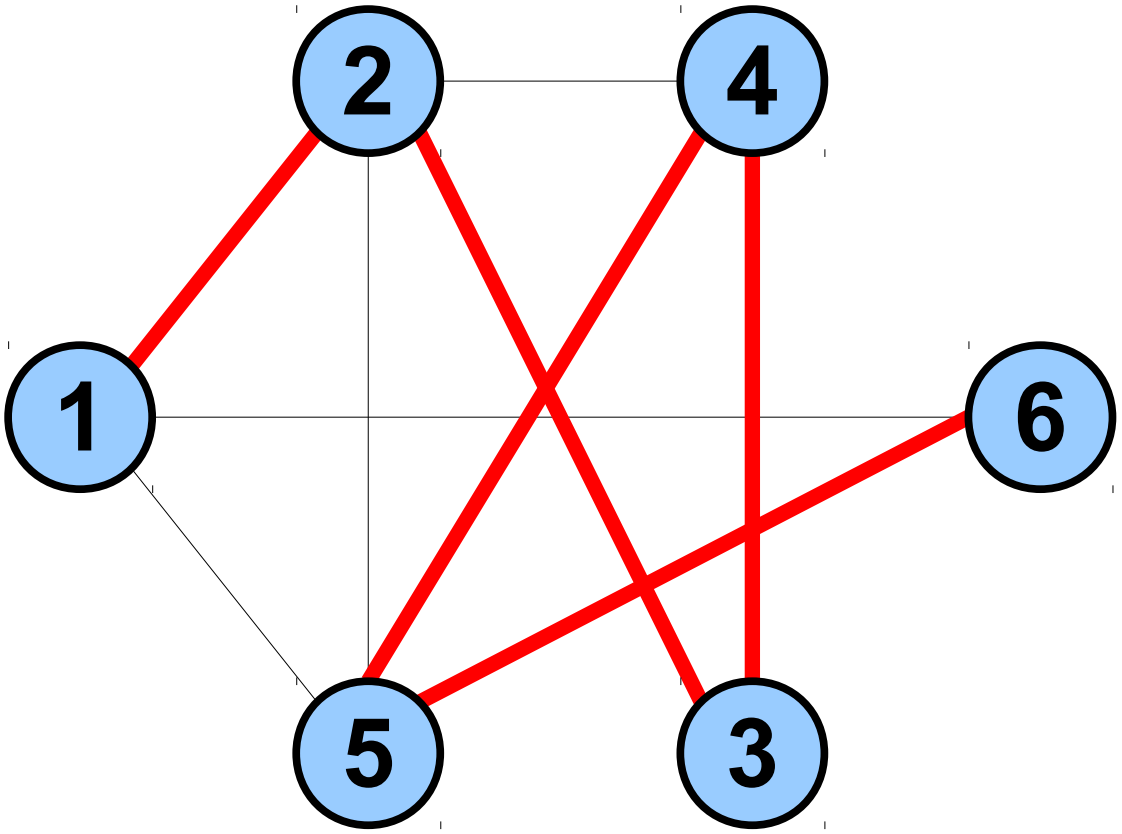
Is there an ascending subsequence of length at least 7?

Verifiers – Again



Is there a simple path that goes through every node exactly once?

Verifiers – Again



Is there a simple path that goes through every node exactly once?

Verifiers

- Recall that a ***verifier*** for L is a TM V such that
  - V halts on all inputs.
  - $w \in L$ iff $\exists c \in \Sigma^*$. V accepts $\langle w, c \rangle$.

Polynomial-Time Verifiers

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

The Complexity Class **NP**

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

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

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

And now…

The

Most Important Question

in

Theoretical Computer Science

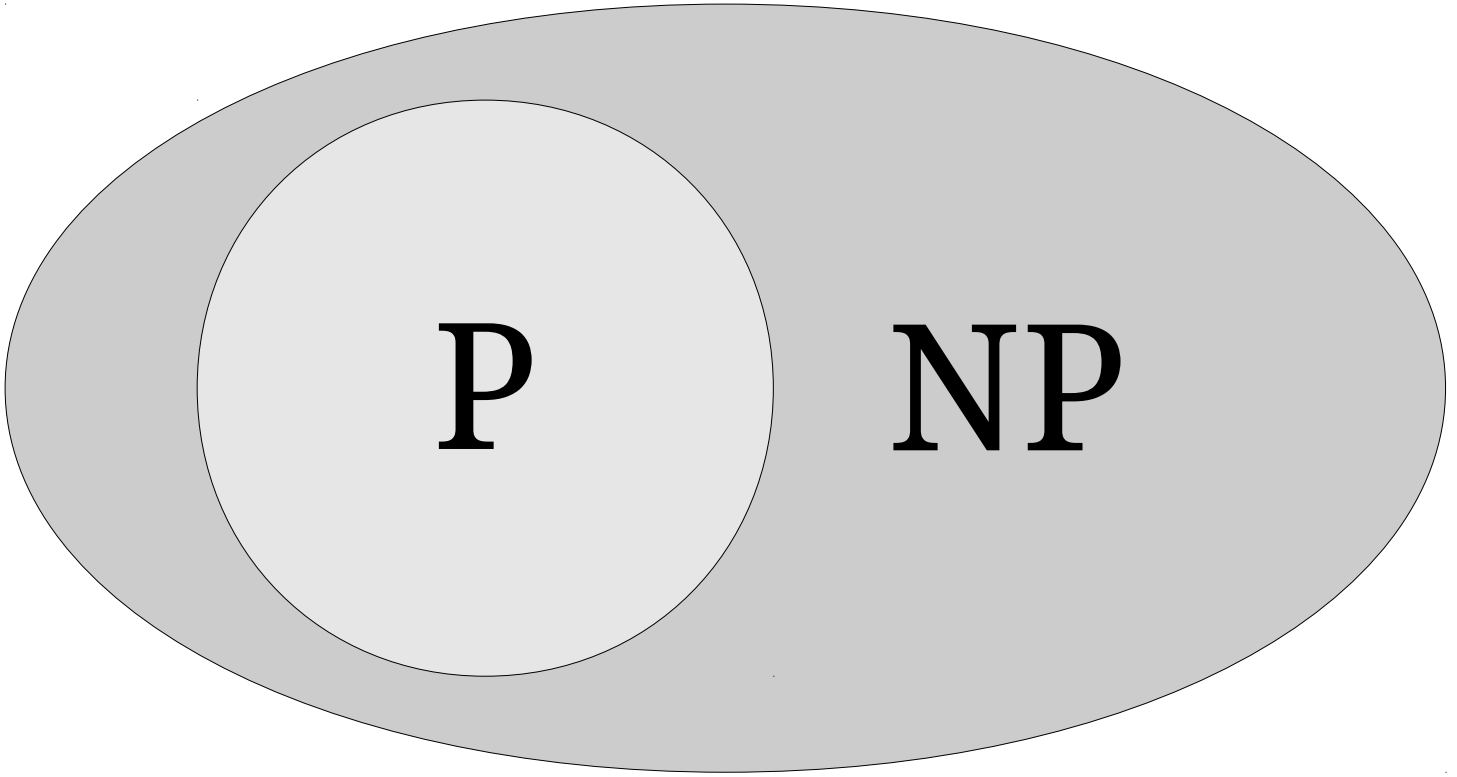
What is the connection between **P** and **NP**?

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

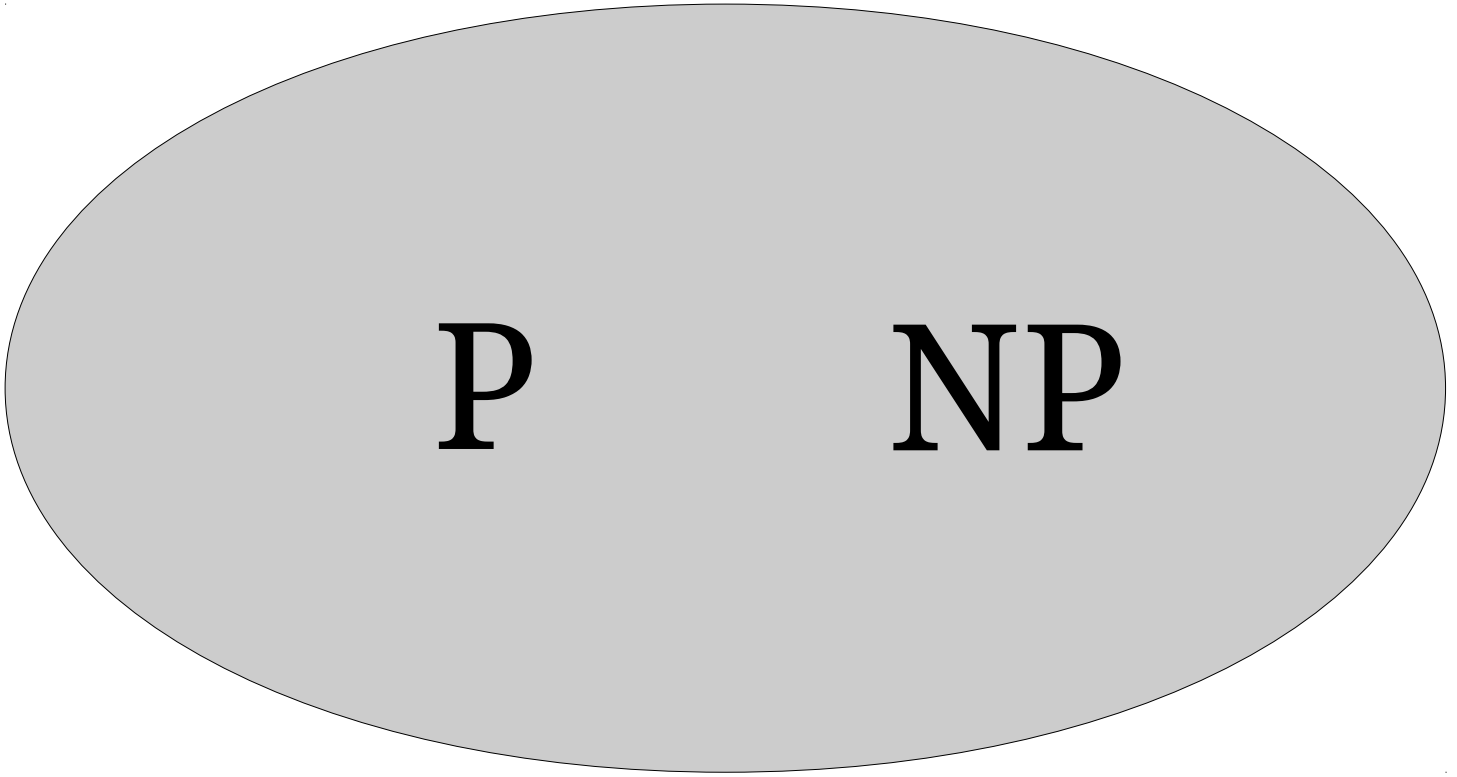
$$\mathbf{NP} = \{\ L \mid \text{There is a polynomial-time verifier for } L\ \}$$

$$\mathbf{P} \subseteq \mathbf{NP}$$

Which Picture is Correct?



Which Picture is Correct?



Does **P** = **NP**?

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

- The $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ question is the most important question in theoretical computer science.
- With the verifier definition of **NP**, one way of phrasing this question is

  *If a solution to a problem can be **checked** efficiently, can that problem be **solved** efficiently?*

- An answer either way will give fundamental insights into the nature of computation.

Why This Matters

- The following problems are known to be efficiently verifiable, but have no known efficient solutions:
  - Determining whether an electrical grid can be built to link up some number of houses for some price (Steiner tree problem).
  - Determining whether a simple DNA strand exists that multiple gene sequences could be a part of (shortest common supersequence).
  - Determining the best way to assign hardware resources in a compiler (optimal register allocation).
  - Determining the best way to distribute tasks to multiple workers to minimize completion time (job scheduling).
  - ***And many more***.
- If **P = NP**, ***all*** of these problems have efficient solutions.
- If **P ≠ NP**, ***none*** of these problems have efficient solutions.

Why This Matters

- If **P = NP**:
  - A huge number of seemingly difficult problems could be solved efficiently.
  - Our capacity to solve many problems will scale well with the size of the problems we want to solve.
- If **P ≠ NP**:
  - Enormous computational power would be required to solve many seemingly easy tasks.
  - Our capacity to solve problems will fail to keep up with our curiosity.

What We Know

- Resolving **P** $\stackrel{?}{=}$ **NP** has proven ***extremely difficult***.
- In the past 44 years:
  - Not a single correct proof either way has been found.
  - Many types of proofs have been shown to be insufficiently powerful to determine whether **P** $\stackrel{?}{=}$ **NP**.
  - A majority of computer scientists believe **P** ≠ **NP**, but this isn't a large majority.
- Interesting read: Interviews with leading thinkers about **P** $\stackrel{?}{=}$ **NP**:
  - http://web.ing.puc.cl/~jabaier/iic2212/poll-1.pdf

The Million-Dollar Question

The Clay Mathematics Institute has offered a **$1,000,000 prize** to anyone who proves or disproves **P = NP**.

The Million-Dollar Question

The Clay Mathematics Institute has offered a **$1,000,000 prize** to anyone who proves or disproves **P = NP**.

Time-Out for Announcements!

Please evaluate this course in Axess.

Your feedback does make a difference.

Problem Set Nine

- Problem Set Nine is due this Friday at 3:00PM.
  - No late days may be used – this is university policy. Sorry about that!
  - We hope that the later questions help give you some review for the final exam.
- Have questions? Stop by office hours or ask on Piazza!

Extra Practice Problems

- We released three sets of cumulative review problems on Friday.
- Solutions will be available in the Gates building starting this afternoon – feel free to stop by to pick them up at your convenience!
  - SCPD students – you should get them over email later today.
- We've released two practice final exams to the course website. We *strongly* recommend blocking out three hours to work through them rather than just skimming over them. It's the best possible way to practice for the exam.

Regrades Processed

- We've finished all the regrade requests for the first midterm exam.
- Regraded midterms will be available outside of class today. Feel free to pick them up from Gates if you aren't able to grab your exam today.
- SCPD students – if you emailed in a regrade request, we should get back to you by later this evening with feedback.

Your Questions!

“You said on Wednesday that you think there is nothing a human brain can do that a Turing Machine can't do. However, it seems like humans are able to confirm whether a program is a secure voting machine (among other things) by looking at the source code?”

Remember that the question is whether it's possible to confirm that *any arbitrary program* that purports to be a secure voting machine is actually secure…

```
int main() {
    string input = getInput();
    int n = countRs(input);

    while (n > 1) {
        if (n % 2 == 0) n = n / 2;
        else n = 3 * n + 1;
    }

    if (countRs(input) > countDs(input)) {
        accept();
    } else {
        reject();
    }
}
```

This is a secure voting machine if and only if the Collatz Conjecture is true.

So no one knows whether this is a secure voting machine!

“Are you religious/do you think it's possible to balance discrete math and spirituality? Realized after we concluded there are some problems mathematics can't solve and I started thinking about the supernatural.”

Mathematical truth is not the same as religious truth, which is not the same as scientific truth, which is not the same as legal truth, etc. Different systems judge truth in different ways and each gives a different insight into the world.

“What myth/false belief do you think is most widespread among Stanford students? Care to dispel it?”

Please, please, please ask this question again on Wednesday. We're short on time today and I don't think I can answer this now, but I'd love to. So please ask this again on Wednesday!

Back to CS103!

What do we know about $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$?

Adapting our Techniques

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 run other TMs as subroutines.
  - ***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**?

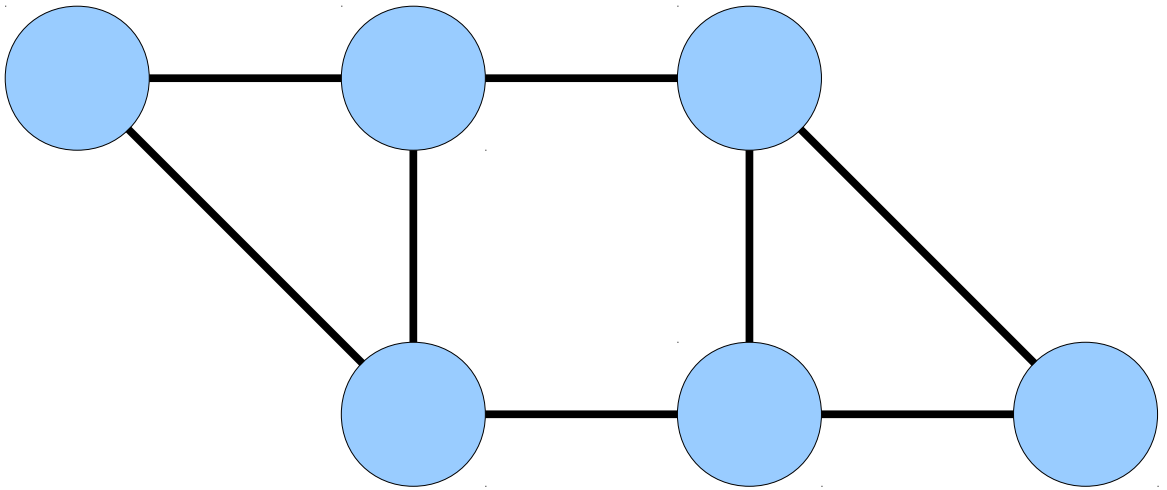
Reducibility

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.

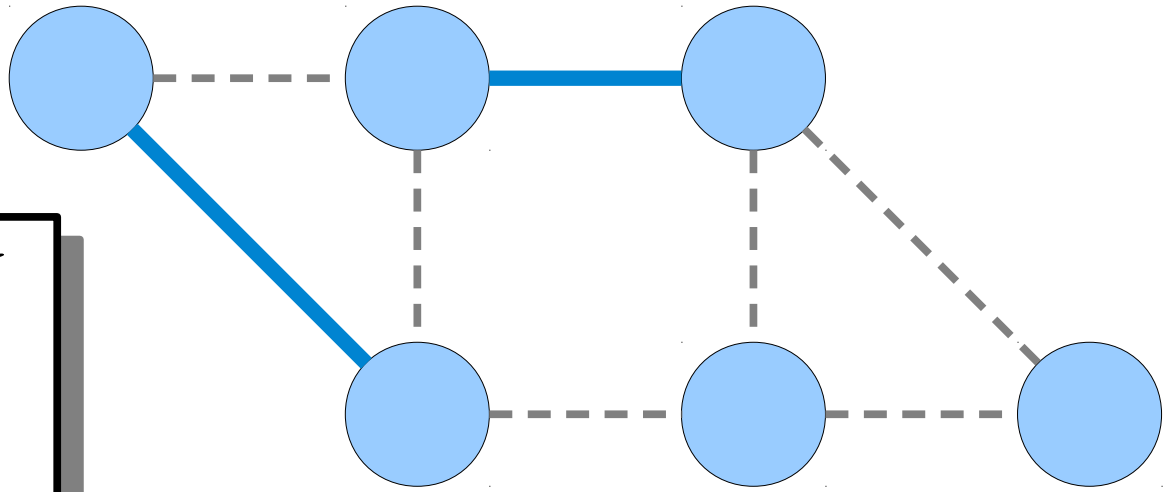
Maximum Matching

- 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

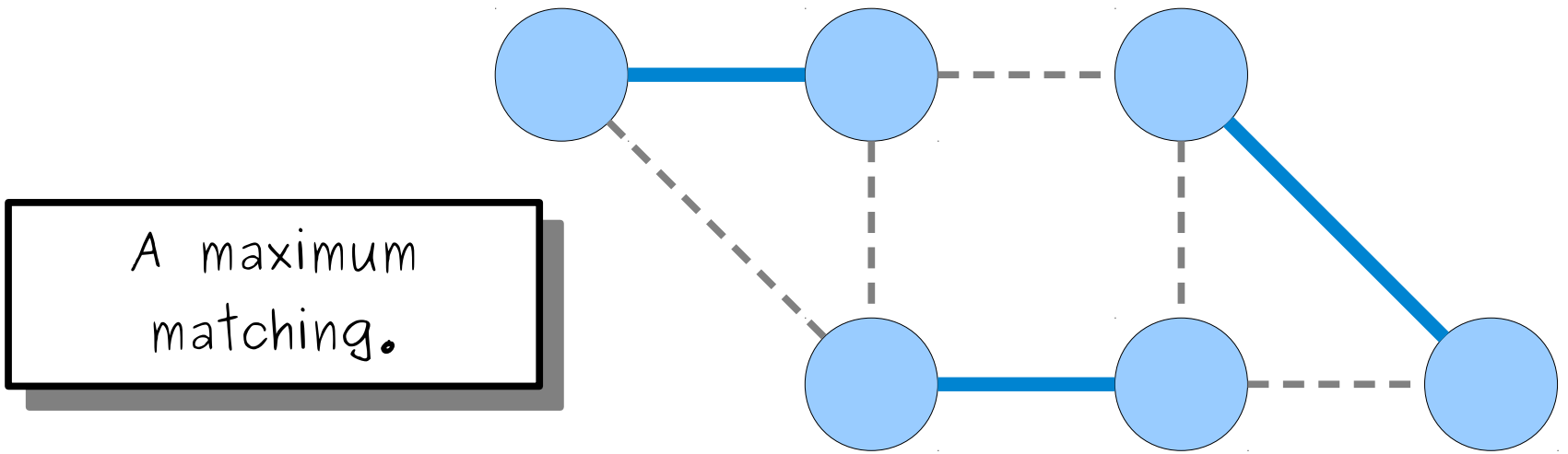
- Given an undirected graph G, a ***matching*** in G is a set of edges such that no two edges share an endpoint.
- A ***maximum matching*** is a matching with the largest number of edges.



A matching, but not a maximum matching.

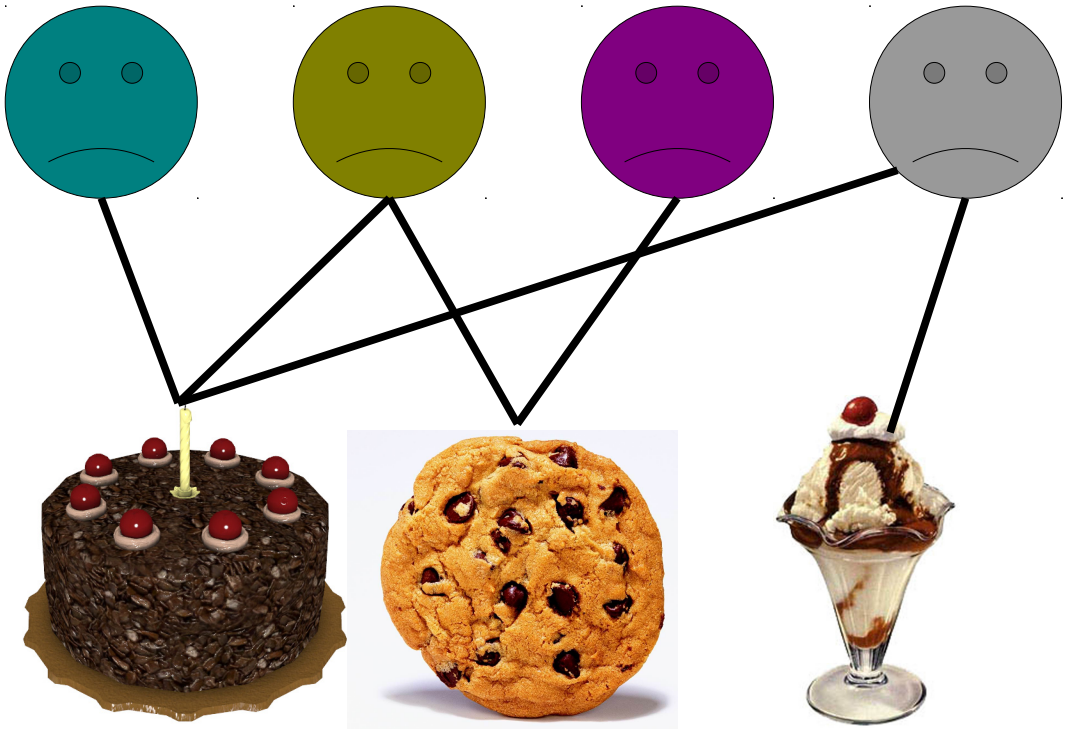
Maximum Matching

- 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.



