LEC 22

CSE 373

P vs. NP

BEFORE WE START

Instructor	Hunter Schafer	
TAs	Ken Aragon	Paul Pham
	Khushi Chaudhari	Mitchell Szeto
	Joyce Elauria	Batina Shikhalieva
	Santino Iannone	Ryan Siu
	Leona Kazi	Elena Spasova
	Nathan Lipiarski	Alex Teng
	Sam Long	Blarry Wang
	Amanda Park	Aileen Zeng

Learning Objectives

After this lecture, you should be able to...

This lecture is optional! It will cover some cool and modern applications of the things we have learned so far but is not part of the ”core” material for CSE 373. Therefore, we won’t highlight learning objectives since this is focused more on showing off a cool topic!

Lecture Outline

- **2-SAT and 2-Coloring**
- Efficiency, P vs. NP
- More Complex: NP-Complete, NP-Hard

2-SAT

Given a Boolean formula over Boolean variables

```
(A || !B) && (!A || B) && (!A || !B) && (A || !C)
```

Is there a setting of these variables to make this formula true?

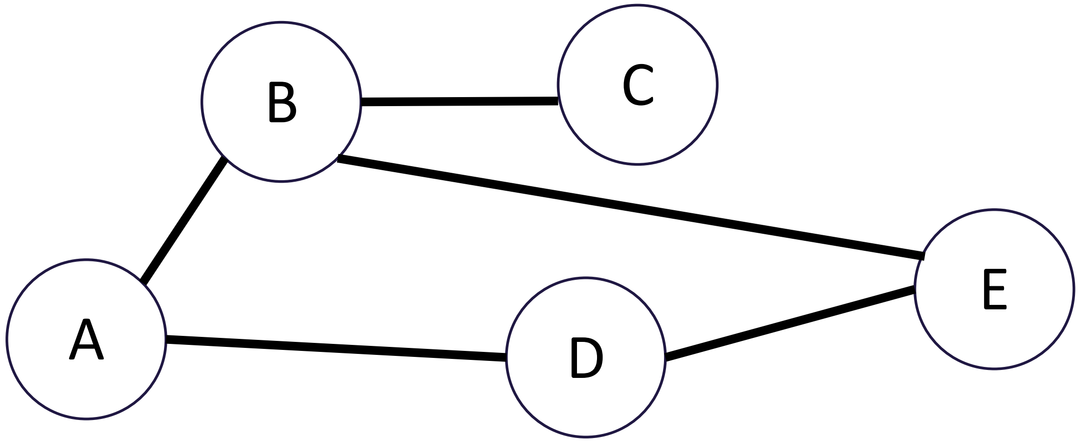
- Called a **satisfiability problem**.

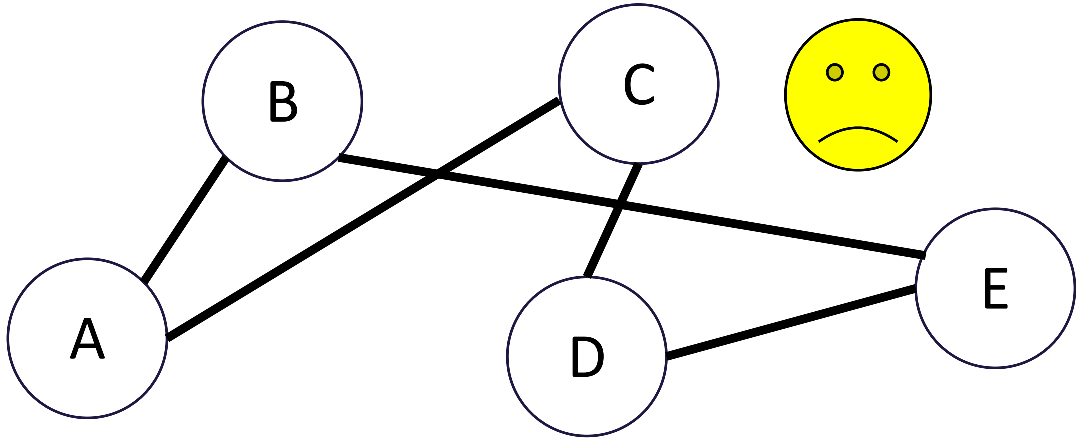
2-SAT is a specific satisfiability problem where each part of the formula uses at most 2 terms.

Fact: There is an efficient algorithm to solve 2-SAT.

2-Coloring

Given a graph, determine if it is possible to color the vertices such that no two vertices that share an edge have the same color.





2-Coloring

Why would we care about coloring a graph?

- Need to divide vertices into two sets, and edges represent conflicts

Could come up with a new algorithm to solve this problem (a modified BFS, a good exercise to try). Instead, let's solve this with a **reduction**.

