P vs. NP

Data Structures and Parallelism

Announcements

Review session in CSE 403 tomorrow at noon.

P3 due tonight.
Exercise 12 & Exercise redos are due tomorrow.

Remember to fill out the token form so we know what exercises you're redoing. Push to gitlab (for exercises 8-11) or submit to gradescope (all others) to submit.

If something weird happens with either of those, submit them to me in an email.

Announcements

Content from today is last stuff we'll test you on for the final.

On Wednesday we'll talk about what you need to know for the "real world" about P vs. NP.

MST lectures have been updated to correct minor bugs in pseudocode.

The way you executed the algorithm in lecture/section was correct, now the pseudocode will match what you did.

Please fill out course feedback forms!

A Longer Example

The best way to really see why topological sorts and strongly connected components are useful would be a bunch of examples.

You'd need to take 421 first.
The second best way is to see one example right now…

This problem doesn't *look like* it has anything to do with graphs

- no maps
- no roads
- no social media friendships

Nonetheless, a graph representation is the best one.

Example Problem: Final Creation

We have a list of types of problems we might want to put on the final.

- ForkJoin code, Hash tables, B-Trees, Graphs,…

To try to make you all happy, we might ask for your preferences. Each of you gives us two preferences of the form "I [do/don't] want a [] problem on the final" *

We'll assume you'll be happy if you get at least one of your two requests.

Final Creation Problem

Given: A list of 2 preferences per student.
Find: A set of questions so every student gets at least one of their preferences (or accurately report no such question set exists).

*This is NOT how I'm making the final.

Final Creation: Take 1

We have Q kinds of questions and S students.

What if we try every possible combination of questions.

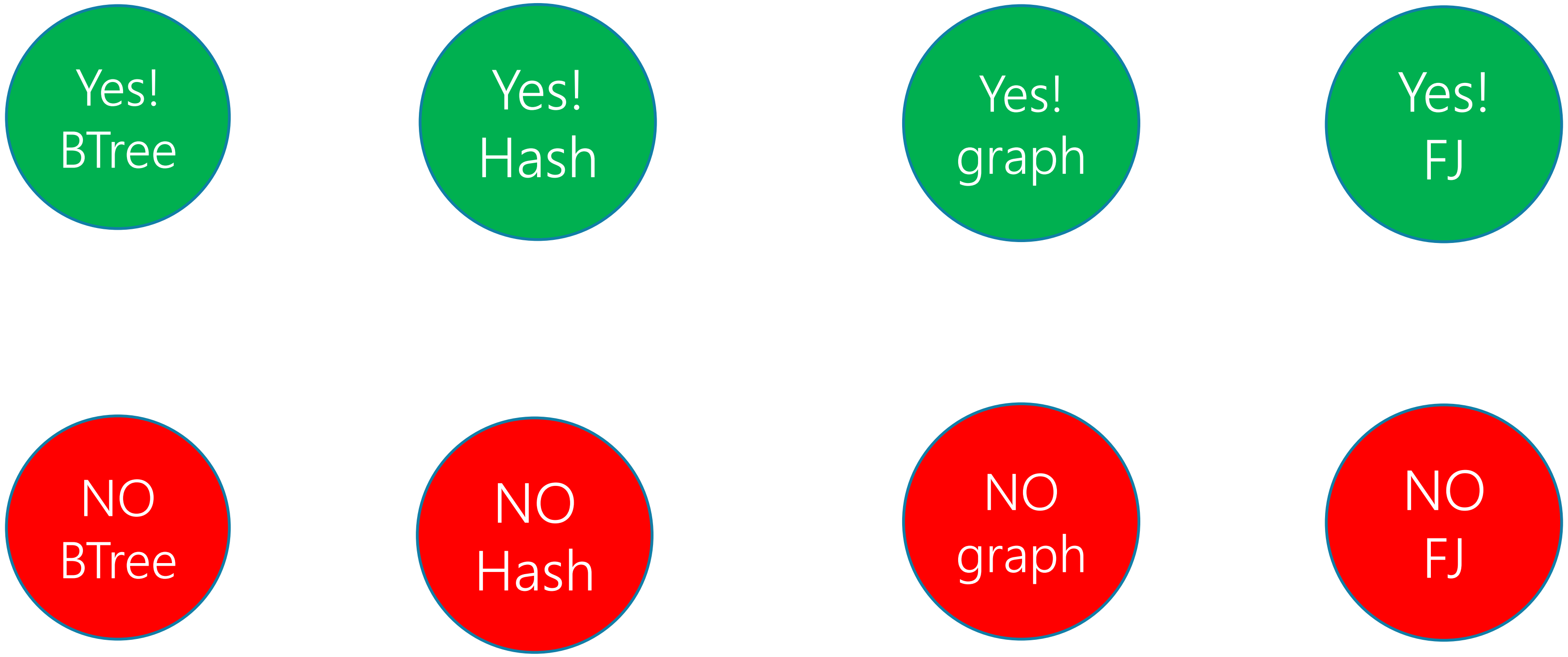
How long does this take? O($2^{Q}S$)

If we have a lot of questions, that's **really** slow.

Final Creation: Take 2

Each student introduces new relationships for data:

Let's say your preferences are in the top table:



