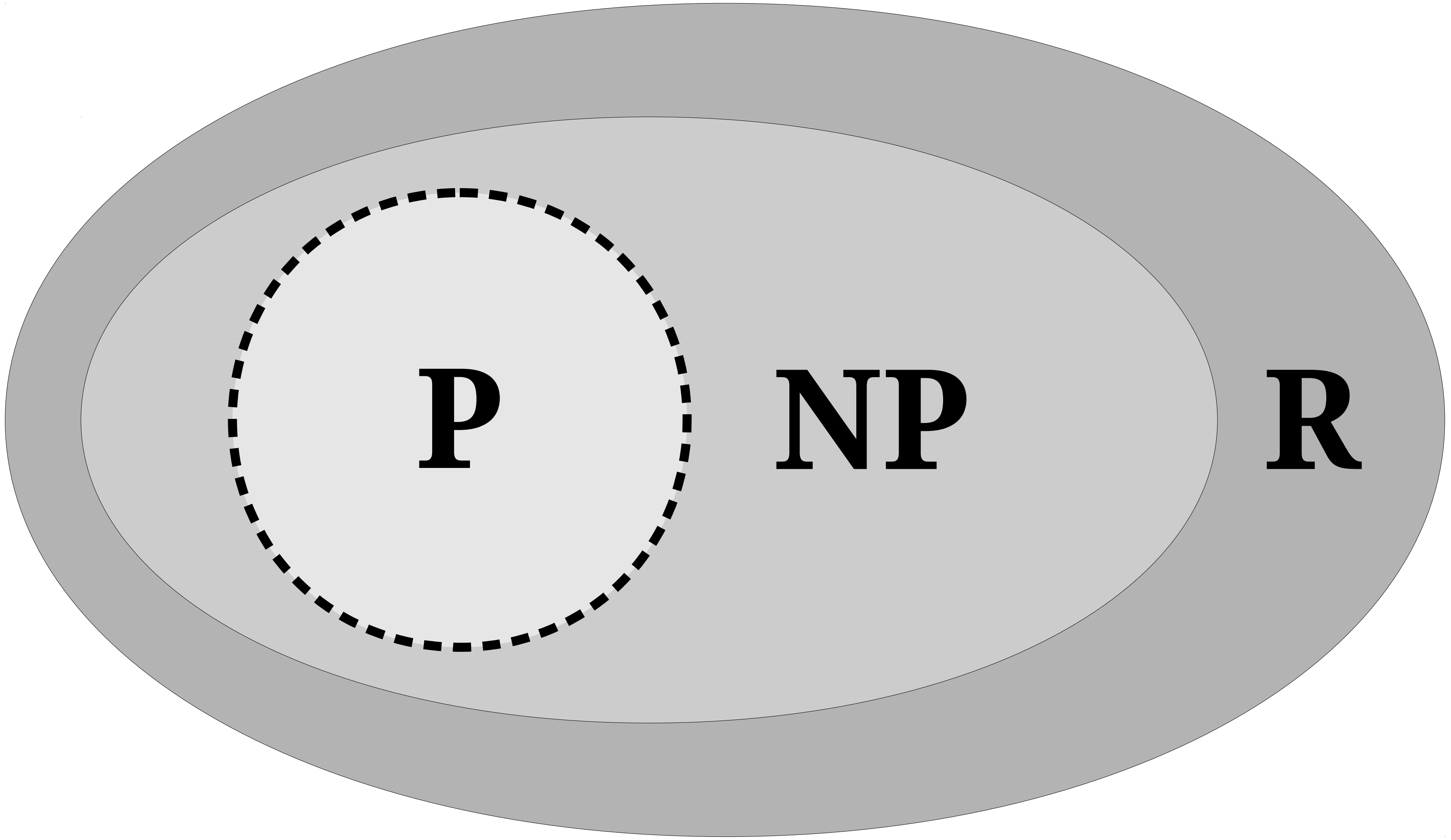
NP-Completeness

Part One

Recap from Last Time

The Limits of Efficient Computation



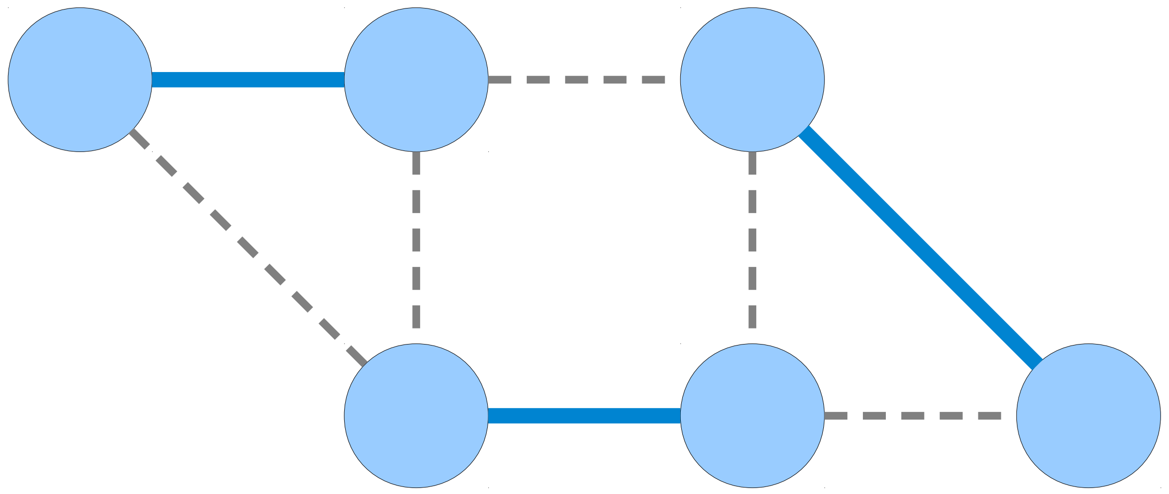
P and **NP** Refresher

- The class **P** consists of all problems *decidable* in polynomial time.
- The class **NP** consists of all problems *verifiable* in polynomial time.
- We don't know whether **P = NP** or not, and unfortunately a theorem by Baker, Gill, and Soloway shows that the approaches we've used with **R** and **RE** won't work here.
- So... what can we do?

Reducibility

Maximum Matching

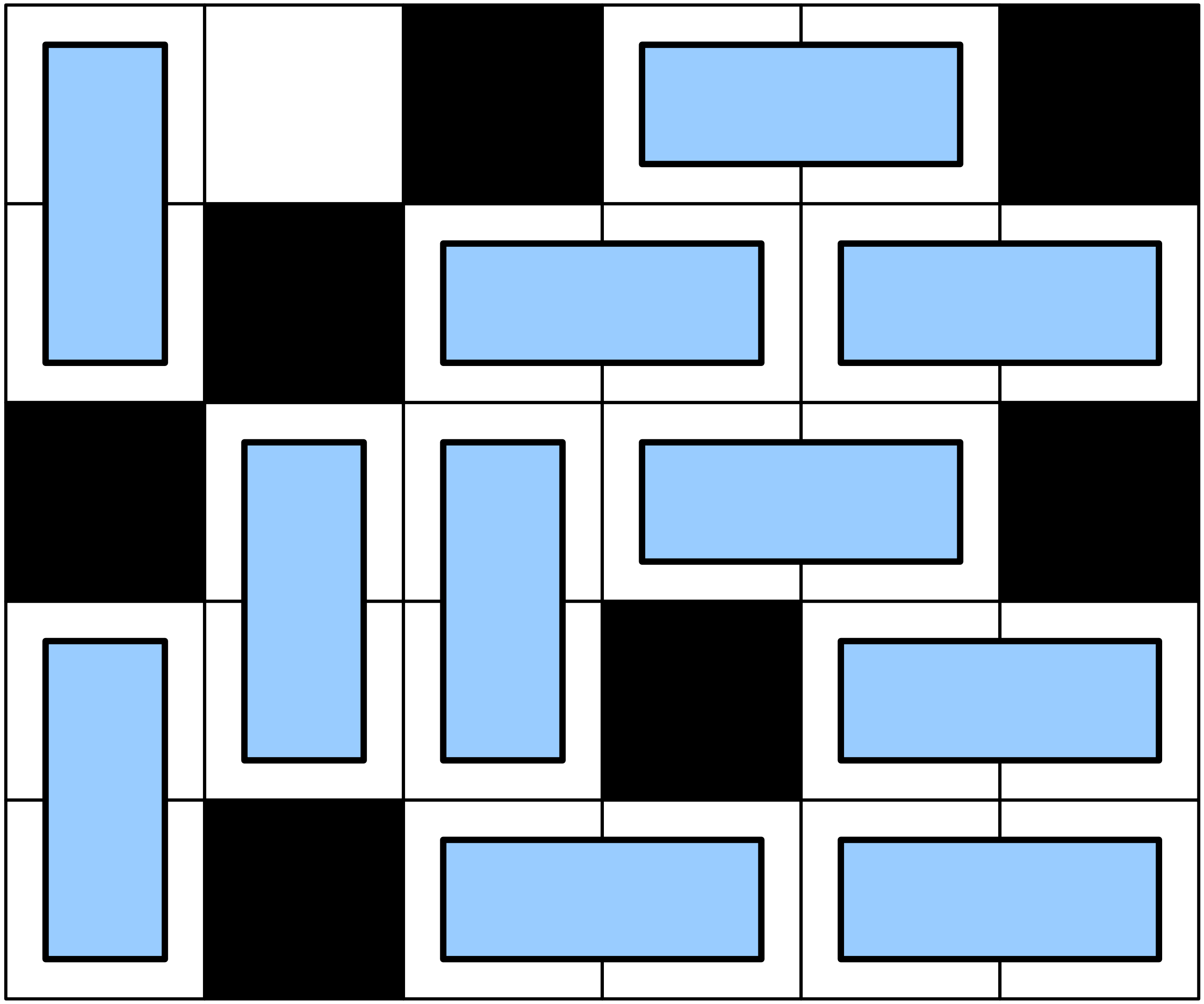
- Given an undirected graph *G*, a ***matching*** in *G* is a set of edges such that no two edges share an endpoint.
- A ***maximum matching*** is a matching with the largest number of edges.