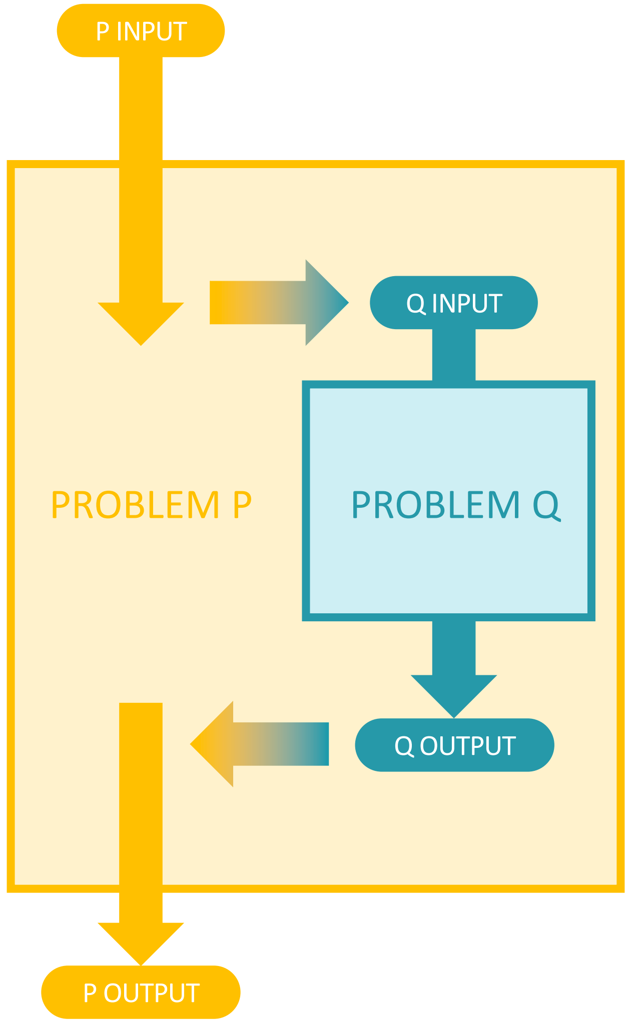
Big Idea

Reduce 2-Coloring to 2-SAT

Review Reductions

- A **reduction** is a problem-solving strategy that involves using an algorithm for problem Q to solve a different problem P
  - Rather than modifying the algorithm for Q, we **modify the inputs/outputs** to make them compatible with Q!
  - "P reduces to Q"

1. Convert input for P into input for Q
2. Solve using algorithm for Q
3. Convert output from Q into output from P



Reduction: 2-Coloring to 2-SAT

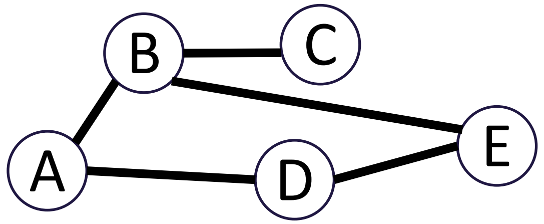
Need to describe 2 steps:

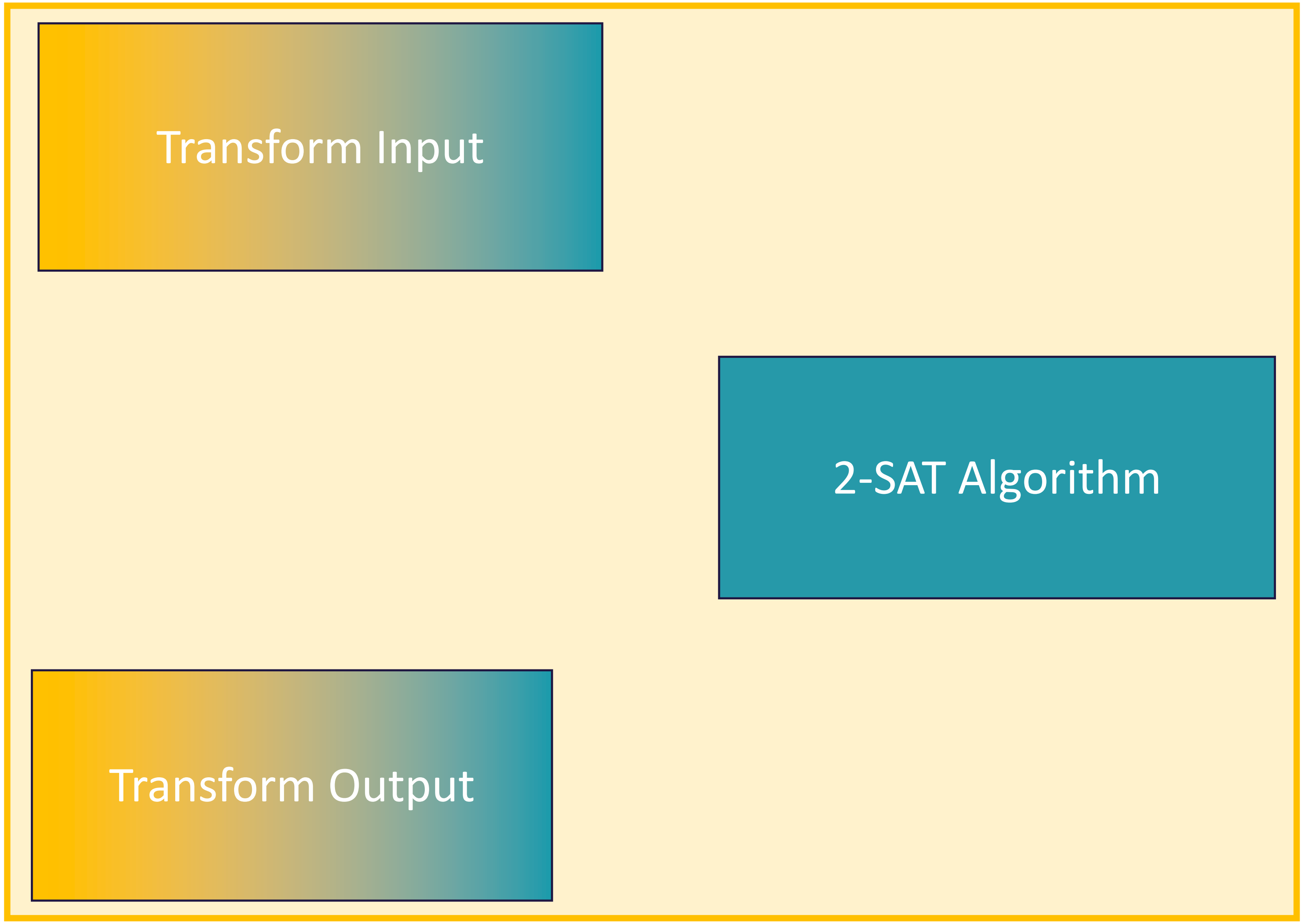
1. Turn a graph for a 2-color problem into an input to 2-SAT
2. How to turn the ANSWER for that 2-SAT input into the answer for the original 2-coloring problem.