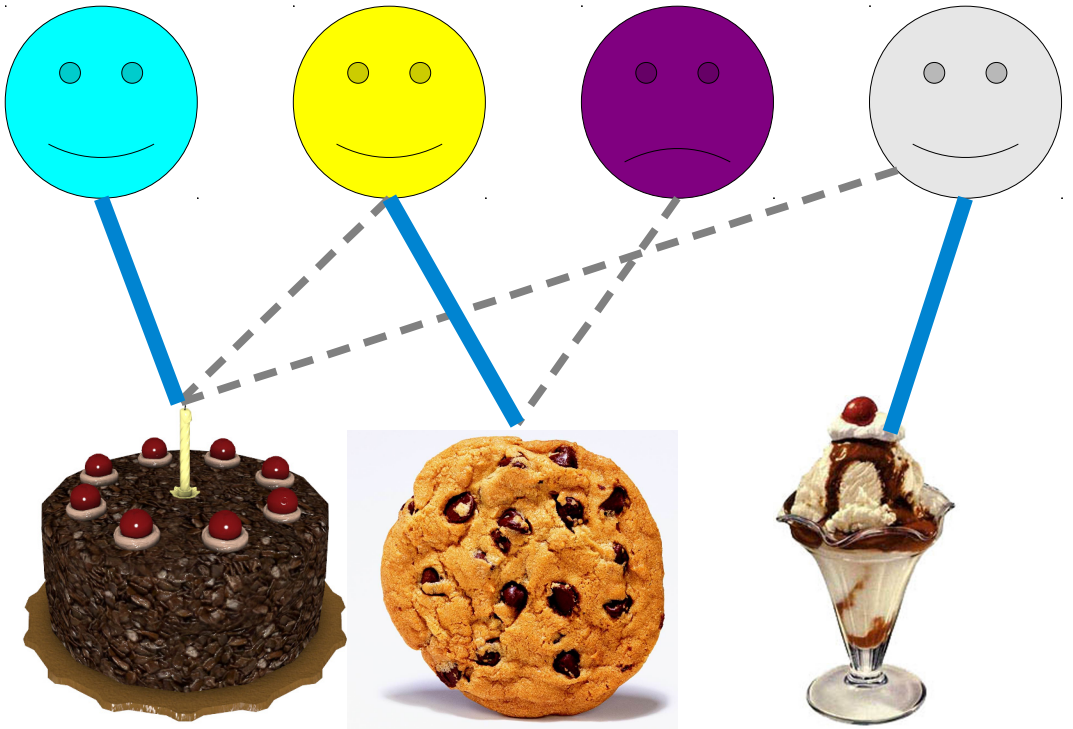
Maximum Matching

- 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

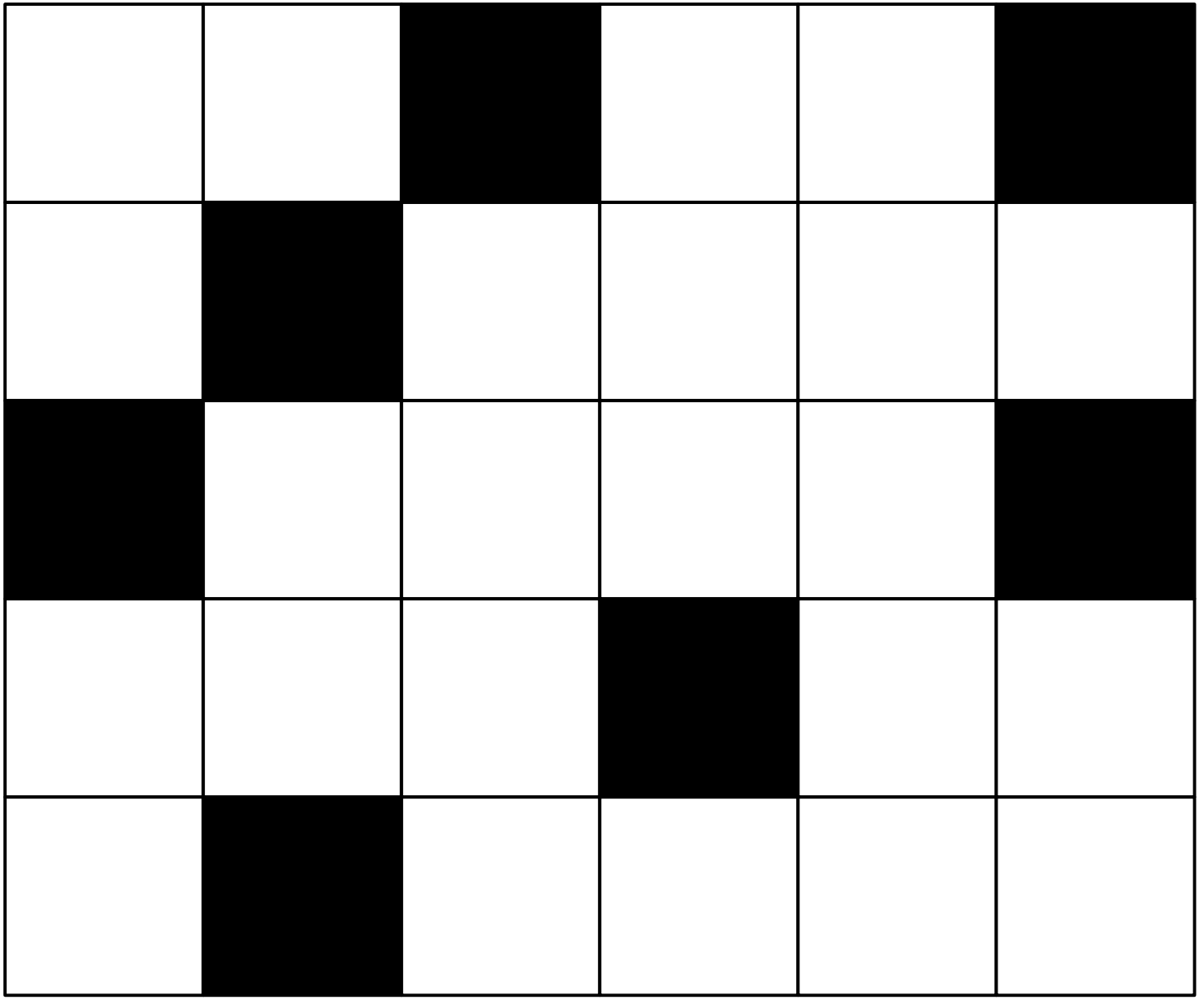
- Given an undirected graph G, a ***matching*** in G is a set of edges such that no two edges share an endpoint.
- A ***maximum matching*** is a matching with the largest number of edges.



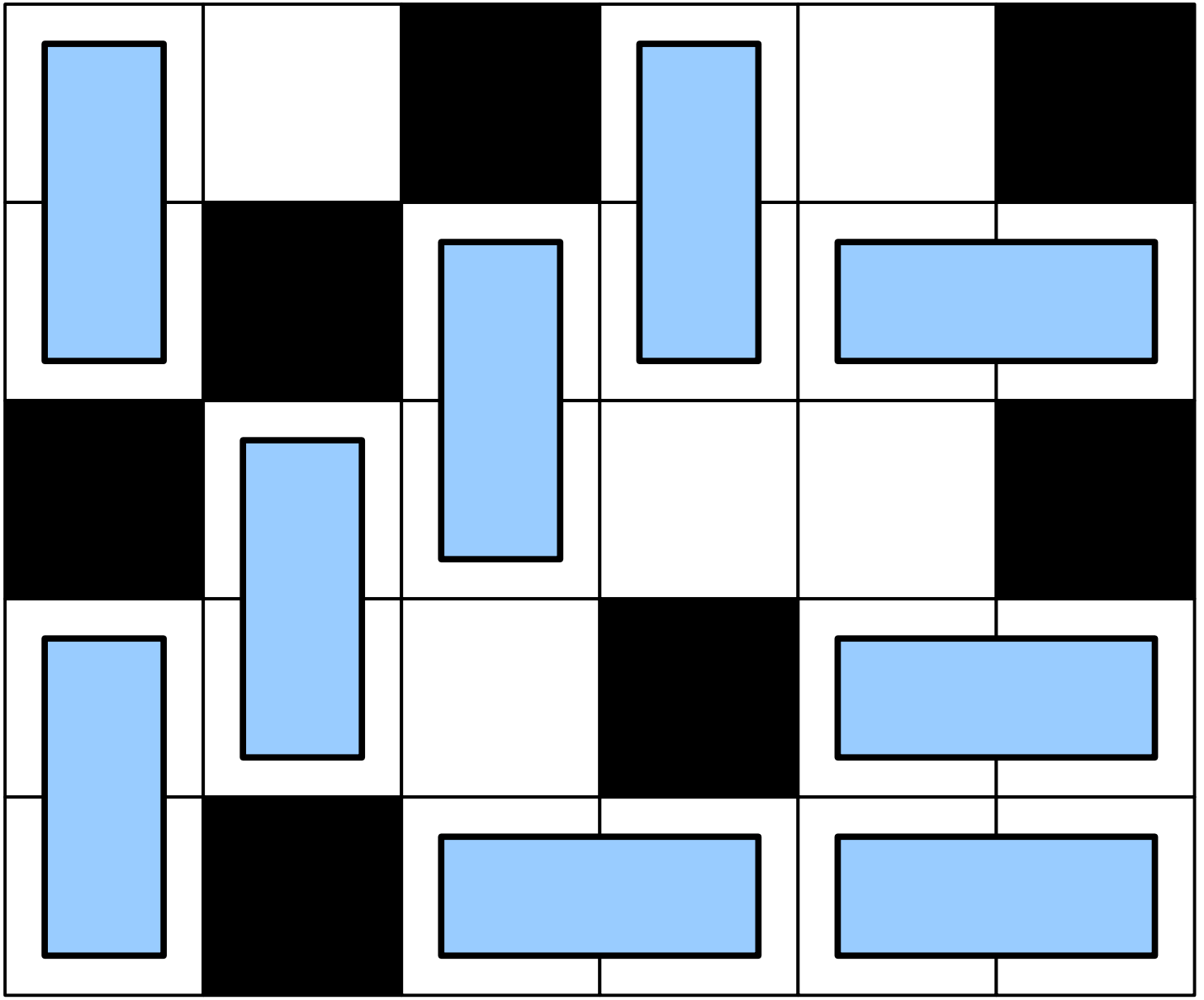
Maximum Matching

- 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?