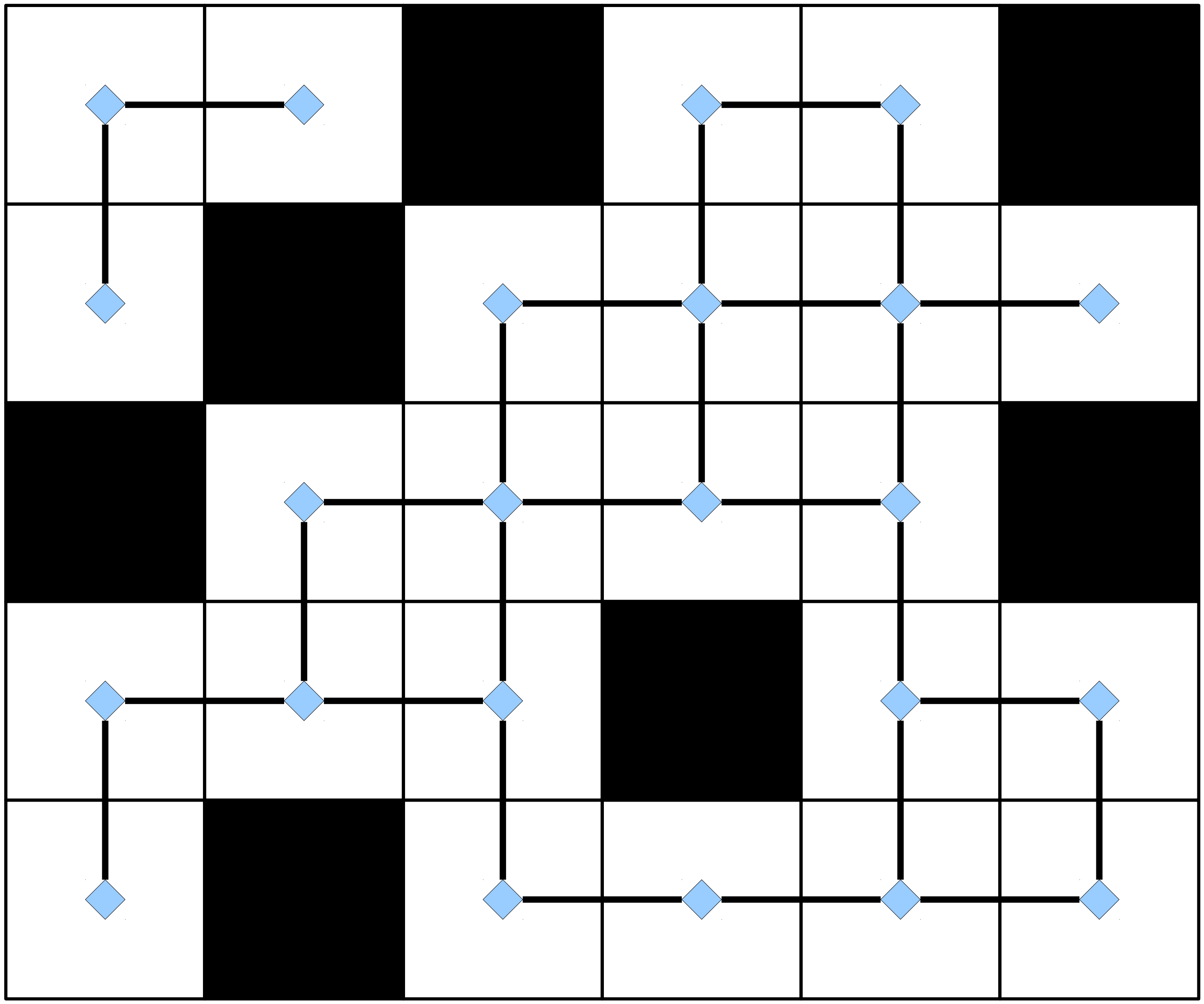
Maximum Matching

- 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?

