CSE 417
Introduction to Algorithms
Winter 2009

NP-Completeness

(Chapter 8)

What can we feasibly compute?

Focus so far in the course has been to give good algorithms for specific problems (and general techniques that help do this).

Now shifting focus to problems where we think this is impossible.

A Brief History of Ideas

From Classical Greece, if not earlier, "logical thought" held to be a somewhat mystical ability

Mid 1800's: Boolean Algebra and foundations of mathematical logic created possible "mechanical" underpinnings

1900: David Hilbert's famous speech outlines program: mechanize all of mathematics?

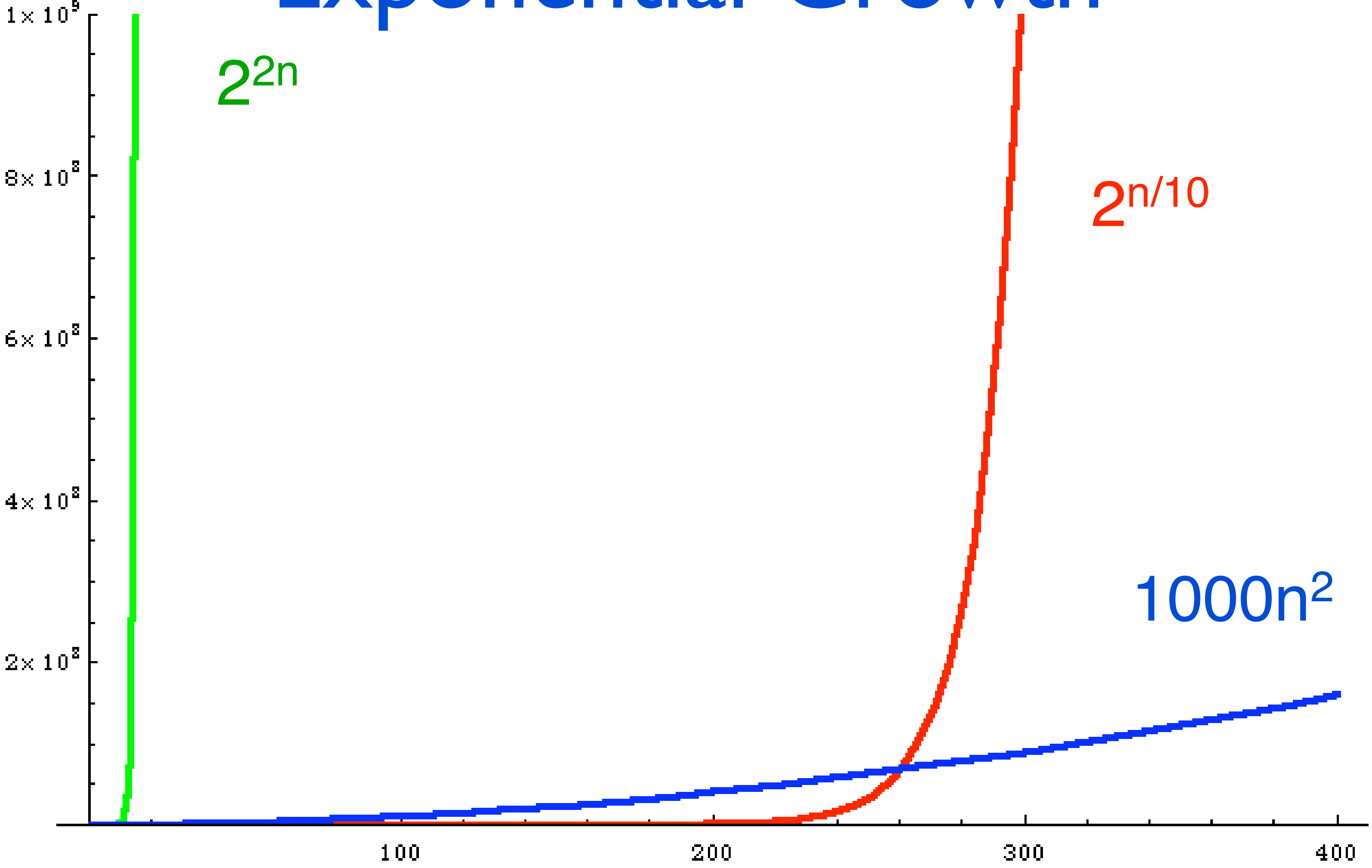
http://mathworld.wolfram.com/HilbertsProblems.html

1930's: Gödel, Church, Turing, et al. prove it's impossible

More History

- 1930/40's
  - What is (is not) computable
- 1960/70's
  - What is (is not) feasibly computable
  - Goal – a (largely) technology-independent theory of time required by algorithms
  - Key modeling assumptions/approximations
    - Asymptotic (Big-O), worst case is revealing
    - Polynomial, exponential time – qualitatively different

Polynomial vs Exponential Growth



Another view of Poly vs Exp

Next year's computer will be 2x faster. If I can solve problem of size n_0 today, how large a problem can I solve in the same time next year?

Complexity	Increase	E.g. $T=10^{12}$	
$O(n)$	$n_0 \rightarrow 2n_0$	10^{12}	2×10^{12}
$O(n^2)$	$n_0 \rightarrow \sqrt{2}\, n_0$	10^6	1.4×10^6
$O(n^3)$	$n_0 \rightarrow \sqrt[3]{2}\, n_0$	10^4	1.25×10^4
$2^{n/10}$	$n_0 \rightarrow n_0+10$	400	410
2^n	$n_0 \rightarrow n_0+1$	40	41

Polynomial versus exponential

We’ll say any algorithm whose run-time is

- polynomial is good
- bigger than polynomial is bad

Note – of course there are exceptions:

- n^{100} is bigger than $(1.001)^n$ for most practical values of n but usually such run-times don’t show up
- There are algorithms that have run-times like $O(2^{sqrt(n)/22})$ and these may be useful for small input sizes, but they're not too common either

Some Algebra Problems (Algorithmic)

Given positive integers a, b, c

Question 1: does there exist a positive integer x such that $ax = c$?