Idea: Encode a variable for each vertex like `v1IsRed`. Then to make sure two neighboring vertices have different colors, use a formula

```
(v1IsRed || v2IsRed) && (!v1IsRed || !v2IsRed)
```

How To Perform Topo Sort?





Context

Saw that reductions can be a powerful tool in solving seemingly unrelated problems.

We will also use reductions as a powerful theoretical tool in proving relationships between problems.

Preview for rest of class:

- If we change 2-SAT to 3-SAT and 2-Coloring to 3-Coloring, the reduction remains essentially the same. **So far, there are no known efficient algorithms to solve 3-SAT.**
- If you came up with an efficient 3-SAT algorithm, you will be able to solve a HUGE set of currently intractable problems including, but not limited to:
    - Traveling Salesperson Problem
    - Folding proteins (inventing new medications)
    - Scheduling UW classes in classrooms to optimize for student demand

Lecture Outline

- 2-SAT and 2-Coloring
- **Efficiency, P vs. NP**
- More Complex: NP-Complete, NP-Hard

Efficiency

So far, we have talked about the efficiency of a particular algorithm. Two noticeable differences in our discussion today:

- We will talk about a *problem* being efficiently solvable if an efficient algorithm exists to solve it.
- We will use a loose definition of efficient: Any polynomial runtime!

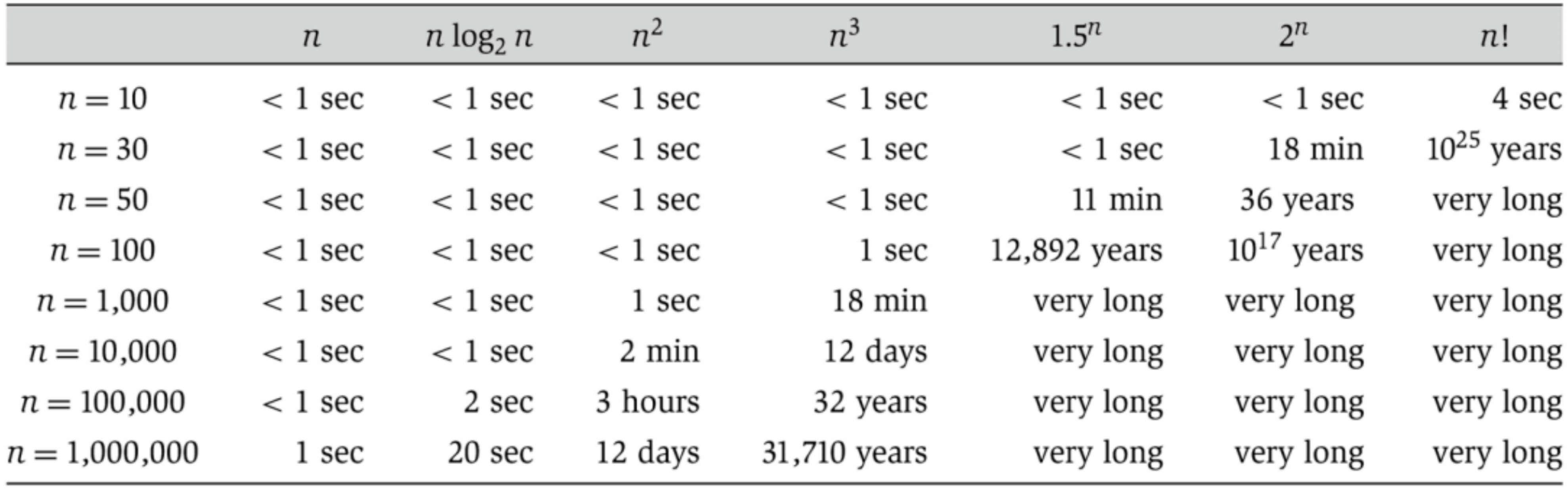
Claim: A polynomial time algorithm runs in time $O(n^k)$ where k is some constant. We consider polynomial time algorithms to be efficient.