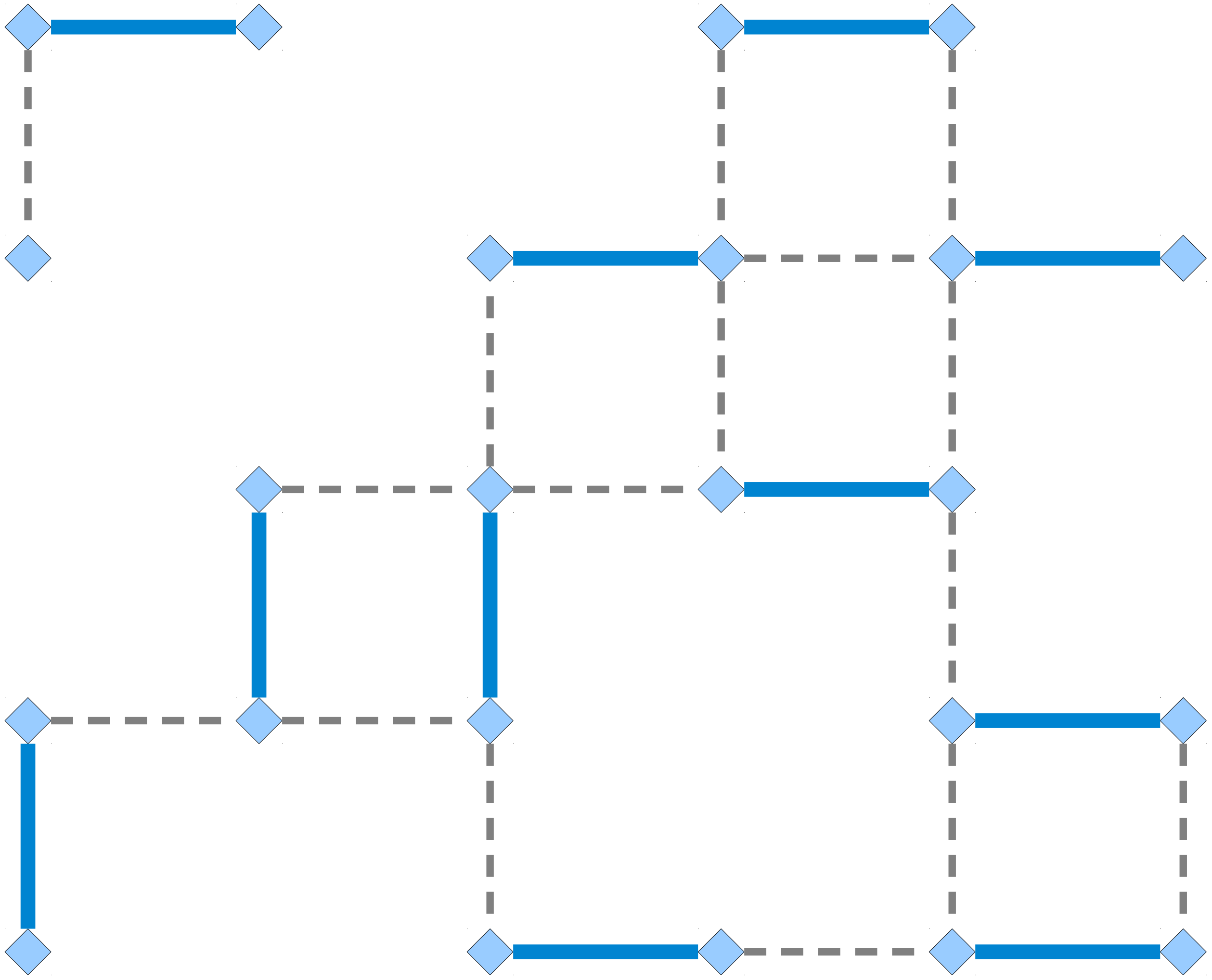
Domino Tiling



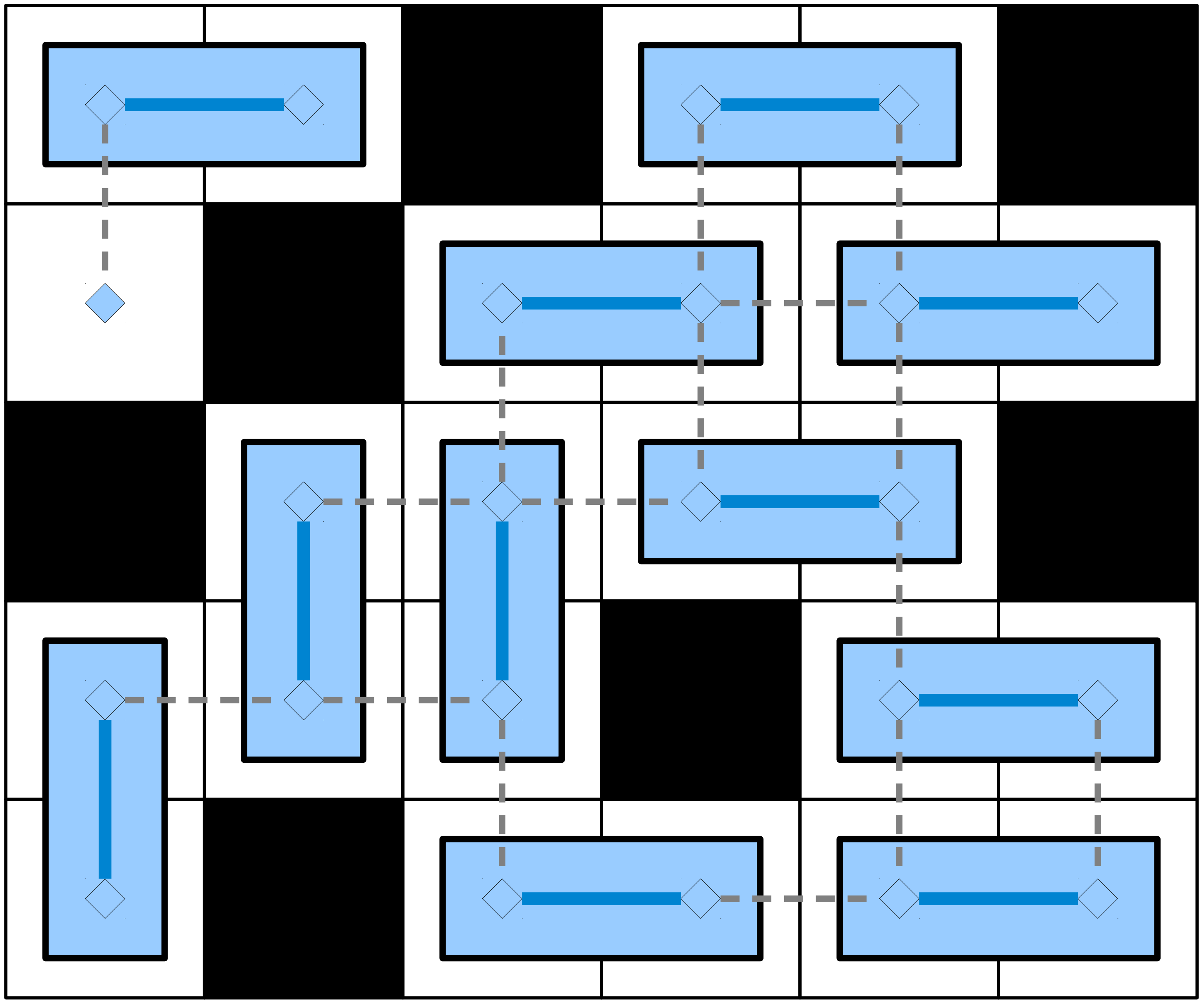
Solving Domino Tiling



Solving Domino Tiling



Solving Domino Tiling



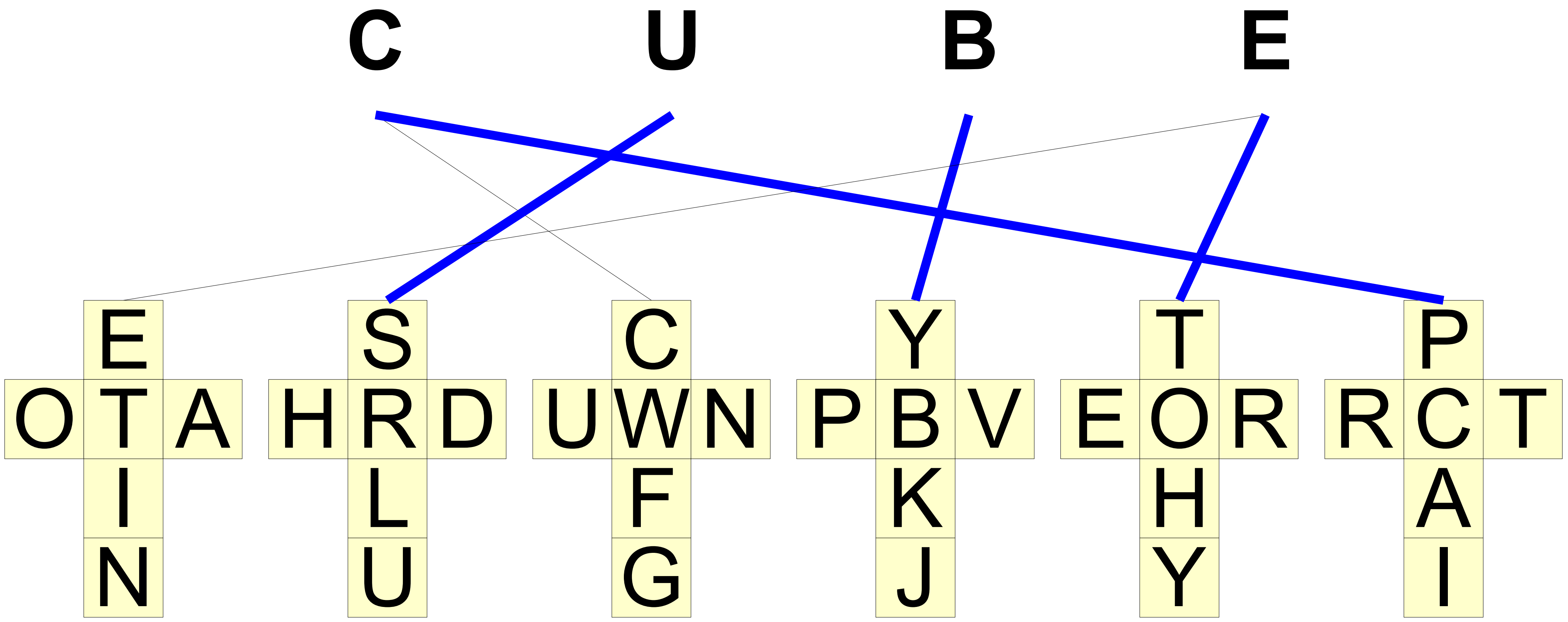
In Pseudocode

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

Boggle Spelling

Given a set of Boggle cubes and a target word or phrase, is it possible to arrange the cubes so that you spell out that word or phrase?

Boggle Spelling



In Pseudocode

```
boolean canSpell(List<Cube> cubes,
                 String word) {

   return hasMatching(makeGraph(cubes, word),
                      word.length());

}
```

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:

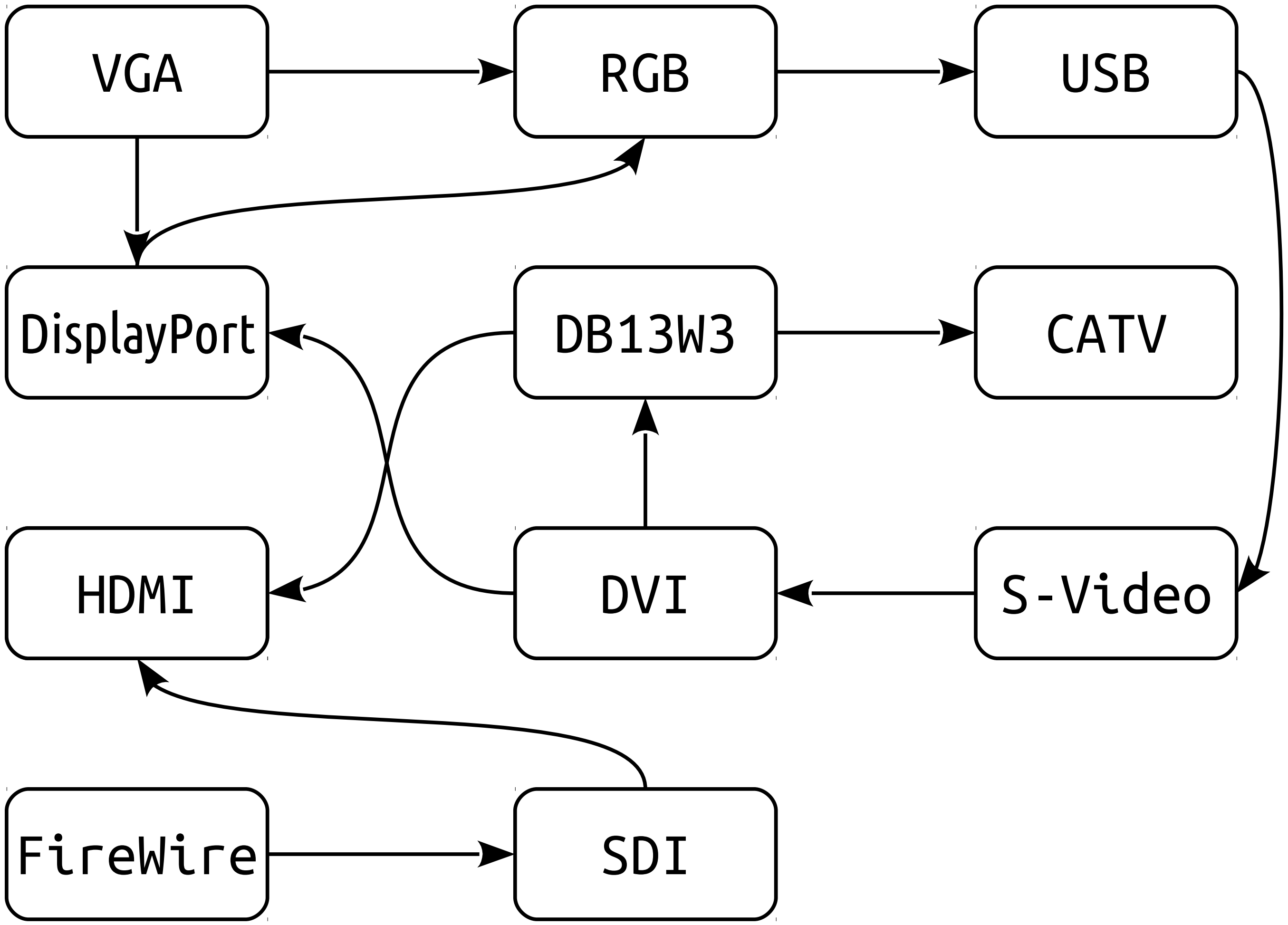
  ***REACHABILITY* =**
  { ⟨*G*, *s*, *t*⟩ | *G* is a directed graph, *s* and *t* are nodes in *G*, and there's a path from *s* to *t* }

- ***Theorem:*** *REACHABILITY* ∈ **P**.
- Given that we can solve the reachability problem in polynomial time, what other problems can we solve in polynomial time?

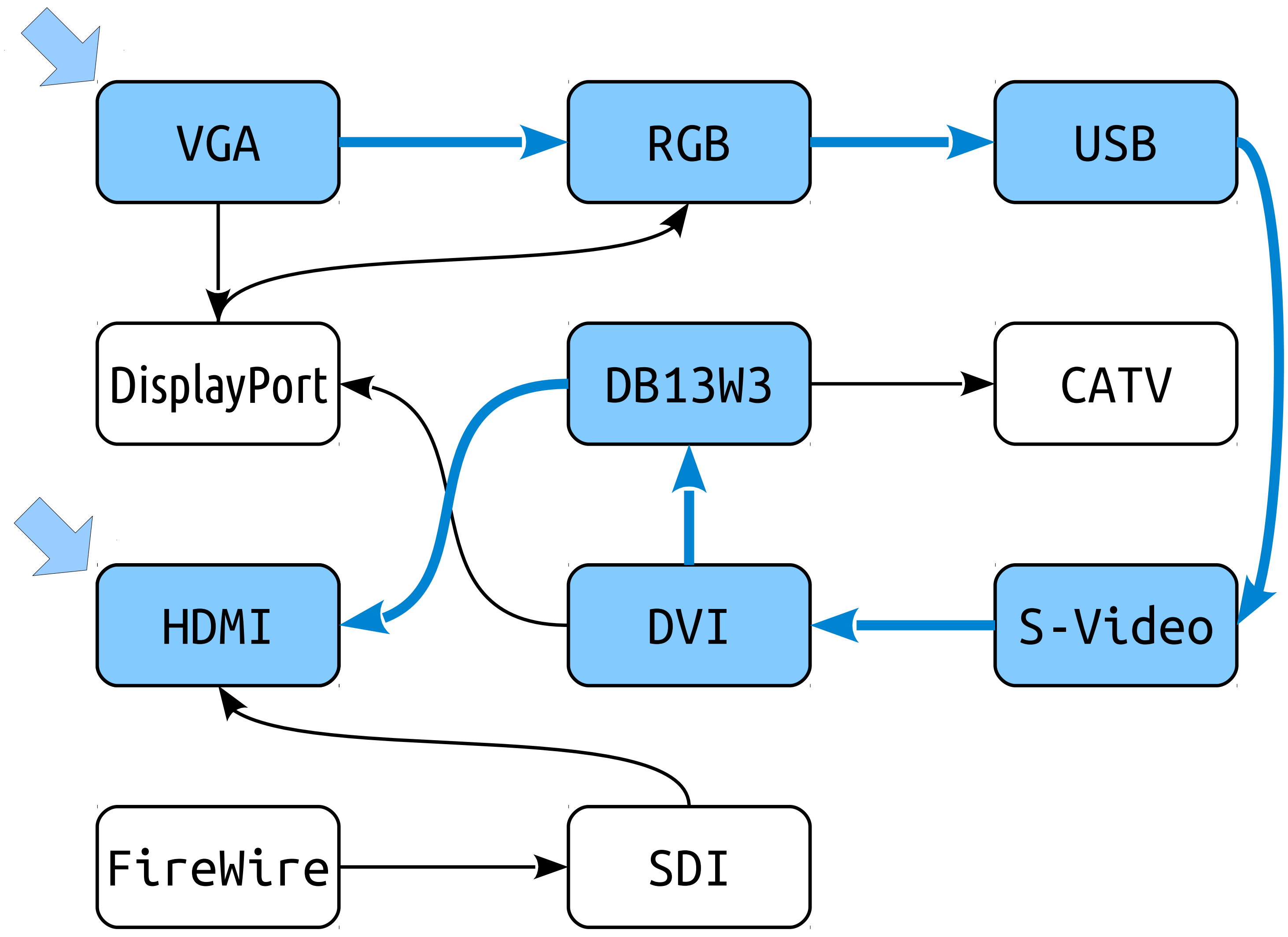
Converter Conundrums

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



In Pseudocode

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

One More!