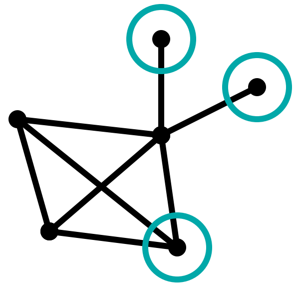
Question 2: does there exist a positive integer x such that $ax^2 + bx = c$?

Question 3: do there exist positive integers x and y such that $ax^2 + by = c$?

Some Problems

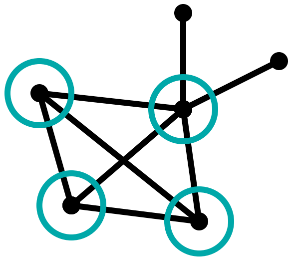
Independent-Set:

Given a graph G=(V,E) and an integer k, is there a subset U of V with |U| ≥ k such that no two vertices in U are joined by an edge.

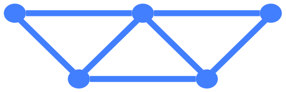
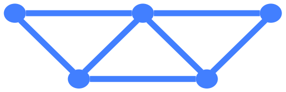
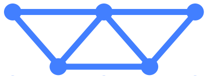
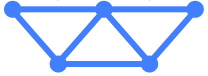


Clique:

Given a graph G=(V,E) and an integer k, is there a subset U of V with |U| ≥ k such that every pair of vertices in U is joined by an edge.



Some Convenient Technicalities

- "Problem" – the general case
  - Ex: The Clique Problem: Given a graph G and an integer k, does G contain a k-clique?
- "Problem Instance" – the specific cases
  - Ex: Does  contain a 4-clique? (no)
  - Ex: Does  contain a 3-clique? (yes)
- Decision Problems – Just Yes/No answer
- Problems as Sets of "Yes" Instances
  - Ex: CLIQUE = { (G,k) | G contains a k-clique }
    - E.g., ( , 4) $\notin$ CLIQUE
    - E.g., ( , 3) $\in$ CLIQUE

Decision problems

Computational complexity usually analyzed using decision problems

answer is just 1 or 0 (yes or no).

Why?

much simpler to deal with

deciding whether G has a k-clique, is certainly no harder than finding a k-clique in G, so a lower bound on deciding is also a lower bound on finding

Less important, but if you have a good decider, you can often use it to get a good finder. (Ex.: does G still have a k-clique after I remove this vertex?)

The class P

Definition: **P** = set of (decision) problems solvable by computers in *polynomial time*. i.e.,

$T(n) = O(n^k)$ for some fixed k (indp of input).

These problems are sometimes called *tractable* problems.

Examples: sorting, shortest path, MST, connectivity, RNA folding & other dyn. prog. – most of this qtr

(exceptions: Change-Making/Stamps, TSP)

Beyond P?

There are many natural, practical problems for which we don't know any polynomial-time algorithms

e.g. CLIQUE:

Given an undirected graph G and an integer k, does G contain a k-clique?

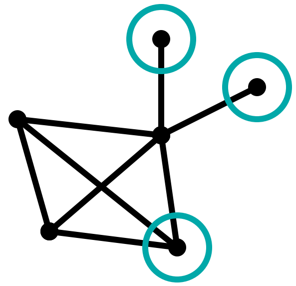
e.g. quadratic Diophantine equations:

Given a, b, $c \in N$, $\exists$ x, $y \in N$ s.t. $ax^2 + by = c$?

Some Problems

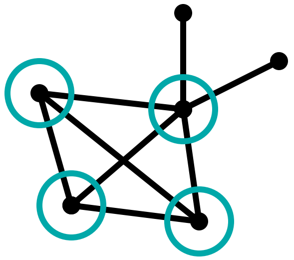
Independent-Set:

Given a graph G=(V,E) and an integer k, is there a subset U of V with |U| ≥ k such that no two vertices in U are joined by an edge.



Clique:

Given a graph G=(V,E) and an integer k, is there a subset U of V with |U| ≥ k such that every pair of vertices in U is joined by an edge.



Some More Problems

Euler Tour:

Given a graph G=(V,E) is there a cycle traversing each edge once.

Hamilton Tour:

Given a graph G=(V,E) is there a simple cycle of length |V|, i.e., traversing each vertex once.

TSP:

Given a weighted graph G=(V,E,w) and an integer k, is there a Hamilton tour of G with total weight ≤ k.

Shortest Path:

Given a digraph G=(V,E), a pair of vertices s,t in V and an integer k, is there a path from s to t of length at most k?

Satisfiability

Boolean variables $x_1, \ldots, x_n$

taking values in $\{0,1\}$. 0=false, 1=true

Literals

x_i or $\neg x_i$ for $i = 1, \ldots, n$

Clause

a logical OR of one or more literals

e.g. $(x_1 \vee \neg x_3 \vee x_7 \vee x_{12})$

CNF formula ("conjunctive normal form")

a logical AND of a bunch of clauses

Satisfiability

CNF formula example

$(x_1 \lor \neg x_3 \lor x_7) \land (\neg x_1 \lor \neg x_4 \lor x_5 \lor \neg x_7)$

If there is some assignment of 0's and 1's to the variables that makes it true then we say the formula is satisfiable

the one above is, the following isn't

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

Satisfiability: Given a CNF formula F, is it satisfiable?