Problem	YES	NO
B-Tree	X	
Hash Table		X
Graph		
Fork Join		

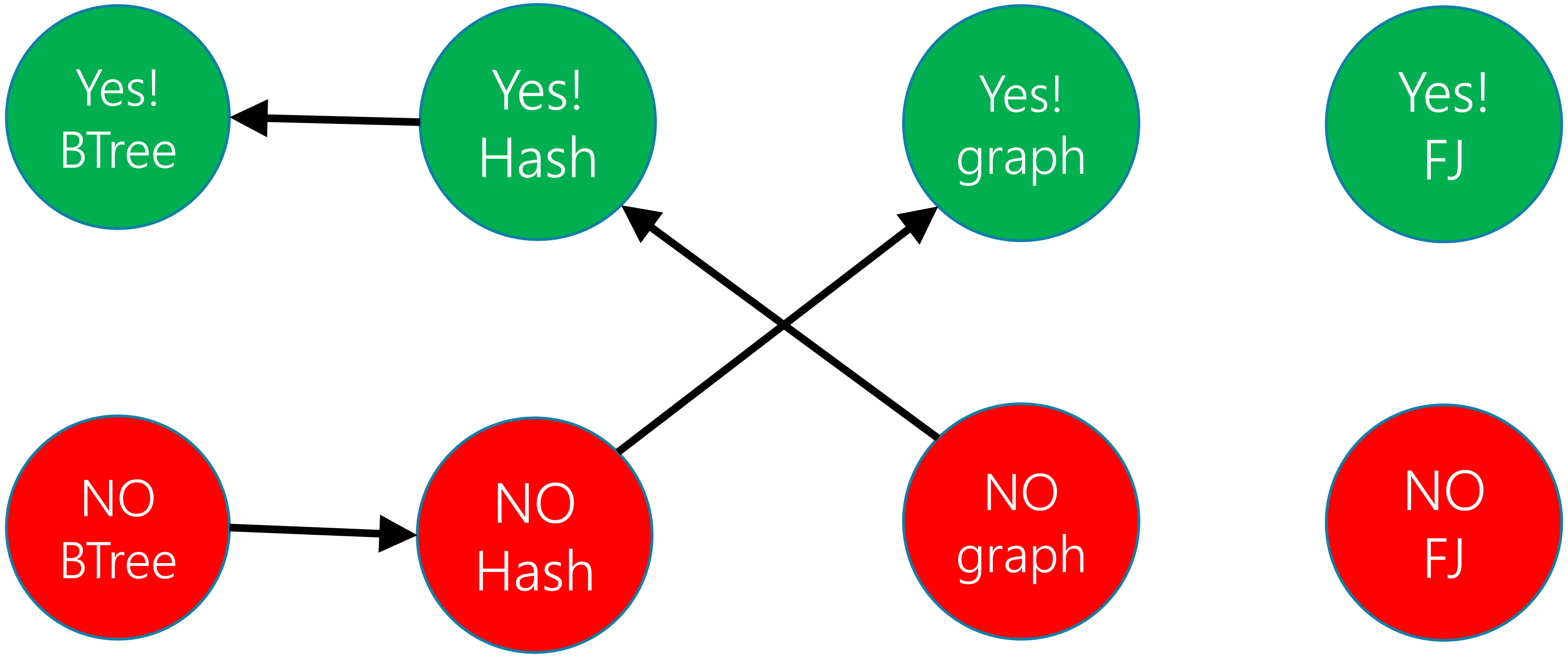
Problem	YES	NO
B-Tree		
Hash Table	X	
Graph	X	
Fork Join		

Final Creation: Take 2

Each student introduces new relationships for data:

Let's say your preferences are represented by this table:

Problem	YES	NO
B-Tree	X	
Hash Table		X
Graph		
Fork Join		



Problem	YES	NO
B-Tree		
Hash Table	X	
Graph	X	
Fork Join		

If we don't include a B-Trees, can you still be happy?
If we do include a hash tables can you still be happy?

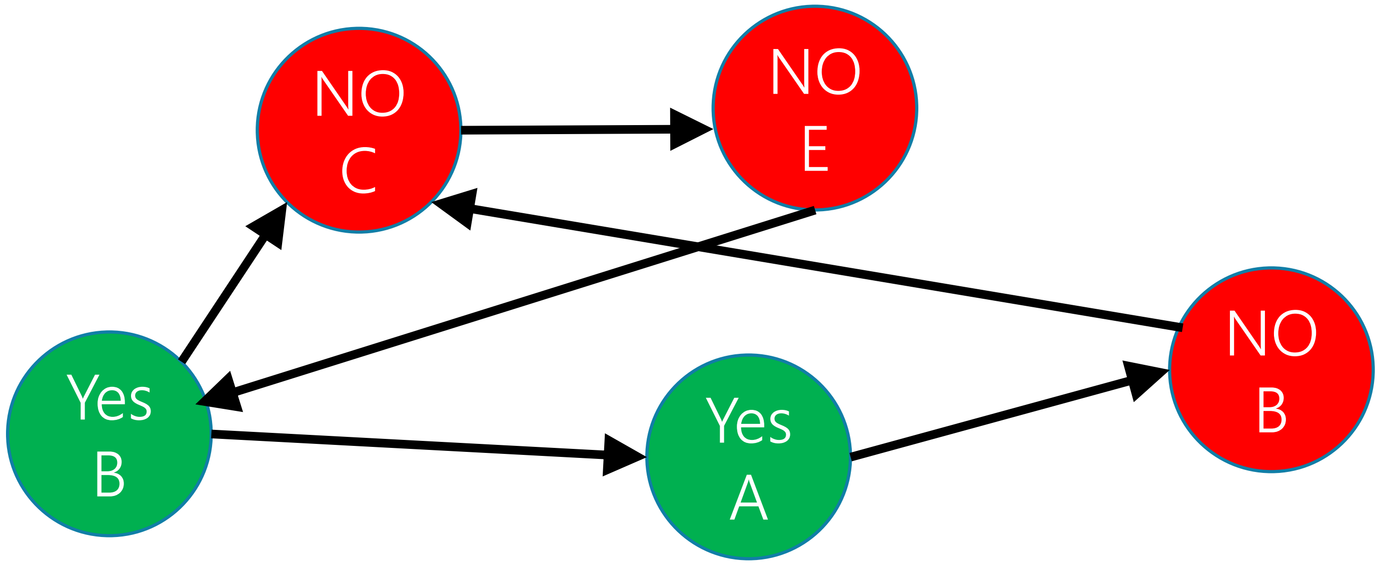
Final Creation: Take 2

Hey we made a graph!

What do the edges mean?

- We need to avoid an edge that goes TRUE THING → FALSE THING

Let's think about a single SCC of the graph.



Final Creation: SCCs

The vertices of a given SCC must either be all true or all false.

Algorithm Step 1: Run SCC on the graph. Check that each question-type-pair are in different SCC.

Now what? Every SCC gets the same value.
- Treat it as a single object!

We want to avoid edges from true things to false things.
- "Trues" seem more useful for us at the end.

Is there some way to start from the end?