- Are they always efficient? Well…. no. Your $O(n^{1000})$ algorithm isn't really efficient.
- These extreme cases are rare, but we could say that polynomial is an absolute minimum qualification to be efficient. A good "low bar"

Running Times

We care about this distinction between polynomial time or not since it’s a VERY strong heuristic for if you can reasonably find a solution to a problem.

A (somewhat old) table from Rosen. Very long means 10^{25} years.



	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Aside Decision Problems

To be a bit pedantic, everything we are going to talk about today only applies to **decision problems.** These are problems where the answer is yes/no.

Why?

- Mostly definitional reasons and it's hard to change lots of research using one definition.
- Almost any problem can be rephrased as a similar decision problem.
  - Instead of "How do we 2-color this graph?" ask "Can we 2-color this graph?"
  - Instead of "Find the shortest path from s to t" ask "Is there are shortest path from s to t of length at most k?"

We will be a bit hand-wavy in this lecture and not focus too much on the decision aspects of these problems, but we thought it was important to mention.

P

P (stands for “Polynomial”)

The set of all decision problems that have an algorithm that runs in time $O(n^k)$ for some constant k

The decision version of *most* problems we’ve seen in this class are in P.

- If we saw a poly time algorithm to solve it, its decision counterpart will be in P.

P is an example of a **complexity class**. A class problems that share some characteristic in terms of how difficult it is to solve them.

P is generally referred to as “problems that can be solved efficiently”.

I’ll Know it When I See It

Another class of problems that we can efficiently verify a solution to. “I’ll Know it When I See it” Problems.

These are decision problems such that

- If the answer is YES, you can verify a potential solution to the problem does result in a YES result.
- This verification takes polynomial time.

Examples:

- If you are a claimed 2-coloring of a graph, you can verify in polynomial that the claimed coloring is a valid 2-colorin
- If someone claims they have a path in a graph of length at most k, you can verify that claim by checking the path itself in poly time.

NP

NP (stands for "Nondeterministic Polynomial")

The set of all decision problems such that if the answer is YES, there is a proof of that which can be verified in polynomial time.

It is a very common misconception that NP stands for "not polynomial".

- Please don't say that. Please!
- Every time this is said, a theoretical computer scientist somewhere sheds a tear.

NP is another complexity class.

- Intuition: P is the set of decision problems we can solve in poly time. NP is the set of decision problems we can verify in poly time if an answer makes the decision YES.

Aside Nondeterministic Polynomail

Where the heck does that name come from and why did no one have the foresight to know people would think NP stands for “not polynomial”?

The concept of a nondeterministic computer is an important concept in CS theory. It’s the concept of having a computer that “knows” the right step to take at each time (or equivalently, exploring all paths at once).

It’s not something we can use in practice, but in theory shows up frequently in statements about computability and efficiency.

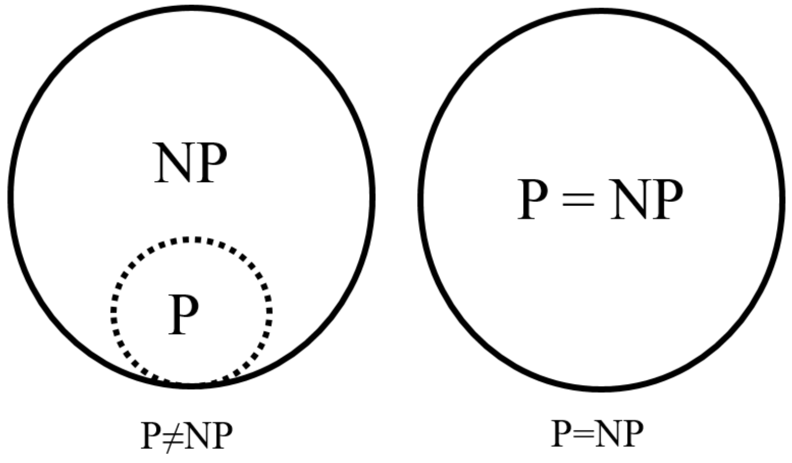
P vs. NP

The fundamental question for computer scientists is answering how P and NP relate to each other.

It's pretty easy to show that $P \subseteq NP$. If you efficiently solve a problem, then you can use that solver if you want to verify an answer. In other words, efficiently solving implies there is an efficient verification.