Satisfiable?

$$\begin{aligned}
&(\ x \lor y \lor z\ ) \land (\ \neg x \lor y \lor \neg z\ ) \land \\
&(\ x \lor \neg y \lor z\ ) \land (\ \neg x \lor \neg y \lor z\ ) \land \\
&(\ \neg x \lor \neg y \lor \neg z\ ) \land (\ x \lor y \lor z\ ) \land \\
&(\ x \lor \neg y \lor z\ ) \land (\ x \lor y \lor \neg z\ )
\end{aligned}$$

$$\begin{aligned}
&(\ x \lor y \lor z\ ) \land (\ \neg x \lor y \lor \neg z\ ) \land \\
&(\ x \lor \neg y \lor \neg z\ ) \land (\ \neg x \lor \neg y \lor z\ ) \land \\
&(\ \neg x \lor \neg y \lor \neg z\ ) \land (\ \neg x \lor y \lor z\ ) \land \\
&(\ x \lor \neg y \lor z\ ) \land (\ x \lor y \lor \neg z\ )
\end{aligned}$$

More History – As of 1970

Many of the above problems had been studied for decades

All had real, practical applications

None had poly time algorithms; exponential was best known

But, it turns out they all have a very deep similarity under the skin

Some Problem Pairs

Euler Tour	Hamilton Tour
2-SAT	3-SAT
Min Cut	Max Cut
Shortest Path	Longest Path

Superficially different; similar computationally

Similar pairs; seemingly different computationally

Common property of these problems: Discrete Exponential Search Loosely–find a needle in a haystack

"Answer" is literally just yes/no, but there's always a somewhat more elaborate "solution" (aka "hint" or "certificate") that *transparently*‡ justifies each "yes" instance (and only those) – but it's *buried in an exponentially large search space of potential solutions.*

‡*Transparently* = verifiable in polynomial time

Example: Clique

“Is there a k-clique in this graph?”

- any subset of k vertices *might* be a clique
- there are *many* such subsets
- I only need to find one
- if I knew where it was, I could describe it succinctly, e.g. "look at vertices 2,3,17,42,...",
- I'd know one if I saw one: "yes, there are edges between 2 & 3, 2 & 17,... so it's a k-clique”
- And if there is *not* a k-clique, I wouldn’t be fooled by a statement like “look at vertices 2,3,17,42,...”

Example: Quad Diophantine Eqns

"Is there an integer solution to this equation?"

- any pair of integers x & y might be a solution
- there are lots of potential pairs
- I only need to find one such pair
- if I knew a solution, I could easily describe it, e.g. "try x=42 and y= 321" [A slight subtlety here: need to be sure there's a solution involving ints with only polynomially many digits...]
- I'd know one if I saw one: "yes, plugging in 42 for x & 321 for y I see ..."
- And wouldn't be fooled by (42,341) if there's no solution

Example: SAT

“Is there a satisfying assignment for this Boolean formula?”

- any assignment might work
- there are lots of them
- I only need one
- if I had one I could describe it succinctly, e.g., “x_1=T, x_2=F, ..., x_n=T"
- I'd know one if I saw one: "yes, plugging that in, I see formula = T...”
- And if the formula is unsatisfiable, I wouldn’t be fooled by , “x_1=T, x_2=F, ..., x_n=F"

The complexity class NP

NP consists of all decision problems where

> You can verify the YES answers efficiently (in polynomial time) given a short (polynomial-size) hint

one among exponentially many;
know it when you see it

And

> No hint can fool your polynomial time verifier into saying YES for a NO instance

(implausible for all exponential time problems)

Precise Definition of NP

A decision problem is in NP iff there is a polynomial time procedure v(-,-), and an integer k such that

> for every YES problem instance x there is a hint h with $|h| \leq |x|^k$ such that v(x,h) = YES
> and
>
> for every NO problem instance x there is no hint h with $|h| \leq |x|^k$ such that v(x,h) = YES

“Hints” sometimes called “Certificates”

Example: CLIQUE is in NP

```
procedure v(x,h)
    if
        x is a well-formed representation of a graph
        G = (V, E) and an integer k,
    and
        h is a well-formed representation of a k-vertex
        subset U of V,
    and
        U is a clique in G,
    then output "YES"
    else output "I'm unconvinced"
```

Is it correct?

For every x = (G,k) such that G contains a k-clique, there is a hint h that will cause v(x,h) to say YES, namely h = a list of the vertices in such a k-clique

and

No hint can fool v into saying yes if either x isn't well-formed (the uninteresting case) or if x = (G,k) but G does not have any cliques of size k (the interesting case)

Another example: SAT ∈ NP

Hint: the satisfying assignment A

Verifier: v(F,A) = syntax(F,A) && satisfies(F,A)

- Syntax: True iff F is a well-formed formula & A is a truth-assignment to its variables
- Satisfies: plug A into F and evaluate

Correctness:

- If F is satisfiable, it has some satisfying assignment A, and we'll recognize it
- If F is unsatisfiable, it doesn't, and we won't be fooled

Keys to showing that a problem is in NP

What's the output? (must be YES/NO)

What's the input? Which are YES?

For every given YES input, is there a hint that would help? Is it polynomial length?

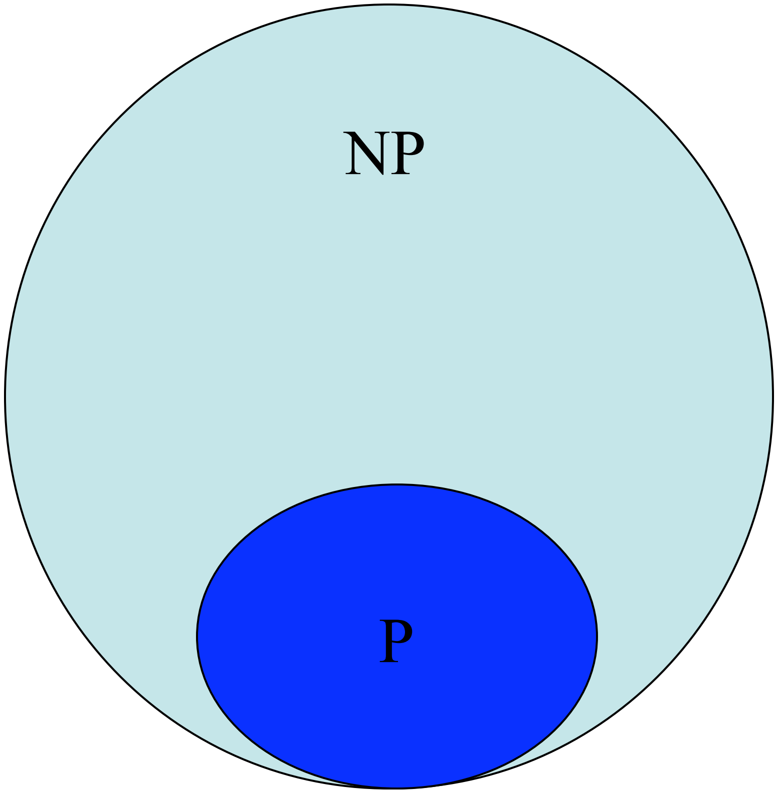
OK if some inputs need no hint

For any given NO input, is there a hint that would trick you?