YES! Topological Sort

Making the Final

Algorithm:
Make the requirements graph.

Find the SCCs.

If an SCC has including and not including a problem, no final is possible.

Run topological sort on the graph of SCC.

Starting from the end:

- if everything in a component is unassigned, set them to true, and set their opposites to false.
- Else If one thing in a component is assigned, assign the same value to the rest of the nodes in the component and the opposite value to their opposites.

Making The Final

This works!!

The proof is a bit more involved. Just trust me.

How fast is it?

O(Q + S). That’s a HUGE improvement.

Some More Context

The Final Making Problem was a type of “Satisfiability” problem.

We had a bunch of variables (include/exclude this question), and needed to satisfy everything in a list of requirements.

Because every requirement was “do at least one of these 2” this was a 2-SAT instance.

What happens if we change the 2 to a 3?

Graph algorithm doesn’t seem to work…

P vs. NP

Taking a Big Step Back

What has this quarter been about?

We’ve taken problems you probably knew how to solve slowly,

And we figured out how to solve them faster.

In some sense, that’s the job of a computer scientist.

Figure out how to take our problems

And make the computer do the hard work for us.

Taking a Big Step Back

Let’s take a big step back, and try to break problems into two types:

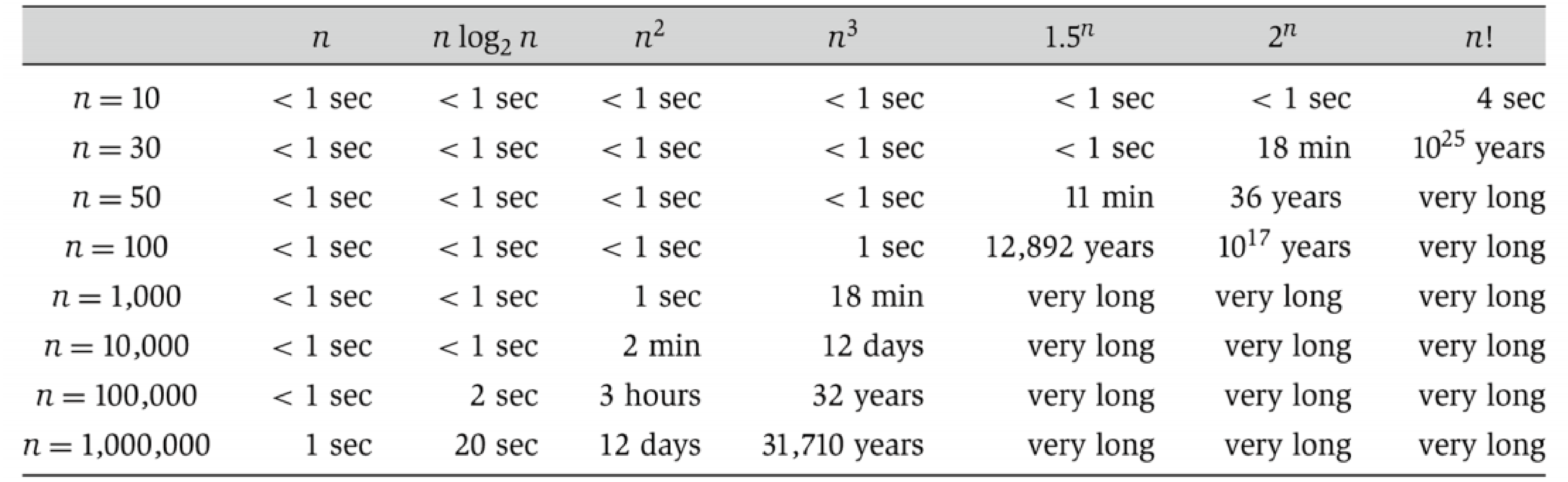
Those for which a computer might be able to help.

And those which would take so long to solve **even on a computer** we wouldn’t expect to solve them.

This is not the same as asking for undecideable problems (like the Halting Problem you saw in 311).

There are problems we could solve in finite time…but we’ll all be long dead before our computer tells us the answer.

Running Times



	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

(somewhat old) table from Rosen. How big of a problem can we solve for an algorithm with the given running times.
"very long" means more than 10^{25} years.

Efficient

We'll consider a problem "efficiently solvable" if it has a polynomial time algorithm.

I.e. an algorithm that runs in time $O(n^k)$ where k is a constant.

Are these algorithms always actually efficient?

Well………no

Your n^{10000} algorithm or even your $2^{2^{2^{2^2}}} \cdot n^3$ algorithm probably aren't going to finish anytime soon.

But these edge cases are rare, and polynomial time is good as a low bar

- If we can't even find an n^{10000} algorithm, we're probably not

Decision Problems

Let’s go back to dividing problems into solvable/not solvable.
For today, we’re going to talk about **decision problems**.

Problems that have a “yes” or “no” answer.

Why?

Theory reasons (you’ll see Wednesday).

But also most problems can be rephrased as very similar decision problems.

E.g. instead of “find the shortest path from s to t” ask
Is there a path from s to t of length at most k?

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 all problems we've solved in this class are in P.

P is an example of a "complexity class"
A set of problems that can be solved under some limitations (e.g. with some amount of memory or in some amount of time).

I’ll know it when I see it.

Another class of problems we want to talk about.

“I’ll know it when I see it” Problems.

Decision Problems such that:

If the answer is YES, you can prove the answer is yes by

- Being given a “proof” or a “certificate”
- Verifying that certificate in polynomial time.

What certificate would be convenient for short paths?

- The path itself. Easy to check the path is really in the graph and really short.

I'll know it when I see it.

More formally,

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's a common misconception that NP stands for "not polynomial"
Please never ever ever ever say that.

Please.

Every time you do a theoretical computer scientist sheds a single tear.

(That theoretical computer scientist is me)

NP

We can **verify** YES instances of NP problems efficiently, but can we **decide** whether the answer is YES or NO efficiently?

I.e. can you bootstrap the ability to check a certificate into the ability to find a certificate?

We don’t know.

This is the P vs. NP problem.

P vs. NP

P (stands for "Polynomial")

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

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.

Claim: P $\subseteq$ NP (do you see why?)

