Complexity Theory

P, NP, and the Limits of Computation

Prof. Antonio Khalil Moretti

SCIS 313: Data Structures and Algorithm Analysis

Spelman College

The Million Dollar Question

Throughout this course:

- Analyzed algorithm efficiency
- Designed fast algorithms
- Compared approaches

But some problems seem fundamentally hard

- No known efficient algorithm
- Despite decades of trying
- Affects crypto, optimization, AI

P vs NP Problem:

Can every problem whose solution is quickly *verifiable* also be quickly *solvable*?

Millennium Prize Problem

$1,000,000 prize

Today:

- What makes problems "hard"?
- Classes: P, NP, NP-complete
- Practical implications

Review: Efficient = Polynomial Time

Polynomial time: $O(n^k)$ for constant k

Complexity	Example	Efficient?
$O(n)$	Linear search	Yes
$O(n \log n)$	Merge sort	Yes
$O(n^2)$	Bubble sort	Yes
$O(n^3)$	Matrix multiplication	Yes
$O(2^n)$	Subset sum (brute)	No
$O(n!)$	TSP (brute)	No

Key: Polynomial = feasible, Exponential = infeasible

Decision Problems

Focus on yes/no questions (simplifies theory)

Optimization:

"What is shortest path A to B?"

Decision:

"Is there path A to B with length $\leq k$?"

Relationship:

- Solving optimization $\rightarrow$ can answer decision (just compare)
- Solving decision $\rightarrow$ can find optimal (try different k values)

More examples:

Knapsack:

- Opt: "What's max value?"
- Decision: "Can get value $\geq V$?"

TSP:

- Opt: "Find shortest tour"
- Decision: "Tour with cost $\leq k$?"

If we can answer "Is there tour ≤ 100?", "Is there tour ≤ 50?", etc., we can binary search to find the minimum!

Class P: Polynomial Time

P = Problems solvable in polynomial time

Definition:

Set of decision problems solvable by deterministic algorithm in $O(n^k)$ time for some constant k

Intuition: P = "easy" problems

Examples in P:

- Sorting, searching
- Graph connectivity
- Shortest path
- MST
- Matching
- Linear programming
- Primality testing*

*Proven in 2002!

Careful: Input Size vs Input Value

For most problems: n = number of items

Sorting array:

- Input: [5, 2, 8, 1, 9]
- n = 5 elements
- Merge sort: $O(n \log n) = O(5 \log 5)$

Graph algorithms:

- Input: Graph with V vertices, E edges
- n = V (or V + E)
- BFS: $O(V + E)$

Input size grows linearly with number of items

For number problems: n = value, input size = log n

Primality testing:

- Input: "Is 1,000,000 prime?"
- Value: n = 1,000,000
- Input size: 20 bits (log 1,000,000)

Naive algorithm:

```
for i in 2..n:
    if n % i == 0: return false
return true
```

Takes $O(n) = O(1{,}000{,}000)$ steps

But input size = 20 bits!

$O(n) = O(2^{\text{input size}})$ - **exponential!**

AKS (2002): $O((\log n)^6)$ - truly polynomial

Class NP: Nondeterministic Polynomial

NP = Problems with polynomial-time verifiable solutions

Definition:

Set of decision problems where "yes" answer has a certificate (proof) verifiable in polynomial time

Key insight:

Verifying a solution $\neq$ **Finding** a solution

NP = "Nondeterministic Polynomial" (NOT "Non-Polynomial"!)

Example: Hamiltonian Path

Problem: Does graph have path visiting each vertex once?

Finding: Hard! Try all paths

Verifying: Easy! Given path, check:

- Visits each vertex once
- Edges exist

Time: $O(n)$ to verify

This is in NP!

What Does "Nondeterministic" Mean?

The magic guesser analogy:

Deterministic (normal):

Must try possibilities one by one

Finding Hamiltonian path:

- Try path 1: A→B→C... ×
- Try path 2: A→C→B... ×
- Try path 3: A→D→E... ×
- ...
- Try path $n!$: Found it! ✓

Exponential time!

We have: Deterministic algorithms

Nondeterministic:

Can "guess" right answer

1. **Guess:** A→D→E→B→C (instant!)
2. **Verify:** Check if valid (poly!)