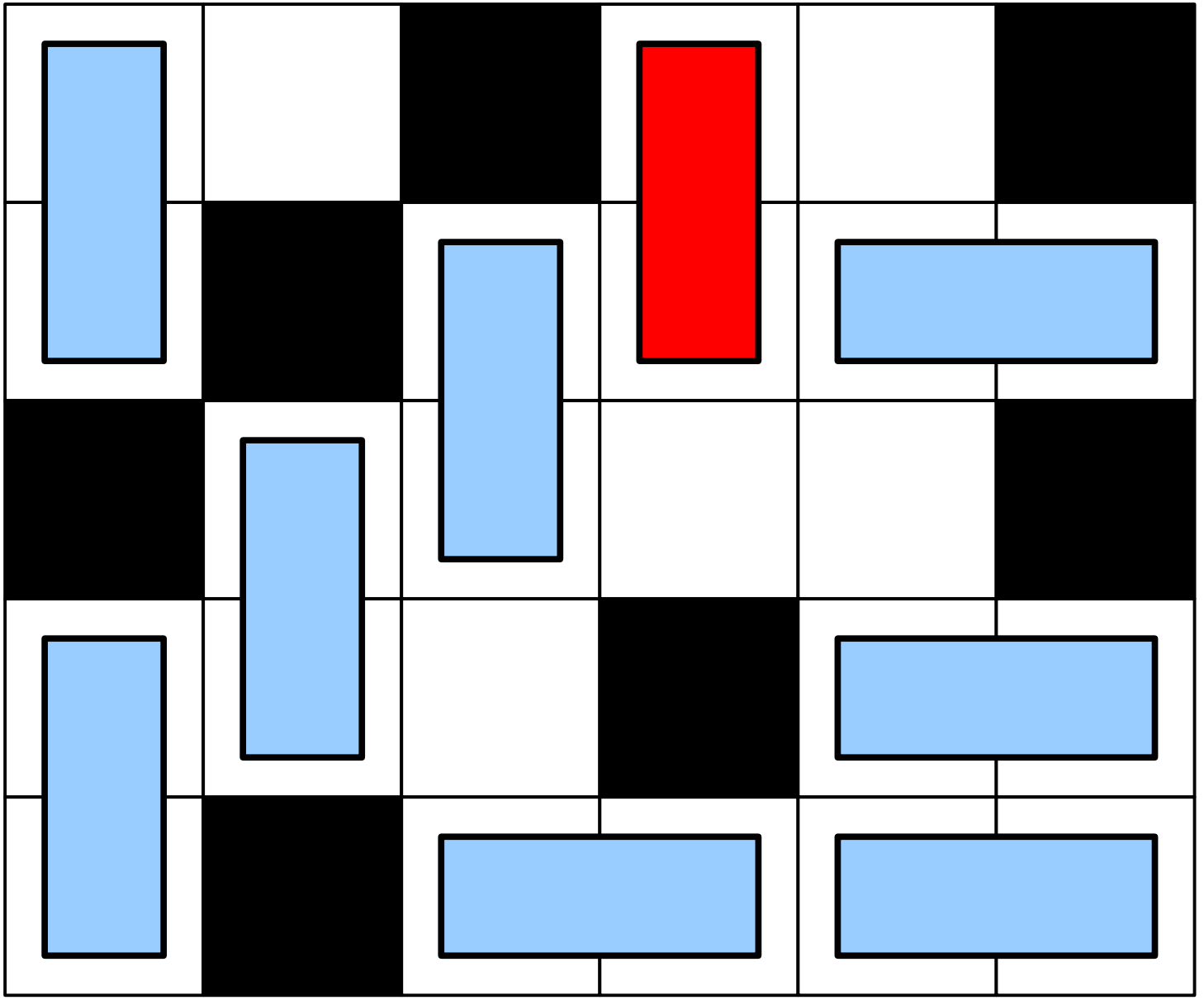
Domino Tiling



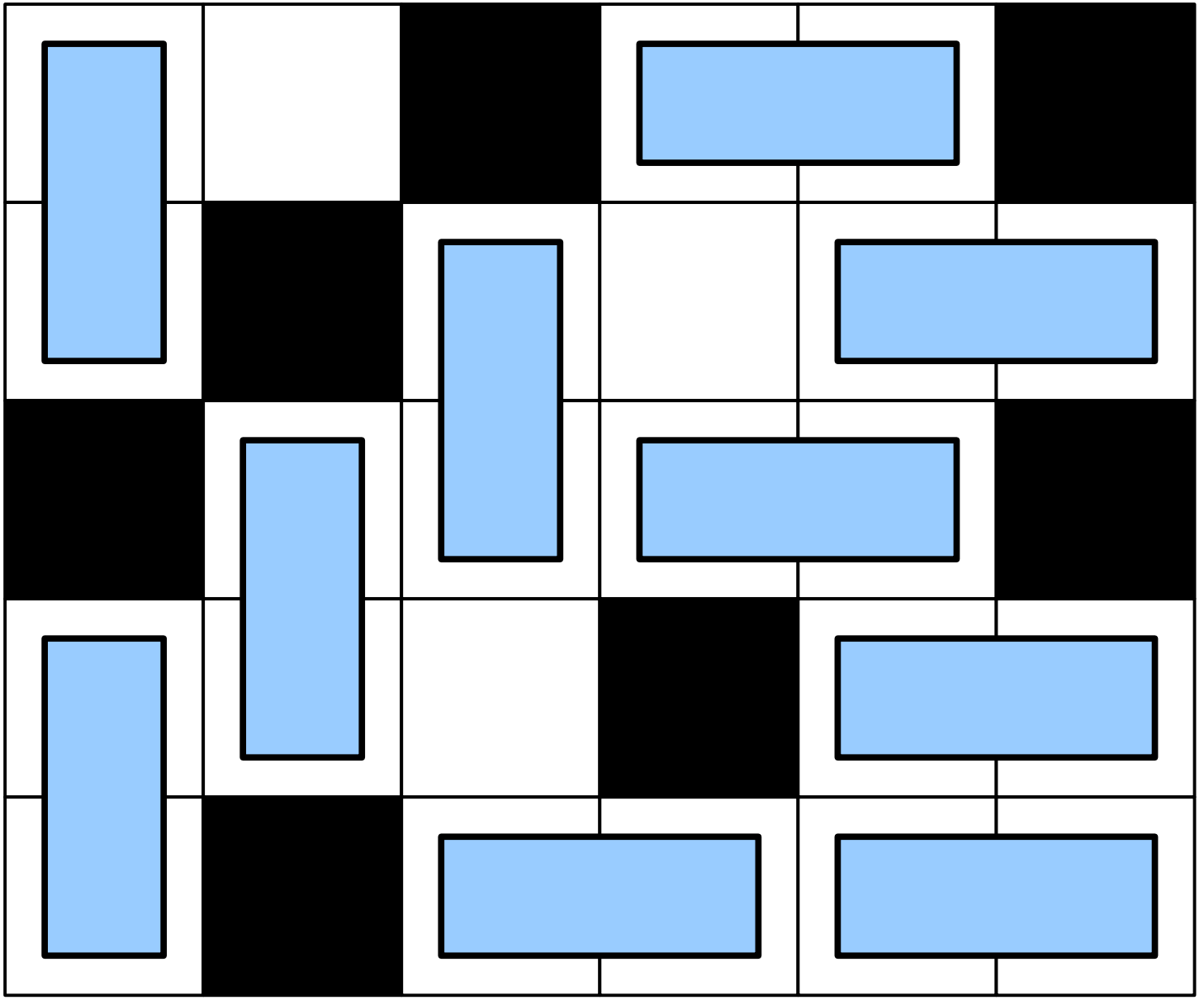
Domino Tiling



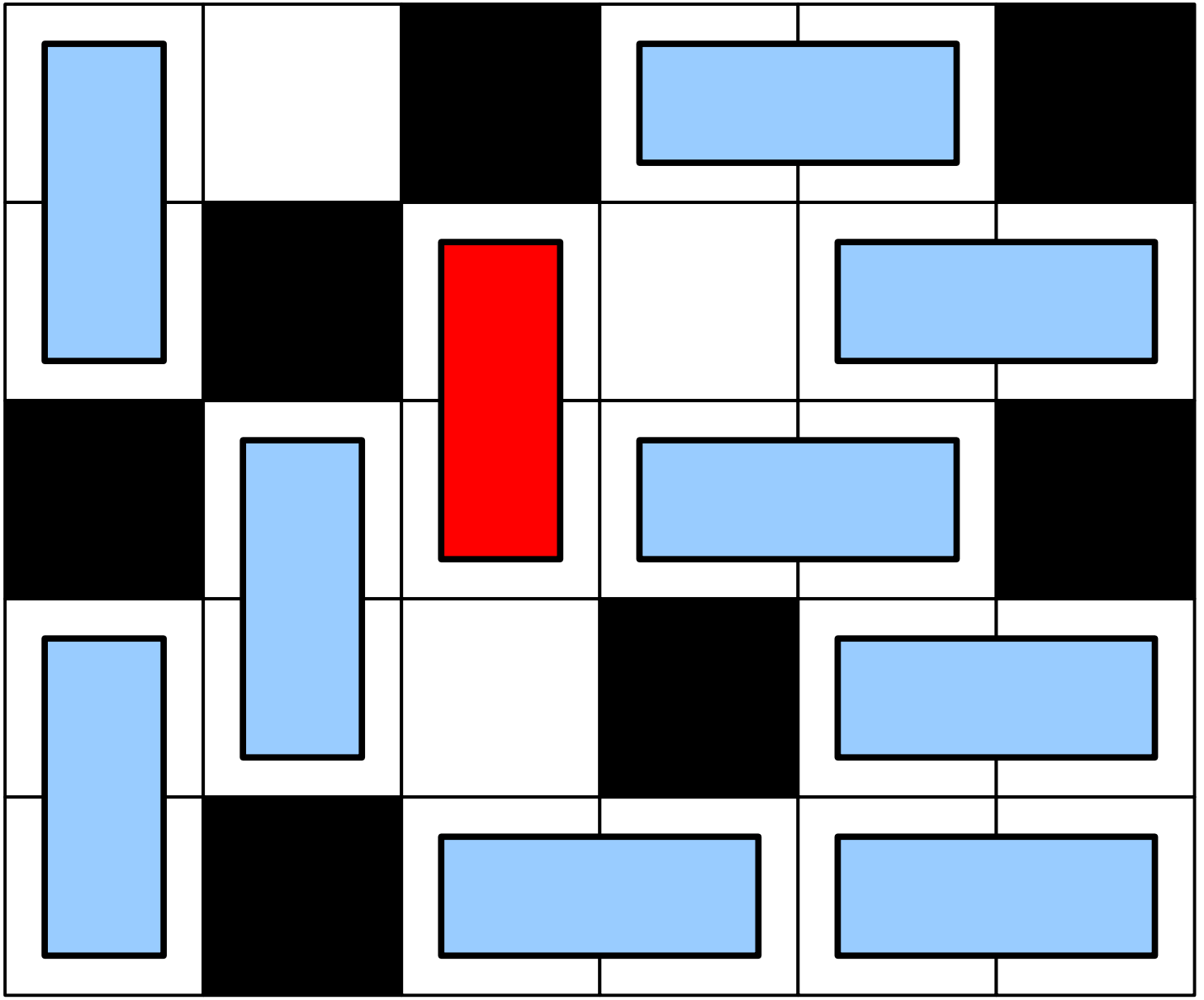
Domino Tiling



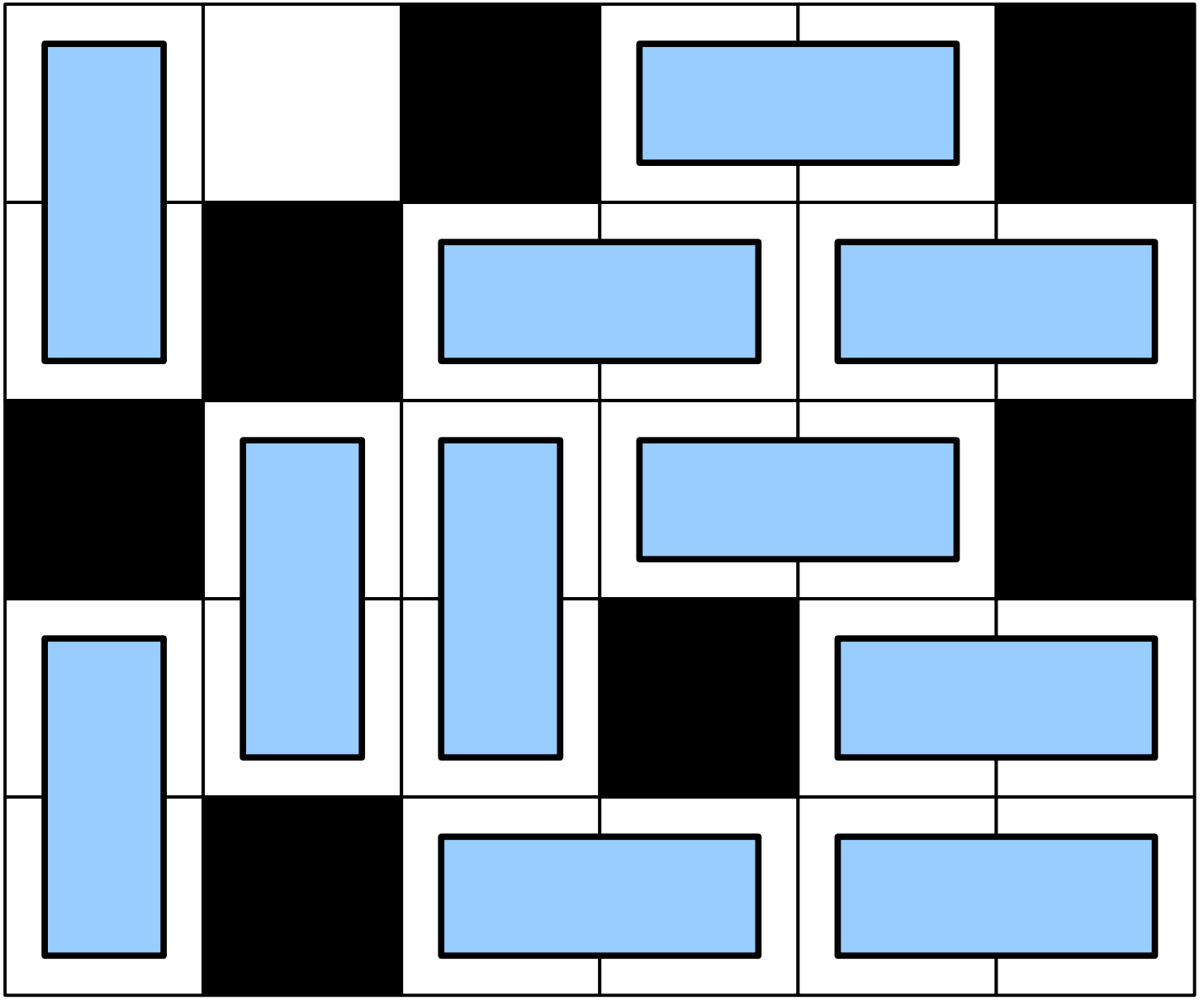
Domino Tiling



Domino Tiling



Domino Tiling



A Domino Tiling Reduction

- Let *MATCHING* be the language defined as follows:

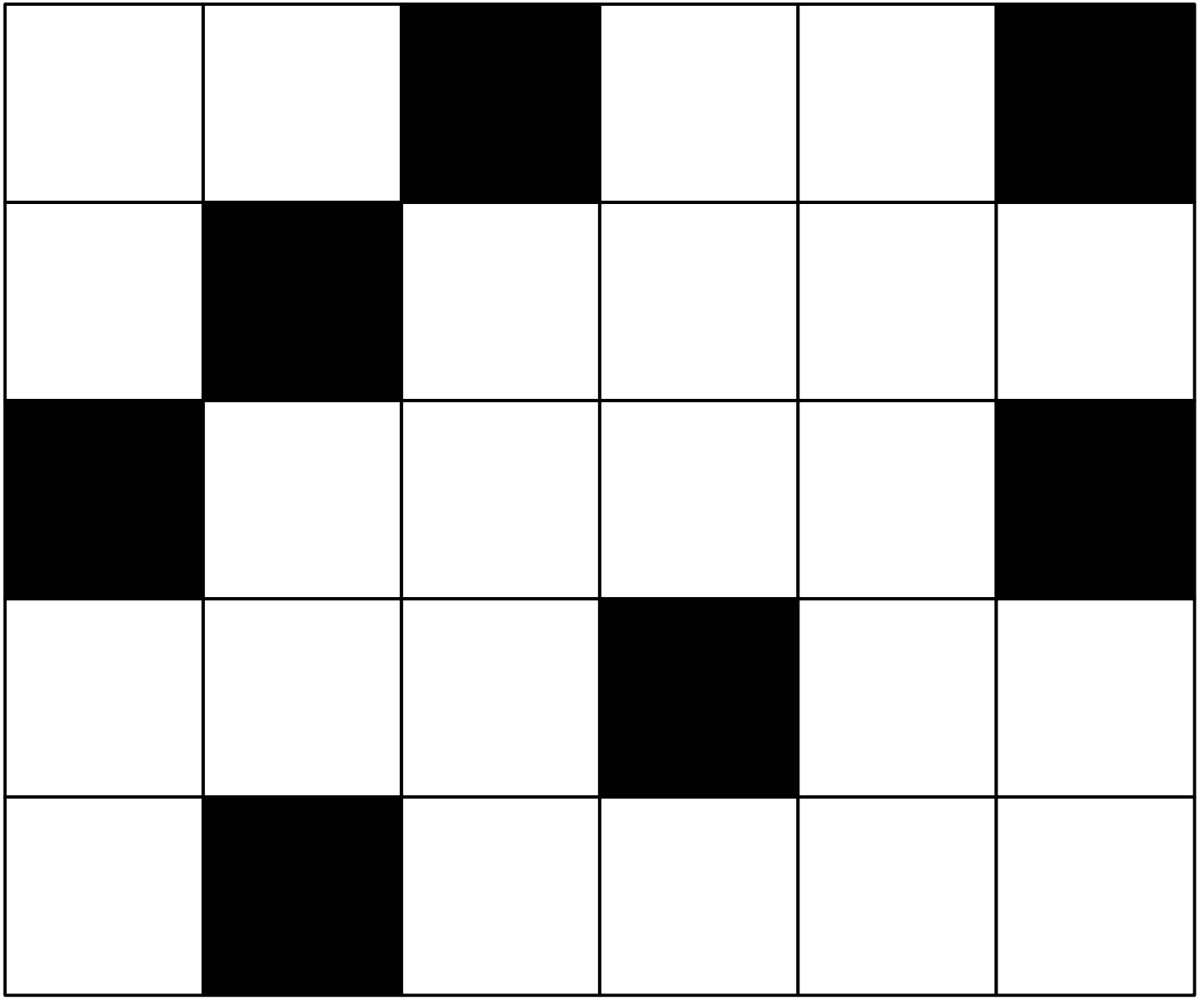
  $MATCHING = \{ \langle G, k \rangle \mid G$ is an undirected graph with a matching of size at least $k \}$

- ***Theorem (Edmonds)***: $MATCHING \in \mathbf{P}$.
- Let *DOMINO* be this language:

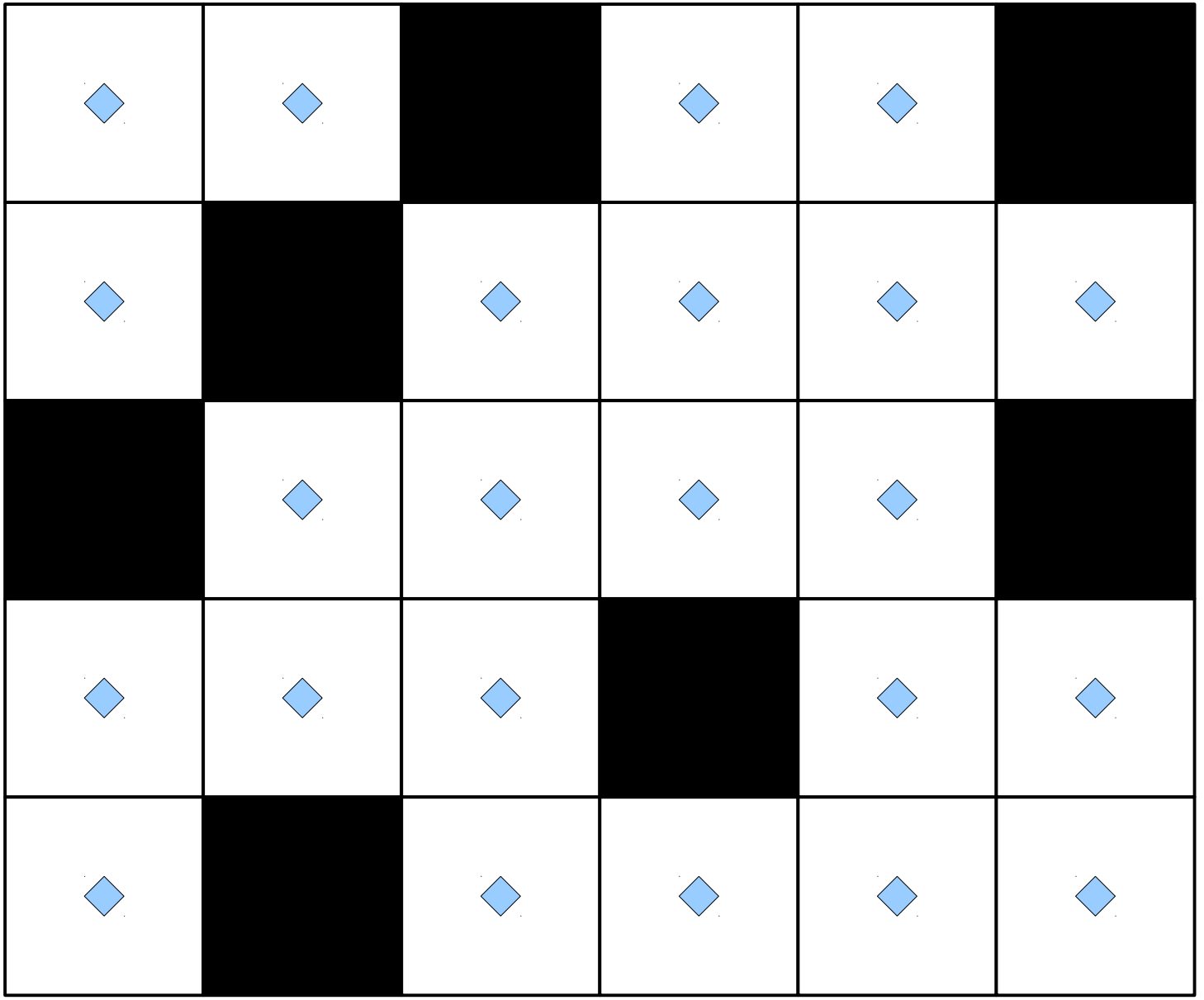
  $DOMINO = \{ \langle D, k \rangle \mid D$ is a grid and k nonoverlapping dominoes can be placed on D. $\}$

- We'll use the fact that $MATCHING \in \mathbf{P}$ to prove that $DOMINO \in \mathbf{P}$.