Reductions

Let's say we want to prove that some problem in NP needs exponential time (i.e. that P is not equal to NP).

Ideally we'd start with a really hard problem in NP.

What does it mean for one problem to be harder than another?

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 polynomially many times

and runs at most polynomially many other operations.

Reductions

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 polynomially many times

and runs at most polynomially many other operations.

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

Let’s say we want to prove that some problem in NP needs exponential time (i.e. that P is not equal to NP).

Ideally we’d start with a really hard problem in NP.

What is the hardest problem in NP?

NP-complete

A problem B is NP-complete if B is in NP and
for all problems A in NP, A reduces to B in polynomial time.

NP-complete

An NP-complete problem is a hardest problem in NP.

Seems like the right place to start for proving P≠NP.

It's also the right place to start for proving P=NP.

A polynomial time algorithm for one NP-complete problem, gives you a polynomial time algorithm for **every** problem in NP.

Examples

There are literally thousands of NP-complete problems.
And some of them look weirdly similar to problems we do know 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

In P

Light Spanning Tree

Given a weighted graph, find a spanning tree (a set of edges that connect all vertices) of weight at most k.

NP-Complete

Traveling Salesperson

Given a weighted graph, find a tour (a walk that visits every vertex and returns to its start) of weight at most k.

The electric company just needs a greedy algorithm to lay its wires.
Amazon doesn’t know a way to optimally route its delivery trucks.

Examples

In P

2-SAT

Given a list of requirements, all of the form "at least one of two must be true"
Set variables so all requirements are satisfied.

NP-Complete

3-SAT

Given a list of requirements, all of the form "at least one of three must be true"
Set variables so all requirements are satisfied.

The "final creation problem" was just 2-SAT.
It didn't look like there was an easy way to solve it…but there was.
Is the same true of 3-SAT? We don't know.

NP-hard

One more class:

NP-hard

Problem B is NP-hard if
for all problems A in NP, A reduces to B in polynomial time.

An NP-hard problem need not be in NP.
Examples?
Find the "best possible" certificate for an NP-hard problem.

Instead of a path of length at least k, find the longest path.
Instead of a tour of weight at most k, find the shortest tour.

NP-hard

NP-hard

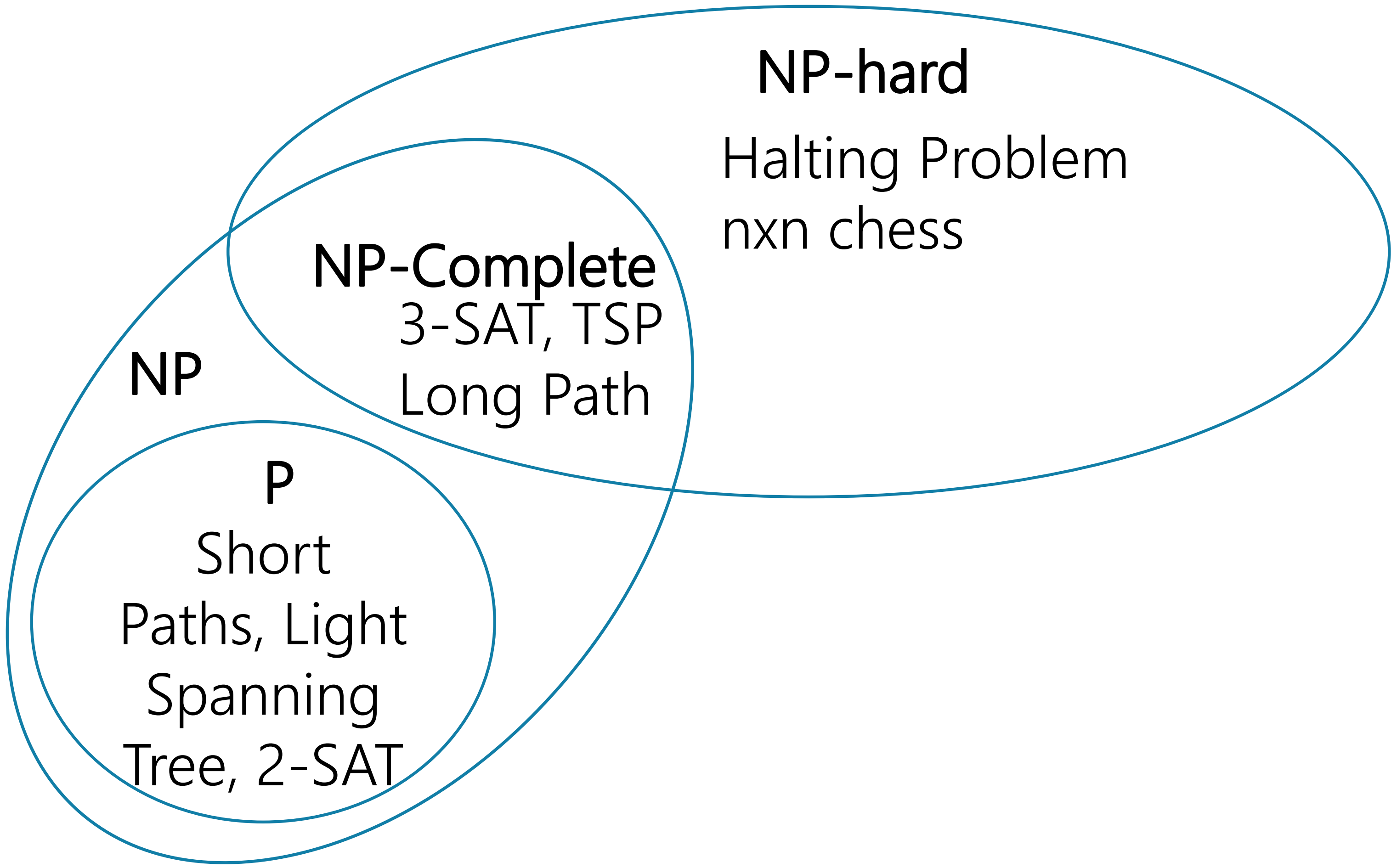
Problem B is NP-complete if
for all problems A in NP, A reduces to B in polynomial time.

Other Examples:
The halting problem is NP-hard (but not NP-complete).
So is n x n chess.