If any guess leads to "yes", answer is "yes"

Like exploring all paths in parallel

We don't have: Magic guessers

So NP = "Can verify guessed solutions quickly"

Origin: Theoretical CS (1950s-60s) not quantum mechanics

What About Quantum Computers?

Common misconception: "Quantum computers solve NP!"

Reality:

Quantum uses superposition to explore multiple states

Sounds like nondeterministic...

But quantum $\neq$ nondeterministic!

Quantum CAN do:

- Factor numbers (Shor's)
- Search faster (Grover's)
- Some specific problems

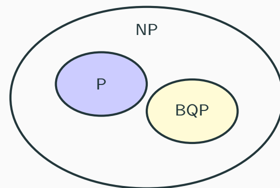
These are in BQP (Bounded Quantum Polynomial)

Quantum CANNOT do:

Solve all NP-complete efficiently

Believed: BQP $\neq$ NP

Quantum gives speedup, not magic



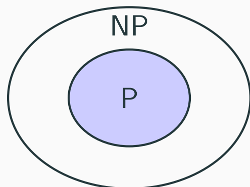
P, BQP overlap; neither contains other

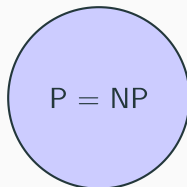
P vs NP: What We Know

Known: P $\subseteq$ NP

If we can solve in polynomial time, we can verify in polynomial time (just solve it!)

Unknown: Does P = NP?





P $\neq$ NP?

P = NP?

Most believe P $\neq$ NP

Why?

- Thousands of hard problems
- Decades of failed attempts
- Verifying seems easier than solving

But no proof!

If P = NP:

- Crypto breaks
- Optimization becomes easy
- Math proofs automated

NP-Complete: The Hardest Problems in NP

Definition:

Problem is NP-complete if:

1. It's in NP
2. Every problem in NP can be reduced to it in poly time

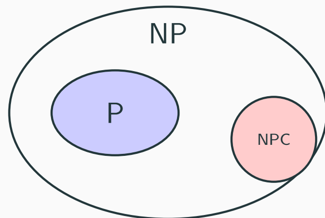
Intuition:

NP-complete = hardest problems in NP

If you solve one NP-complete problem efficiently, you solve ALL of NP!

Key property:

If ANY NP-complete problem is in P, then P = NP



NP-Complete sits at the boundary

Example: 3-SAT (Boolean Satisfiability)

Problem: Given boolean formula in CNF, can variables make it true?

Small example (3 variables):

$(x_1 \lor x_2 \lor \neg x_3) \land (\neg x_1 \lor x_3 \lor x_4) \land (x_2 \lor \neg x_3 \lor \neg x_4)$

With 4 variables: $2^4 = 16$ assignments

Solution:

$x_1 = T, x_2 = T, x_3 = F, x_4 = T$

Clause 1: $T \lor T \lor T = T$ ✓

Clause 2: $F \lor F \lor T = T$ ✓

Clause 3: $T \lor T \lor F = T$ ✓

Verifying: Check 3 clauses, $O(n)$!

Example: 3-SAT (Boolean Satisfiability)

Problem: Given boolean formula in CNF, can variables make it true?

Small example (3 variables):

$(x_1 \lor x_2 \lor \neg x_3) \land (\neg x_1 \lor x_3 \lor x_4) \land (x_2 \lor \neg x_3 \lor \neg x_4)$

With 4 variables: $2^4 = 16$ assignments

Solution:

$x_1 = T, x_2 = T, x_3 = F, x_4 = T$

Clause 1: $T \lor T \lor T = T$ ✓

Clause 2: $F \lor F \lor T = T$ ✓

Clause 3: $T \lor T \lor F = T$ ✓

Verifying: Check 3 clauses, $O(n)$!

Realistic instance (100 variables):

$(x_1 \lor x_7 \lor \neg x_{23}) \land$

$(x_{45} \lor \neg x_2 \lor x_{89}) \land$

$\vdots$ (300 clauses)

$\vdots$

Finding: Try $2^{100} \approx 10^{30}$ assignments!