Satisfiability

- A propositional logic formula φ is called ***satisfiable*** if there is some assignment to its variables that makes it evaluate to true.
  - $p \land q$ is satisfiable.
  - $p \land \neg p$ is unsatisfiable.
  - $p \to (q \land \neg q)$ is satisfiable.
- An assignment of true and false to the variables of φ that makes it evaluate to true is called a ***satisfying assignment***.

SAT

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

Given a propositional logic formula φ, is φ satisfiable?

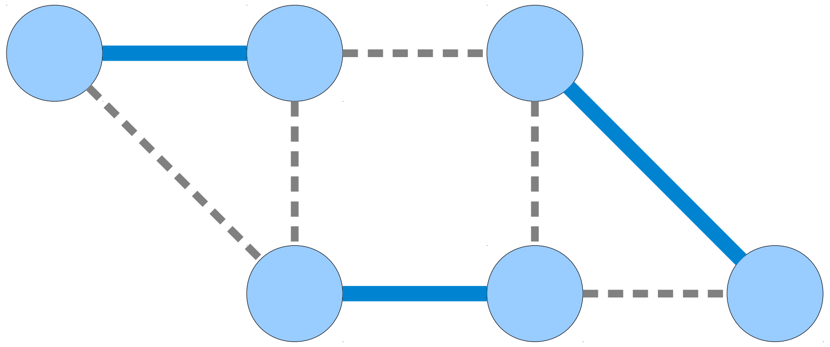
- Formally:

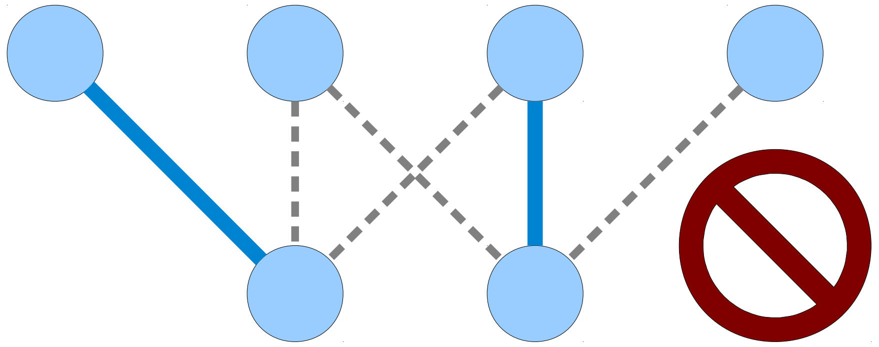
SAT **= { ⟨φ⟩ | φ is a satisfiable PL formula }**

- ***Claim:*** *SAT* ∈ **NP**.
  - (Do you see why?)

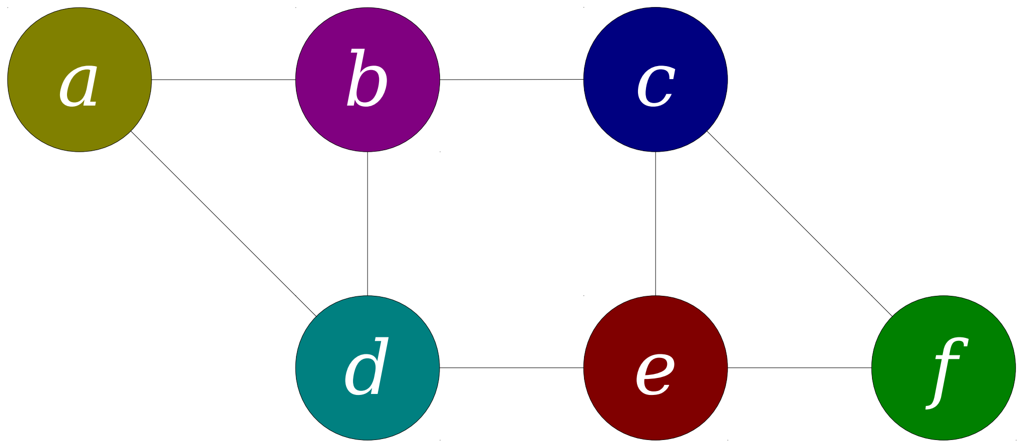
Perfect Matchings

- A ***perfect matching*** in a graph G is a matching where every node is adjacent to some edge in the matching.



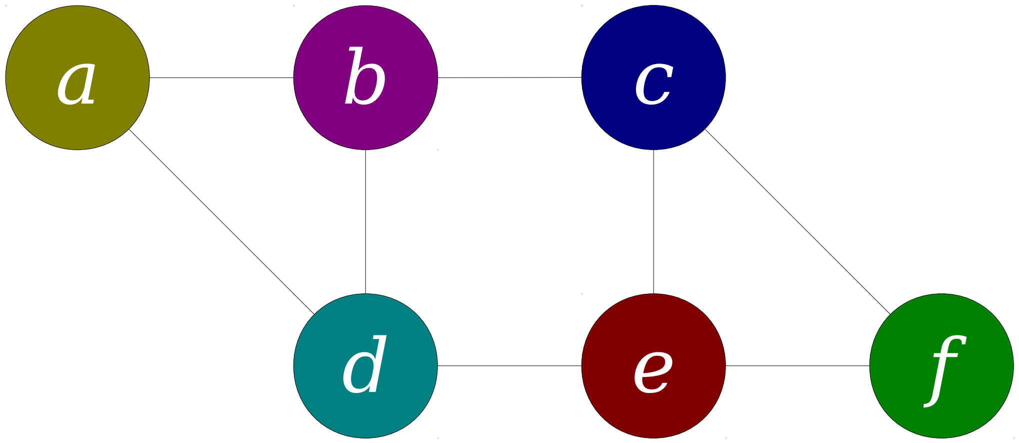


- ***Claim:*** We can reduce the problem of determining whether a graph has a perfect matching to the boolean satisfiability problem.



For each edge $\{u, v\}$, we'll introduce a propositional variable p_{uv} meaning “edge $\{u, v\}$ is included in the perfect matching.”

- ☐ p_{ab}
- ☐ p_{ad}
- ☐ p_{bc}
- ☐ p_{bd}
- ☐ p_{ce}
- ☐ p_{cf}
- ☐ p_{de}
- ☐ p_{ef}



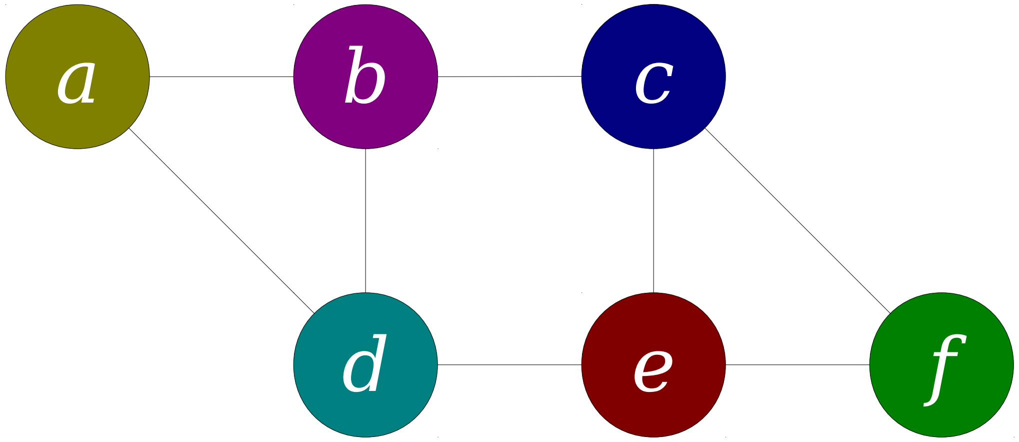
For each edge $\{u, v\}$, we'll introduce a propositional variable p_{uv} meaning "edge $\{u, v\}$ is included in the perfect matching."

We need to enforce the following:

- Every node is adjacent to at least one edge. ("perfect")

$$(p_{ab} \lor p_{ad}) \land (p_{ab} \lor p_{bc} \lor p_{bd}) \land (p_{bc} \lor p_{ce} \lor p_{cf}) \land$$
$$(p_{ad} \lor p_{bd} \lor p_{de}) \land (p_{ce} \lor p_{de} \lor p_{ef}) \land (p_{cf} \lor p_{ef})$$

☐ p_{ab}
☐ p_{ad}
☐ p_{bc}
☐ p_{bd}
☐ p_{ce}
☐ p_{cf}
☐ p_{de}
☐ p_{ef}



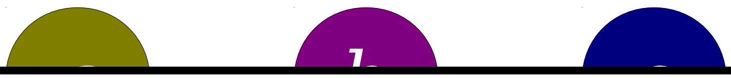
For each edge $\{u, v\}$, we'll introduce a propositional variable p_{uv} meaning “edge $\{u, v\}$ is included in the perfect matching.”