Complexity Classes

NP = Polynomial-time **verifiable**

P = Polynomial-time **solvable**



Solving NP problems without hints

The most obvious algorithm for most of these problems is brute force:

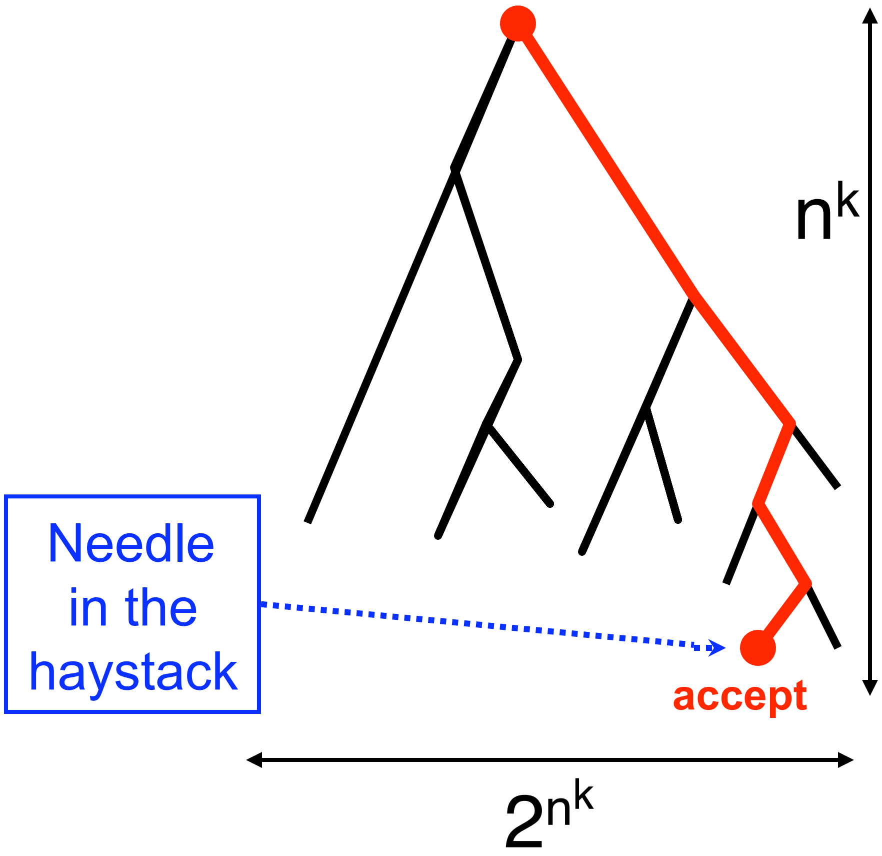
- try all possible hints; check each one to see if it works.
- Exponential time:
  - 2^n truth assignments for n variables
  - n! possible TSP tours of n vertices
  - $\binom{n}{k}$ possible k element subsets of n vertices
  - etc.

…and to date, every alg, even much less-obvious ones, are slow, too

P vs NP vs Exponential Time

Theorem: Every problem in NP can be solved deterministically in exponential time

Proof: “hints” are only n^k long; try all 2^{n^k} possibilities, say by backtracking. If any succeed, say YES; if all fail, say NO.



P and NP

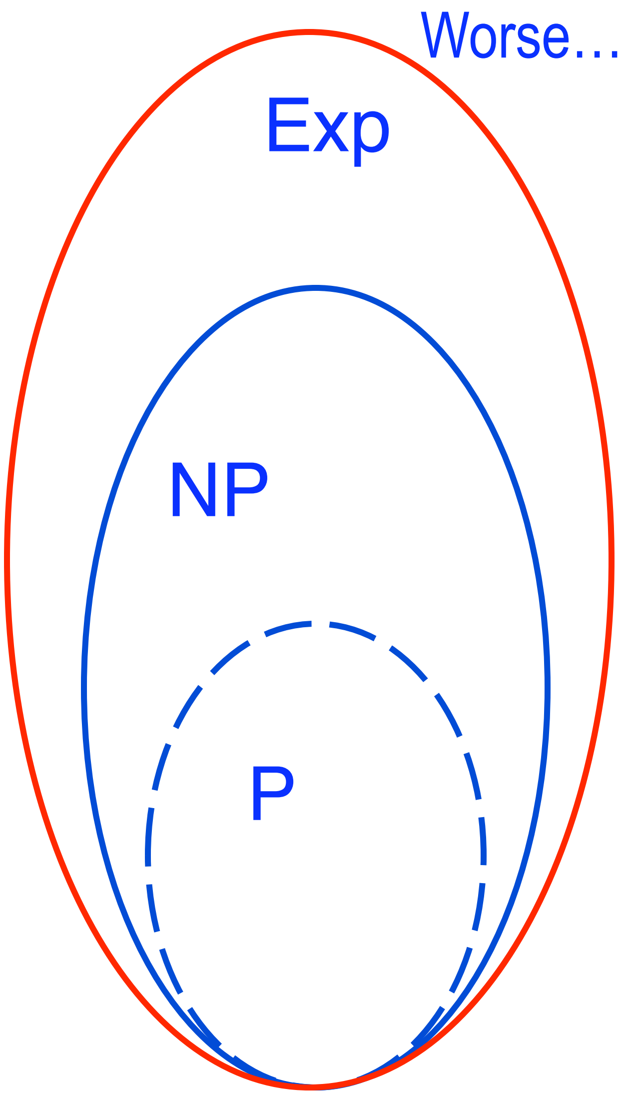
Every problem in P is in NP

one doesn't even need a hint for problems in P so just ignore any hint you are given

Every problem in NP is in exponential time

I.e., $P \subseteq NP \subseteq Exp$

We know $P \neq Exp$, so either $P \neq NP$, or $NP \neq Exp$ (most likely both)



P vs NP

Theory

- P = NP ?
- Open Problem!
- I bet against it

Practice

- Many interesting, useful, natural, well-studied problems known to be NP-complete
- With rare exceptions, no one routinely succeeds in finding exact solutions to large, arbitrary instances

Shortest Path

"Is there a short path (< k) from A to B in this graph?”

