Lecture 21: Complexity Theory

Announcements:

- The final approaches fast!
- Three HW's: fork/join, concurrency, complexity (all due by Wednesday afternoon!)
- Monday: no lecture, but instead time to work on your HW's / study hall.
  - I will hold OH monday instead of friday.
- Wednesday: no lecture, still more study hall
  - I can go through P/NP section material if people are interested.
- We will try to grade HW's by Wednesday night, so you have feedback in time.
- Complexity concept check due Monday (instead of friday)
- No section next week (day of final)
- Final thurs 5-7pm Sieg 134

This quarter we focused on fast algorithms and the data structures behind them.

- We could've used a linked list as our dictionary (slow)
- instead we saw how to use AVL trees or hashing to get faster runtimes!

In general, computer scientists often divide problems into two types

1. "easy" problems: computer can solve in time $O(n^c)$ (c = constant)
2. "hard" problems: computer cannot solve in time $O(n^c)$

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

complexity theory's goal: **prove** that certain problems are **hard** (for computers)

What is a **problem**? It is simply a set of inputs/outputs

ex 1| Minimum spanning tree

input = graph output = any MST

ex 2| "tell me if this list is sorted"

input = list $\in \mathbb{R}^n$ output = $\{0,1\}$ ($\{$yes, no$\}$)

ex 3| "sort this array"

input = list output = list.

Def| An **instance** is a specific input to a problem.

Complexity Theorists focus on **decision problems** (output is "yes" or "no")

so instead of "find shortest path"

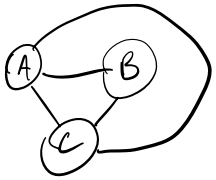
ask "is there a path of length $\leq k$?"

(this is because if you can solve the "decision version" then you can solve the "search version"!)

Examples!!!

(1) **Eulerian Path:** "Does G have a path that uses each edge exactly once?"

Theorem| $\exists$ Eulerian Path iff exactly 0 or 2 vertices have odd degree



intuition: suppose every vertex is even degree

ALG: starting at u, keep following edges you haven't used yet, until you cannot anymore.

claim: This process stops at u.

$\Rightarrow$ Eulerian Path is EASY: $\exists$ $O(|V| + |E|)$ algo solving it (why?)

(2) Hamiltonian Path: "Does G have a path that touches every vertex exactly once?!"

This seems like a similar problem!

ALG: for each permutation of the $|V|$ vertices, check if that path exists

runtime?

better algorithm?

... we think Hamiltonian Path is HARD

Def P is the class of all problems solvable in time $O(n^c)$ where c is constant

"solvable" = $\exists$ an algorithm with this runtime

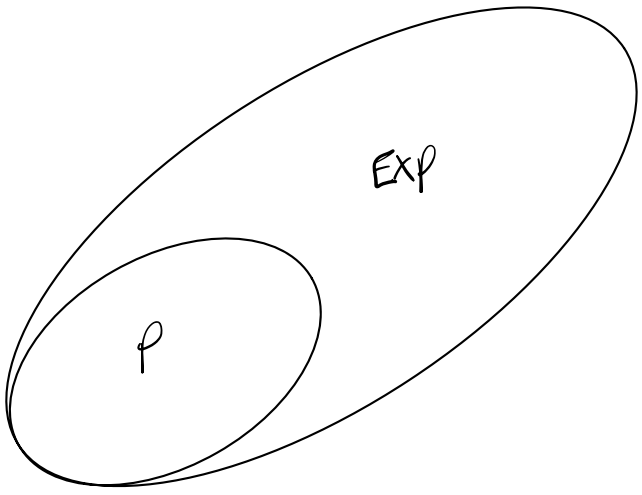
P stands for "polynomial time". It is a set of problems.

Def EXP is the class of problems that can be solved in $O(2^{n^c})$ time, c = constant.

questions: 1) is Eulerian Path $\in P$?

2) is Hamiltonian Path $\in P$?

3) If $f \in P$ then $f \in EXP$?



Def NP is the class of problems for which YES instances have a "proof" or "certificate" that can be verified in polynomial time $O(n^c)$, c = const.

more formally: $f \in NP$ iff $\exists$ algorithm $A(x,y)$ and polynomials p and q, s.t.

① $\forall x \in f^{-1}(1)$, $A(x,y)$ runs in time $p(|x|)$

② $\forall x \in f^{-1}(1)$ $\exists y$, $|y| \leq q(|x|)$, s.t. $A(x,y) = 1$.