We need to enforce the following:

- Every node is adjacent to at least one edge. (“perfect”)
- Every node is adjacent to at most one edge. (“matching”)

$$
\begin{gathered}
(p_{ab} \lor p_{ad}) \land (p_{ab} \lor p_{bc} \lor p_{bd}) \land (p_{bc} \lor p_{ce} \lor p_{cf}) \land \\
(p_{ad} \lor p_{bd} \lor p_{de}) \land (p_{ce} \lor p_{de} \lor p_{ef}) \land (p_{cf} \lor p_{ef}) \land \\
\neg(p_{ab} \land p_{ad}) \land \\
\neg(p_{ab} \land p_{bc}) \land \neg(p_{ab} \land p_{bd}) \land \neg(p_{bc} \land p_{bd}) \land \\
\neg(p_{bc} \land p_{ce}) \land \neg(p_{bc} \land p_{cf}) \land \neg(p_{ce} \land p_{cf}) \land \\
\neg(p_{ad} \land p_{bd}) \land \neg(p_{ad} \land p_{de}) \land \neg(p_{bd} \land p_{de}) \land \\
\neg(p_{ce} \land p_{de}) \land \neg(p_{ce} \land p_{ef}) \land \neg(p_{de} \land p_{ef}) \land \\
\neg(p_{cf} \land p_{ef})
\end{gathered}
$$

- ☐ p_{ab}
- ☐ p_{ad}
- ☐ p_{bc}
- ☐ p_{bd}
- ☐ p_{ce}
- ☐ p_{cf}
- ☐ p_{de}
- ☐ p_{ef}



Notice that this formula is the many-way AND of lots of smaller formulas of the form

$$(l_1 \lor l_2 \lor \ldots \lor l_n)$$

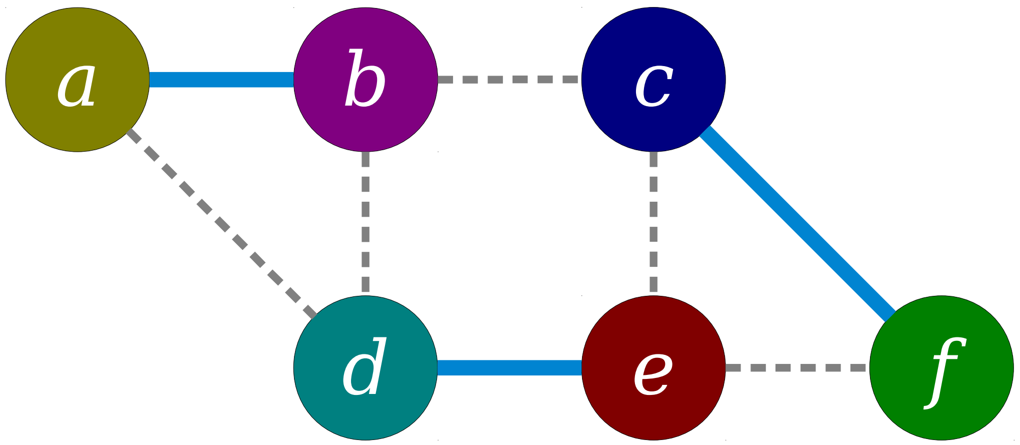
where each l_k is either a variable or its negation. Each l_k is called a ***literal***, and the multi-way OR of literals is called a ***clause***.

To satisfy this formula, we need to choose truth values so that every clause has at least one true literal.

· Every node is adjacent to at most one edge. ("matching")

$$(p_{ab} \lor p_{ad}) \land (p_{ab} \lor p_{bc} \lor p_{bd}) \land (p_{bc} \lor p_{ce} \lor p_{cf}) \land$$
$$(p_{ad} \lor p_{bd} \lor p_{de}) \land (p_{ce} \lor p_{de} \lor p_{ef}) \land (p_{cf} \lor p_{ef}) \land$$
$$(\neg p_{ab} \lor \neg p_{ad}) \land$$
$$(\neg p_{ab} \lor \neg p_{bc}) \land (\neg p_{ab} \lor \neg p_{bd}) \land (\neg p_{bc} \lor \neg p_{bd}) \land$$
$$(\neg p_{bc} \lor \neg p_{ce}) \land (\neg p_{bc} \lor \neg p_{cf}) \land (\neg p_{ce} \lor \neg p_{cf}) \land$$
$$(\neg p_{ad} \lor \neg p_{bd}) \land (\neg p_{ad} \lor \neg p_{de}) \land (\neg p_{bd} \lor \neg p_{de}) \land$$
$$(\neg p_{ce} \lor \neg p_{de}) \land (\neg p_{ce} \lor \neg p_{ef}) \land (\neg p_{de} \lor \neg p_{ef}) \land$$
$$(\neg p_{cf} \lor \neg p_{ef})$$

☐ p_{ab}
☐ p_{ad}
☐ p_{bc}
☐ p_{bd}
☐ p_{ce}
☐ p_{cf}
☐ p_{de}
☐ p_{ef}



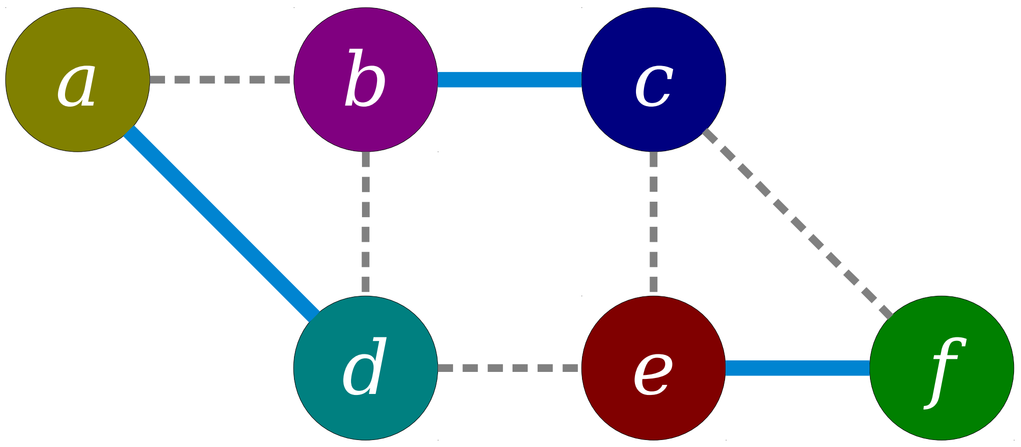
For each edge $\{u, v\}$, we'll introduce a propositional variable p_{uv} meaning “edge $\{u, v\}$ is included in the perfect matching.”

We need to enforce the following:

- Every node is adjacent to at least one edge. (“perfect”)
- Every node is adjacent to at most one edge. (“matching”)

$$(p_{ab} \lor p_{ad}) \land (p_{ab} \lor p_{bc} \lor p_{bd}) \land (p_{bc} \lor p_{ce} \lor p_{cf}) \land$$
$$(p_{ad} \lor p_{bd} \lor p_{de}) \land (p_{ce} \lor p_{de} \lor p_{ef}) \land (p_{cf} \lor p_{ef}) \land$$
$$(\neg p_{ab} \lor \neg p_{ad}) \land$$
$$(\neg p_{ab} \lor \neg p_{bc}) \land (\neg p_{ab} \lor \neg p_{bd}) \land (\neg p_{bc} \lor \neg p_{bd}) \land$$
$$(\neg p_{bc} \lor \neg p_{ce}) \land (\neg p_{bc} \lor \neg p_{cf}) \land (\neg p_{ce} \lor \neg p_{cf}) \land$$
$$(\neg p_{ad} \lor \neg p_{bd}) \land (\neg p_{ad} \lor \neg p_{de}) \land (\neg p_{bd} \land \neg p_{de}) \land$$
$$(\neg p_{ce} \lor \neg p_{de}) \land (\neg p_{ce} \lor \neg p_{ef}) \land (\neg p_{de} \lor \neg p_{ef}) \land$$
$$(\neg p_{cf} \lor \neg p_{ef})$$

T	p_{ab}
F	p_{ad}
F	p_{bc}
F	p_{bd}
F	p_{ce}
T	p_{cf}
T	p_{de}
F	p_{ef}



$(p_{ab} \vee p_{ad}) \wedge (p_{ab} \vee p_{bc} \vee p_{bd}) \wedge (p_{bc} \vee p_{ce} \vee p_{cf}) \wedge$
$(p_{ad} \vee p_{bd} \vee p_{de}) \wedge (p_{ce} \vee p_{de} \vee p_{ef}) \wedge (p_{cf} \vee p_{ef}) \wedge$
$(\neg p_{ab} \vee \neg p_{ad}) \wedge$
$(\neg p_{ab} \vee \neg p_{bc}) \wedge (\neg p_{ab} \vee \neg p_{bd}) \wedge (\neg p_{bc} \vee \neg p_{bd}) \wedge$
$(\neg p_{bc} \vee \neg p_{ce}) \wedge (\neg p_{bc} \vee \neg p_{cf}) \wedge (\neg p_{ce} \vee \neg p_{cf}) \wedge$
$(\neg p_{ad} \vee \neg p_{bd}) \wedge (\neg p_{ad} \vee \neg p_{de}) \wedge (\neg p_{bd} \wedge \neg p_{de}) \wedge$
$(\neg p_{ce} \vee \neg p_{de}) \wedge (\neg p_{ce} \vee \neg p_{ef}) \wedge (\neg p_{de} \vee \neg p_{ef}) \wedge$
$(\neg p_{cf} \vee \neg p_{ef})$