- Any path might work
- There are lots of them
- I only need one
- If I knew one I could describe it succinctly, e.g., "go from A to node 2, then node 42, then ... ”
- I'd know one if I saw one: "yes, I see there's an edge from A to 2 and from 2 to 42... and the total length is < k”
- And if there isn’t a short path, I wouldn’t be fooled by, e.g., "go from A to node 2, then node 42, then ... ”

Longest Path

"Is there a long path (> k) from A to B in this graph?"

- Any path might work
- There are lots of them
- I only need one
- If I knew one I could describe it succinctly, e.g., "go from A to node 2, then node 42, then ... "
- I'd know one if I saw one: "yes, I see there's an edge from A to 2 and from 2 to 42... and the total length is > k"
- And if there isn't a long path, I wouldn't be fooled by, e.g., "go from A to node 2, then node 42, then ... "

Mostly Long Paths

"Are the *majority* of paths from A to B long (>k)?"

Any path might work

Yes! → There are lots of ther

I only need one

If I knew one I ... it

succinctly, ... to node

2, then ...

I'd k... one: "yes, I

see an ... A to 2 and from

2 to 42... ...otal length > k"

And if there isn't a long path, I wouldn't be fooled …

This problem is not believed to be in NP; probably *harder*

No, this is a *collective* property of the set of all paths in the graph, and no one path overrules the rest

Problems in P can also be verified in polynomial-time

Short Path: Given a graph G with edge lengths, is there a path from s to t of length ≤ k?

- Verify: Given a purported path from s to t, is it a path, is its length ≤ k?

Small Spanning Tree: Given a weighted undirected graph G, is there a spanning tree of weight ≤ k?

- Verify: Given a purported spanning tree, is it a spanning tree, is its weight ≤ k?

(But the hints aren't really needed in these cases…)

NP: Summary so far

P = “poly time solvable”

NP = “poly time verifiable” *(nondeterministic poly time solvable)*

Defined only for decision problems, but fundamentally about search: can cast *many* problems as searching for a poly size, poly time verifiable “solution” in a 2^{poly} size “search space”.

Examples:

- is there a big clique? Space = all big subsets of vertices; solution = one subset; verify = check all edges
- is there a satisfying assignment? Space = all assignments; solution = one asgt; verify = eval formula

Sometimes we can do that quickly (is there a small spanning tree?); P = NP would mean we can *always* do that.

Does P = NP?

This is an open question.

To show that P = NP, we have to show that every problem that belongs to NP can be solved by a polynomial time deterministic algorithm.

Would be very cool, but no one has shown this yet.

(And it seems unlikely to be true.)

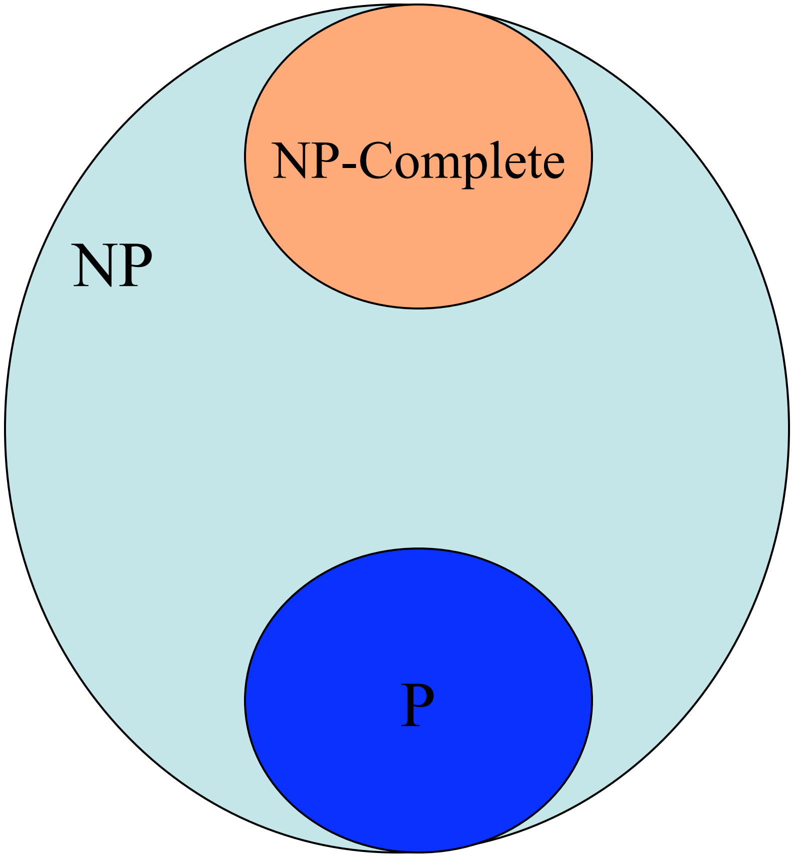
(Also seems daunting: there are infinitely many problems in NP; do we have to pick them off one at a time…?)

Complexity Classes

NP = Poly-time **verifiable**

P = Poly-time **solvable**

NP-Complete =
"Hardest" problems in **NP** (formal defn later)



Reductions: a useful tool

Definition: To reduce A to B means to solve A, given a subroutine solving B.