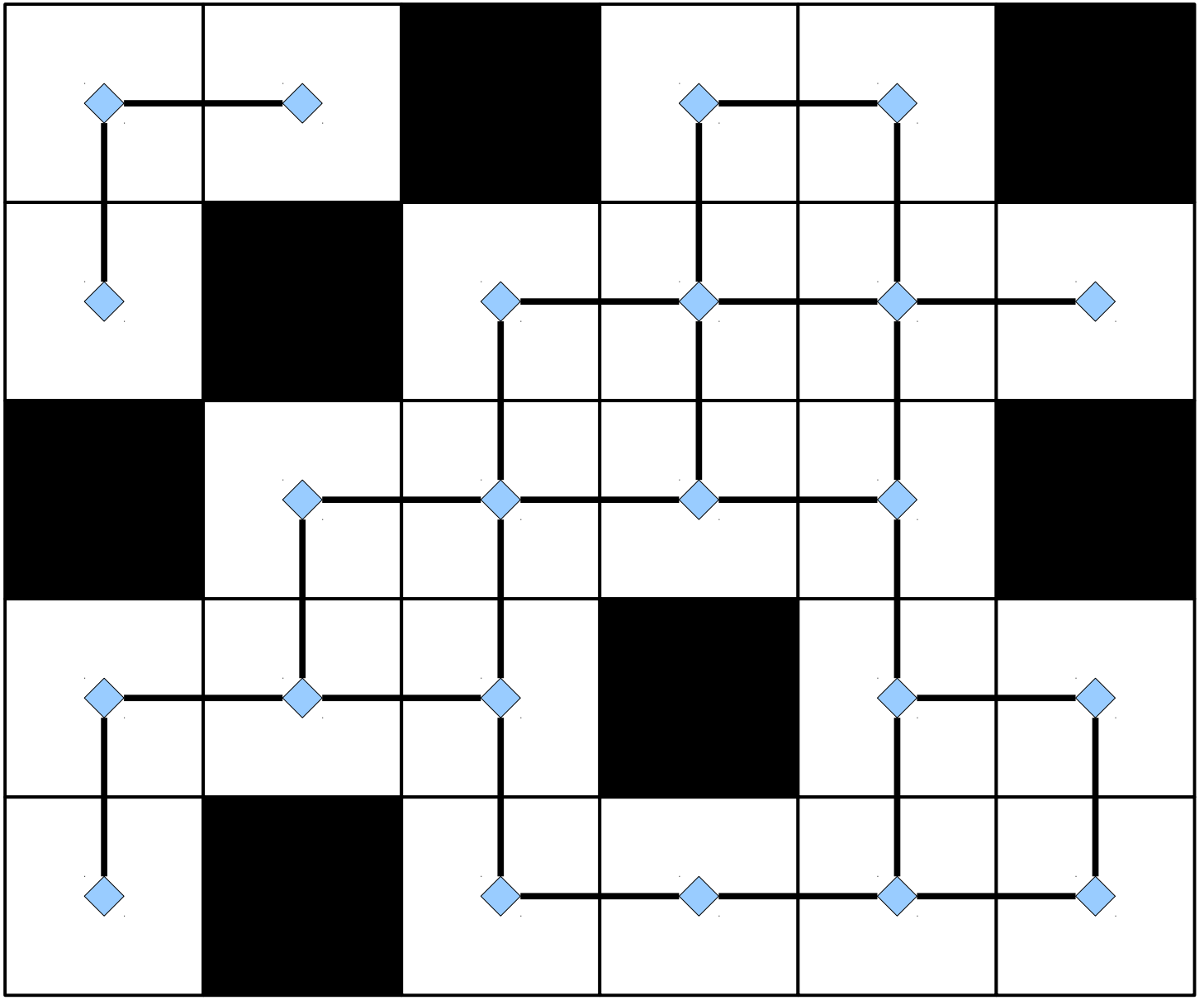
Solving Domino Tiling



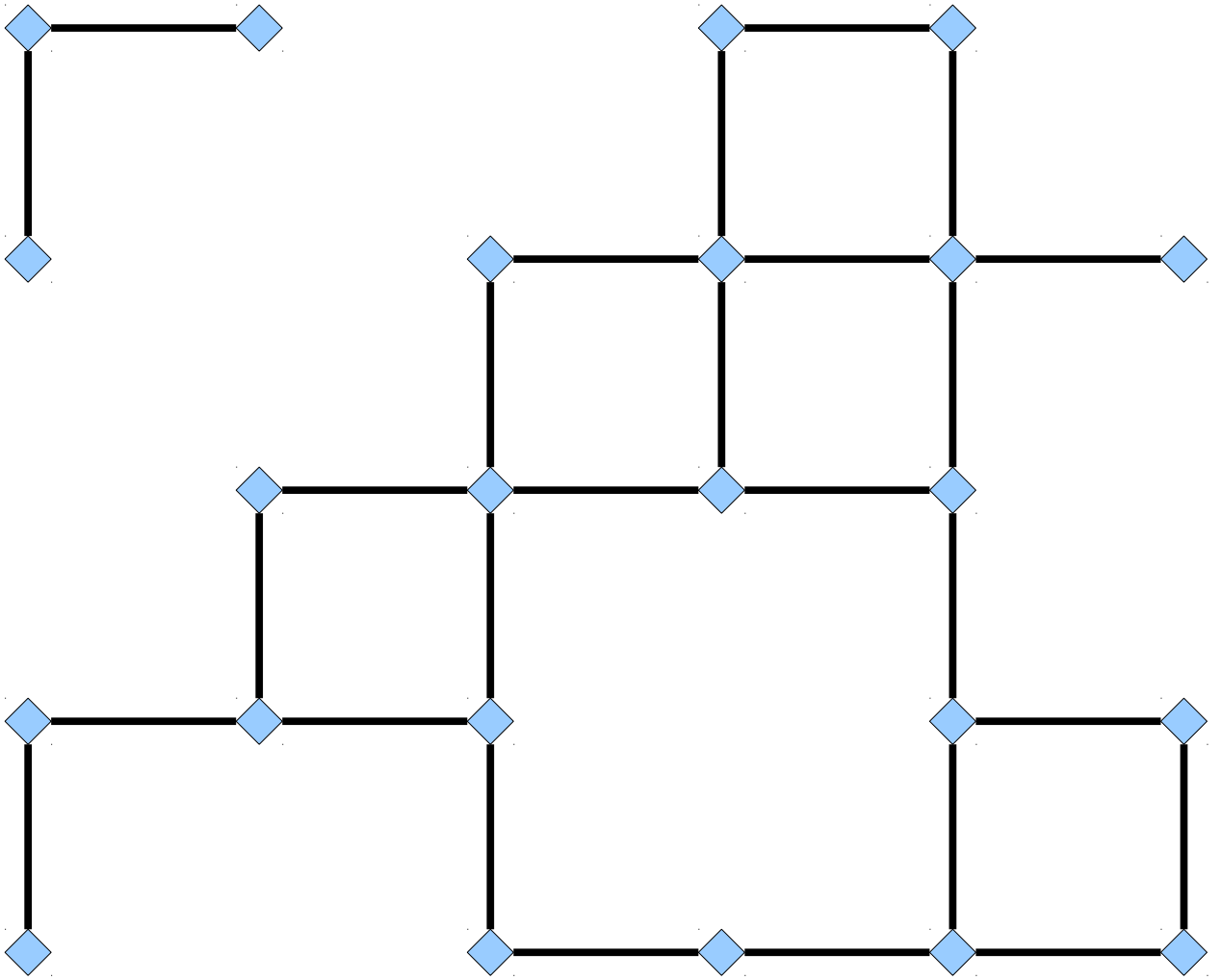
Solving Domino Tiling



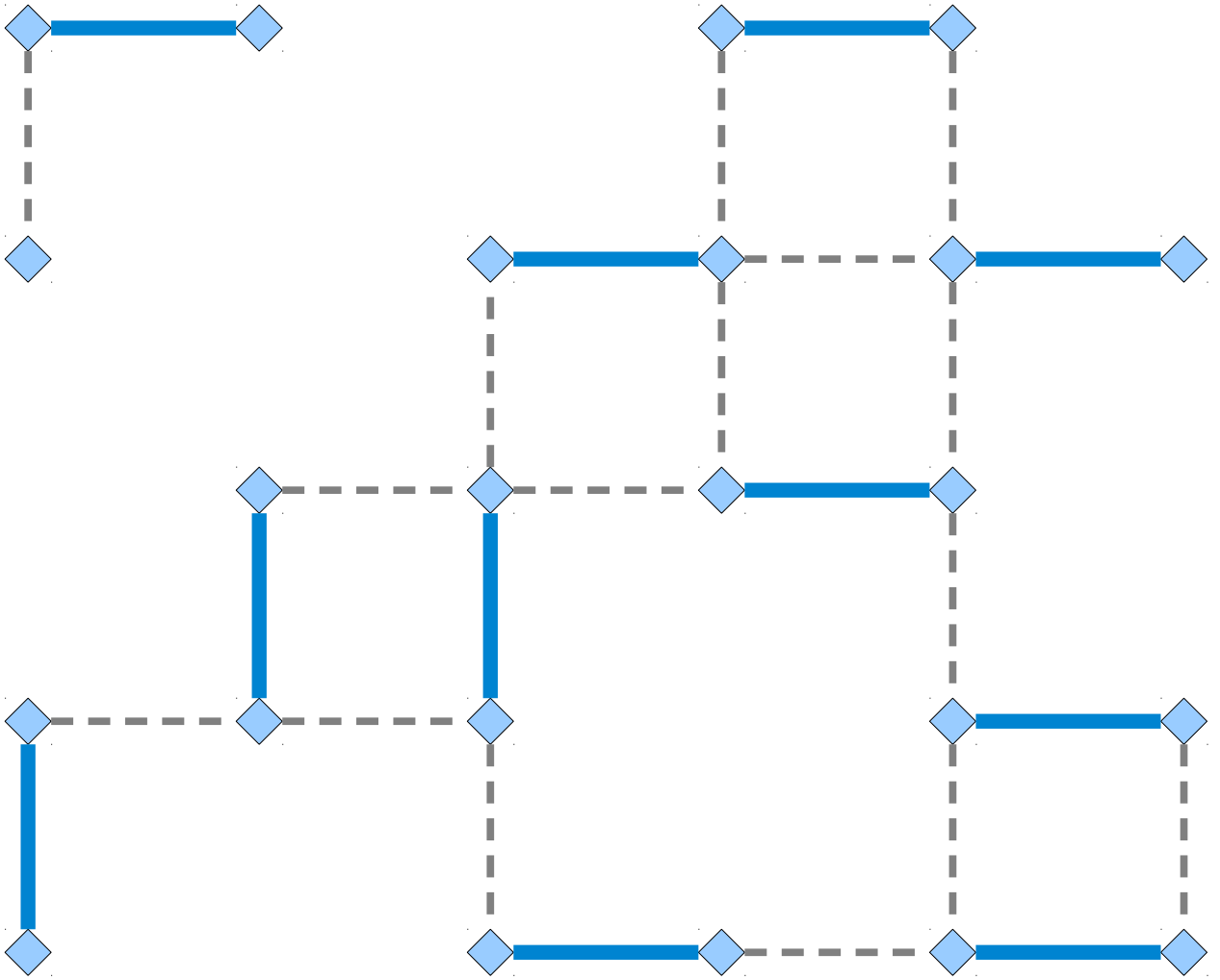
Solving Domino Tiling



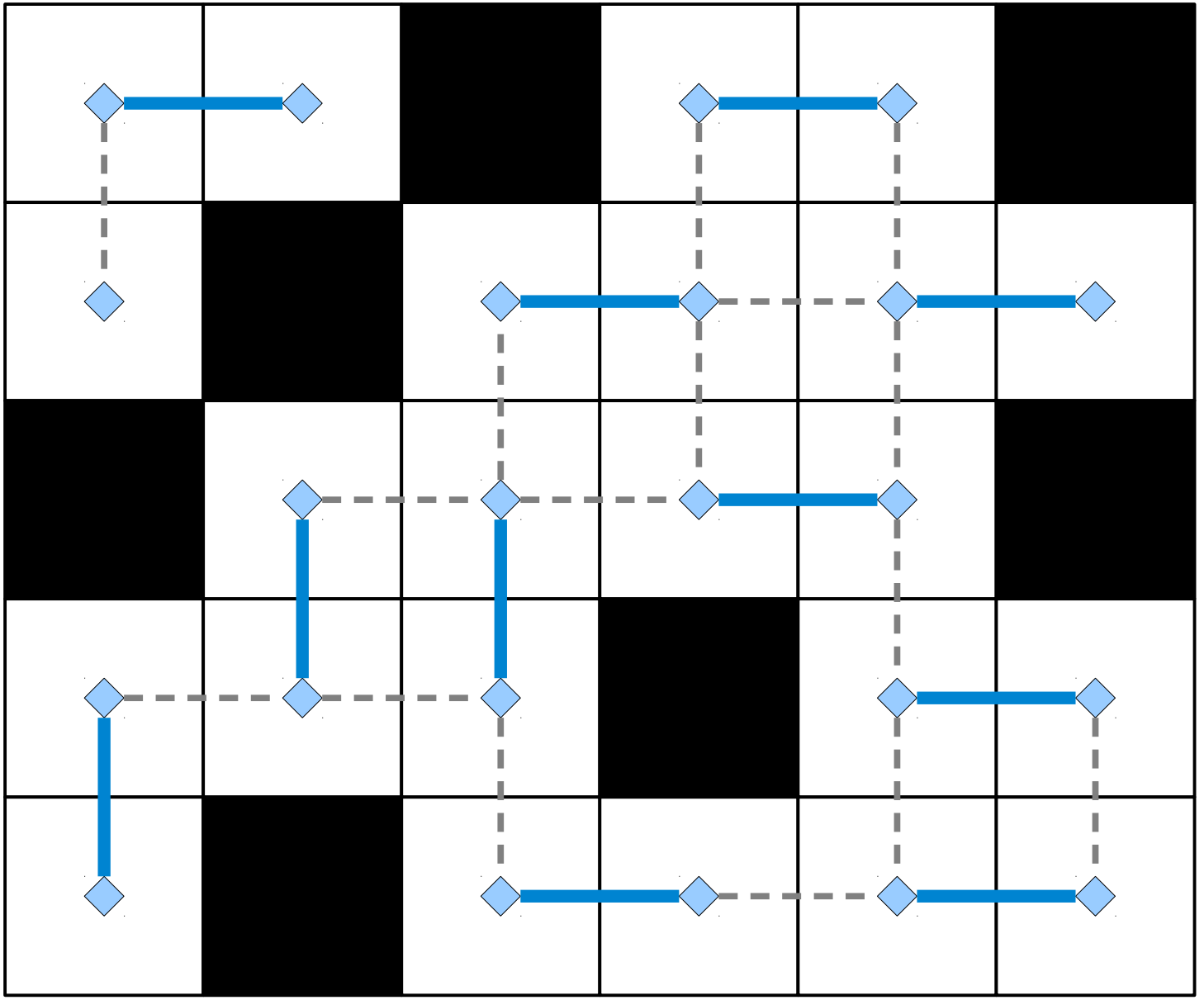
Solving Domino Tiling



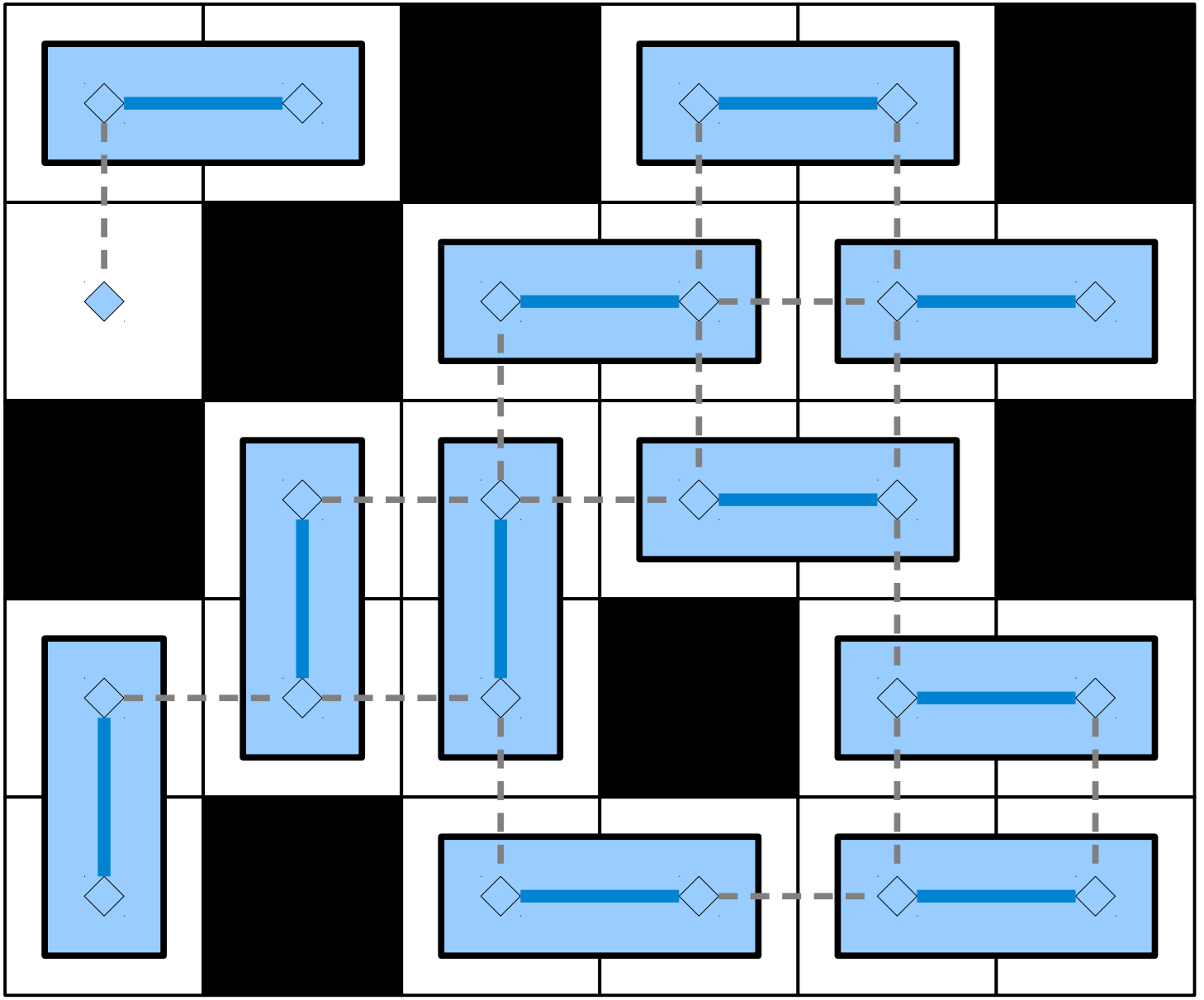
Solving Domino Tiling



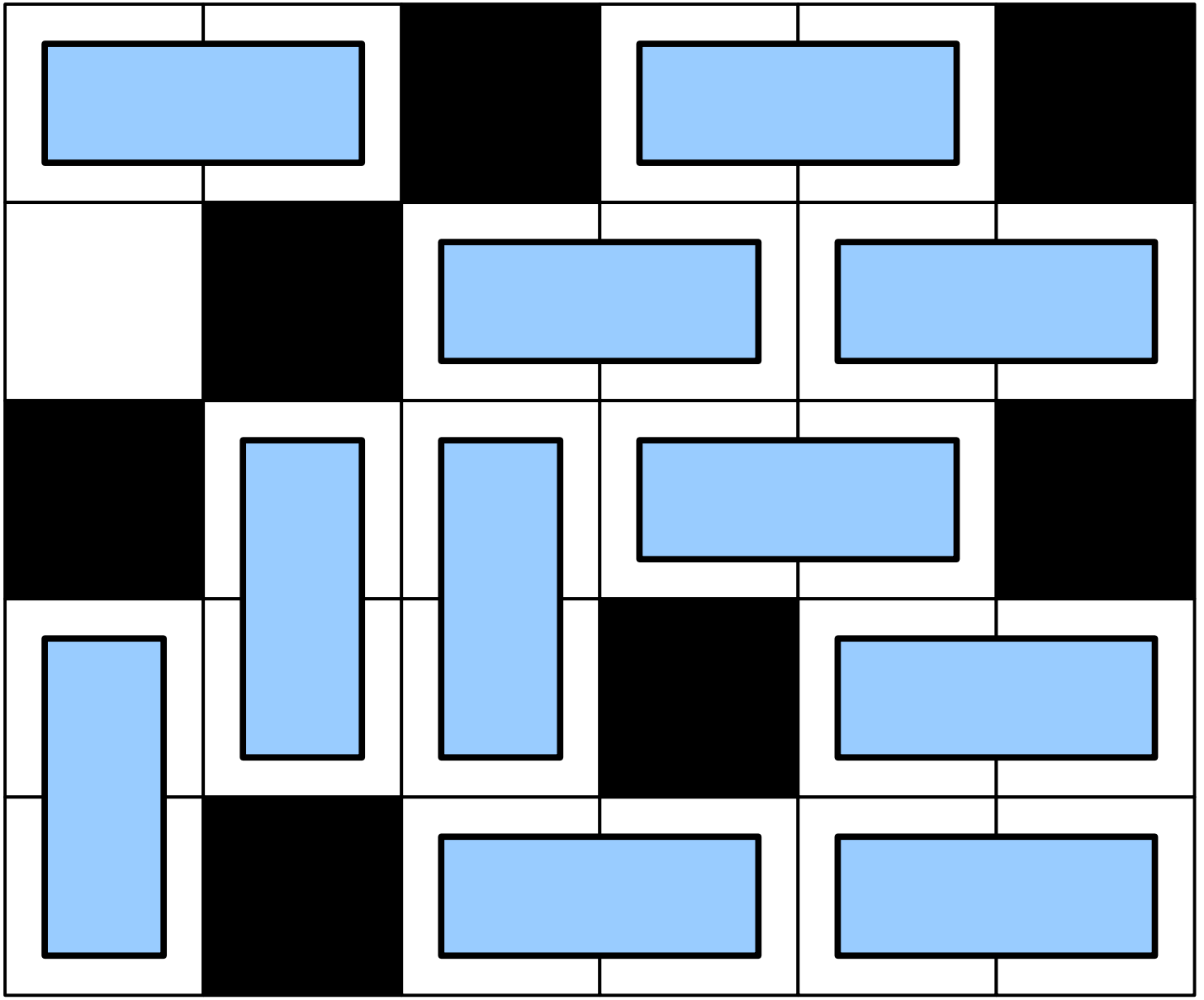
Solving Domino Tiling



Solving Domino Tiling



Solving Domino Tiling



In Pseudocode

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

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

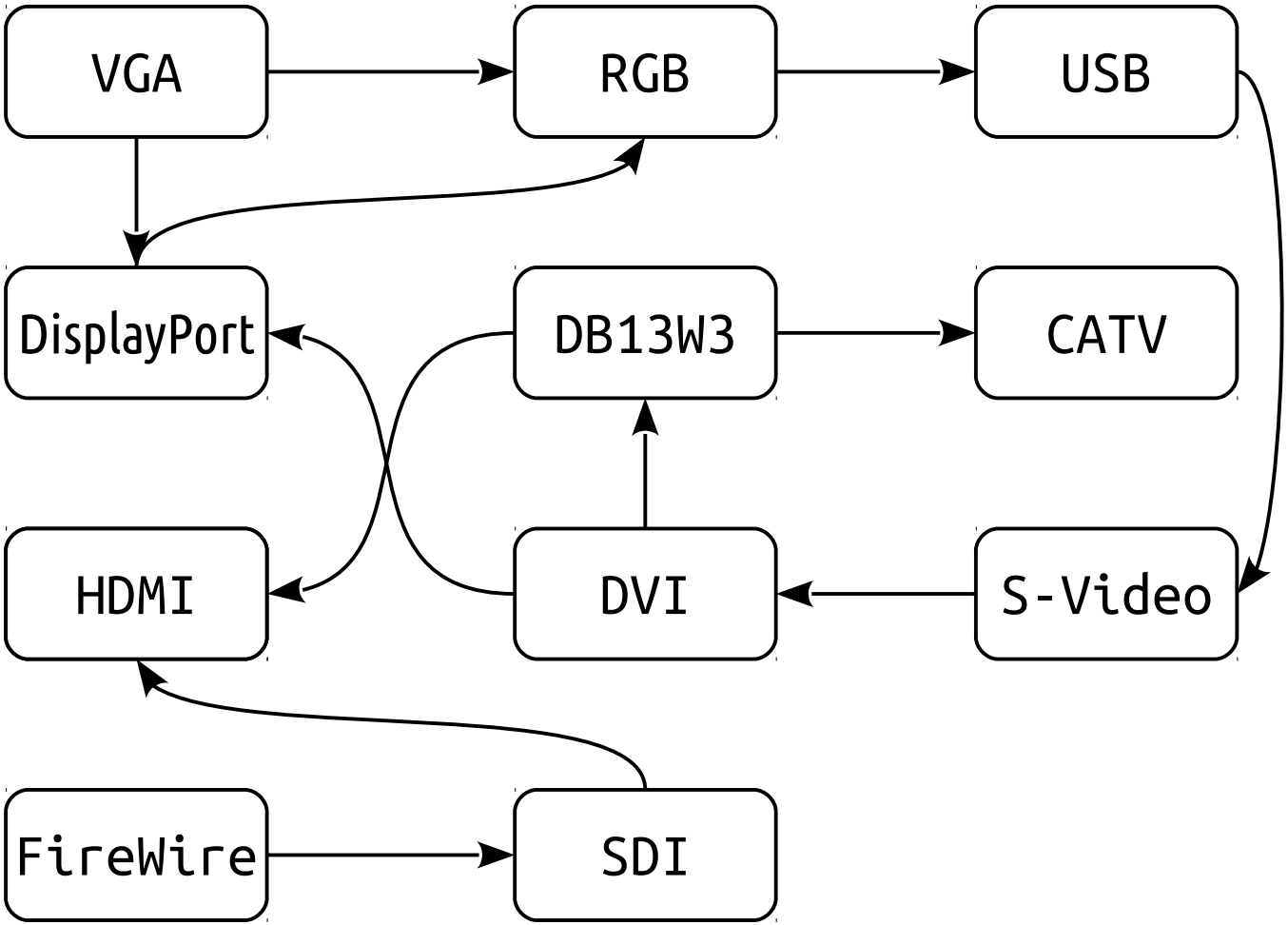
Connectors

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

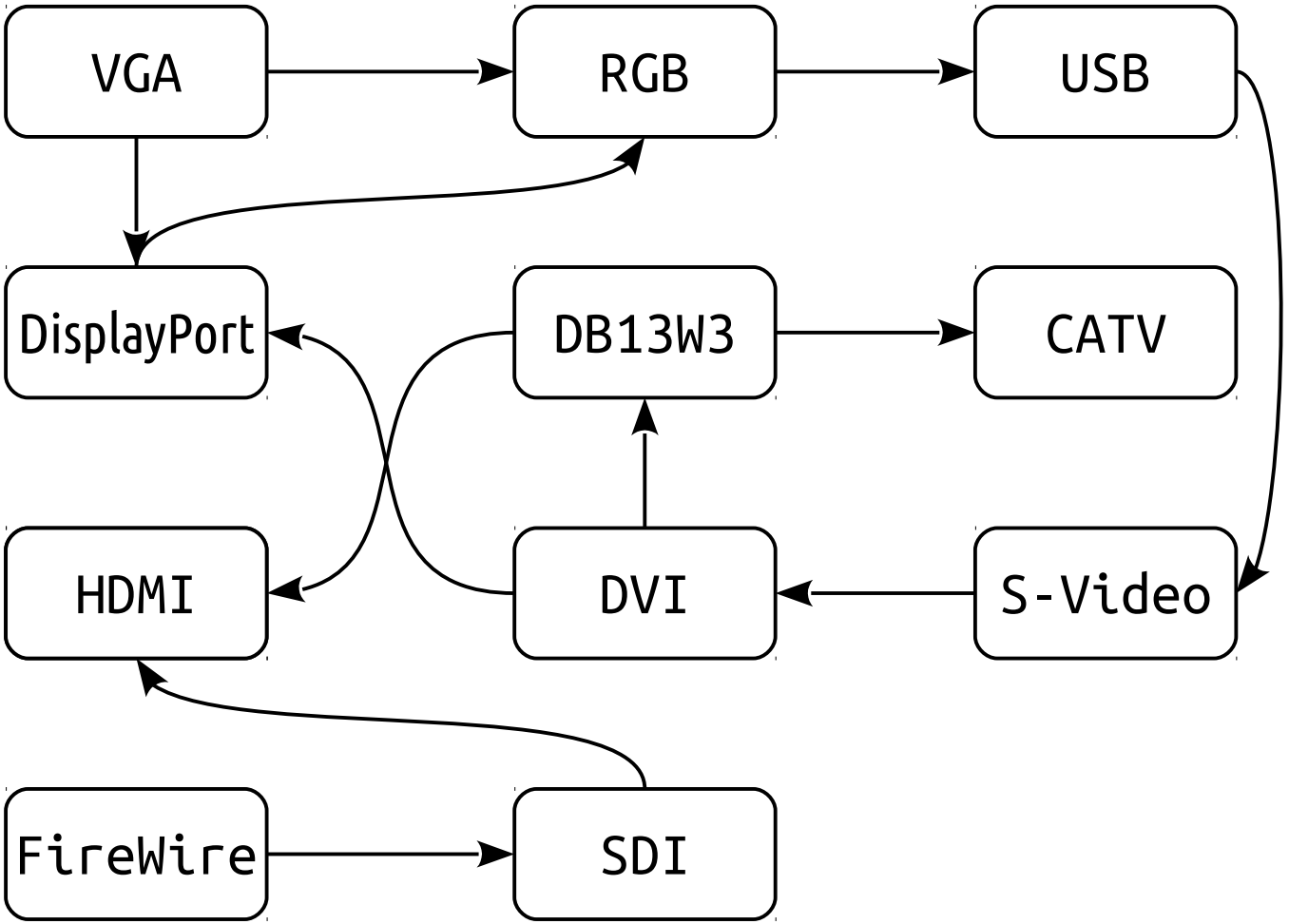
Converter Conundrums

Connectors

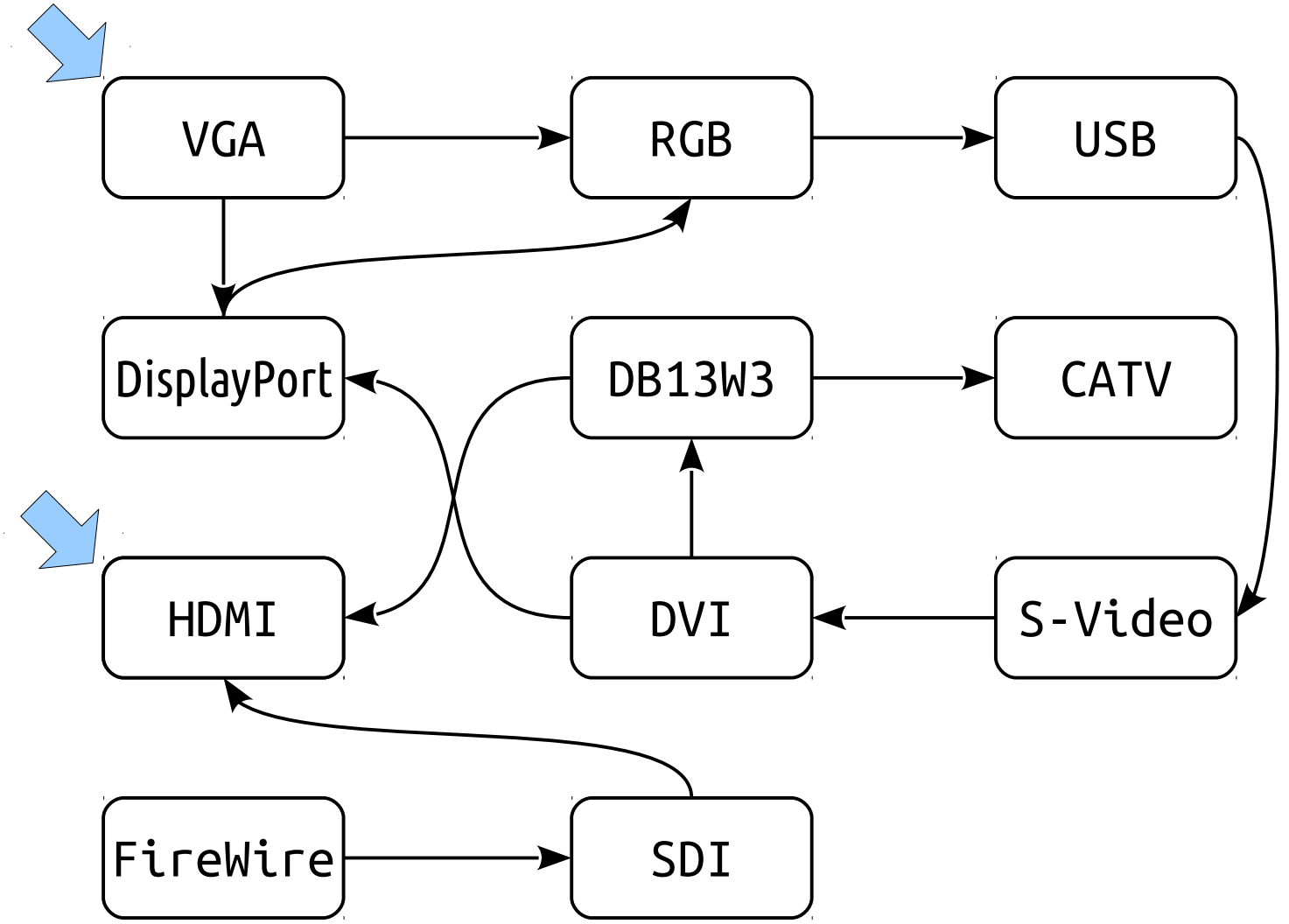
RGB to USB
VGA to DisplayPort
DB13W3 to CATV
DisplayPort to RGB
DB13W3 to HDMI
DVI to DB13W3
S-Video to DVI
FireWire to SDI
VGA to RGB
DVI to DisplayPort
USB to S-Video
SDI to HDMI