③ $\forall x \in f^{-1}(0)$ and $\forall y$ of length $\leq q(|x|)$, $A(x,y) = 0$.

examples: **Hamiltonian path $\in$ NP** because if G has a Hamiltonian path, then the "proof" can be the path itself

the "verifier" checks that the path it's given

① is a valid path

② touches each vertex once

3-COLOR: "can I color vertices of G with ≤ 3 colors such that all neighbors are colored differently?"

3-COLOR $\in$ NP : proof = assignment of colors to vertices

verifier = checks if ① only 3 colors used

② each edge has endpoints colored differently.

2-COLOR: "can I color vertices of G with ≤ 2 colors such that all neighbors are colored differently?"

2-COLOR $\in$ P ALG: run BFS. If layer of vertex is even, color 0

If layer is odd, color 1

(if you must color vertex with a different color than it already has, output NO)

Theorem $P \subseteq NP$

proof let $f \in P$. So there exists an algorithm that runs in polynomial time deciding f.

if $f(x) = 1$, then the "proof" is empty, and the verifier

just runs the algorithm to see whether $f(x) = 1$.

ex "Is there an MST of weight $\leq k$" $\in NP$

(1) we could have the "proof" be a MST of weight $\leq k$
and the "verifier" check that the proof a) is a tree, and
b) has cost $\leq k$.

(2) we could also have the "proof" be nothing at all.
and let the verifier run Prim's / Kruskal's algo.

Theorem $NP \subseteq EXP$

proof Let $f \in NP$. given x, consider the following algorithm:

ALG: (1) write down all possible "proofs" of length $\leq n^c$ that $f(x) = 1$

(2) run verifier on all these proofs
if verifier succeeds on any, output 1. Otherwise, output 0.

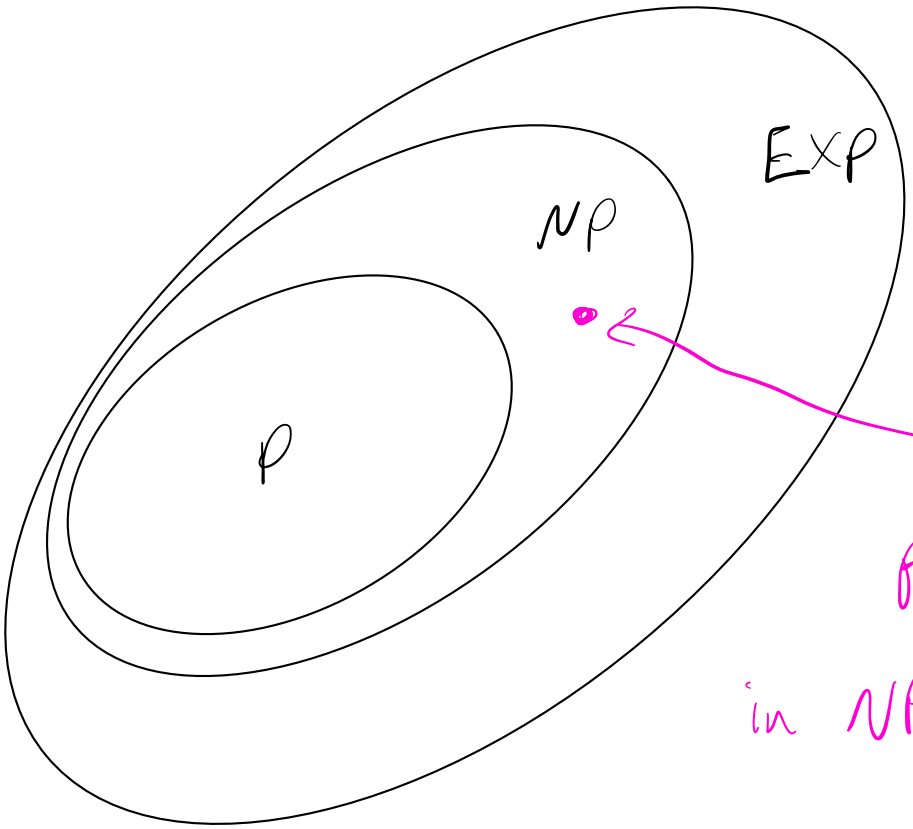
runtime of ALG?

$\Rightarrow f \in EXP$ ∎

BIG COMPLEXITY THEORY QUESTION: $P \stackrel{?}{=} NP$

intuitively: If I can "verify" a good solution quickly, can I use this to come up with a fast algorithm myself?

... most people think the answer is no.



Is there any problem here?

in NP but not in P?