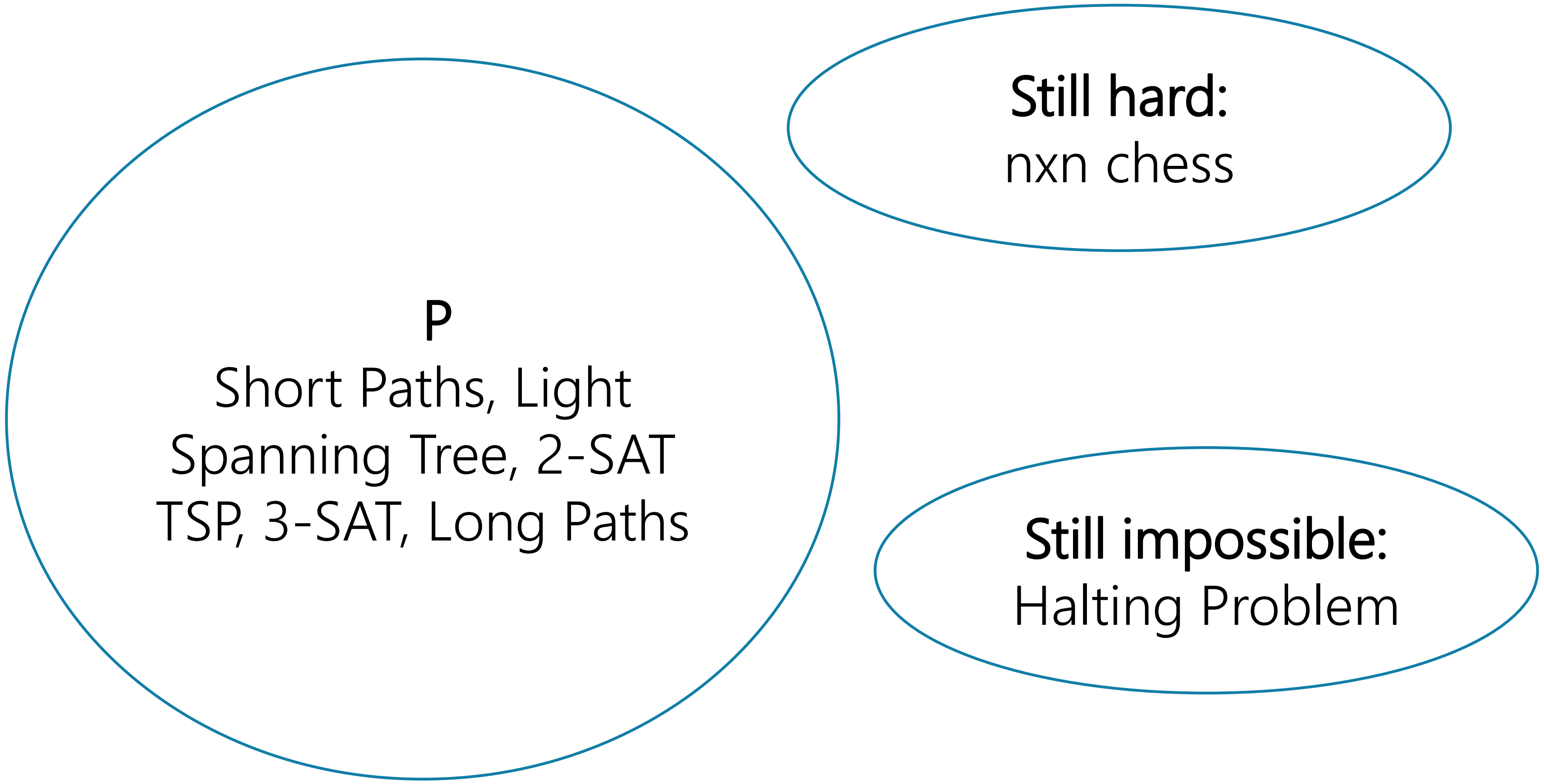
n x n Chess

Given an n x n chessboard, can white force a win with perfect play?

What The World Looks Like (We Think)



What The World Looks Like (If P=NP)


P
Short Paths, Light Spanning Tree, 2-SAT TSP, 3-SAT, Long Paths
Still hard:
nxn chess
Still impossible:
Halting Problem

Why P vs. NP matters

Not tested on final

P (stands for "Polynomial")

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

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.

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.

NP-hard

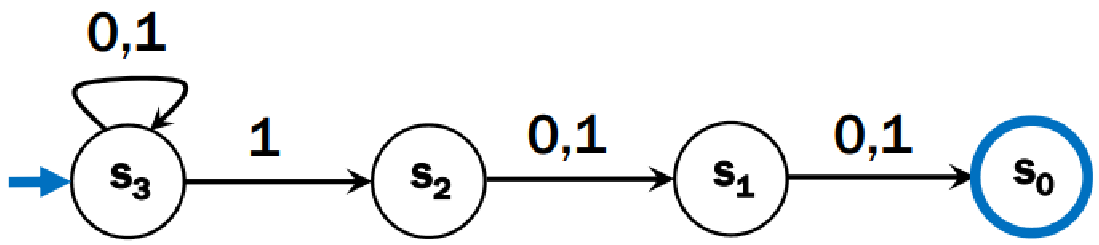
Problem B is NP-hard if
for all problems A in NP, A reduces to B in polynomial time.