The big question is: **Is P = NP?**

- Most computer scientists think no.
- Verifying a solutions to problems seems fundamentally easier than solving the problem itself.
- No one has proven this though!



Why should you care about P vs NP?

Even though most of us are convinced $P \neq NP$, why do people care?

It's your chance to:

- $1,000,000. The Clay Mathematics Institute will give a million dollars to whoever solves P vs. NP (or any of their other 5 problems).
- To get ~~a Turing Award~~ the Turing Award renamed after you.

Why should you Care if P = NP?

Suppose P = NP. Specifically, that you found an efficient algorithm for one of these “NP-complete” problems (next video). What would you do?

- Get $1,000,000 from the Clay Math Institute. But what’s next?
- Put mathematicians out of work
- Decrypt (essentially) all current internet communication.
    - No more secure online shopping, banking or messaging, or really online *anything*.
- Maybe find the cure for cancer?
- A world where P = NP is a very very different place from the world we live now.

Why Should you Care if $P \neq NP$?

We already expect this is the case, why prove it?

- Tells us something fundamental about the universe and computation.
- For some questions, there is not a clever way to find the right answer.
  - Even though you'll know the answer once you see it.

To prove $P \neq NP$, we need to better understand differences between problems.

- Why do some problems allow easy solutions and other don't?
- What is the structure of these problems?

We don't care about P vs. NP just because it has a huge effect on what the world looks like (even though that's one reason). We will learn a LOT about computation along the way.

Lecture Outline

- 2-SAT and 2-Coloring
- Efficiency, P vs. NP
- **More Complex: NP-Complete, NP-Hard**

P vs. NP

Most computer scientists thing $P \neq NP$. No one has proven this yet, but there is an important subset of NP that will be useful when it comes to understand the hardness of problems.

To prove $P \neq NP$, "all you have to do" is show there is a problem in NP that requires exponential time to solve.

- We know lots of problems that we only *currently* have exponential algorithms for, but that doesn't mean there is some unknown efficient algorithm we haven't found yet!

Before we introduce these other sets of problems, we have to make one clarification to the idea of reductions.

Reductions (again)

To talk about hardness of problems in NP, we need to bring back the idea of reductions to solve problems. One addition to the idea of a reduction is the idea of an efficient reduction.

Polynomial Time Reducible

We say A reduces to B in polynomial time, if there is an algorithm for A that:

- Calls a black box for B at most a polynomial number of times
- Runs at most a polynomial number of other operations

All this means is the reduction itself is totally polynomial.

- If the runtime of the algorithm for B is also polynomial time, then this whole procedure is polynomial.

If A reduces to B, then A should be “easier” than B.

- If we can solve B, we can definitely solve A.
- Usually denoted $A \leq_P B$

NP-complete

If we want to prove there is a problem in NP that isn't in P, we should probably pick the hardest one we can think of!

What is the hardest problem in NP?

NP-complete

Problem B is NP-complete if:

- B is in NP and
- For all problems A in NP, A reduces to B in polynomial time

Any problem you can prove is NP-complete is one of the hardest problems in NP (since all problems in NP reduce to it)

Why NP-complete?

Seems like the right place to start to prove $P \neq NP$. If it's the hardest problem in NP, it's probably the one that requires that highest runtime.

Turns out, it's also the right place to start for proving $P = NP$!

- If you can find a polynomial time algorithm for one NP-complete problem, it gives you a polynomial time algorithm for **every** problem in NP.

Examples

There are literally thousands of NP-complete problems. Some look weirdly similar to problems we care about or ones that we have efficient algorithms for.

In P

Short Path

Given a directed graph, report if there is a path from s to t of length **at most** k.

NP-complete

Long Path

Given a directed graph, report if there is a path from s to t of length **at least** k.

Examples

There are literally thousands of NP-complete problems. Some look weirdly similar to problems we care about or ones that we have efficient algorithms for.

In P

Light Spanning Tree

Given a weighted graph, report of there is a spanning tree of weight at most k.

NP-complete

Traveling Salesperson

Given a weighted graph, find a tour (cycle that visits every vertex once before returning to start) of weight at most k.

Examples

There are literally thousands of NP-complete problems. Some look weirdly similar to problems we care about or ones that we have efficient algorithms for.

In P

2-SAT

Given a Boolean formula of the form "at least one of two must be true", determine if a setting of variables can make the whole formula true.