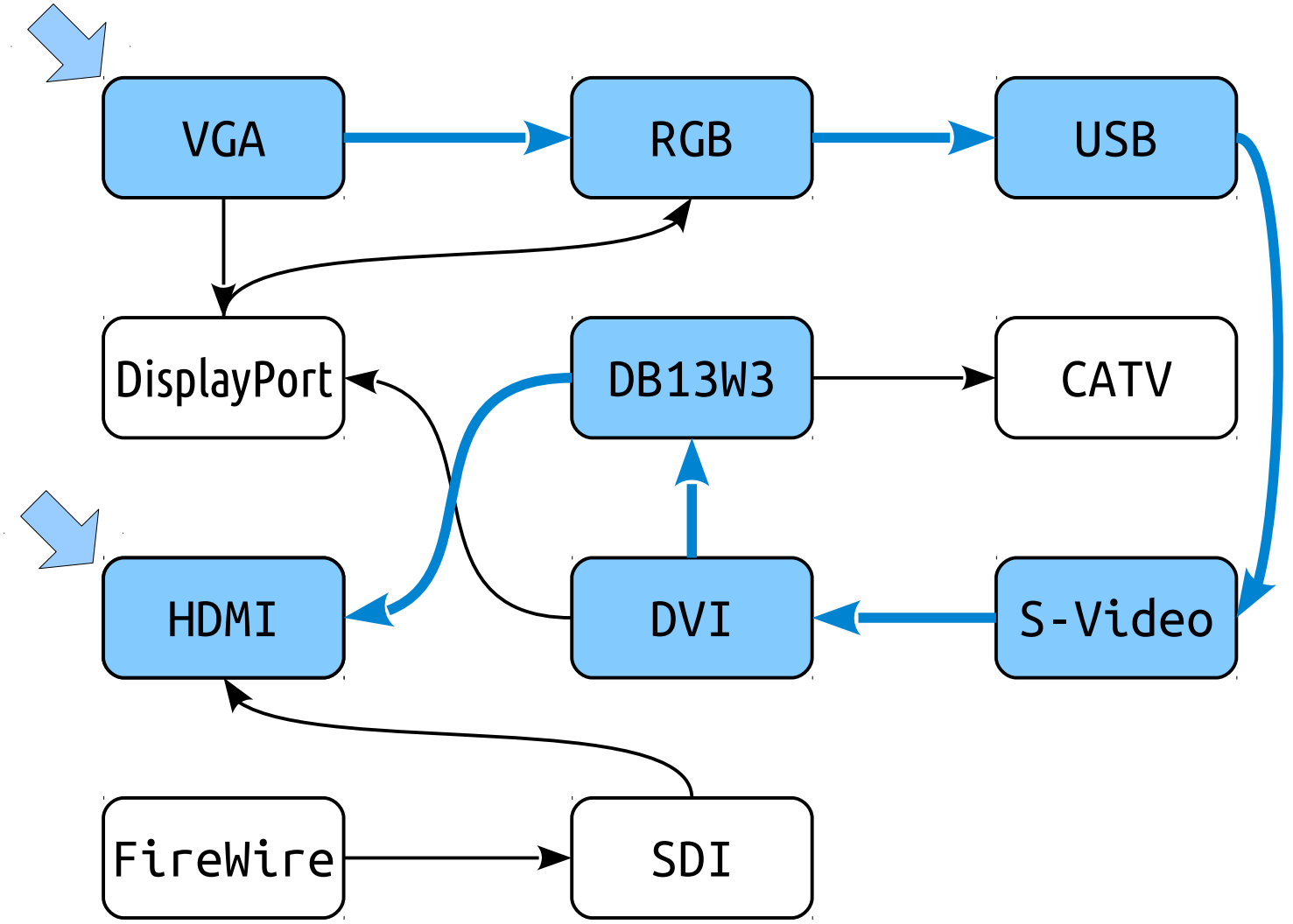
Converter Conundrums



Converter Conundrums



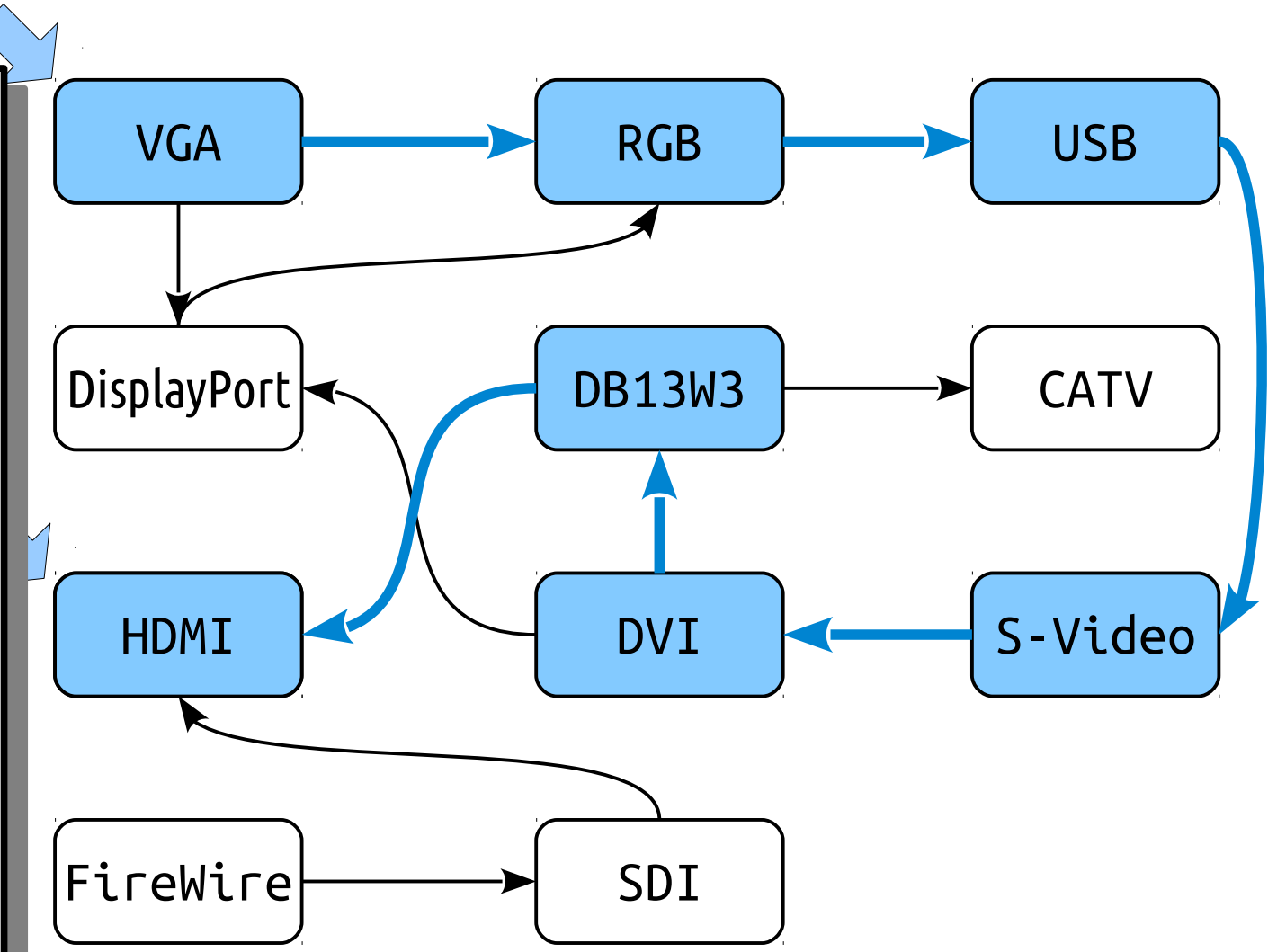
Converter Conundrums



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);
}
```

A Commonality

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

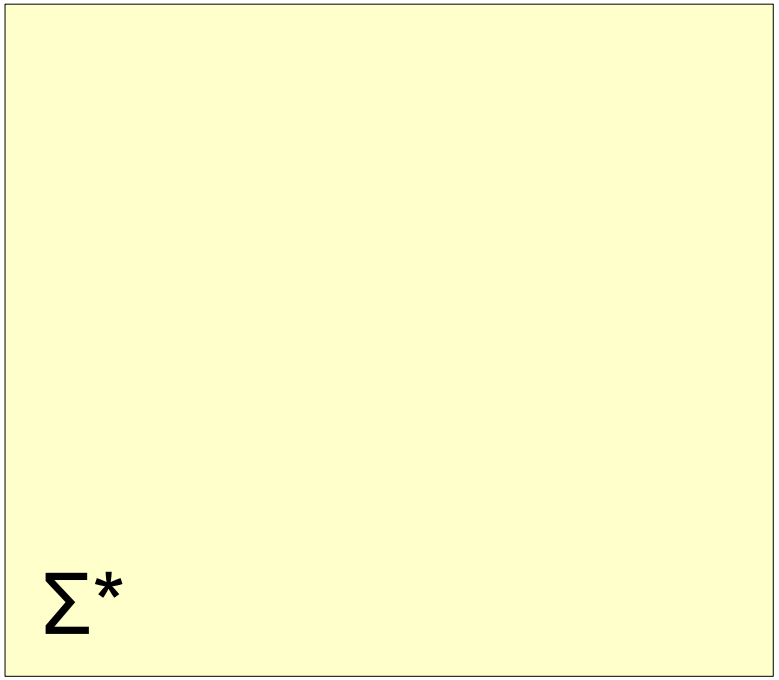
$$\forall w \in \Sigma^*. (w \in A \leftrightarrow f(w) \in B)$$

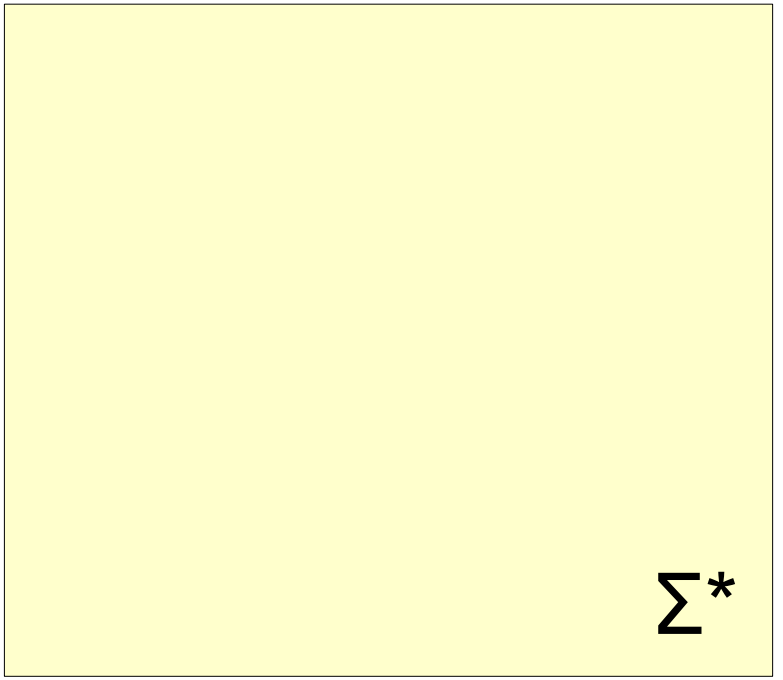
- ***If you want to know whether $w \in A$, you can instead ask whether $f(w) \in B$.***
  - Every $w \in A$ maps to some $f(w) \in B$.
  - Every $w \notin A$ maps to some $f(w) \notin B$.

Polynomial-Time Reductions

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

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

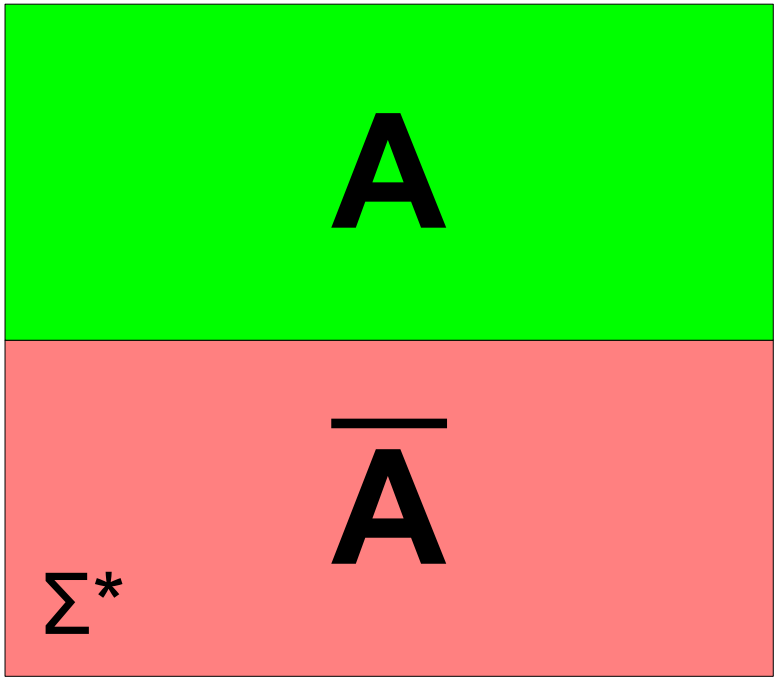


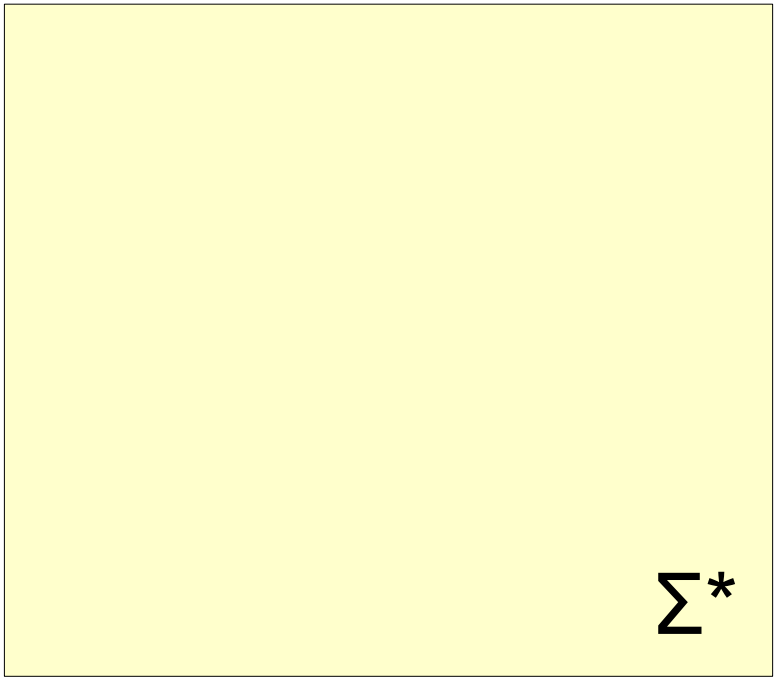


Polynomial-Time Reductions

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

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

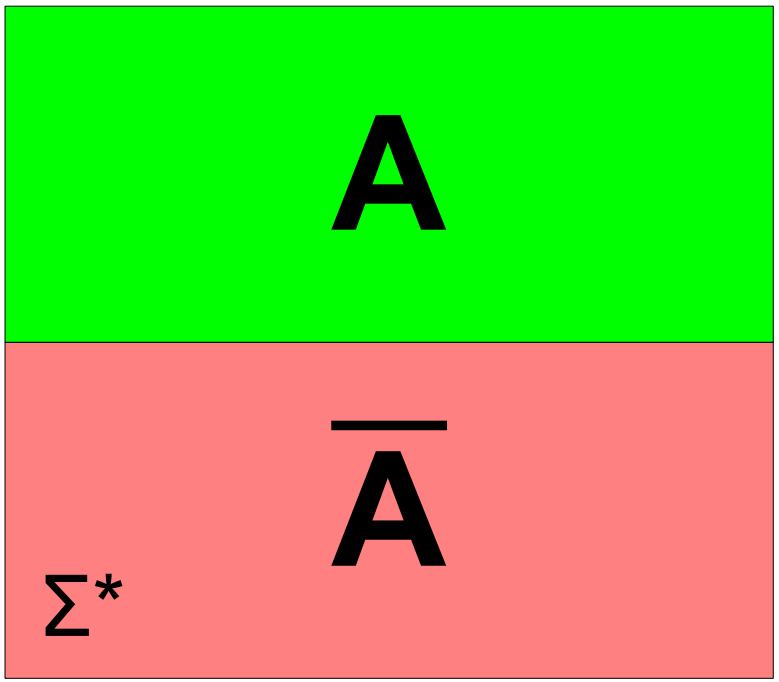


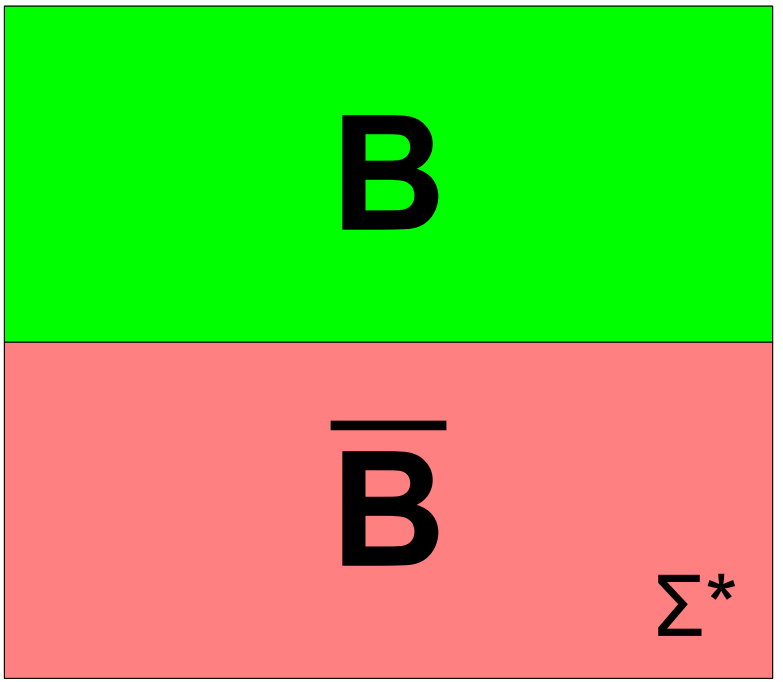


Polynomial-Time Reductions

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

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

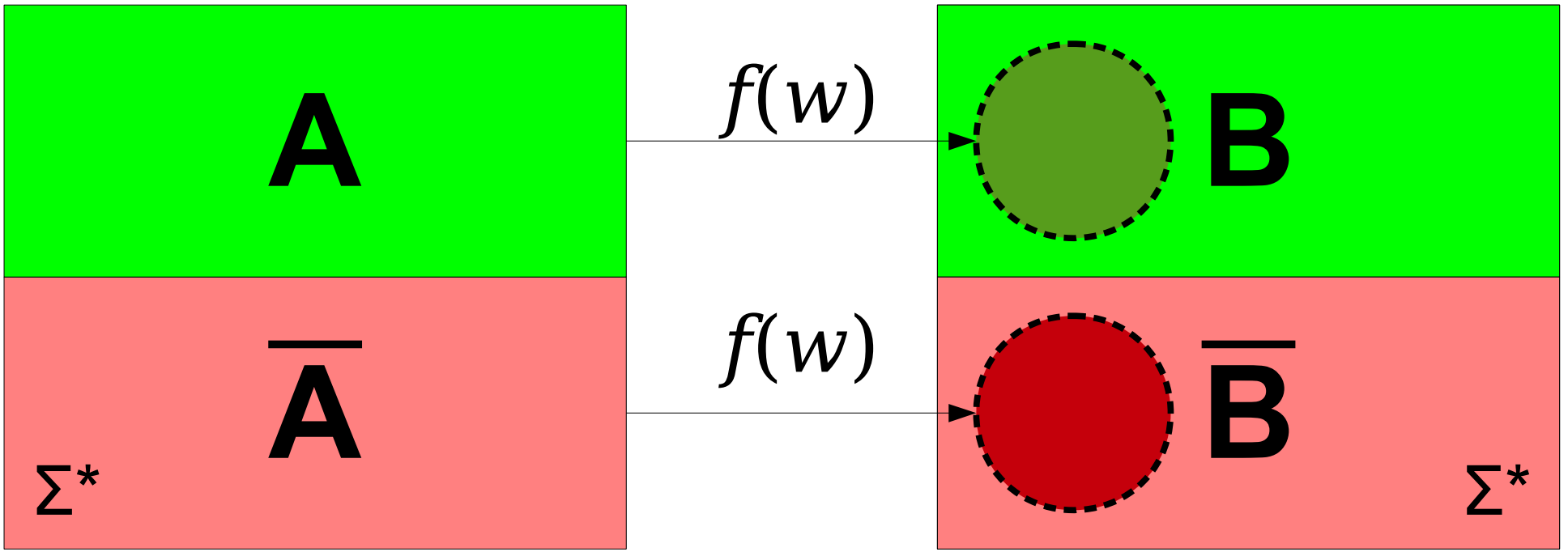




Polynomial-Time Reductions

- 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."

Two Key Theorems

- ***Theorem 1:*** If $A \leq_p B$ and $B \in \mathbf{P}$, then $A \in \mathbf{P}$.
- ***Theorem 2:*** If $A \leq_p B$ and $B \in \mathbf{NP}$, then $A \in \mathbf{NP}$.
- Intuitively, if $A \leq_p B$, then A is "not harder than" B and B is "at least as hard as" A.
- ***Proof idea:*** For (1), show that applying the reduction from A to B and solving B solves A in polynomial time. For (2), show that applying the reduction from A to B lets us use a verifier for B to verify answers to A.

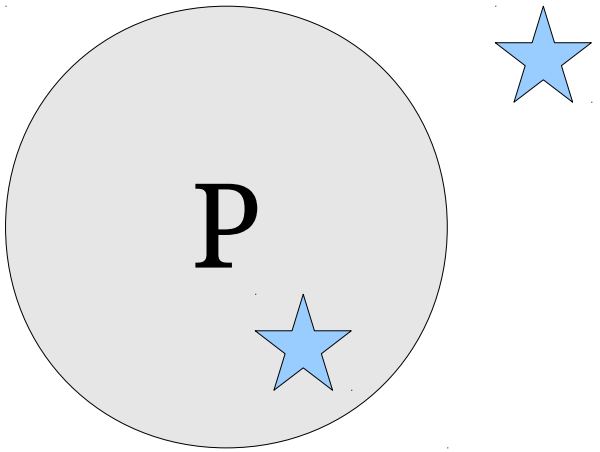
Reducibility, Summarized