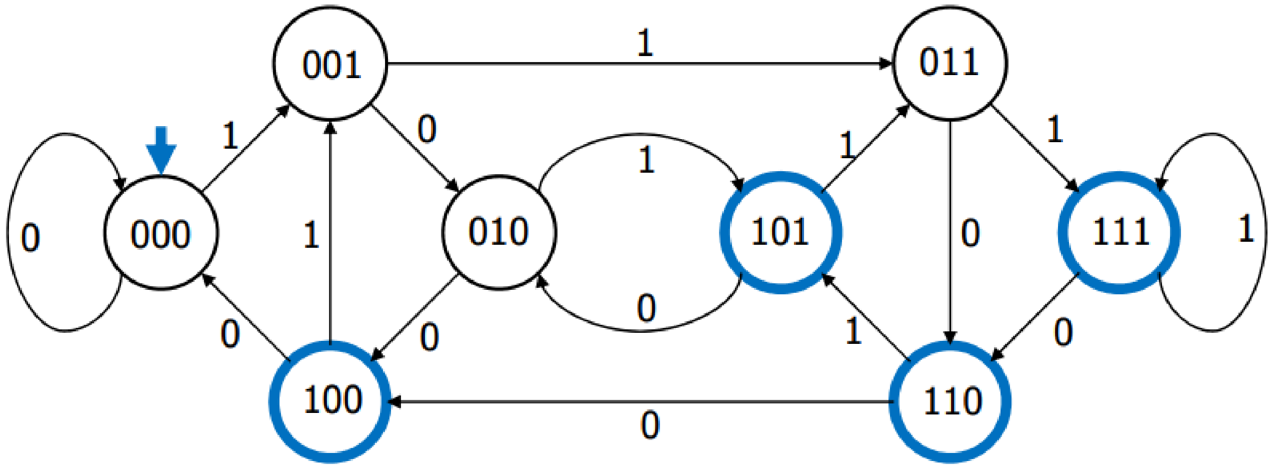
Why is it called NP?

You’ve seen nondeterministic computation before.
Way back in 311.

NFAs would “magically” decide among a set of valid transitions.
Always choosing one that would lead to an accept state (if such a transition exists).

An NFA and a DFA for the language “binary strings with a 1 in the 3rd position from the end.”





From Kevin & Paul’s 311 Lecture 23.

Nondeterminism

What would a nondeterministic **computer** look like?

It can run all the usual commands,

But it can also magically (i.e. nondeterministically) decide to set any bit of memory to 0 or 1.

Always choosing 0 or 1 to cause the computer to output YES,

(if such a choice exists).

If we had a nondeterministic computer…

Can you think of a polynomial time algorithm on a nondeterministic computer to:

Solve 2-SAT?

Solve 3-SAT?

If we had a nondeterministic computer…

Can you think of a polynomial time algorithm on a nondeterministic computer to:

Solve 2-SAT?

Just run our regular deterministic polynomial time algorithm

Or nondeterministically guess variable settings, output if they work.

Solve 3-SAT?

nondeterministically guess variable settings, output if they work.

Analogue of NFA/DFA equivalence

You showed in 311 that the set of languages decided by NFAs and DFAs were the same.

I.e. NFAs didn't let you solve more problems than DFAs.

But it did sometimes make the process a lot easier.

There are languages such that the best DFA is exponentially larger than the best NFA. (like the one from a few slides ago).

P vs. NP is an analogous question. Does non-determinism let us use exponentially fewer resources to solve some problems?

History, and Why P vs. NP?

Not tested on the final.

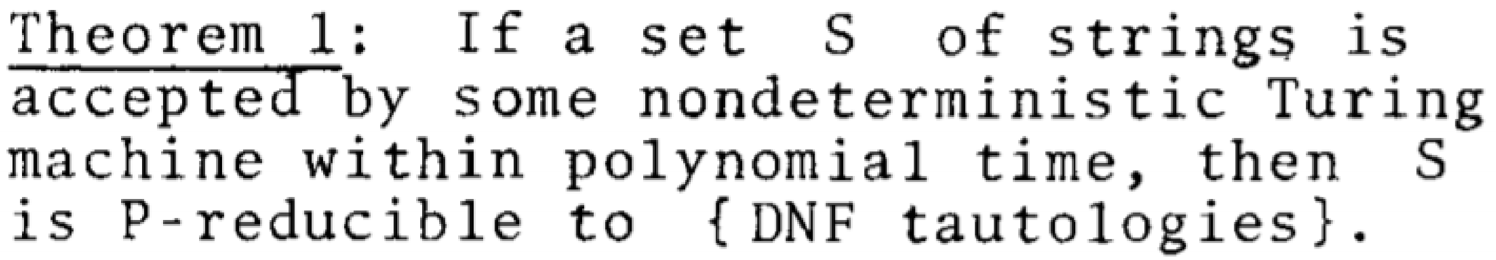
NP-Completeness

An NP-complete problem is a **universal language** for encoding "I'll know it when I see it" problems.

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

Cook-Levin Theorem (1971)