If checking 1 billion/second:

$10^{30}/10^9 = 10^{21}$ seconds

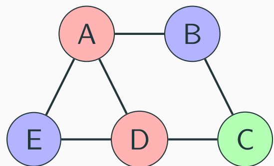
$=$ 31 trillion years (age of universe: 14 billion years!)

But verifying: Check 300 clauses in milliseconds!

Example: Graph Coloring

Problem: Can graph be colored with k colors (no adjacent vertices same color)?

Small: 3-colorable

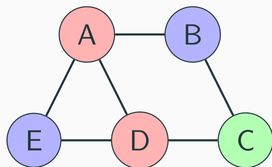


Easy to verify colors work!

Example: Graph Coloring

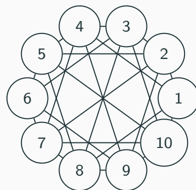
Problem: Can graph be colored with k colors (no adjacent vertices same color)?

Small: 3-colorable



Easy to verify colors work!

Large: harder to find!



With n vertices, k colors: k^n assignments!

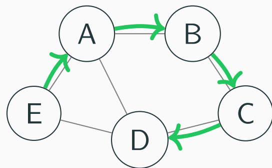
10 vertices, 3 colors = 59,049 to check

Finding: Exponential. **Verifying:** Check each edge, $O(|E|)$!

Example: Hamiltonian Path

Problem: Does graph have path visiting each vertex exactly once?

Small graph - has path:



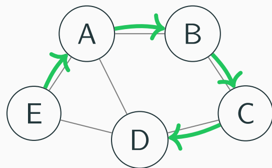
Path: E→A→B→C→D ✓

Verifying: Check 5 vertices visited once, edges exist. $O(n)$!

Example: Hamiltonian Path

Problem: Does graph have path visiting each vertex exactly once?

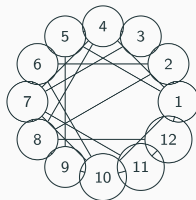
Small graph - has path:



Path: E→A→B→C→D ✓

Verifying: Check 5 vertices visited once, edges exist. $O(n)$!

Large graph - harder to find!



Finding: Try $n!$ orderings!

12 vertices = 479 million paths

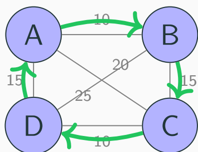
20 vertices = 2.4 quintillion paths!

But given a path, verify in seconds!

Example: Traveling Salesman Problem (TSP)

Problem: Visit all cities exactly once, return to start, minimize total distance

Small example (4 cities):



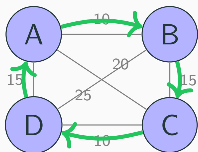
Only 3 distinct tours to check!

Best: A→B→C→D→A (cost 50)

Example: Traveling Salesman Problem (TSP)

Problem: Visit all cities exactly once, return to start, minimize total distance

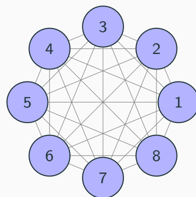
Small example (4 cities):



Only 3 distinct tours to check!

Best: A→B→C→D→A (cost 50)

Larger example (8 cities):



2,520 distinct tours!

10 cities: 181,440 tours

15 cities: 43 billion tours!

Finding: Must check exponentially many. **Verifying:** Just add up distances, $O(n)$!

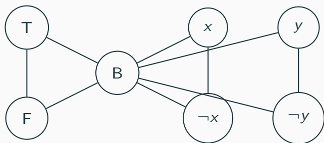
Reductions: Example - 3-SAT to Graph Coloring

Idea: Show 3-SAT $\leq_P$ Graph Coloring (if we can solve coloring, we can solve 3-SAT)

Given 3-SAT formula:

$(x \lor y \lor \neg z)$

Build graph: (simplified)



T, F, B use 3 colors

Each variable and negation form pair (opposite colors)

Key property:

3-SAT satisfiable $\iff$ graph 3-colorable!

If satisfiable:

- True literals get color T
- False literals get color F
- Can 3-color the graph

If 3-colorable:

- Variables with color T = true
- Variables with color F = false
- Satisfies formula

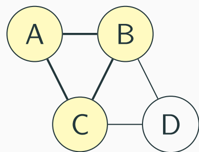
Since 3-SAT is NP-complete, graph

More NP-Complete Problems

Thousands proven through reduction chains!