NP-complete

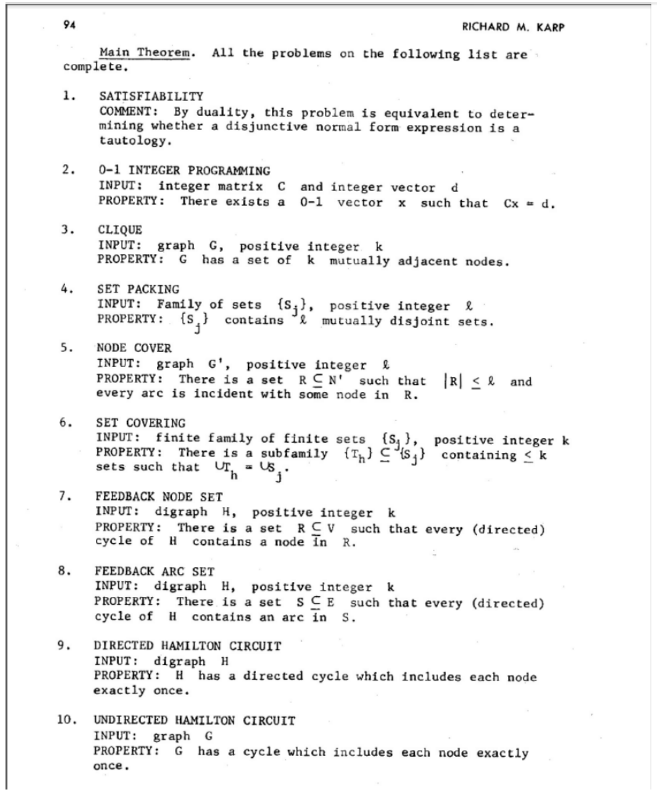
3-SAT

Given a Boolean formula of the form "at least one of three must be true", determine if a setting of variables can make the whole formula true.

NP-complete

If you find an efficient algorithm for an NP-complete problem, you have an efficient algorithm for **every** problem in NP.

Started with Cook-Levin Theorem (1971) that proved that SAT (general version of 3-SAT) is NP-complete. A year later…



94 RICHARD M. KARP

Main Theorem. All the problems on the following list are complete.

1. SATISFIABILITY
COMMENT: By duality, this problem is equivalent to determining whether a disjunctive normal form expression is a tautology.

2. 0-1 INTEGER PROGRAMMING
INPUT: integer matrix C and integer vector d
PROPERTY: There exists a 0-1 vector x such that $Cx = d$.

3. CLIQUE
INPUT: graph G, positive integer k
PROPERTY: G has a set of k mutually adjacent nodes.

4. SET PACKING
INPUT: Family of sets $\{S_j\}$, positive integer ℓ
PROPERTY: $\{S_j\}$ contains ℓ mutually disjoint sets.

5. NODE COVER
INPUT: graph G', positive integer ℓ
PROPERTY: There is a set $R \subseteq N'$ such that $|R| \le \ell$ and every arc is incident with some node in R.

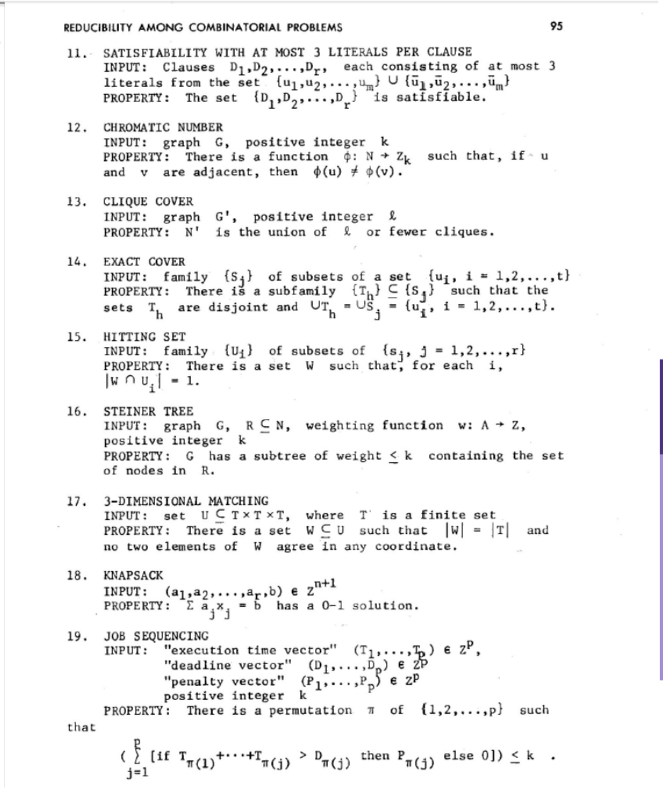
6. SET COVERING
INPUT: finite family of finite sets $\{S_j\}$, positive integer k
PROPERTY: There is a subfamily $\{T_h\} \subseteq \{S_j\}$ containing $\le k$ sets such that $\cup T_h = \cup S_j$.

7. FEEDBACK NODE SET
INPUT: digraph H, positive integer k
PROPERTY: There is a set $R \subseteq V$ such that every (directed) cycle of H contains a node in R.

8. FEEDBACK ARC SET
INPUT: digraph H, positive integer k
PROPERTY: There is a set $S \subseteq E$ such that every (directed) cycle of H contains an arc in S.