Example: reduce MEDIAN to SORT

Solution: sort, then select (n/2)nd

Example: reduce SORT to FIND_MAX

Solution: FIND_MAX, remove it, repeat

Example: reduce MEDIAN to FIND_MAX

Solution: transitivity: compose solutions above.

Reductions: Why useful

Definition: To reduce A to B means to solve A, given a subroutine solving B.

Fast algorithm for B implies fast algorithm for A

(nearly as fast; takes some time to set up call, etc.)

If every algorithm for A is slow, then no algorithm for B can be fast.

"complexity of A" ≤ *"complexity of B" + "complexity of reduction"*

SAT and 3SAT

Satisfiability: A Boolean formula in conjunctive normal form (CNF) is satisfiable if there exists an assignment of 0's and 1's to its variables such that the value of the expression is 1.

Example:

$$S=(x \vee y \vee \neg z) \wedge (\neg x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z)$$

Example above is satisfiable. (E.g., set $x=1$, $y=1$ and $z=0$.)

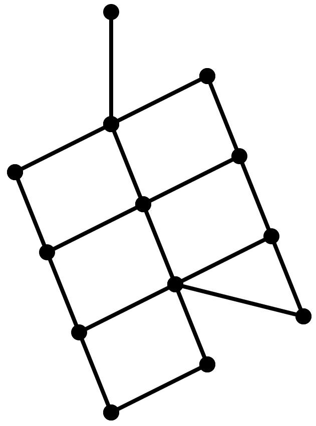
SAT = the set of satisfiable CNF formulas

3SAT = … having at most 3 literals per clause

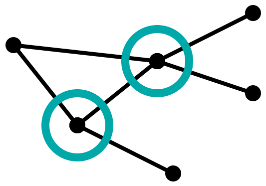
Another NP problem: Vertex Cover

Input: Undirected graph G = (V, E), integer k.

Output: True iff there is a subset C of V of size ≤ k such that every edge in E is incident to at least one vertex in C.

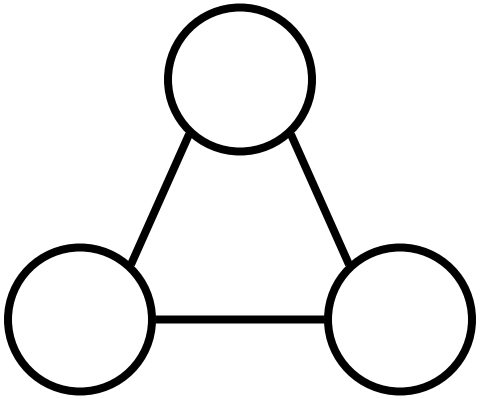


Example: Vertex cover of size ≤ 2.