F	p_{ab}
T	p_{ad}
T	p_{bc}
F	p_{bd}
F	p_{ce}
F	p_{cf}
F	p_{de}
T	p_{ef}

A Commonality

- All 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

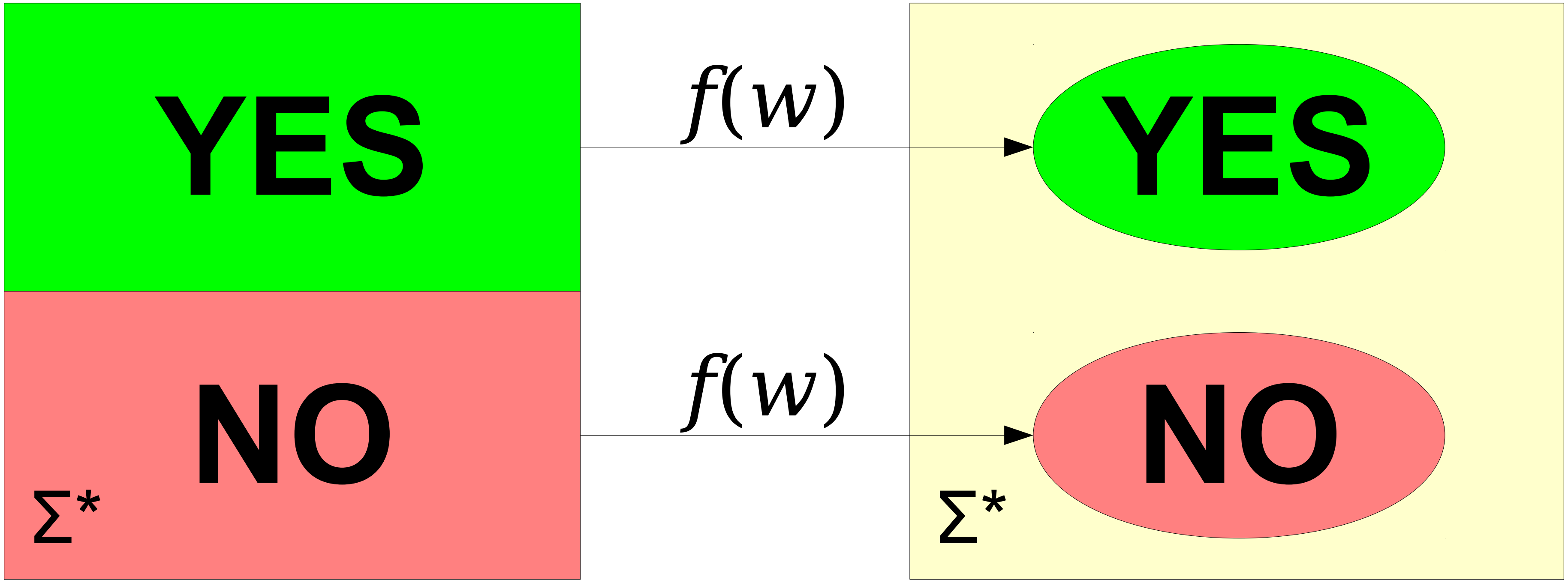
$$\forall w \in \Sigma^*. \ (w \in A \leftrightarrow f(w) \in B)$$

- ***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

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



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 $A \in \mathbf{NP}$.

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 over the break.
- Solutions are available in the Gates building – feel free to stop by to pick them up at your convenience!
- We will be releasing a practice final exam later this week; details on Wednesday.
- Are there any topics you'd like *even more* review on? If so, let us know!

Midterm Regrades

- We will release a midterm regrade request form online tomorrow.
- Please only submit regrade requests if you believe we made an actual grading error. Do not submit regrades if
    - you have a wrong answer and want more points,
    - you don't understand a deduction, or
    - you feel that our criteria are unfair.
- Our definition of “regrade” is “completely regrade the question from scratch according to the grading criteria.” As a result, if you submit a regrade request for a reason other than “we made a grading error,” you are likely to lose points.

Career Advice Panel

- BEAM and Stanford Career Education are putting together a panel event tonight featuring students talking about navigating the CS career landscape.
- The panel looks great – I know many of the students on it and they'll have some really interesting things to say.
- Interested? Follow along on YouTube starting at 5:15PM:

  https://youtu.be/FE5vys-nTG8
- Want to submit questions? Use this link:

  http://goo.gl/forms/9VkCph7NOA

Your Questions!

“Keith, I feel like I know the material but I keep messing up on the tests (it's so easy to mess up on 4 question tests). I feel like I'm going to fail 103 what should I do?”

I'm sorry to hear things are so stressful! Hang in there. You can do this!

When you're learning something for the first time, one of the hardest skills to learn – but one of the most valuable – is being able to recognize when something doesn't seem right. When you get to the point where you have a good "immune system" or a good "sense of smell," you'll become less likely to make little mistakes and you'll get better at catching errors before you even make them.

Developing this skill takes time and practice. Obviously, it's good to do a bunch of practice problems, but be sure to get the TAs to look over your work and offer feedback. Look over the mistakes you've made in the past. Do you really understand what the mistake was? If not, ask your Friendly Neighborhood TA and find out. Your goal on the exam is to make new mistakes, and the more practice you get reviewing your work and finding out where your weaknesses are, the better you'll learn the material.

“What is the biggest risk you've ever taken? How did it turn out?”

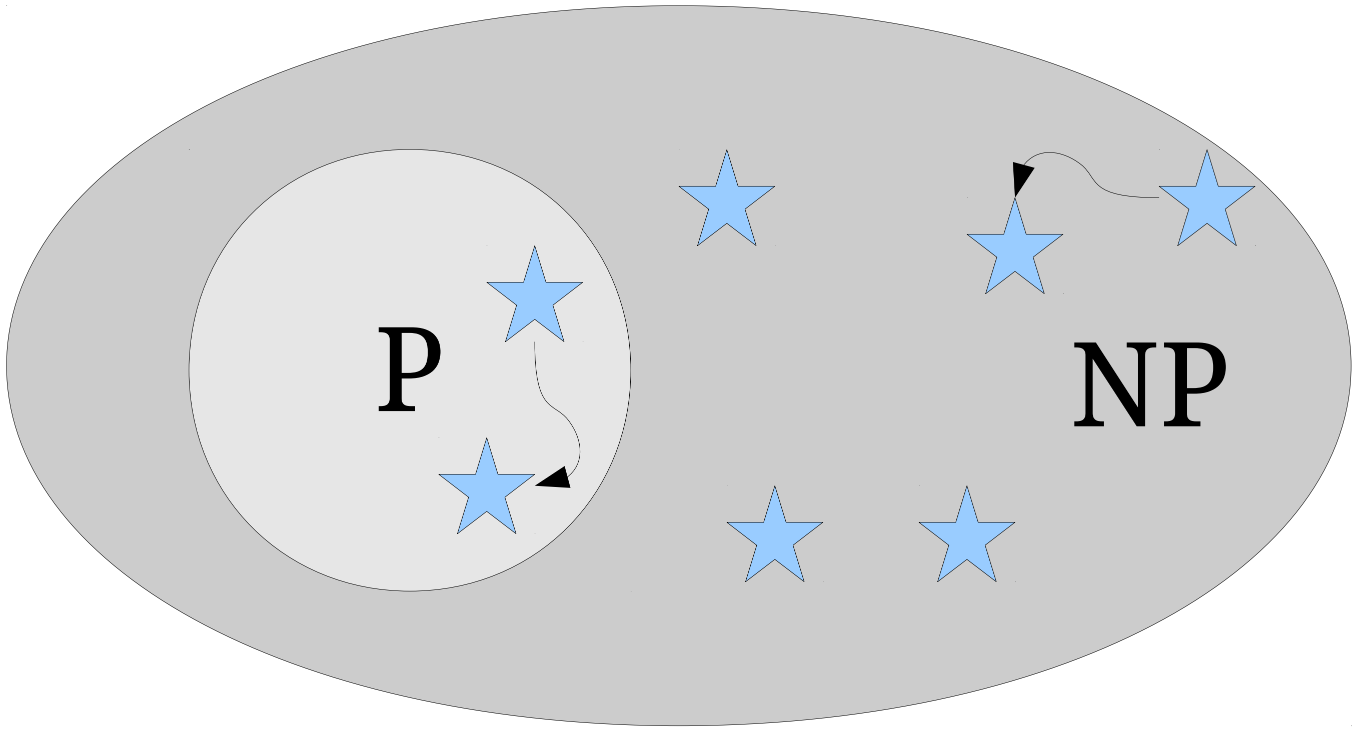
I'm just going to talk about this one in class. I don't feel comfortable posting this answer for everyone on the Internet. Sorry about that!

Back to CS103!

NP-Hardness and **NP**-Completeness

Polynomial-Time Reductions

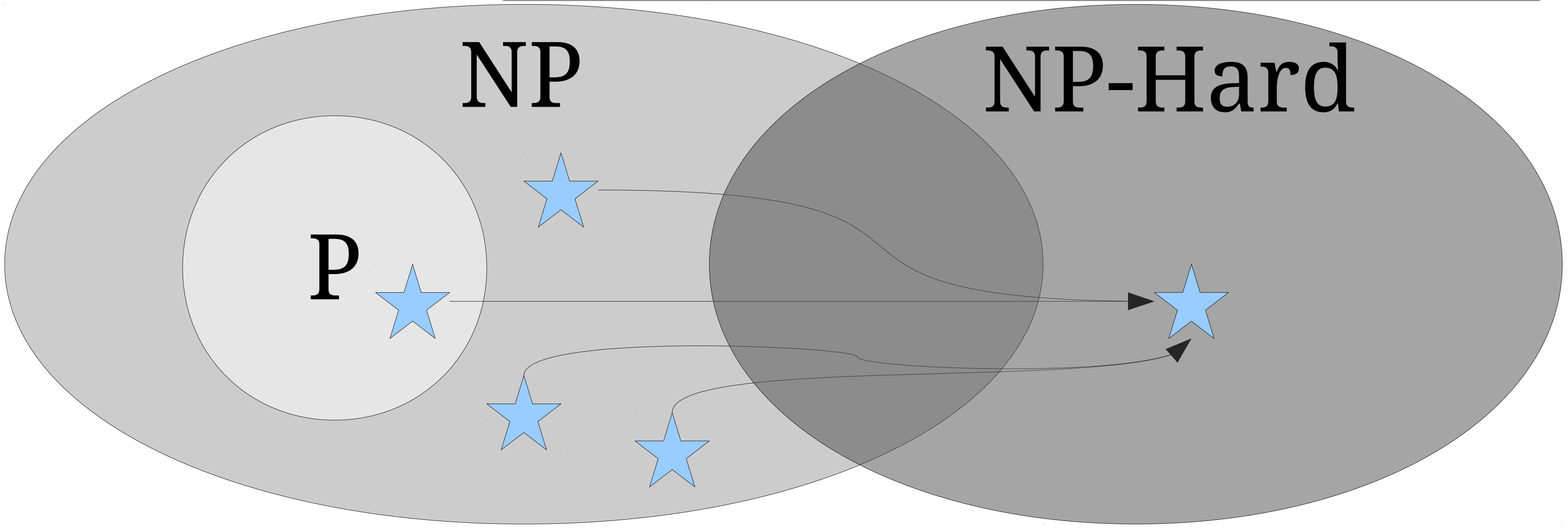
- If $A \leq_P B$ and $B \in \mathbf{P}$, then $A \in \mathbf{P}$.
- If $A \leq_P B$ and $B \in \mathbf{NP}$, then $A \in \mathbf{NP}$.