- 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

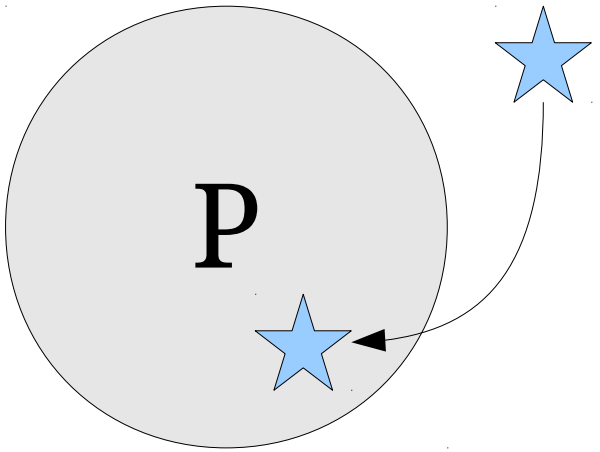
Polynomial-Time Reductions

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



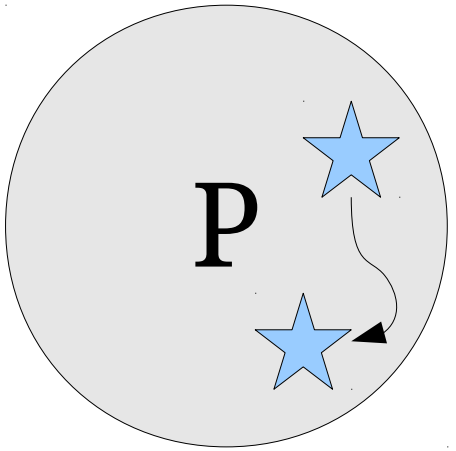
Polynomial-Time Reductions

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



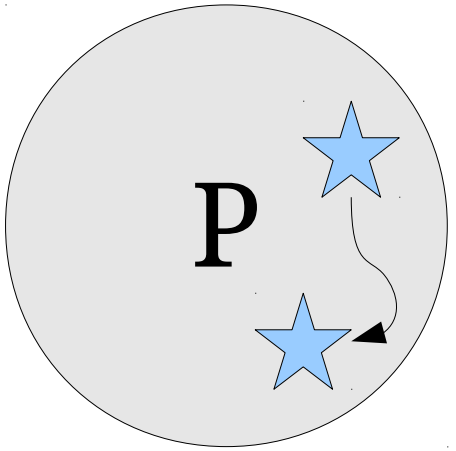
Polynomial-Time Reductions

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



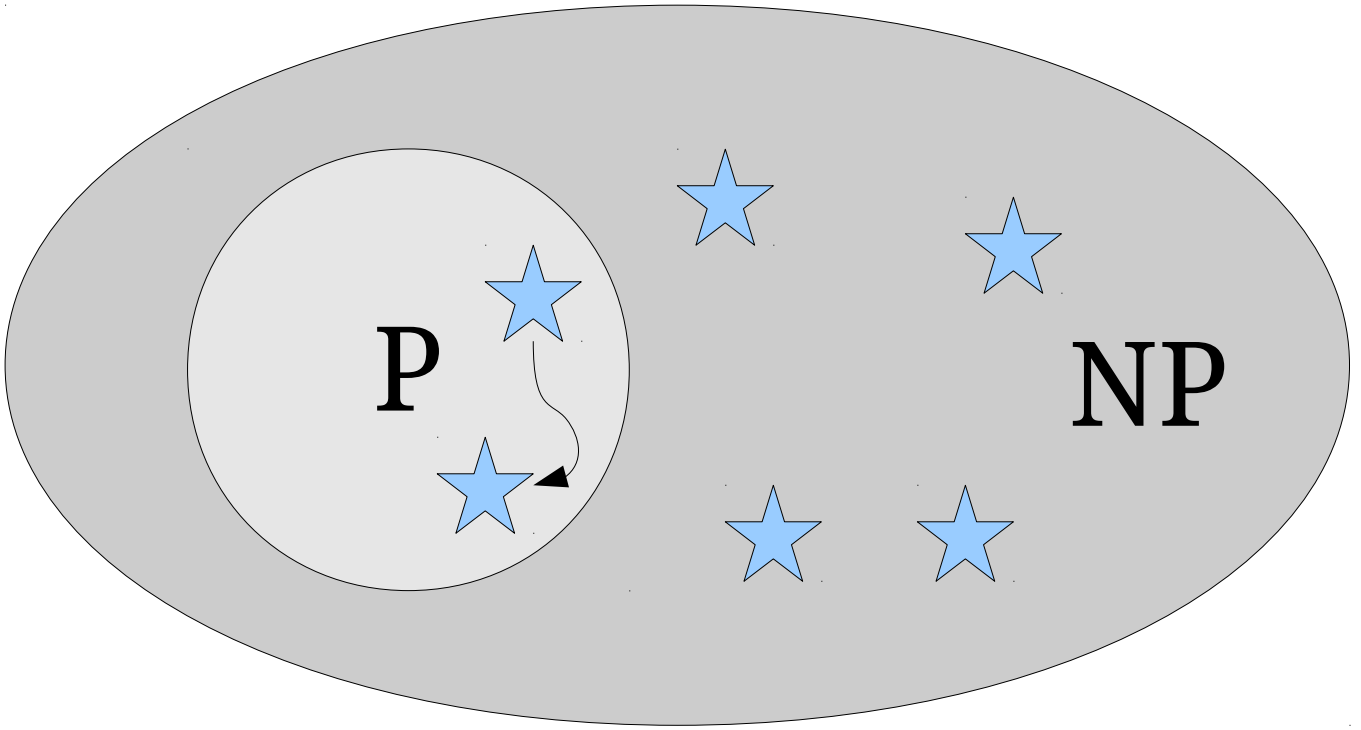
Polynomial-Time Reductions

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



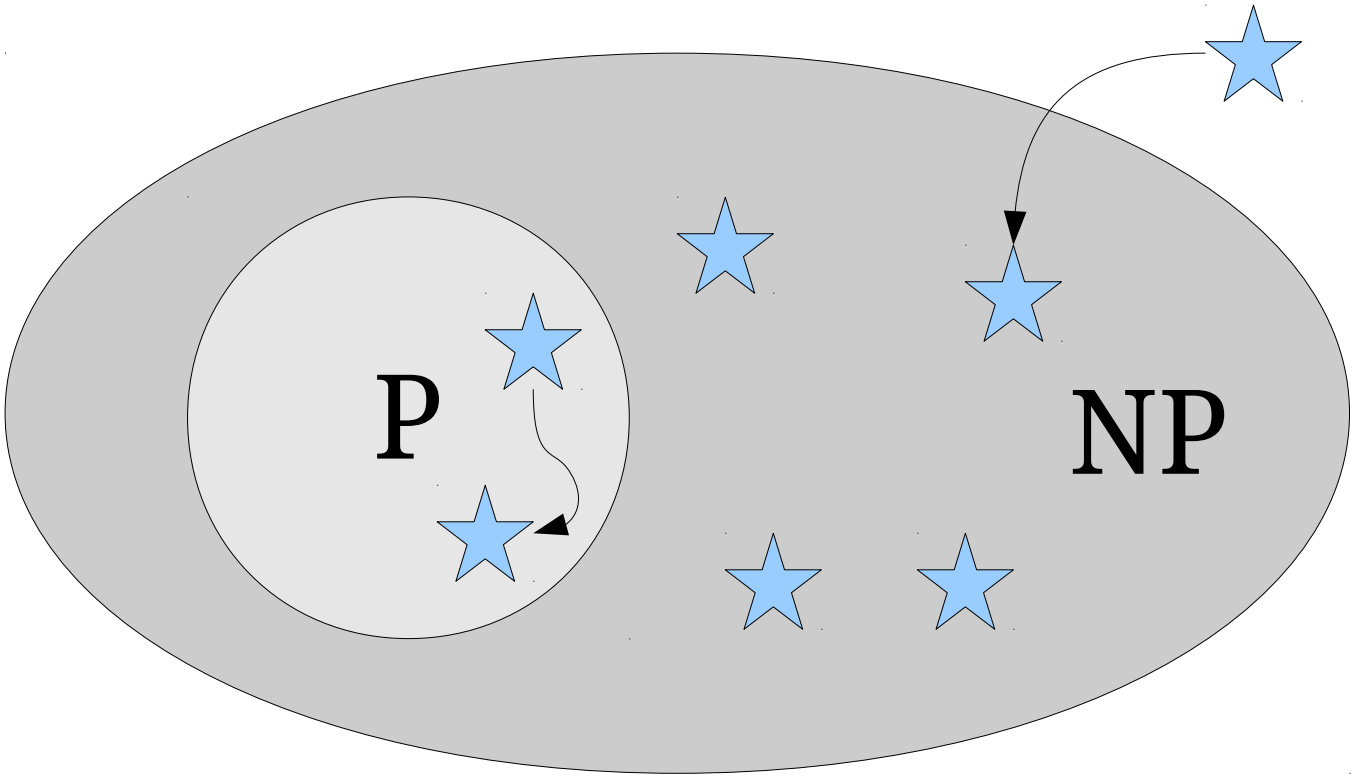
Polynomial-Time Reductions

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



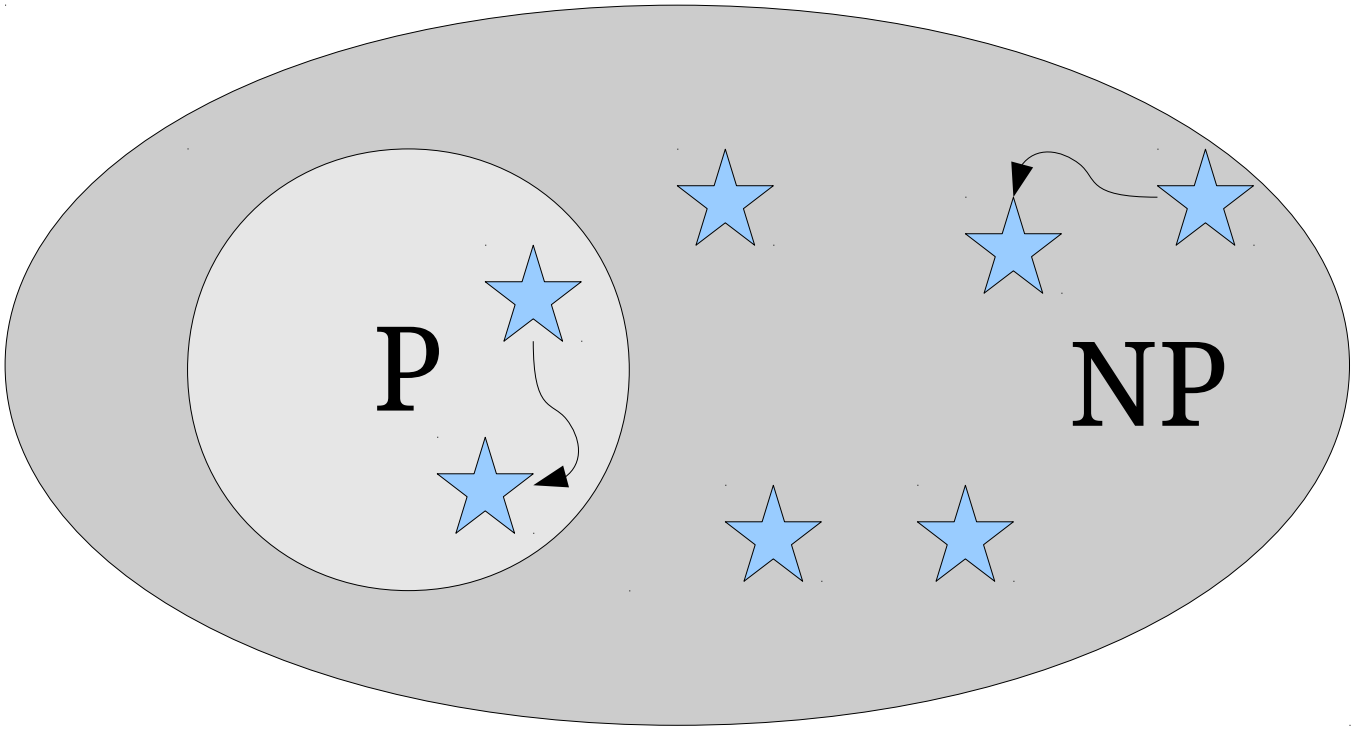
Polynomial-Time Reductions

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



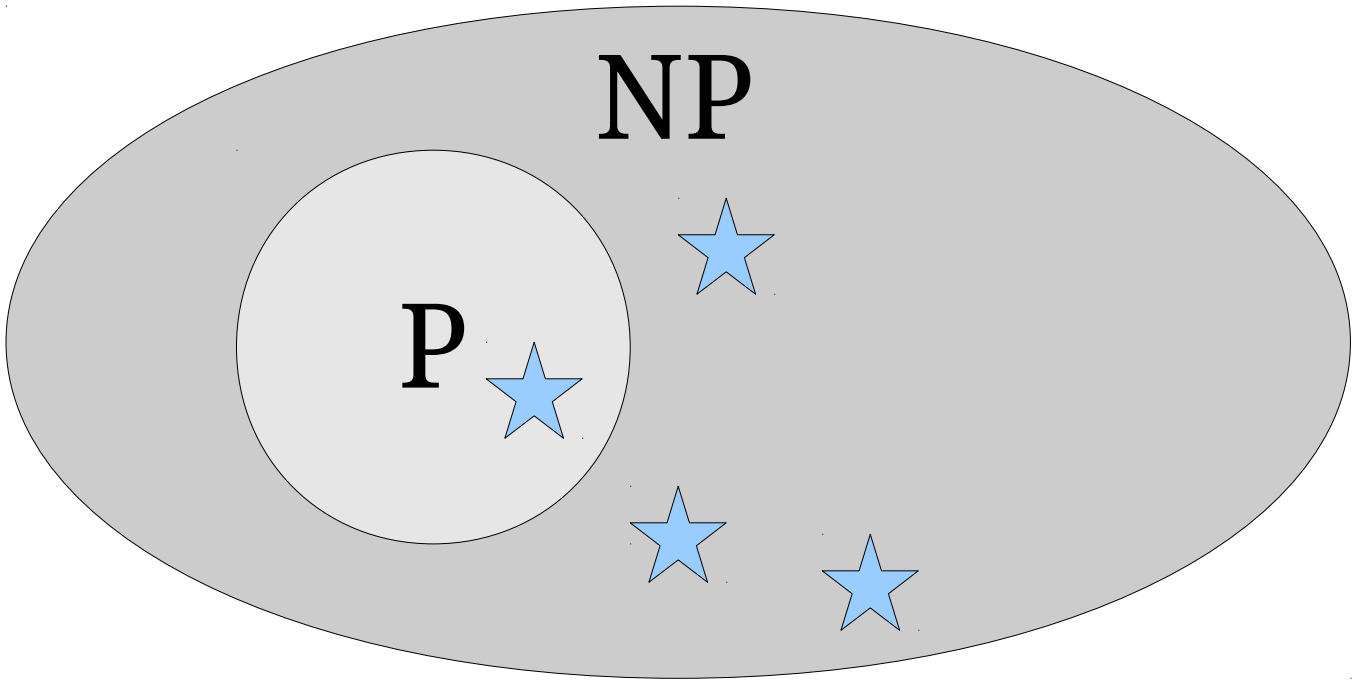
Polynomial-Time Reductions

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



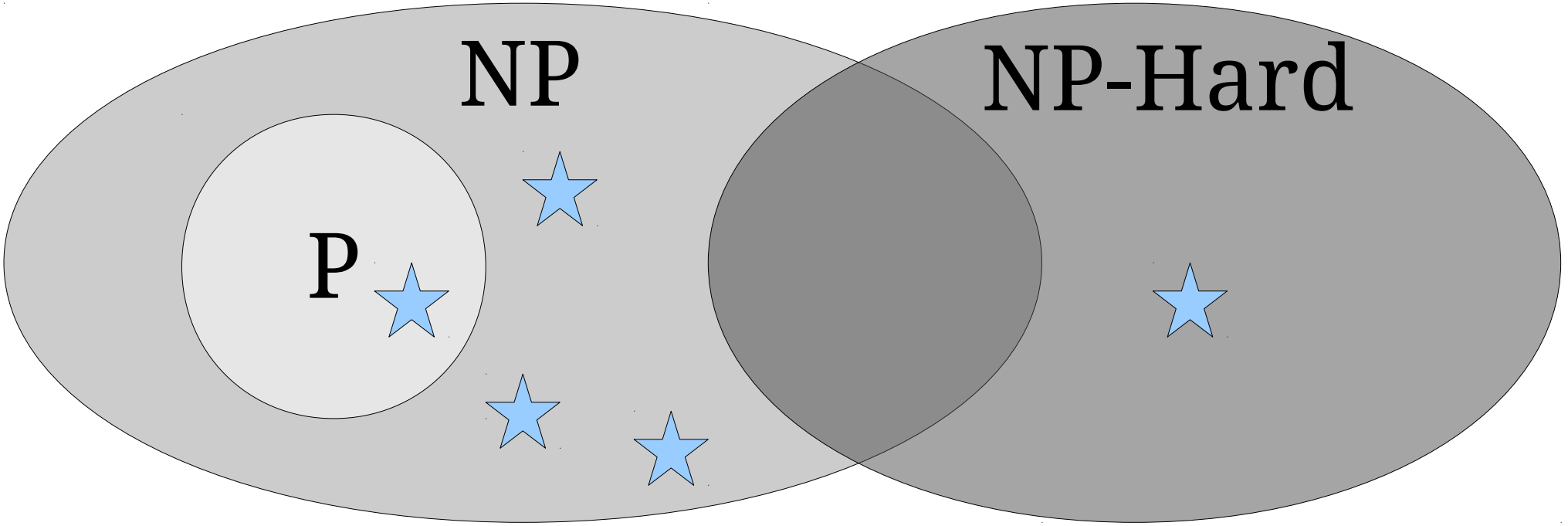
NP-Hardness

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



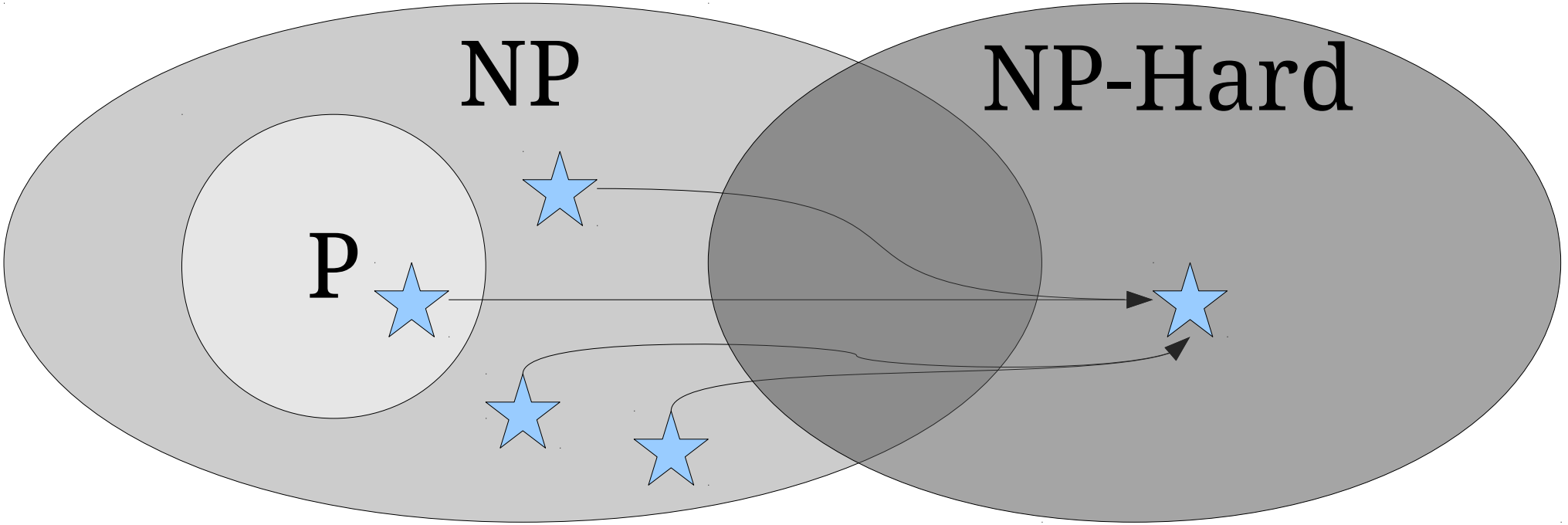
NP-Hardness

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



NP-Hardness

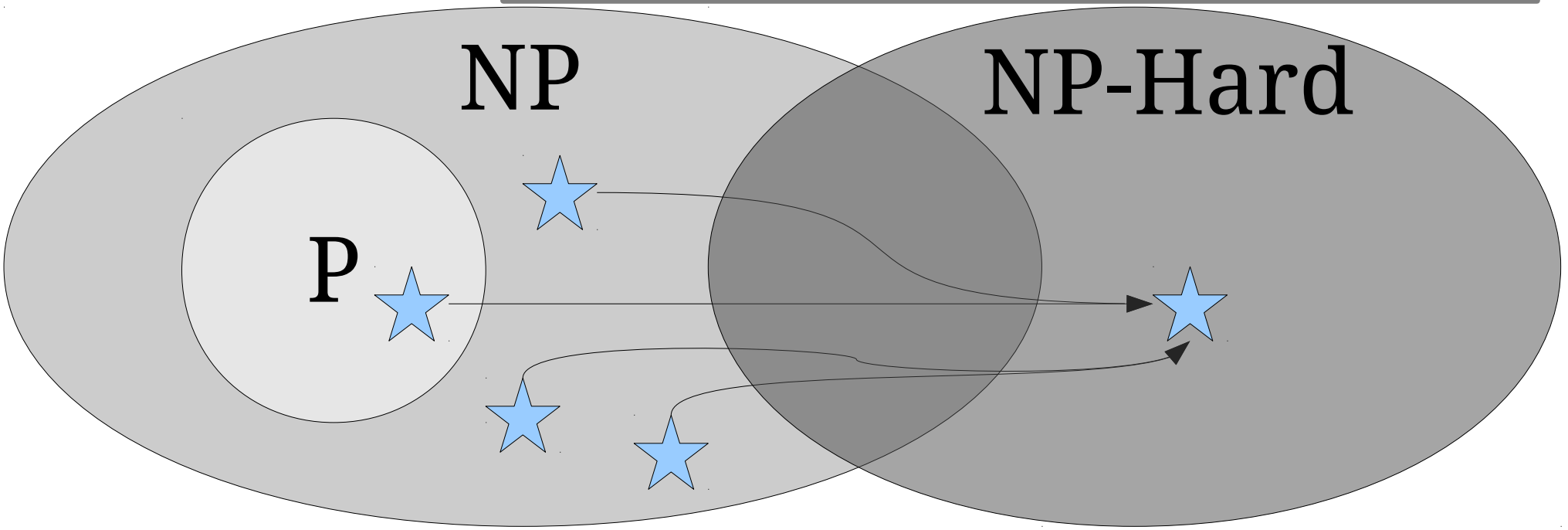
- A language L is called ***NP-hard*** if for *every* $A \in \mathbf{NP}$, we have $A \leq_{\mathrm{P}} L$.



NP-Hardness

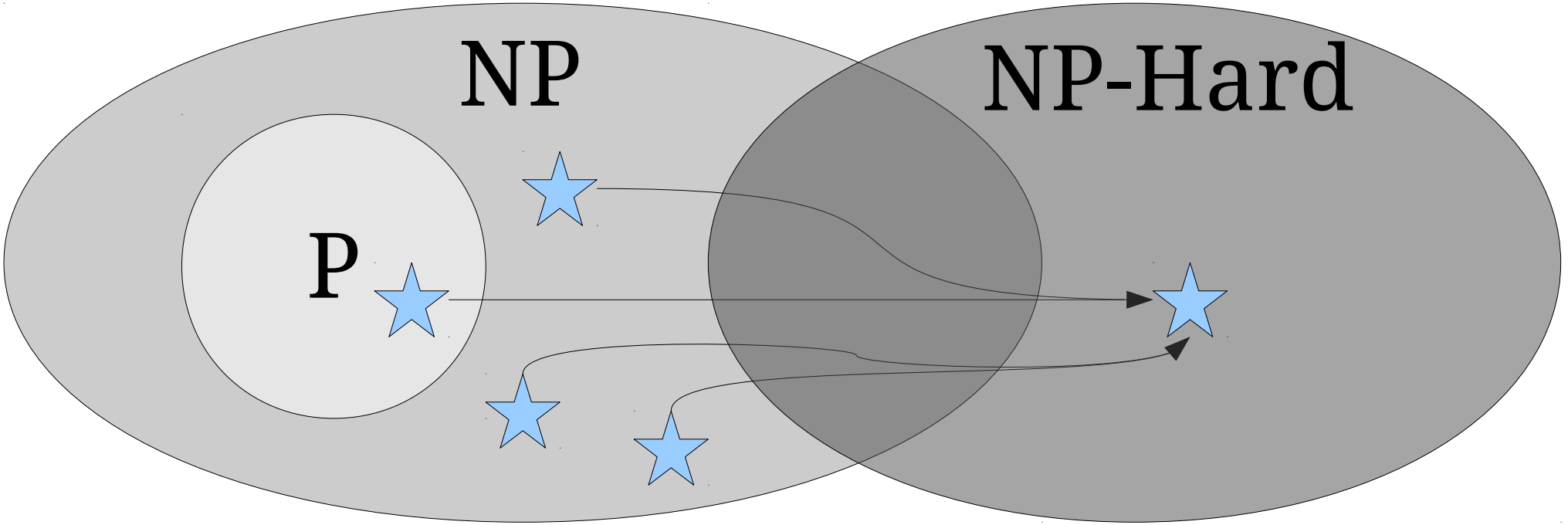
- A language L is called ***NP-hard*** if for *every* $A \in \mathbf{NP}$, we have $A \leq_P L$.

Intuitively: L has to be at least as hard as every problem in **NP**, since an algorithm for L can be used to decide all problems in **NP**.



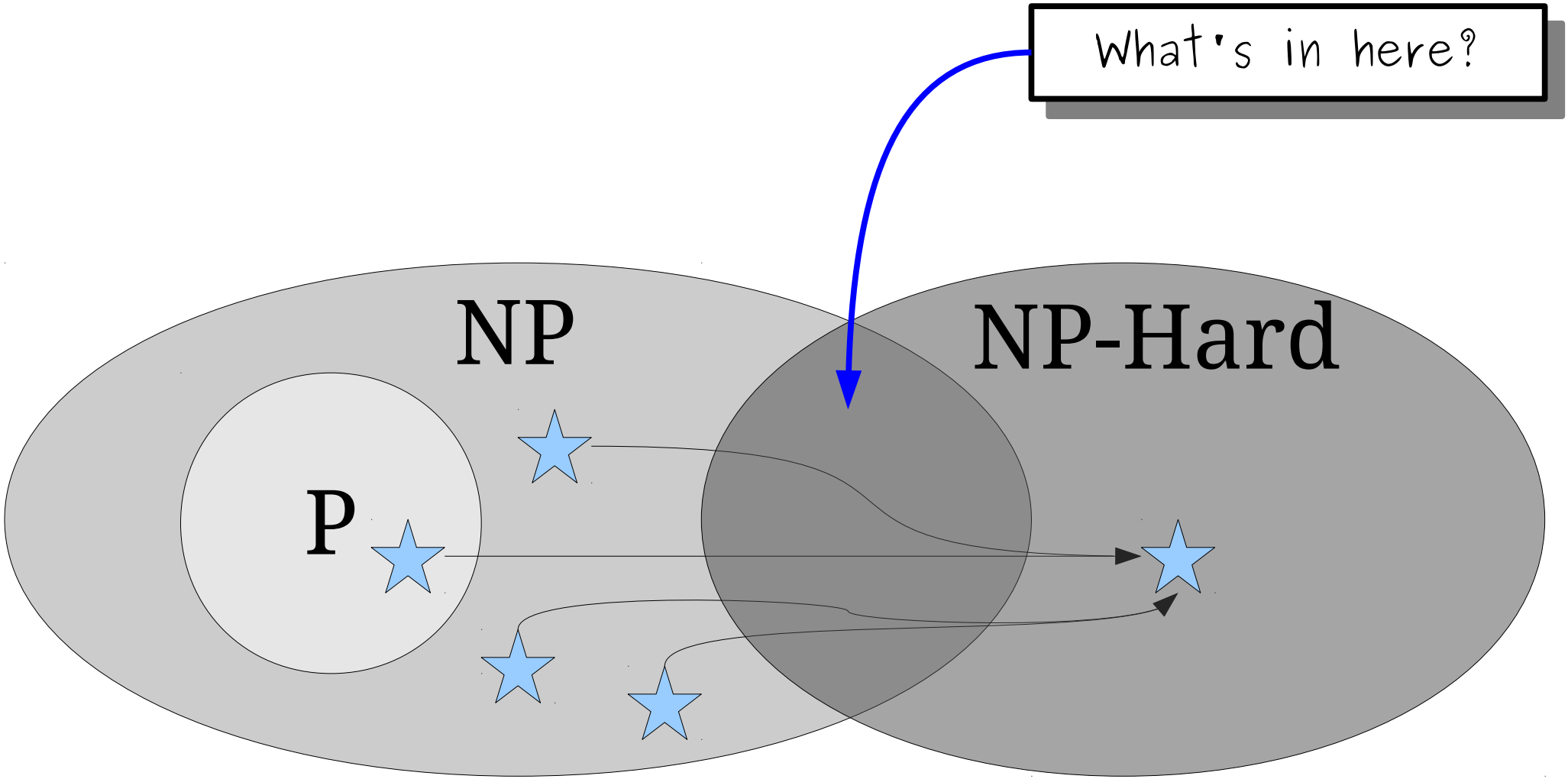
NP-Hardness

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



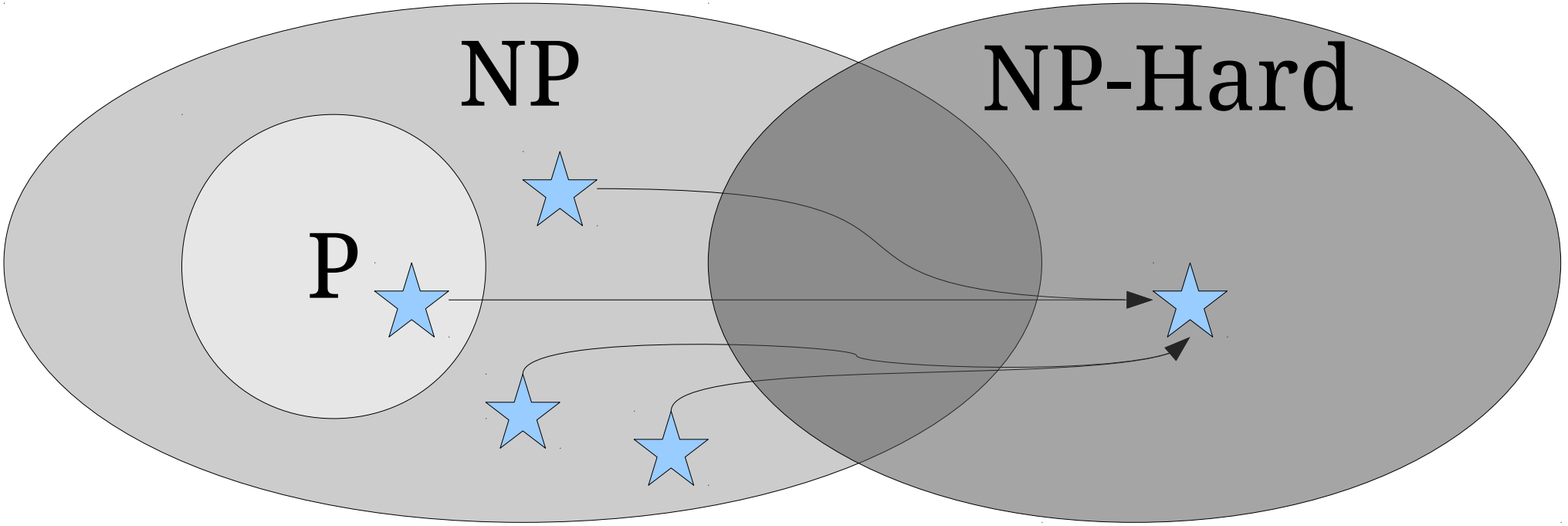
NP-Hardness

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



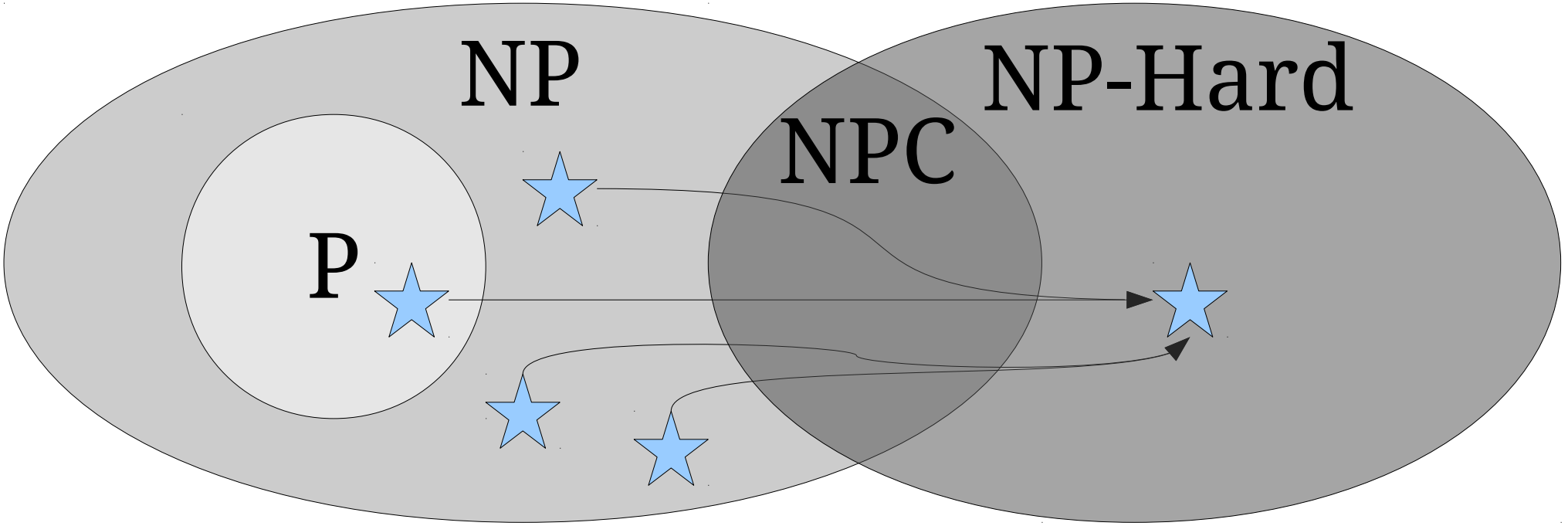
NP-Hardness