9. DIRECTED HAMILTON CIRCUIT
INPUT: digraph H
PROPERTY: H has a directed cycle which includes each node exactly once.

10. UNDIRECTED HAMILTON CIRCUIT
INPUT: graph G
PROPERTY: G has a cycle which includes each node exactly once.



REDUCIBILITY AMONG COMBINATORIAL PROBLEMS 95

11. SATISFIABILITY WITH AT MOST 3 LITERALS PER CLAUSE
INPUT: Clauses $D_1, D_2, \ldots, D_r$, each consisting of at most 3 literals from the set $\{u_1, u_2, \ldots, u_m\} \cup \{\bar{u}_1, \bar{u}_2, \ldots, \bar{u}_m\}$
PROPERTY: The set $\{D_1, D_2, \ldots, D_r\}$ is satisfiable.

12. CHROMATIC NUMBER
INPUT: graph G, positive integer k
PROPERTY: There is a function $\phi: N \to Z_k$ such that, if u and v are adjacent, then $\phi(u) \ne \phi(v)$.

13. CLIQUE COVER
INPUT: graph G', positive integer ℓ
PROPERTY: N' is the union of ℓ or fewer cliques.

14. EXACT COVER
INPUT: family $\{S_j\}$ of subsets of a set $\{u_i,\ i = 1,2,\ldots,t\}$
PROPERTY: There is a subfamily $\{T_h\} \subseteq \{S_j\}$ such that the sets T_h are disjoint and $\cup T_h = \cup S_j = \{u_i,\ i = 1,2,\ldots,t\}$.

15. HITTING SET
INPUT: family $\{U_i\}$ of subsets of $\{s_j,\ j = 1,2,\ldots,r\}$
PROPERTY: There is a set W such that, for each i, $|W \cap U_i| = 1$.

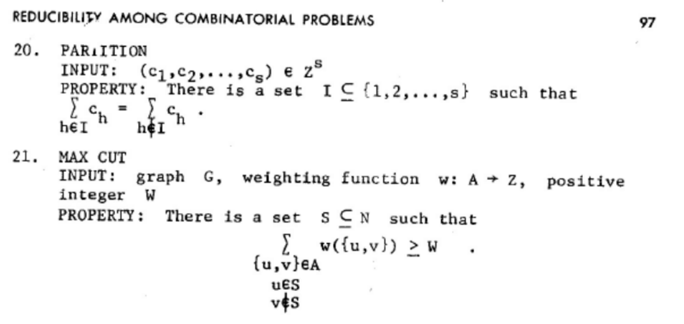
16. STEINER TREE
INPUT: graph G, $R \subseteq N$, weighting function $w: A \to Z$, positive integer k
PROPERTY: G has a subtree of weight $\le k$ containing the set of nodes in R.

17. 3-DIMENSIONAL MATCHING
INPUT: set $U \subseteq T \times T \times T$, where T is a finite set
PROPERTY: There is a set $W \subseteq U$ such that $|W| = |T|$ and no two elements of W agree in any coordinate.

18. KNAPSACK
INPUT: $(a_1, a_2, \ldots, a_r, b) \in Z^{n+1}$
PROPERTY: $\Sigma a_j x_j = b$ has a 0-1 solution.

19. JOB SEQUENCING
INPUT: "execution time vector" $(T_1, \ldots, T_p) \in Z^p$,
"deadline vector" $(D_1, \ldots, D_p) \in Z^p$
"penalty vector" $(P_1, \ldots, P_p) \in Z^p$
positive integer k
PROPERTY: There is a permutation π of $\{1,2,\ldots,p\}$ such that

$$\left(\sum_{j=1}^{p} [\text{if } T_{\pi(1)} + \cdots + T_{\pi(j)} > D_{\pi(j)} \text{ then } P_{\pi(j)} \text{ else } 0]\right) \le k \ .$$



REDUCIBILITY AMONG COMBINATORIAL PROBLEMS 97

20. PARTITION
INPUT: $(c_1, c_2, \ldots, c_s) \in Z^s$
PROPERTY: There is a set $I \subseteq \{1,2,\ldots,s\}$ such that $\sum_{h \in I} c_h = \sum_{h \notin I} c_h$.

21. MAX CUT
INPUT: graph G, weighting function $w: A \to Z$, positive integer W
PROPERTY: There is a set $S \subseteq N$ such that

$$\sum_{\substack{\{u,v\} \in A \\ u \in S \\ v \notin S}} w(\{u,v\}) \ge W \ .$$

Karp's Theorem (1972)

A lot of problems people care about are NP-complete

NP-complete

But wait! There's more!

By 1979, at least 300 problems had been proven NP-complete.

Garey and Johnson wrote a textbook with 100 pages listing known NP-complete problems.

No one has made a super comprehensive list since, but there are literally thousands and there are new ones found each year.

NP-Hard

NP-hard

Problem B is NP-complete if:

- For all problems A in NP, A reduces to B in polynomial time

NP-hard are the problems that are "at least as hard as the hardest problems in NP".