NP-Hardness

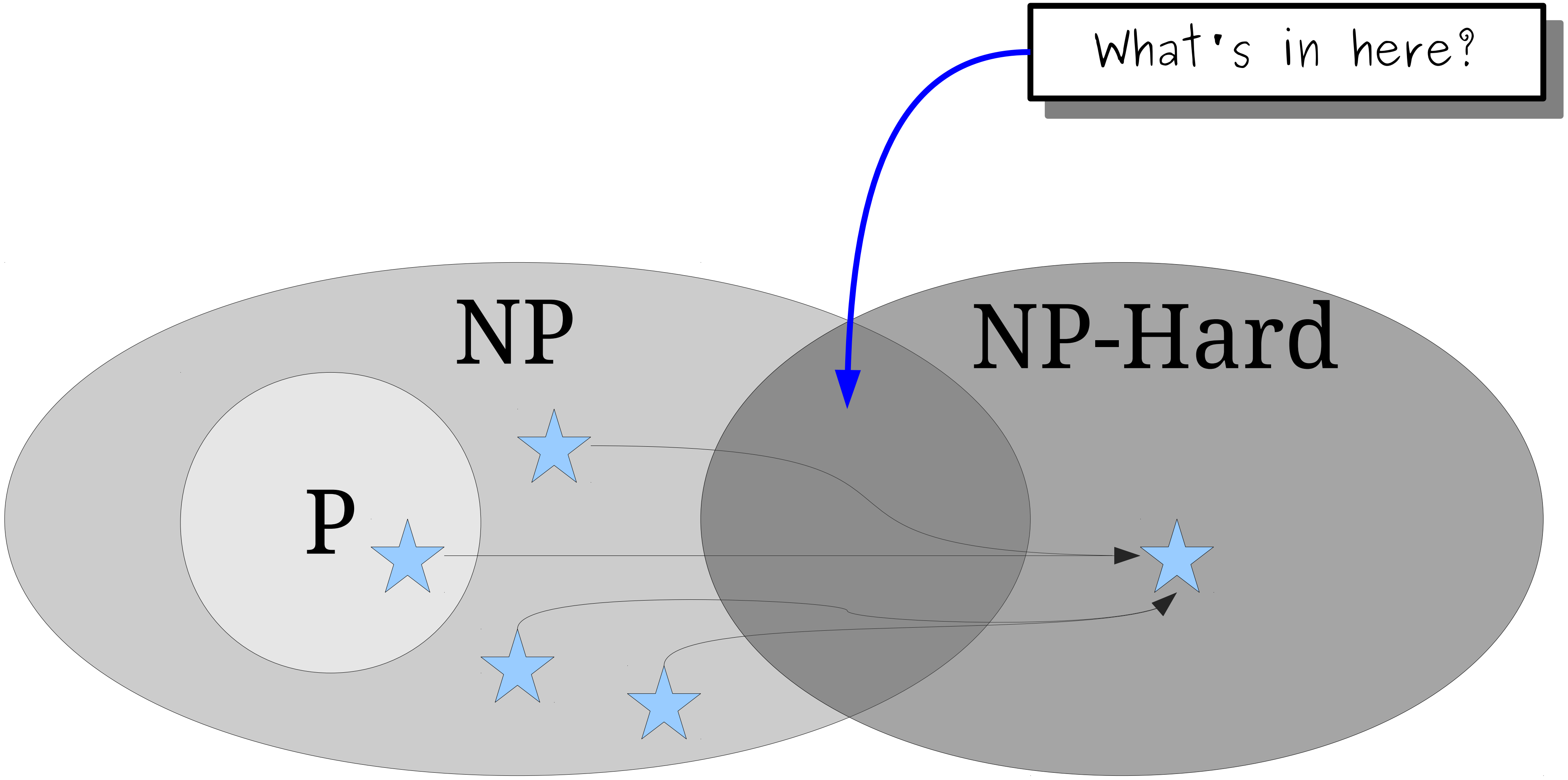
- A language L is called ***NP-hard*** if for *every* $A \in$ **NP**, we have $A \leq_P L$.

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



NP-Hardness

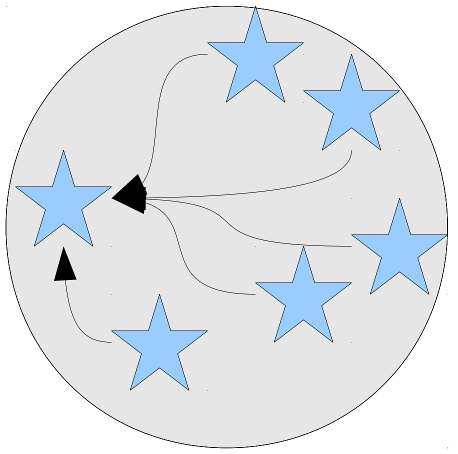
- A language L is called ***NP-hard*** if for *every* $A \in$ **NP**, we have $A \leq_P L$.



The Tantalizing Truth

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

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

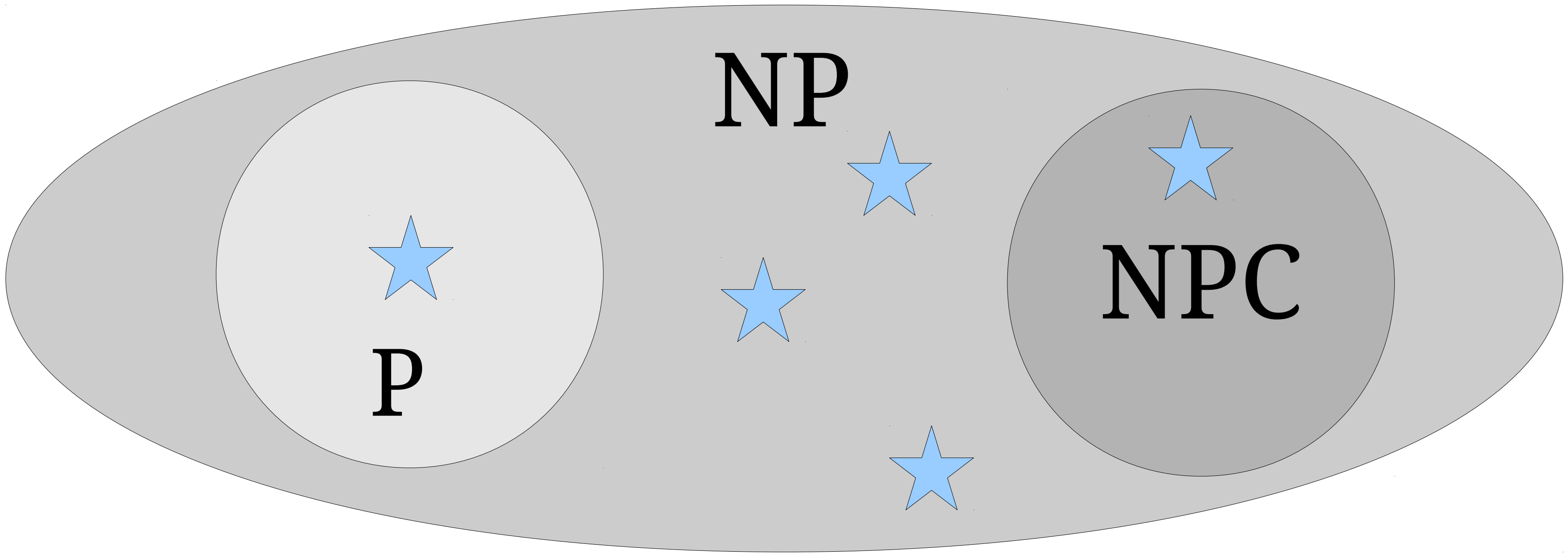


P = NP

The Tantalizing Truth

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

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.
  - $p \land q$ is satisfiable.
  - $p \land \neg p$ is unsatisfiable.
  - $p \to (q \land \neg q)$ is satisfiable.
- An assignment of true and false to the variables of φ that makes it evaluate to true is called a ***satisfying assignment***.

SAT

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

Given a propositional logic formula φ, is φ satisfiable?

- Formally:

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

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

Proof Idea: Given a 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

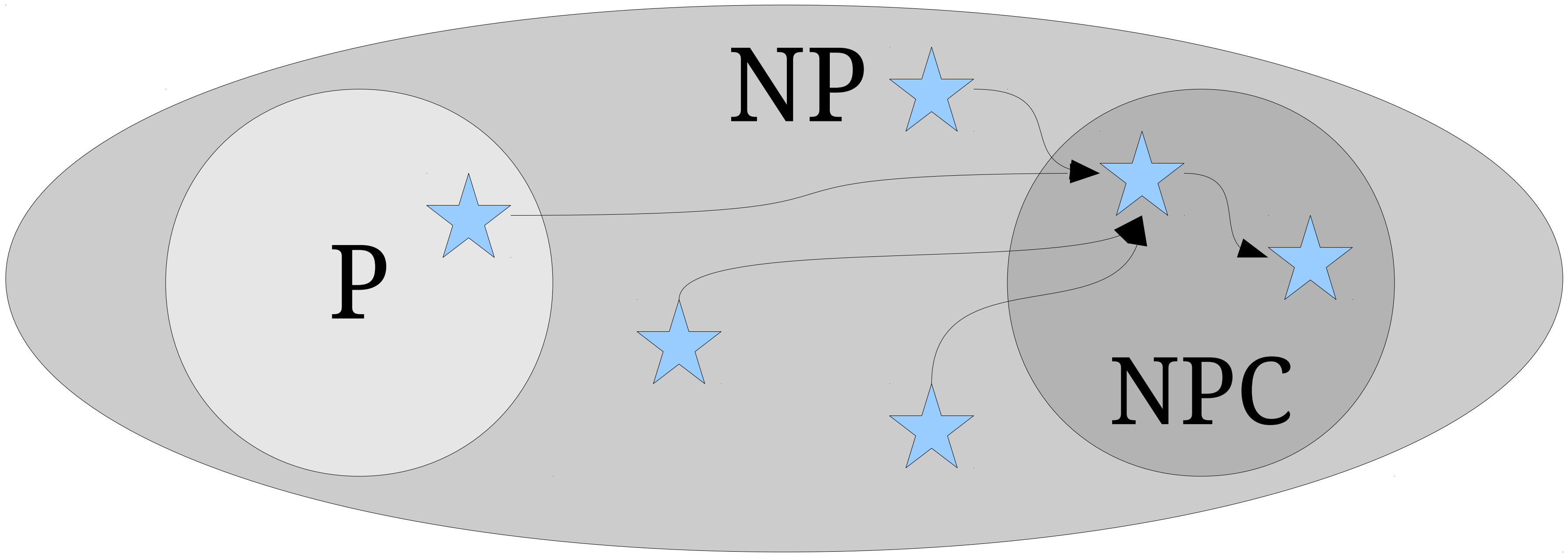
- Resolving $\mathbf{P} \overset{?}{=} \mathbf{NP}$ is equivalent to just figuring out how hard SAT is.
  - If SAT $\in \mathbf{P}$, then $\mathbf{P} = \mathbf{NP}$.
  - If SAT $\notin \mathbf{P}$, then $\mathbf{P} \neq \mathbf{NP}$.
- We've turned a huge, abstract, theoretical problem about solving 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