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



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.

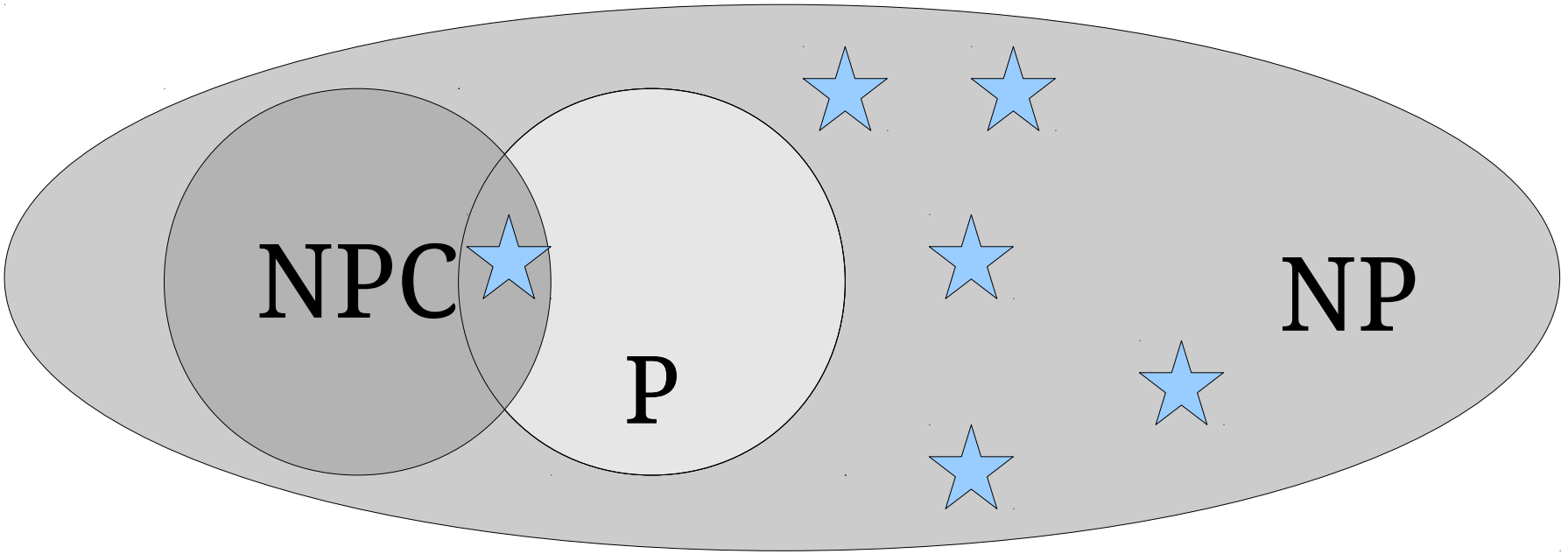


The Tantalizing Truth

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

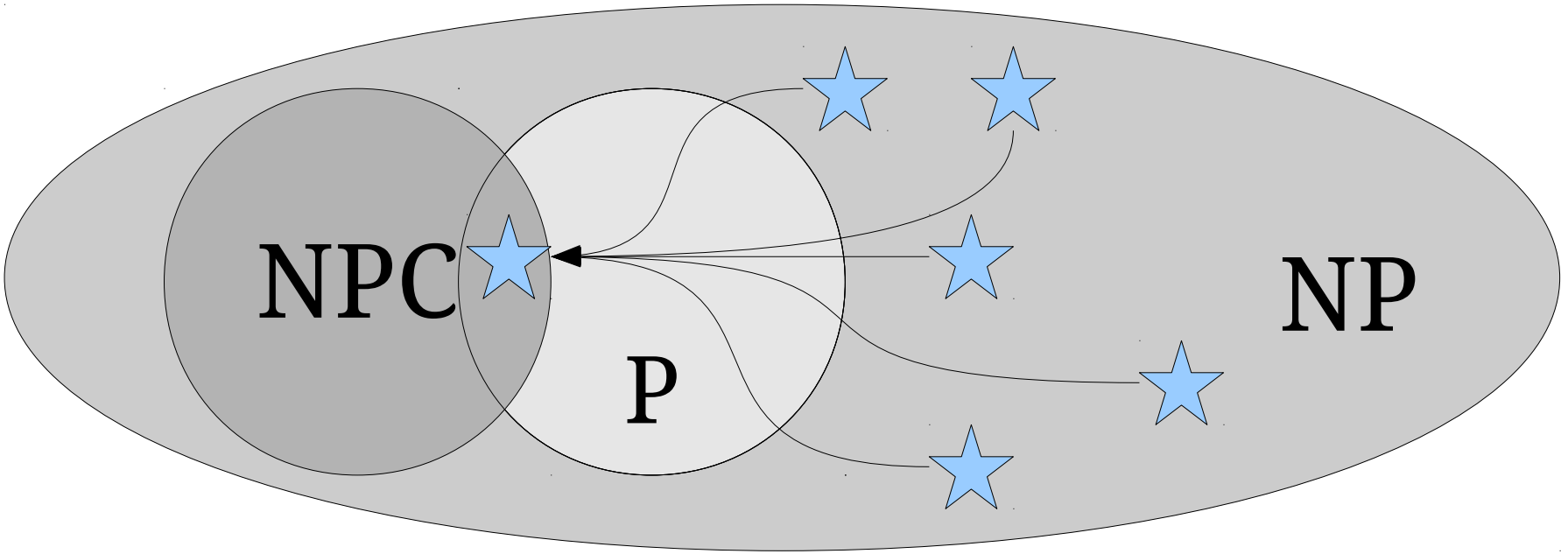
The Tantalizing Truth

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



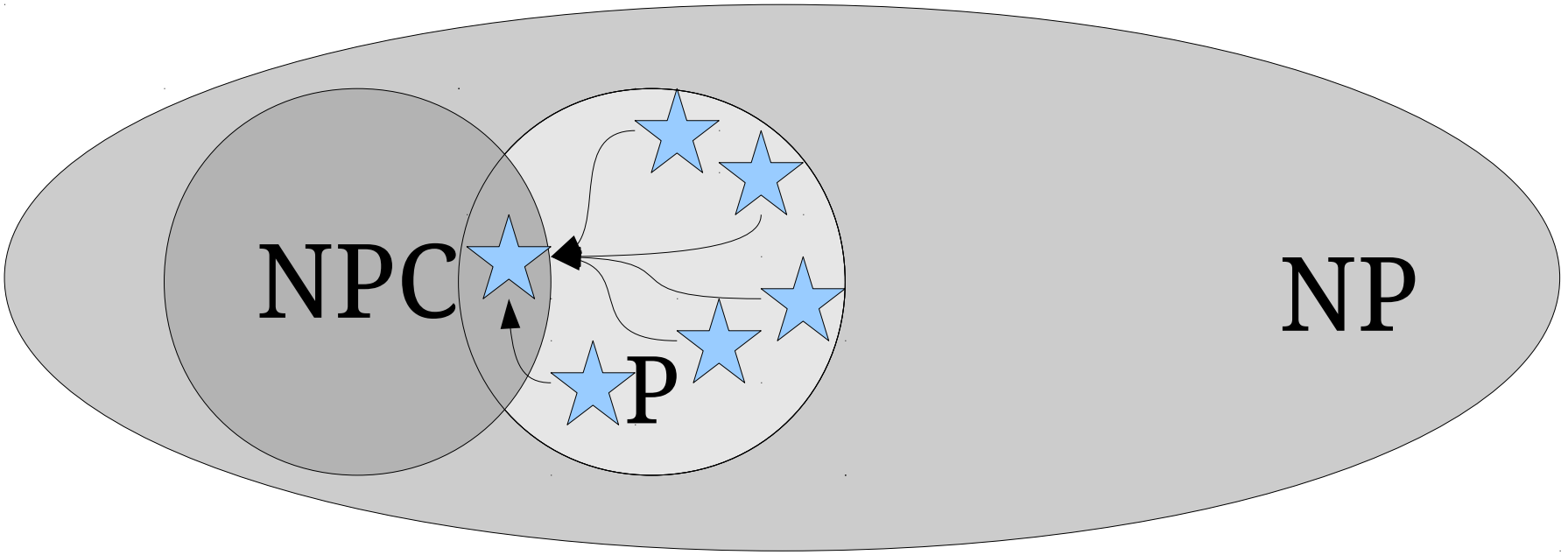
The Tantalizing Truth

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



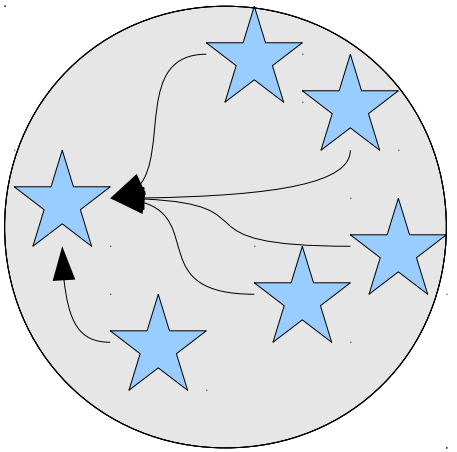
The Tantalizing Truth

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**.

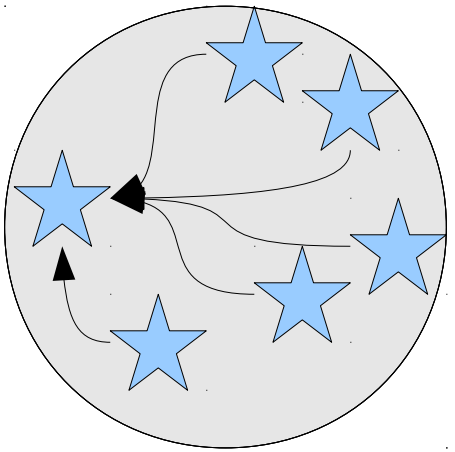


P = NP

The Tantalizing Truth

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

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

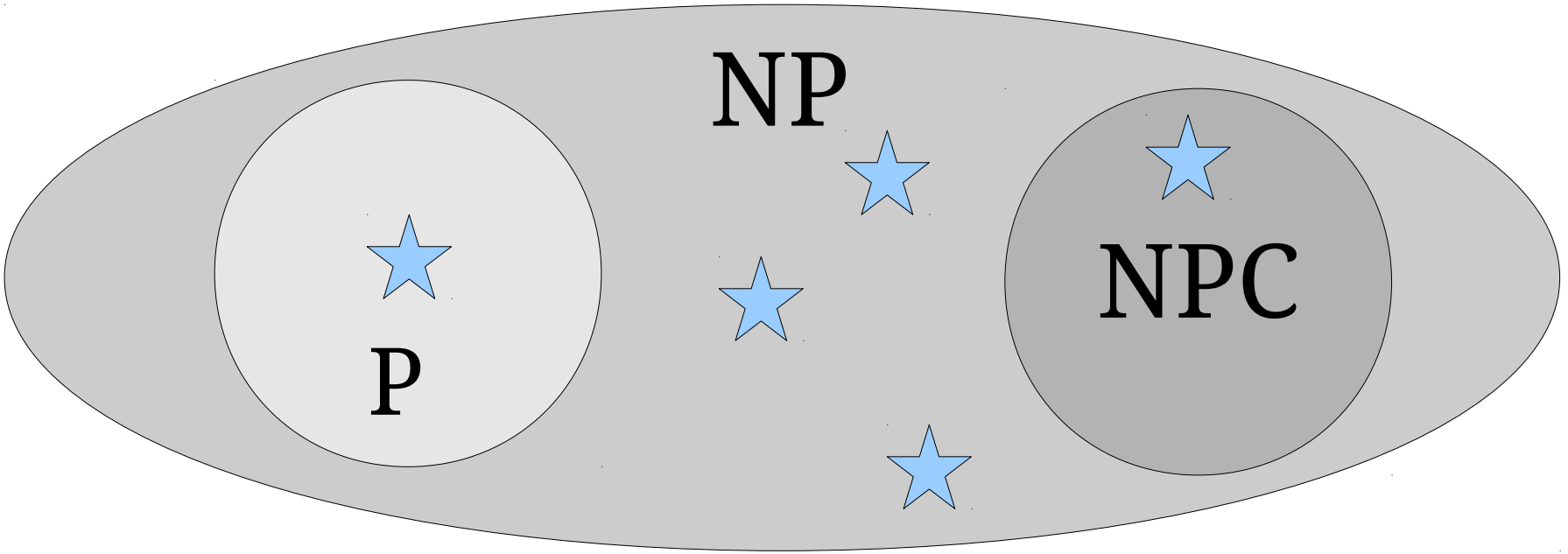


P = NP

The Tantalizing Truth

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

Proof: Suppose that L is an **NP**-complete language not in **P**. Since L is **NP**-complete, we know that $L \in$ **NP**. Therefore, we know that $L \in$ **NP** and $L \notin$ **P**, so **P** ≠ **NP**. ■



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

Proof: Read Sipser or take CS154!

Why All This Matters

- Resolving $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ is equivalent to just figuring out how hard SAT is.
  - If SAT $\in \mathbf{P}$, then $\mathbf{P} = \mathbf{NP}$.
  - If SAT $\notin \mathbf{P}$, then $\mathbf{P} \neq \mathbf{NP}$.
- We've turned a huge, abstract, theoretical problem about solving problems versus checking solutions into the concrete task of seeing how hard one problem is.
- You can get a sense for how little we know about algorithms and computation given that we can't yet answer this question!

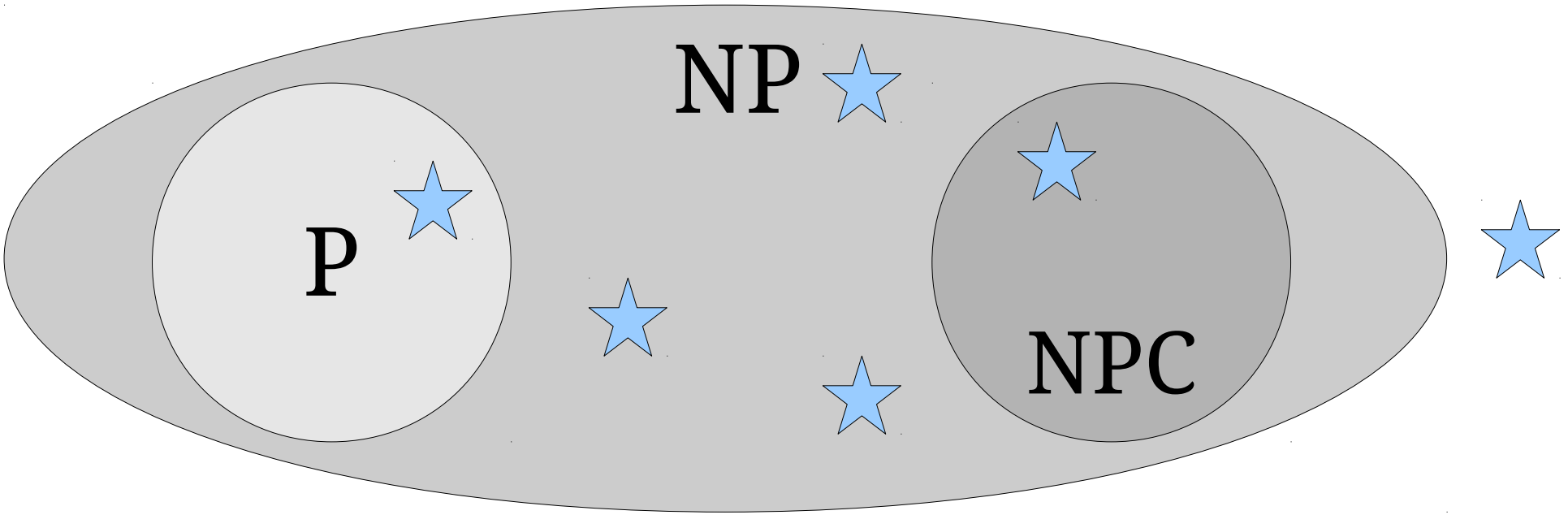
Finding Additional **NP**-Complete Problems

NP-Completeness

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