One more fun example

"is there MST of weight $\leq k$" $\in P$

"is there a walk that visits each vertex exactly once, and returns to start, and has weight $\leq k$" $\in NP$

electric company can optimize its grid, but Amazon doesn't know

how to optimally route it's drivers!! (without solving P vs. NP, that is)

REDUCTIONS

Def We say that problem **A is polynomial time reducible to B** if there is an algorithm which, using a (hypothetical) polynomial time algorithm for B, solves A in polynomial time.

notation: $A \leq_p B$

intuition: A is no harder to solve than B, because if we can solve B, then the reduction tells us we can also solve A!

Warning: Don't get it backwards!

Silly example: suppose A = I have ants in my kitchen, I want them gone.

B = I blow up my house, and rebuild it.

Then $A \leq_p B$, because I can run my algo for B, and it gets rid of the ants.

A is no harder than B.

another example: $3\text{-COLOR} \leq_p 4\text{-COLOR}$

ALG: on input G, add a new vertex, and connect it to all other vertices

call this graph G'

return 4-COLOR(G')

need to check: 1) if G 3-colorable, then G' 4-colorable
2) if G not 3-colorable, then G' not 4-colorable
} correctness

3) ALG runs in polynomial time, assuming 4-COLOR does] poly-time.

Def: A is NP-complete iff ① $\forall B \in NP$, $B \leq_p A$
② and $A \in NP$.

motivation: these are the "hardest" problems in NP

so if we want to show $P \neq NP$, these would be the best candidates! (most likely not to have a poly time algo)

Def: A is NP-hard iff $\forall B \in NP$, $B \leq_p A$.

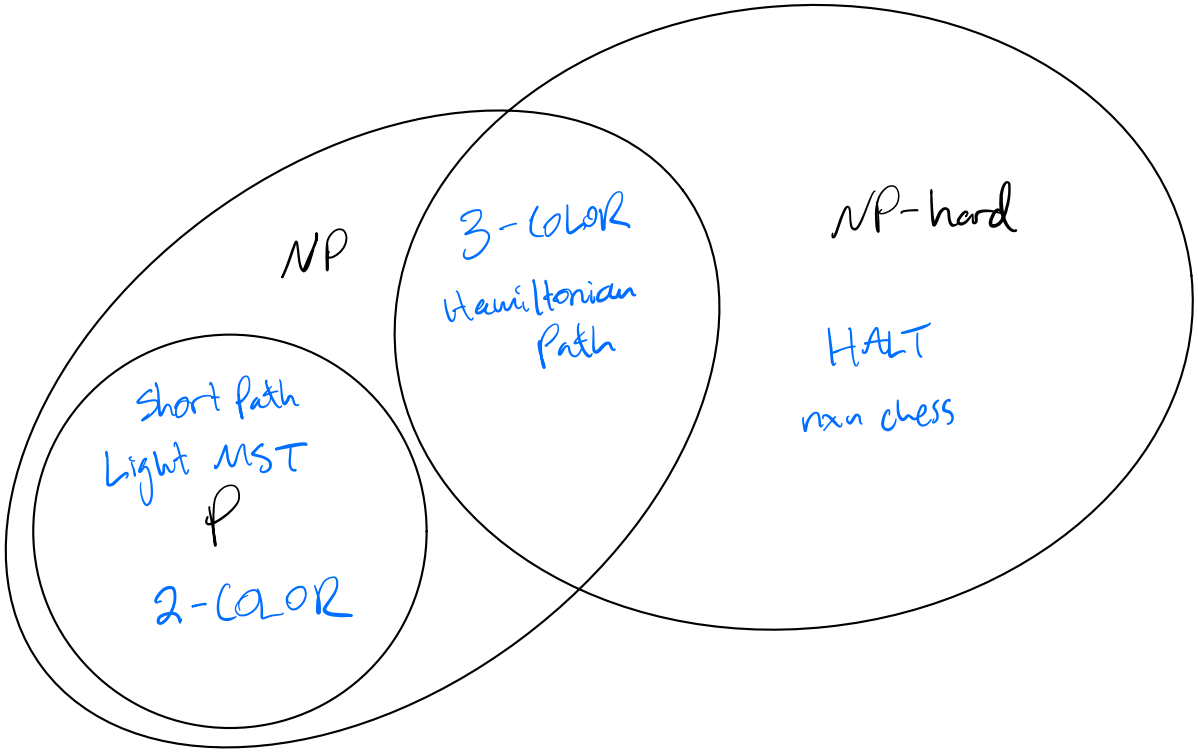
Remark: Many problems are NP hard, but not in NP

ex] halting problem is NP-hard (but certainly not NP-complete!)

nxn chess: is there an optimal strategy for white?

not easy to verify!

What we think the world looks like



if P=NP, then P "grows" and "consumes" all of NP.

History / Why You Should Care