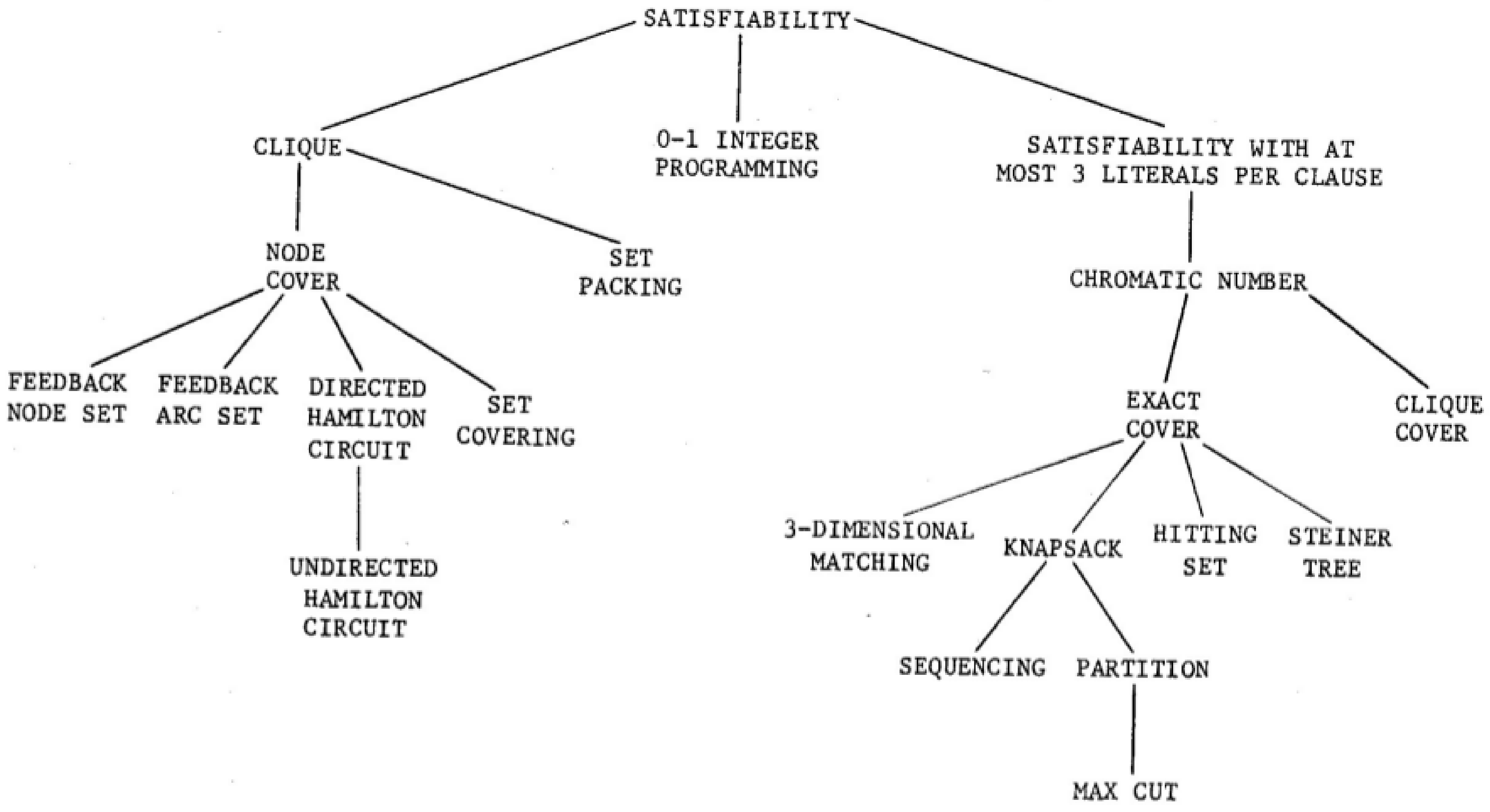
NP-Completeness

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

Theorem: Let A and B be languages where $A \in$ **NPC** and $B \in$ **NP**. If $A \leq_P B$, then $B \in$ **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!*

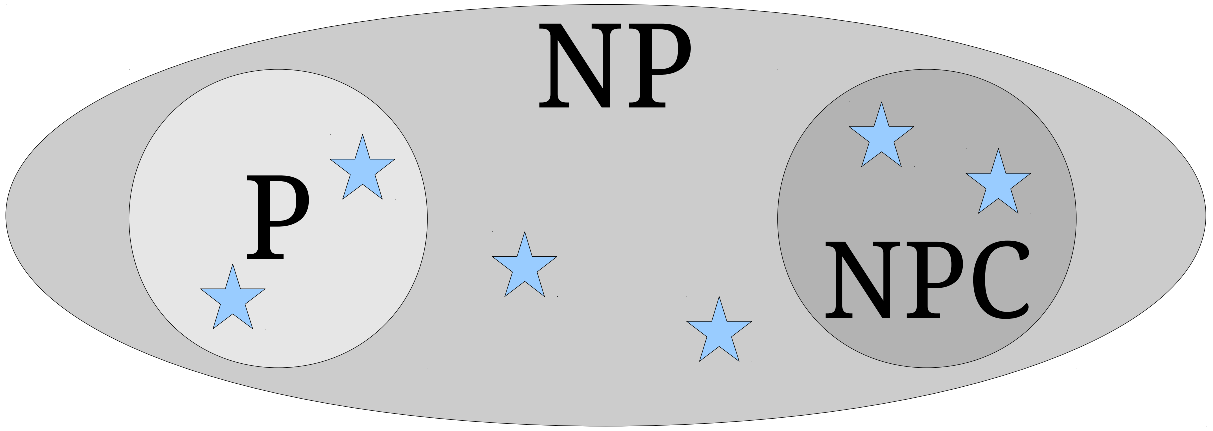
A Feel for **NP**-Completeness

- 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.*

Some Recent News

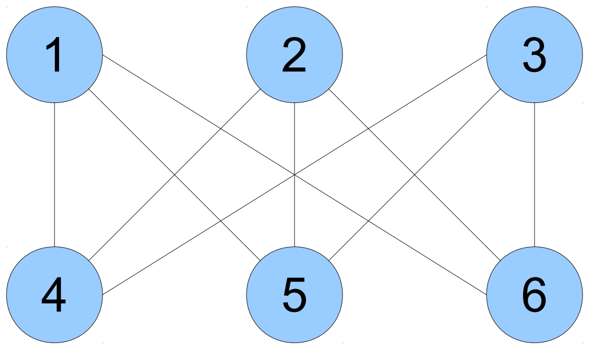
Intermediate Problems

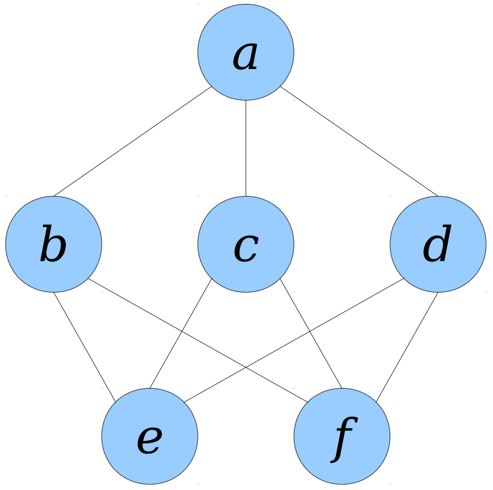
- With few exceptions, every problem we've discovered in **NP** has either
  - definitely been proven to be in **P**, or
  - definitely been proven to be **NP**-complete.
- A problem that's **NP**, not in **P**, but not **NP**-complete is called ***NP-intermediate***.
- ***Theorem (Ladner):*** There are **NP**-intermediate problems if and only if **P ≠ NP**.



Graph Isomorphism

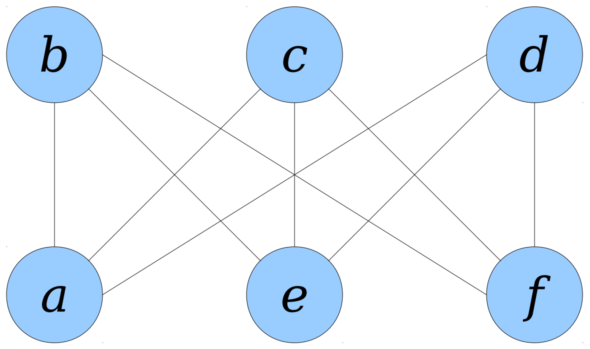
- The ***graph isomorphism problem*** is the following: given two graphs G_1 and G_2, is there a way to relabel the nodes in G_1 so that the resulting graph is G_2?
- This problem is in **NP**, but no one knows whether it's in **P**, whether it's **NP**-complete, or whether it's **NP**-intermediate.

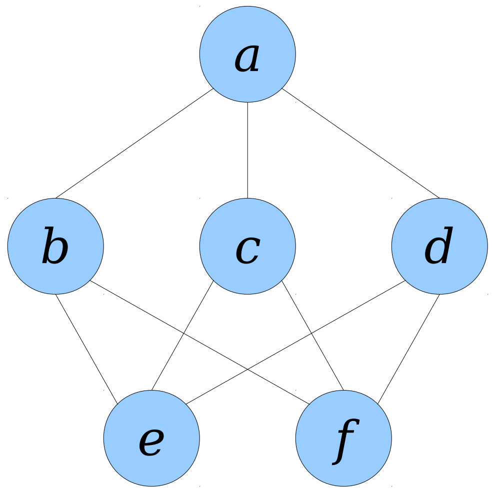




Graph Isomorphism

- The ***graph isomorphism problem*** is the following: given two graphs G_1 and G_2, is there a way to relabel the nodes in G_1 so that the resulting graph is G_2?
- This problem is in **NP**, but no one knows whether it's in **P**, whether it's **NP**-complete, or whether it's **NP**-intermediate.





Recent News

- Right now, a mathematician named Laszlo Babai is presenting a proof that graph isomorphism can be solved in “almost” polynomial time.
- This is a huge deal for a few reasons:
  - This particular problem has resisted any improvements for a long time. Remember – we don't have many good tools for reasoning about **P** and **NP**!
  - The particular technique he's using is somewhat novel and interesting. Remember – we don't have many good tools for reasoning about **P** and **NP**!
  - This probably means that the problem will eventually be proven to be in **P**, meaning that we'll probably develop some really clever new algorithmic technique while solving it. Remember – we don't have many good tools for reasoning about **P** and **NP**!

Next Time

- **Additional NP-Complete Problems**
  - 3SAT
  - Independent Set
  - 3-Colorability