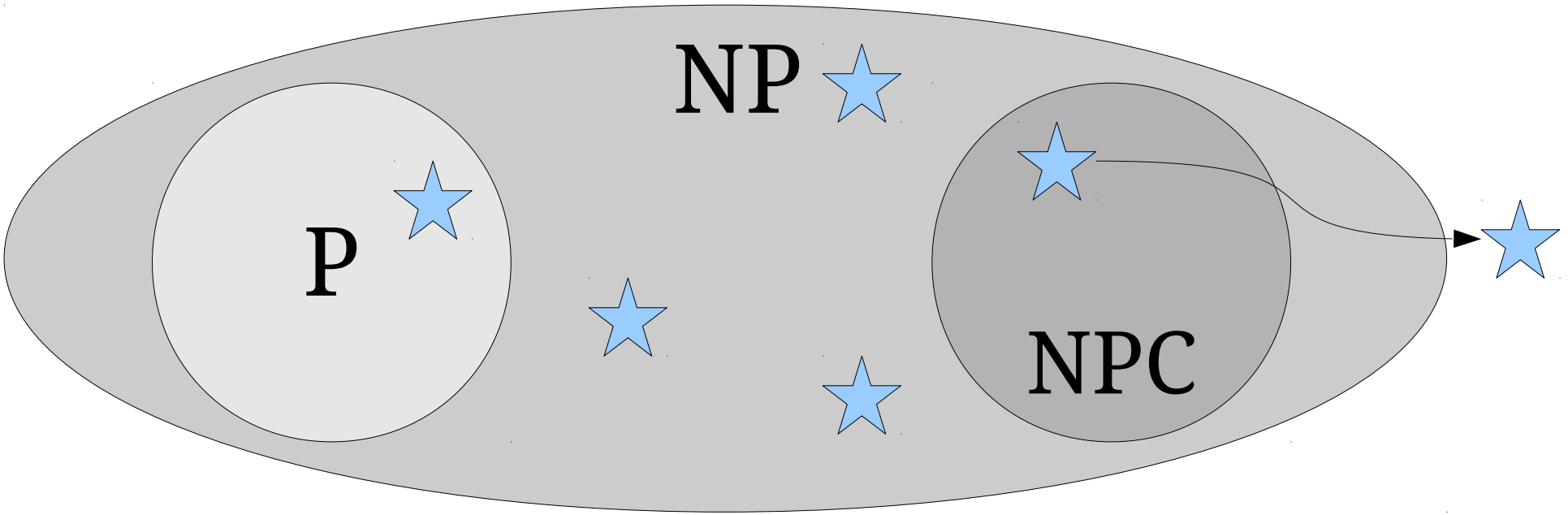
NP-Completeness

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



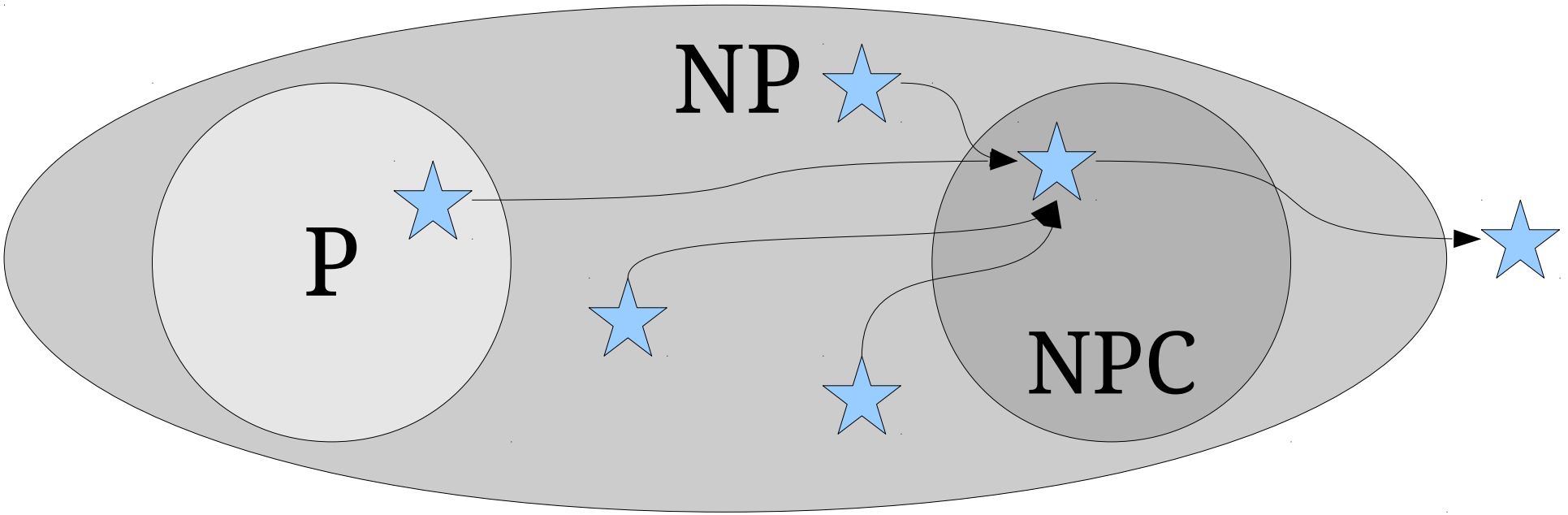
NP-Completeness

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



NP-Completeness

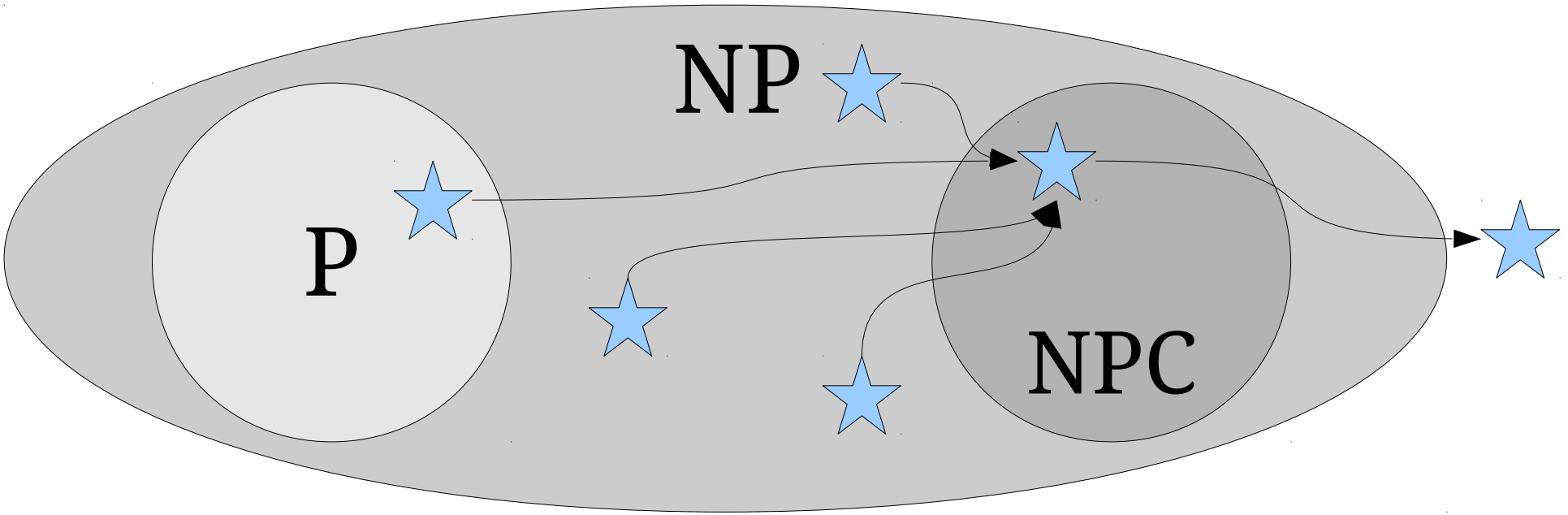
Theorem: Let A and B be languages. If $A \leq_P B$ and A is **NP**-hard, then B is **NP**-hard.



NP-Completeness

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

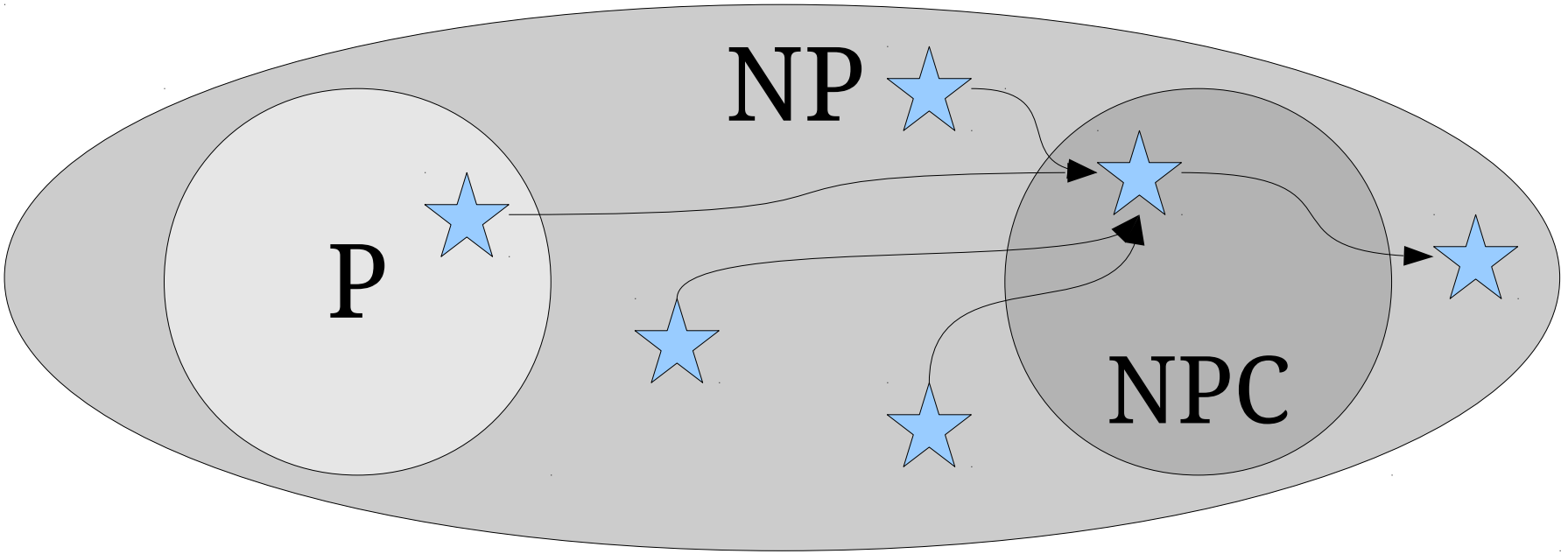
Theorem: Let A and B be languages where $A \in \mathbf{NPC}$ and $B \in \mathbf{NP}$. If $A \leq_P B$, then $B \in \mathbf{NPC}$.



NP-Completeness

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

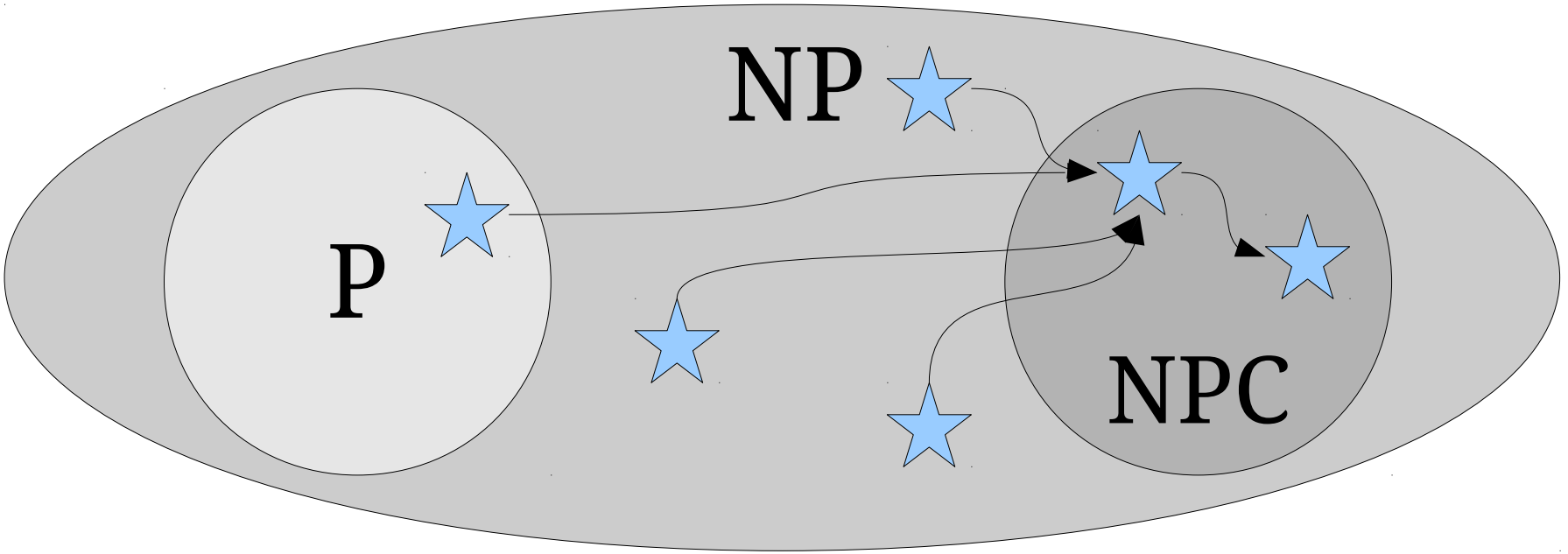
Theorem: Let A and B be languages where $A \in \mathbf{NPC}$ and $B \in \mathbf{NP}$. If $A \leq_P B$, then $B \in \mathbf{NPC}$.



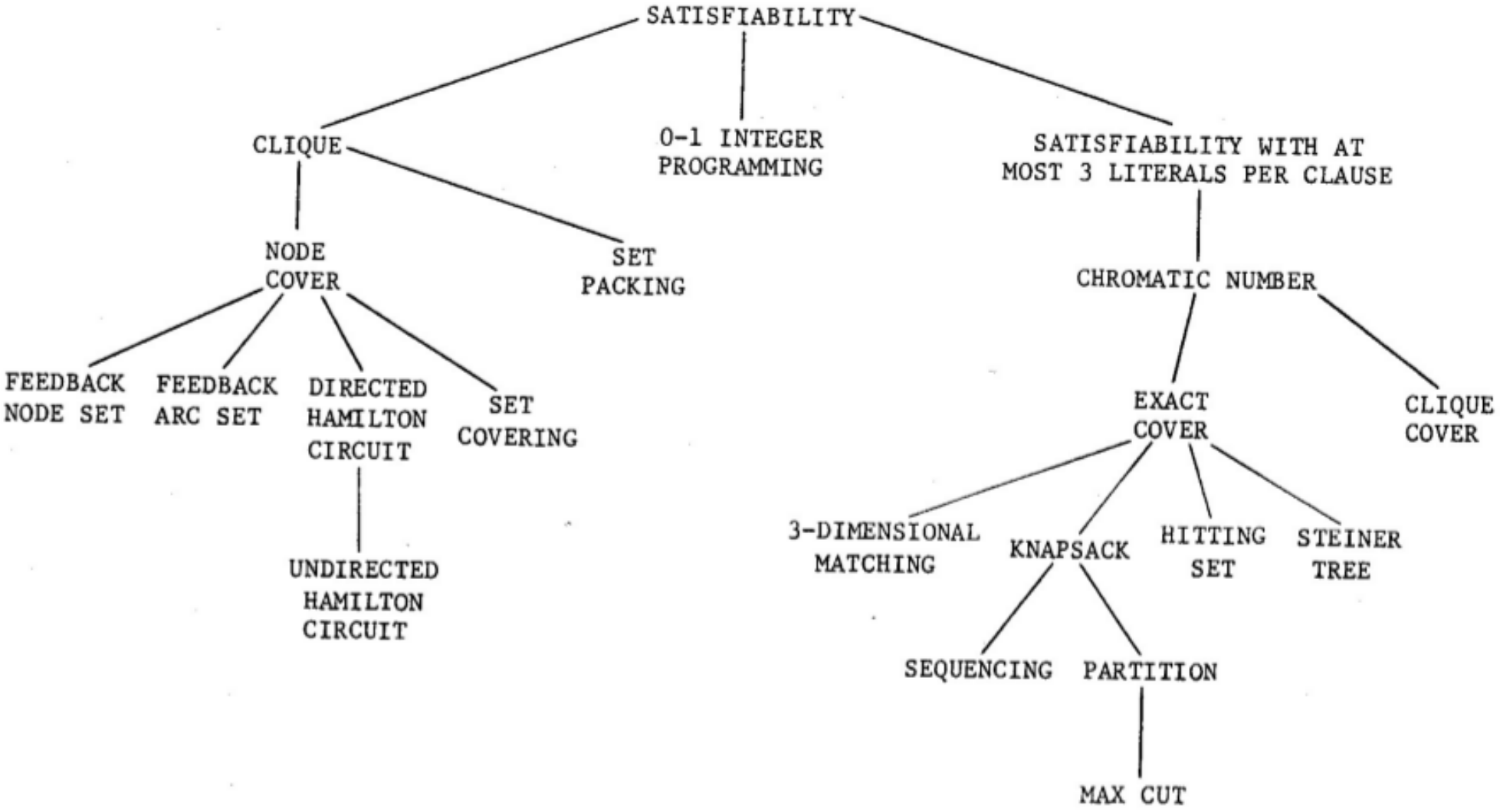
NP-Completeness

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

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



Karp's 21 **NP**-Complete Problems



Sample **NP**-Complete Problems

- Given a graph, is that graph 3-colorable?
- Given a graph, does that graph contain a Hamiltonian path?
- Given a set of cities and a set of substation locations, can you provide power to all the cities using at most k substations?
- Given a set of jobs and workers who can perform those tasks in parallel, can you complete all the jobs in at most T units of time?
- Given a set of numbers S, can you split S into two disjoint sets with the same sum?
- *And many, many more!*

A Feel for **NP**-Completeness

- There are **NP**-complete problems in
  - formal logic (SAT),
  - graph theory (3-colorability),
  - operations research (job scheduling),
  - number theory (partition problem),
  - *and basically everywhere*.
- *You will undoubtedly encounter **NP**-complete problems in the real world.*