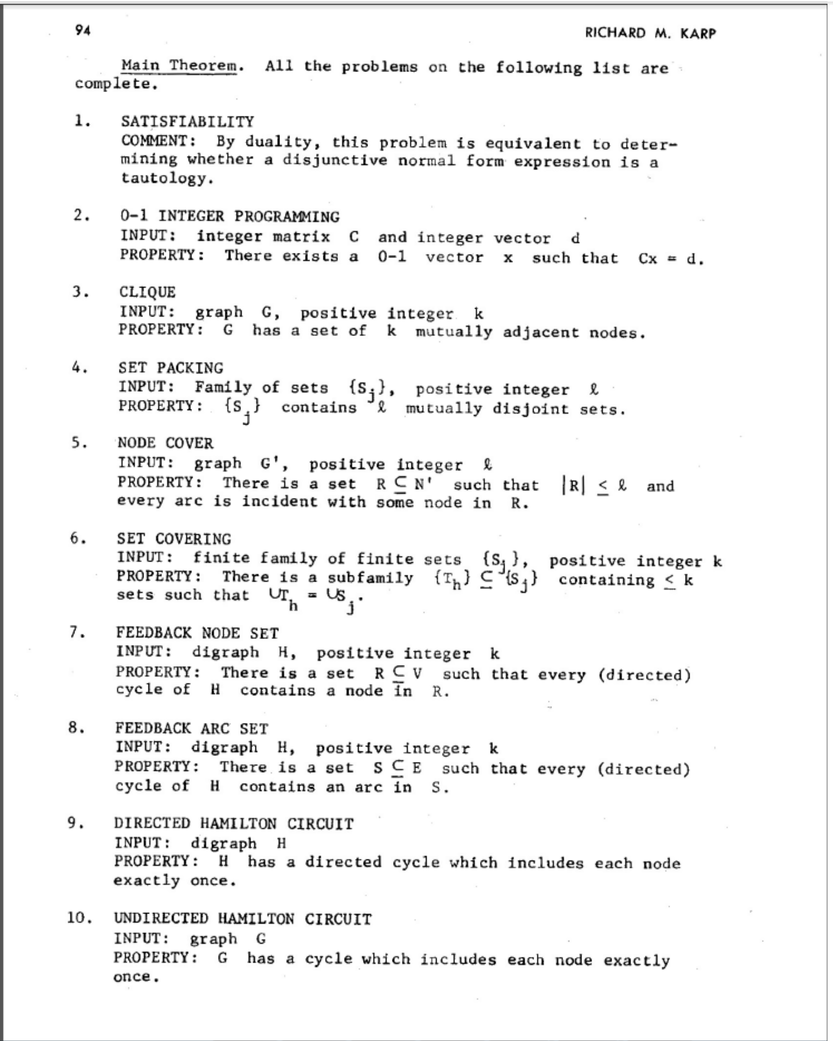
SAT is NP-complete



Theorem 1: If a set S of strings is accepted by some nondeterministic Turing machine within polynomial time, then S is P-reducible to {DNF tautologies}.

NP-Complete Problems

But Wait! There's more!



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 $|R| \leq \ell$ and every arc is incident with some node in R.

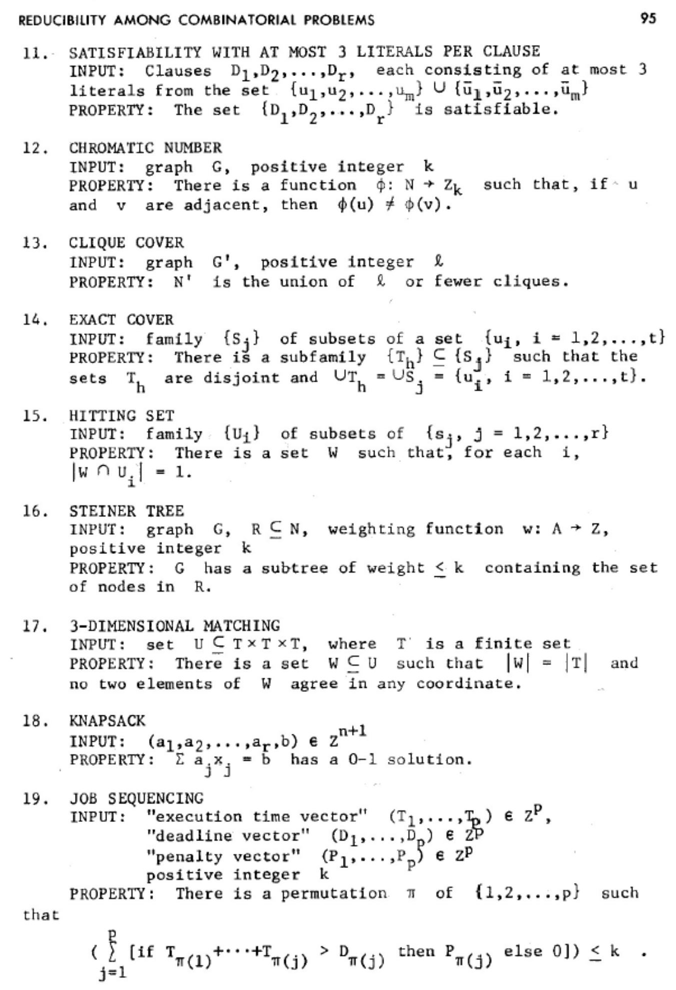
6. SET COVERING
INPUT: finite family of finite sets $\{S_j\}$, positive integer k
PROPERTY: There is a subfamily $\{T_h\} \subseteq \{S_j\}$ containing $\leq k$ sets such that $\cup T_h = \cup S_j$.

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 $\phi(u) \neq \phi(v)$.

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, $|W \cap U_i| = 1$.

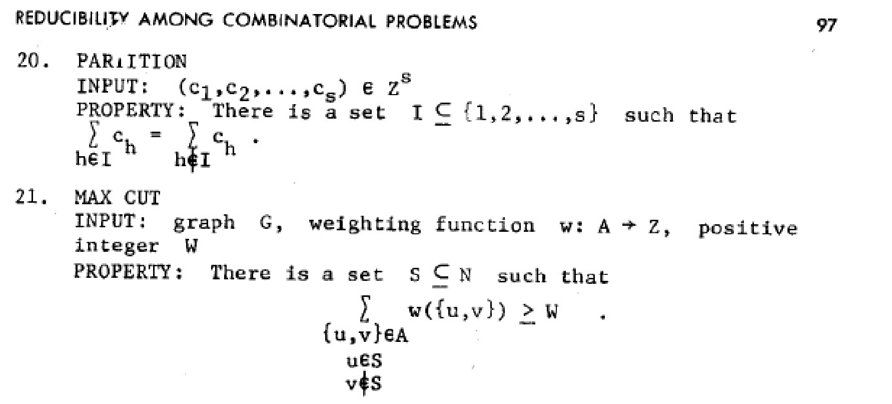
16. STEINER TREE
INPUT: graph G, $R \subseteq N$, weighting function $w: A \to Z$, positive integer k
PROPERTY: G has a subtree of weight $\leq k$ containing the set of nodes in R.

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

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



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

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

Karp's Theorem (1972)