Clique:

Does graph have complete subgraph of size k?



Clique of size 3: {A,B,C}

Vertex Cover:

Can k vertices cover all edges?

Subset Sum:

Given numbers, does subset sum to T?

Example: {3, 7, 2, 5}, $T = 12$?

Yes: $\{7, 5\}$ or $\{3, 7, 2\}$

Partition:

Can set be split into equal sums?

Hamiltonian Path:

Path visiting each vertex exactly once?

All NP-complete via reductions!

If P = NP: The World Would Change

If someone proves P = NP tomorrow...

Cryptography collapses:

Modern internet security relies on:

- Factoring large numbers is hard
- RSA encryption: easy to encrypt, hard to decrypt without key

If P = NP: Breaking encryption becomes easy!

- No more secure online shopping
- No more private communication
- Banking systems vulnerable
- National security at risk

Would need entirely new cryptography!

But also amazing breakthroughs:

Drug design: Protein folding becomes easy → cure diseases

AI & optimization: Every scheduling, routing, packing problem solved optimally

Mathematics: Automated theorem proving → new discoveries

Traveling salesman: Perfect delivery routes instantly

Most believe P ≠ NP because:

- Decades of brilliant minds failed
- Verifying truly seems easier than solving

What Does This Mean Practically?

If you encounter NP-complete problem:

Don't try to find poly-time algorithm!
(Probably doesn't exist)

Instead:

- Use approximation algorithms
- Use heuristics (may not be optimal)
- Solve special cases efficiently
- Accept exponential for small inputs
- Use randomization

Real-world examples:

Traveling Salesman:

- Approximation: within $2\times$ optimal
- Heuristics: nearest neighbor
- Works well in practice

SAT Solvers:

- Exponential worst-case
- Modern solvers handle millions of variables
- Used in chip design, AI planning

Knowing a problem is NP-complete guides approach!

Beyond P and NP

Class	Description
P	Solvable in polynomial time
NP	Verifiable in polynomial time
NP-Complete	Hardest problems in NP
NP-Hard	At least as hard as NP (may not be in NP)
co-NP	Complement of NP ("no" answers verifiable)
PSPACE	Solvable with polynomial space
EXPTIME	Solvable in exponential time

Known: P $\subseteq$ NP $\subseteq$ PSPACE $\subseteq$ EXPTIME

Unknown: Whether any of these are equal!

Beyond NP: Undecidable Problems

Some problems are impossible to solve algorithmically!

Halting Problem:

Given program and input, does it halt?

Turing (1936): This is *undecidable*

No algorithm can solve it for all inputs!

Proof sketch (contradiction):

1. Assume halting solver H exists
2. Build program P that does opposite
3. Ask: Does P(P) halt?
4. Contradiction!

Other undecidable problems:

- Does program output "Hello"?
- Are two programs equivalent?
- Does equation have integer solutions? (Hilbert's 10th)

Hierarchy:

Decidable

P $\subseteq$ NP $\subseteq$ ...

Undecidable

Halting, ...

Summary

Complexity classes:

- **P:** Efficiently solvable
- **NP:** Efficiently verifiable
- **NP-Complete:** Hardest in NP

Open question: P = NP?
Most believe no, but unproven

Practical impact:
Recognizing NP-complete problems guides algorithm design

When faced with hard problem:

- Is it in P? Great!
- Is it NP-complete?
  - Use approximations
  - Use heuristics
  - Solve special cases
- Is it undecidable? Can't solve in general

Understanding limits is as important as designing algorithms!

Course Journey

What we've covered:

Analysis: Big-O, recurrences, amortized analysis

Data Structures: Trees, graphs, balanced trees, heaps, hash tables

Algorithm Design: Greedy, dynamic programming, divide and conquer

Theory: Complexity classes, limits of computation

Key takeaways:

- Algorithm choice matters enormously
- Trade-offs everywhere: time vs space, optimal vs fast
- Some problems are inherently hard
- Understanding complexity guides design

You now think like a computer scientist!