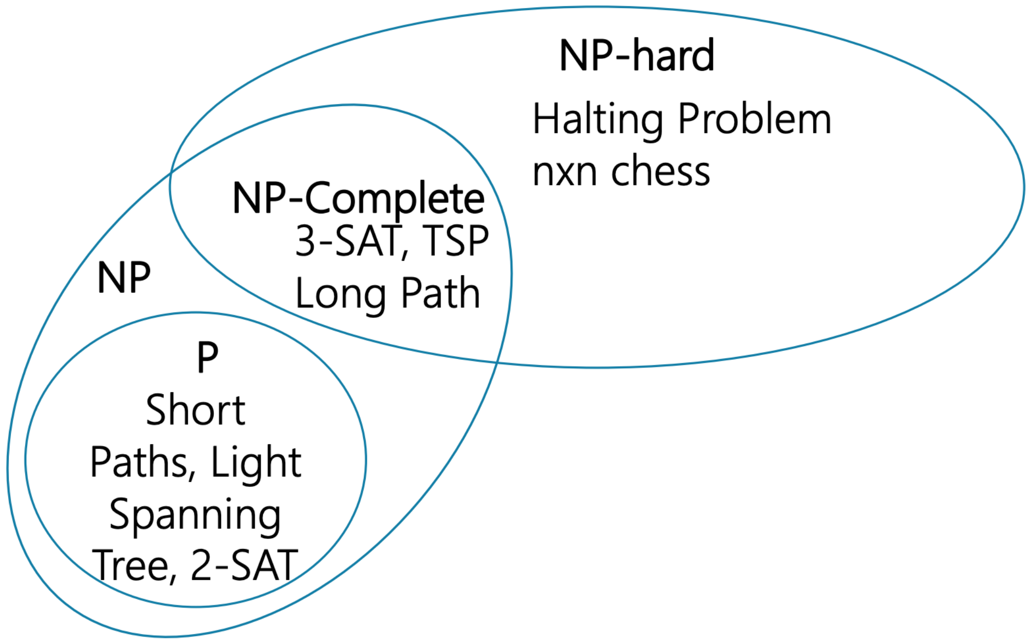
- NP-complete = NP and NP-hard.

Many problems that are in NP-hard that aren't in NP (i.e. no efficient verification). The edges of NP-hard is the "there be dragons" of our problem space.

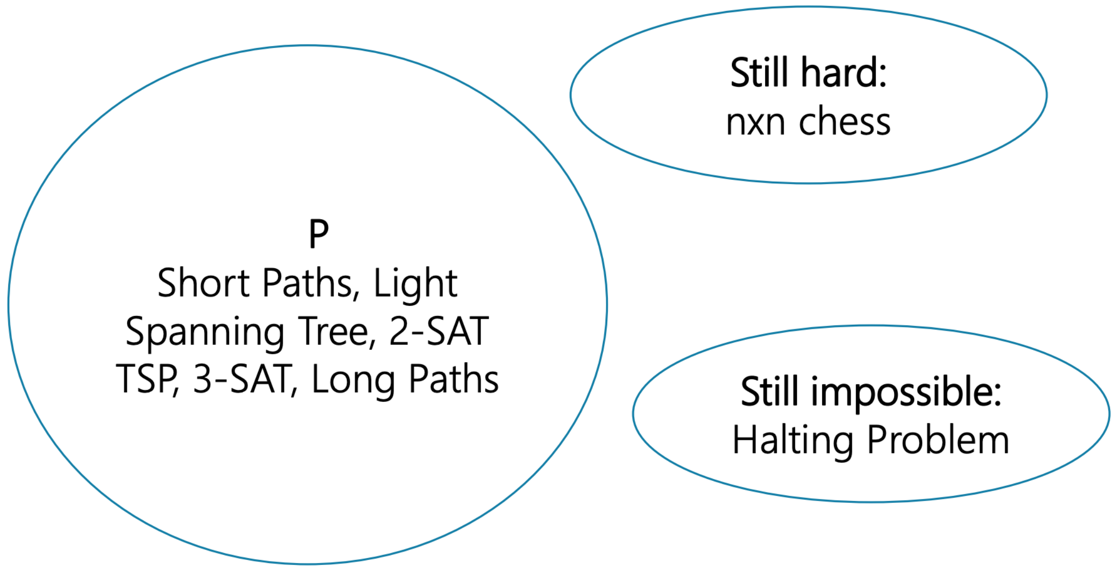
- Determining if you can win a game of n x n chess (really hard).
- Determining if your P2 code is stuck in an infinite loop or just taking a long time to run (undecidable).

WorldView

If $P \neq NP$ (what we think)



If $P = NP$



NP-complete in Practice

Since it's unlikely that P = NP, if you know your problem is NP-complete you likely can't solve it efficiently. Three common options:

1. Maybe it's a special case we understand better.
   - While SAT is NP-complete, 2-SAT is not!
2. Even if a problem is NP-complete, ”nice” instances might exist that can be solved quickly.
   - Common: Reduce your problem to a SAT instance and us a SAT-solver to help you solve it quickly (worst case, still might be exponential)
3. Approximation Algorithms: Just like with TSP, make an efficient algorithm that can approximately compute an answer if only an approximate guess is good enough.