A lot of problems people care about are NP-complete

NP-Complete Problems

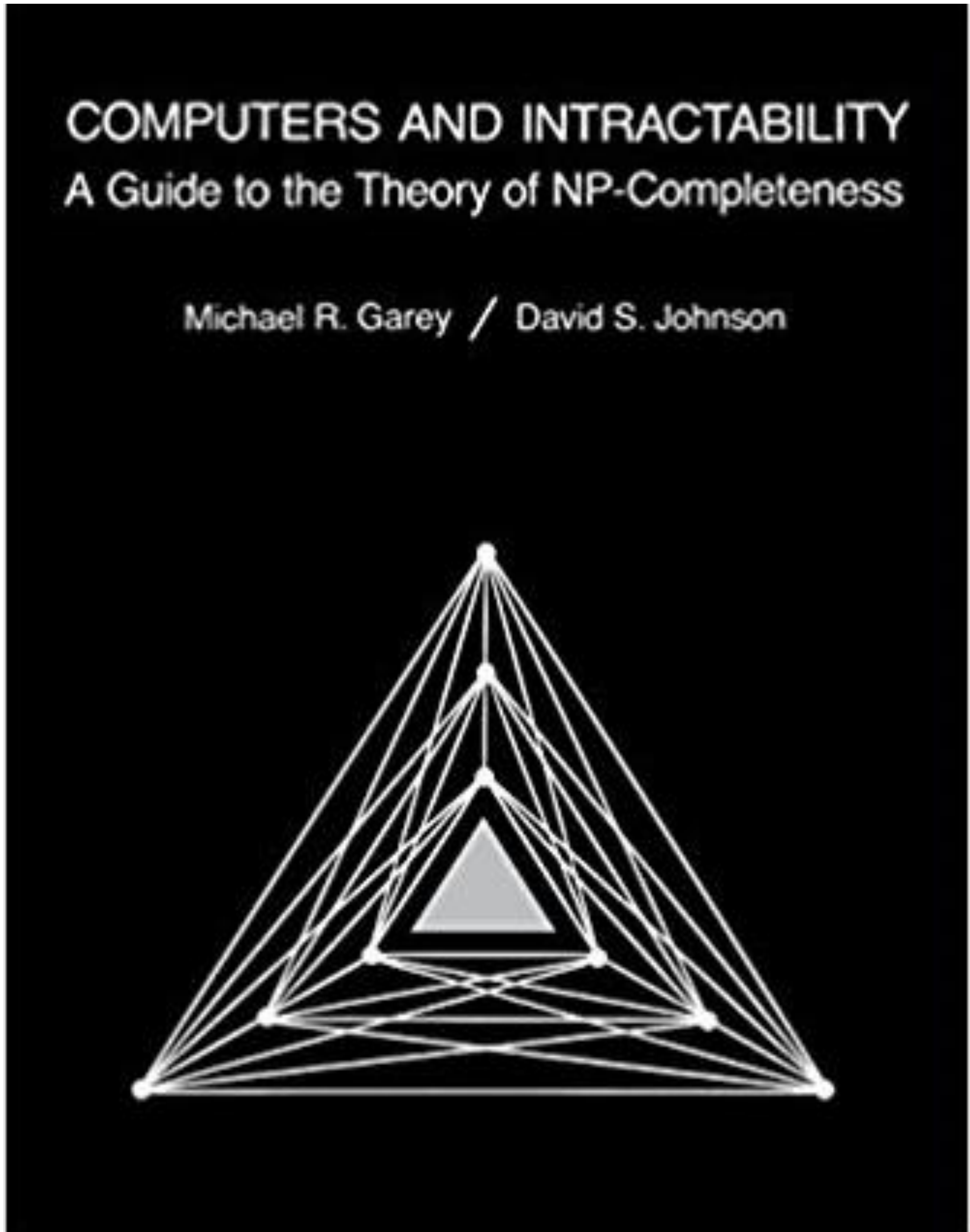
But Wait! There's more!

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

Garey and Johnson put a list of all the NP-complete problems they could find in this textbook.

Took them almost 100 pages to just list them all.

No one has made a comprehensive list since.


COMPUTERS AND INTRACTABILITY
A Guide to the Theory of NP-Completeness
Michael R. Garey / David S. Johnson

NP-Complete Problems

But Wait! There's more!

In the last month, mathematicians and computer scientists have put papers on the arXiv claiming to show (at least) 14 more problems are NP-complete.

If you spend enough time trying to use computers to solve your problems, you will run into an NP-complete problem sooner or later.

What do you do?

Dealing with NP-Completeness

Option 1: Maybe it's a special case we understand

Maybe you don't need to solve the general problem, just a special case

- 2-SAT vs. 3-SAT

Option 2: Maybe it's a special case we *don't* understand (yet)

There are algorithms that are known to run quickly on "nice" instances. Maybe your problem has one of those.

One approach: Turn your problem into a SAT instance, find a solver and cross your fingers.

Dealing with NP-Completeness

Option 3: Approximation Algorithms

You might not be able to get an exact answer, but you might be able to get close.

Optimization version of Traveling Salesperson

Given a weighted graph, find a tour (a walk that visits every vertex and returns to its start) of minimum weight.

Algorithm:
Find a minimum spanning tree.
Have the tour follow the visitation order of a DFS of the spanning tree.
Theorem: This tour is at most twice as long as the best one.

Why should you care about P vs. NP

Most computer scientists are convinced that P≠NP.

Why should you care about this problem?

It’s your chance for:

$1,000,000. The Clay Mathematics Institute will give $1,000,000 to whoever solves P vs. NP (or any of the 5 remaining problems they listed)

To get a Turing Award

Why should you care about P vs. NP

Most computer scientists are convinced that P≠NP.

Why should you care about this problem?

It’s your chance for:

$1,000,000. The Clay Mathematics Institute will give $1,000,000 to whoever solves P vs. NP (or any of the 5 remaining problems they listed)

To get ~~a Turing Award~~ the Turing Award renamed after you.

Why Should You Care if P=NP?

Suppose P=NP.

Specifically that we found a genuinely in-practice efficient algorithm for an NP-complete problem. What would you do?

- $1,000,000 from the Clay Math Institute obviously, but what's next?

Why Should You Care if P=NP?

We found a genuinely in-practice efficient algorithm for an NP-complete problem. What would you do?

- Another $5,000,000 from the Clay Math Institute
- Put mathematicians out of work.
- Decrypt (essentially) all current internet communication.
- No more secure online shopping or online banking or online messaging…or online *anything.*
- Maybe find the cure for cancer?
- A world where P=NP is a very very different place from the world we live in now.

Why Should You Care if P≠NP?

We already expect P≠NP. Why should you care when we finally prove it?

P≠NP says something fundamental about the universe.

For some questions there is not a clever way to find the right answer

- Even though you'll know it when you see it.

There is actually a way to obscure information.

Why Should You Care if P≠NP?

To prove P≠NP we need to better understand the differences between problems.

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

We don't care about P vs NP just because it has a huge effect about what the world looks like.

We will learn a lot about computation along the way.