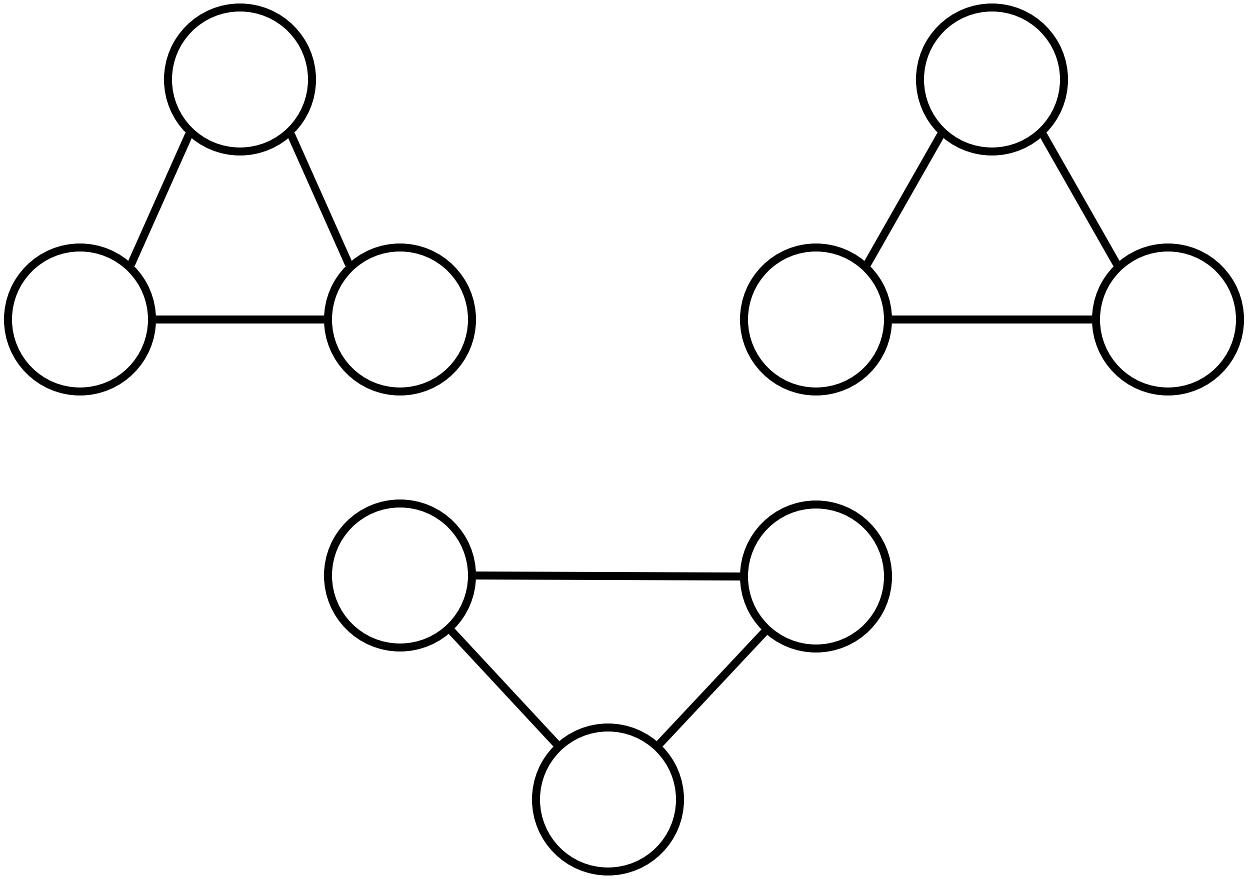


In NP? Exercise

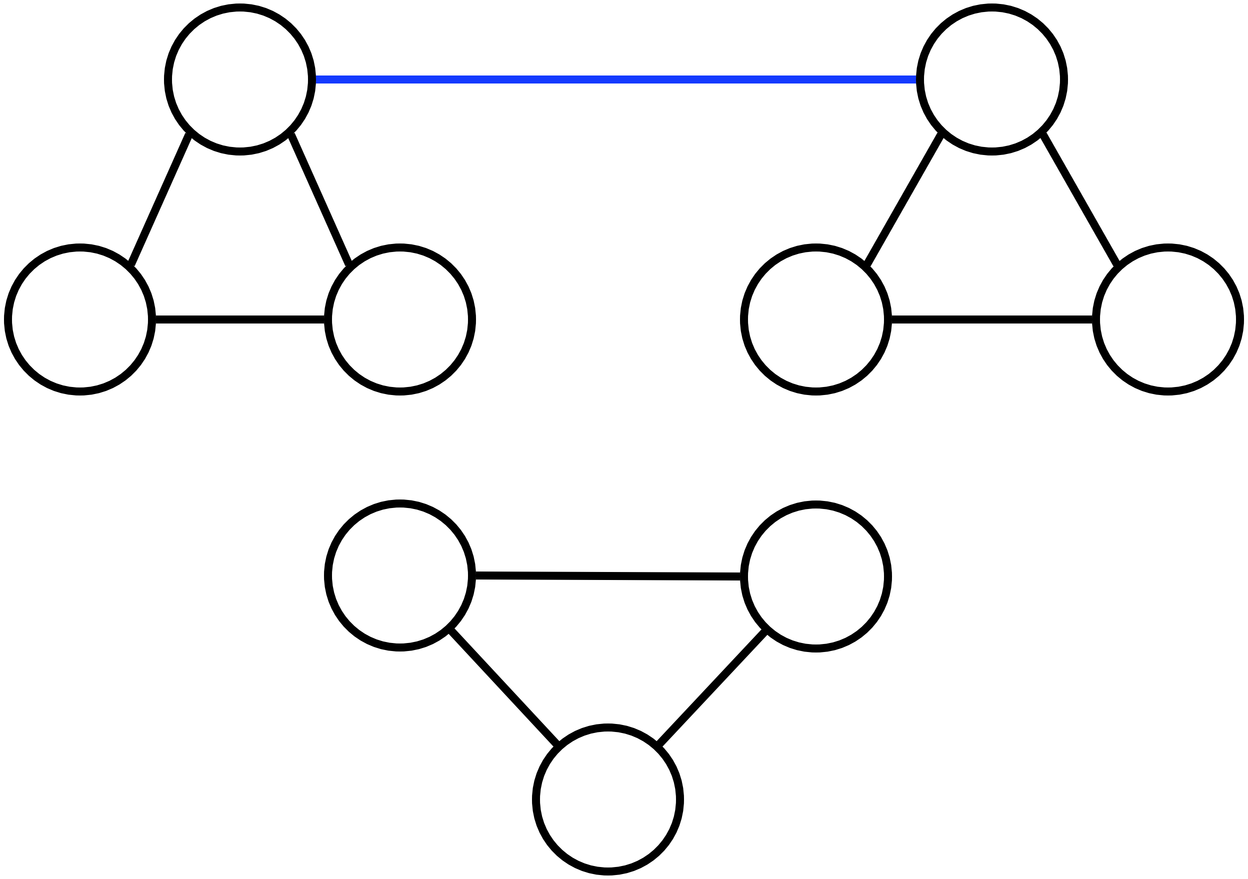
3SAT $\leq_p$ VertexCover



3SAT $\leq_p$ VertexCover

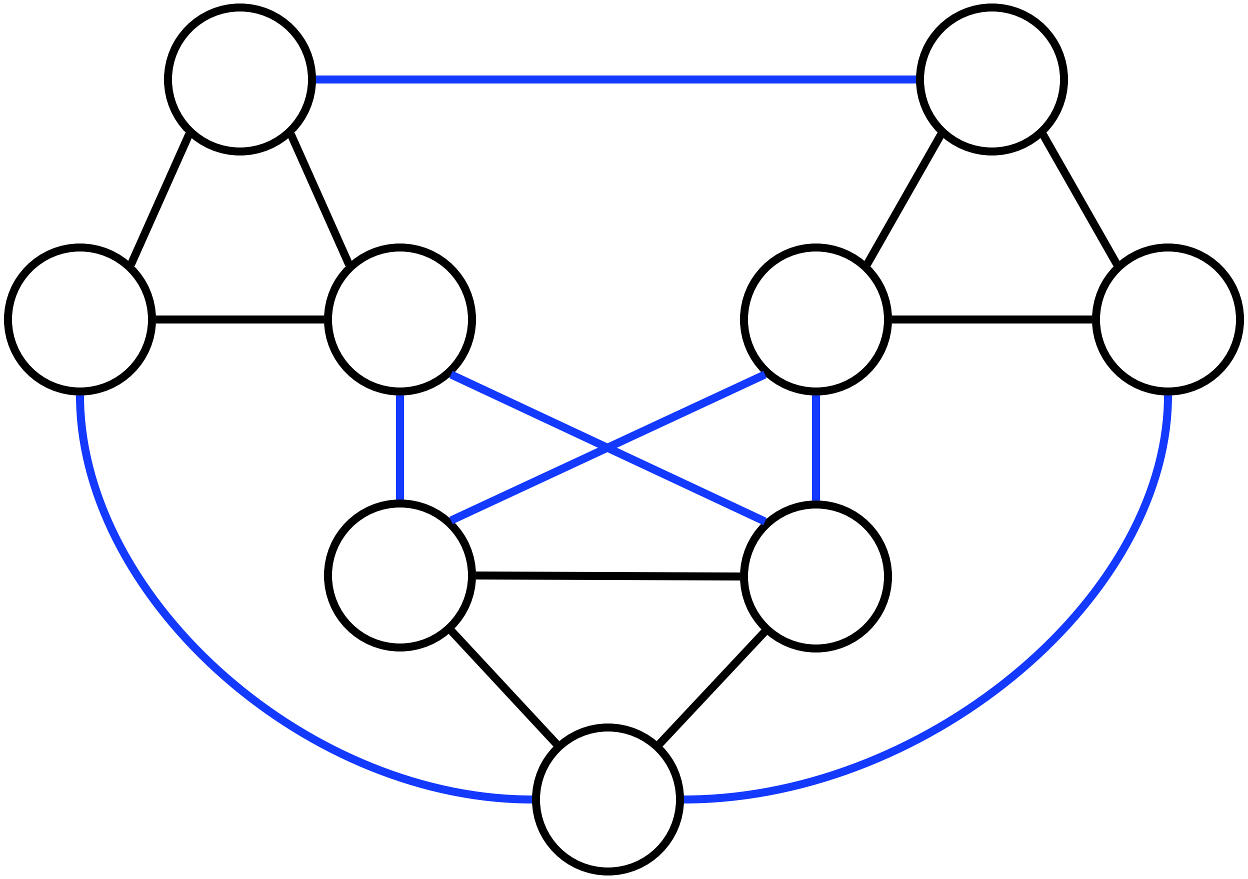


3SAT $\leq_p$ VertexCover



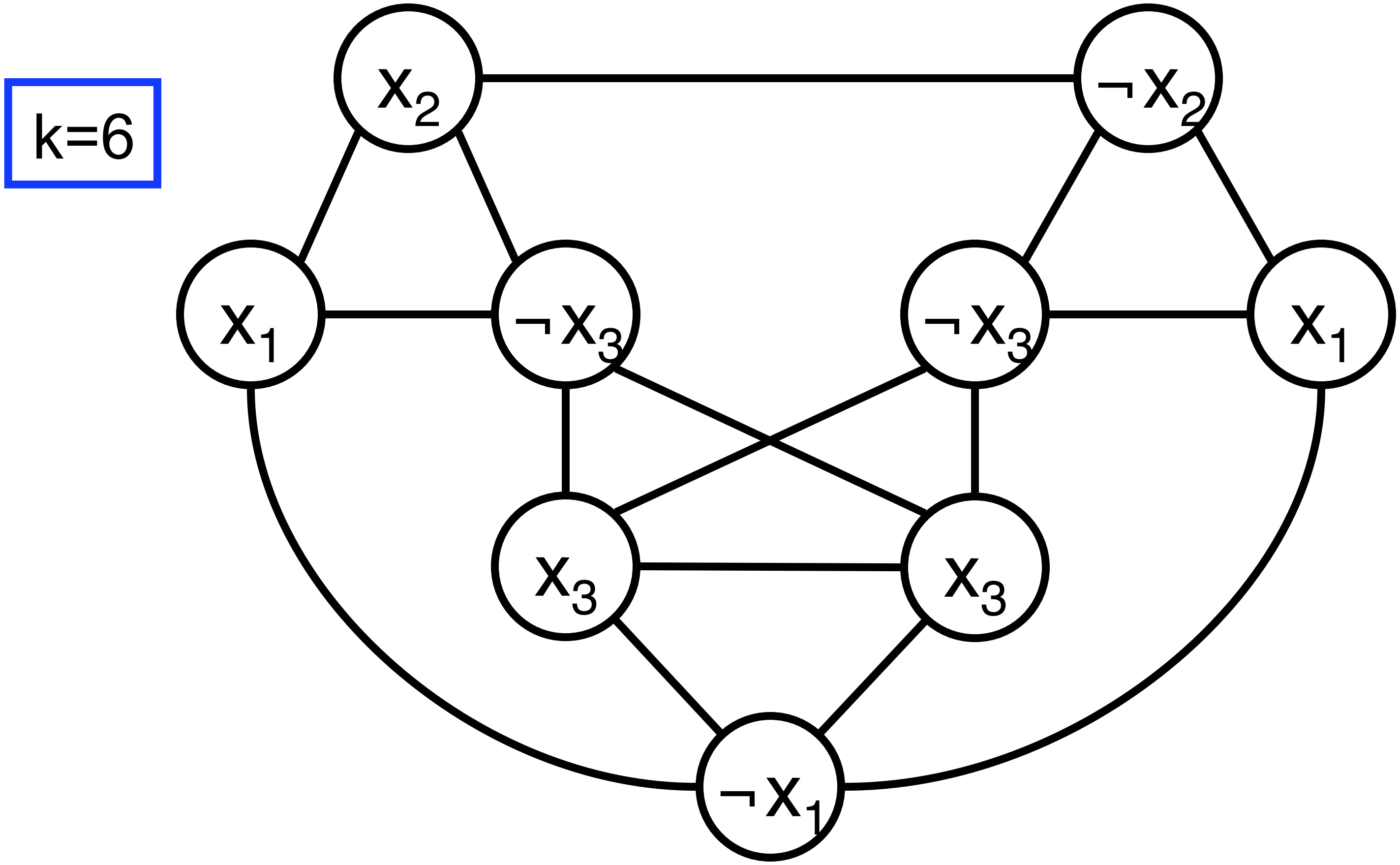
3SAT $\leq_p$ VertexCover

k=6



3SAT $\leq_p$ VertexCover

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



3SAT $\leq_p$ VertexCover

f (

3-SAT Instance:

- Variables: x_1, x_2, …
- Literals: $y_{i,j}$, $1 \leq i \leq q$, $1 \leq j \leq 3$
- Clauses: $c_i = y_{i1} \vee y_{i2} \vee y_{i3}$, $1 \leq i \leq q$
- Formula: $c = c_1 \wedge c_2 \wedge \ldots \wedge c_q$

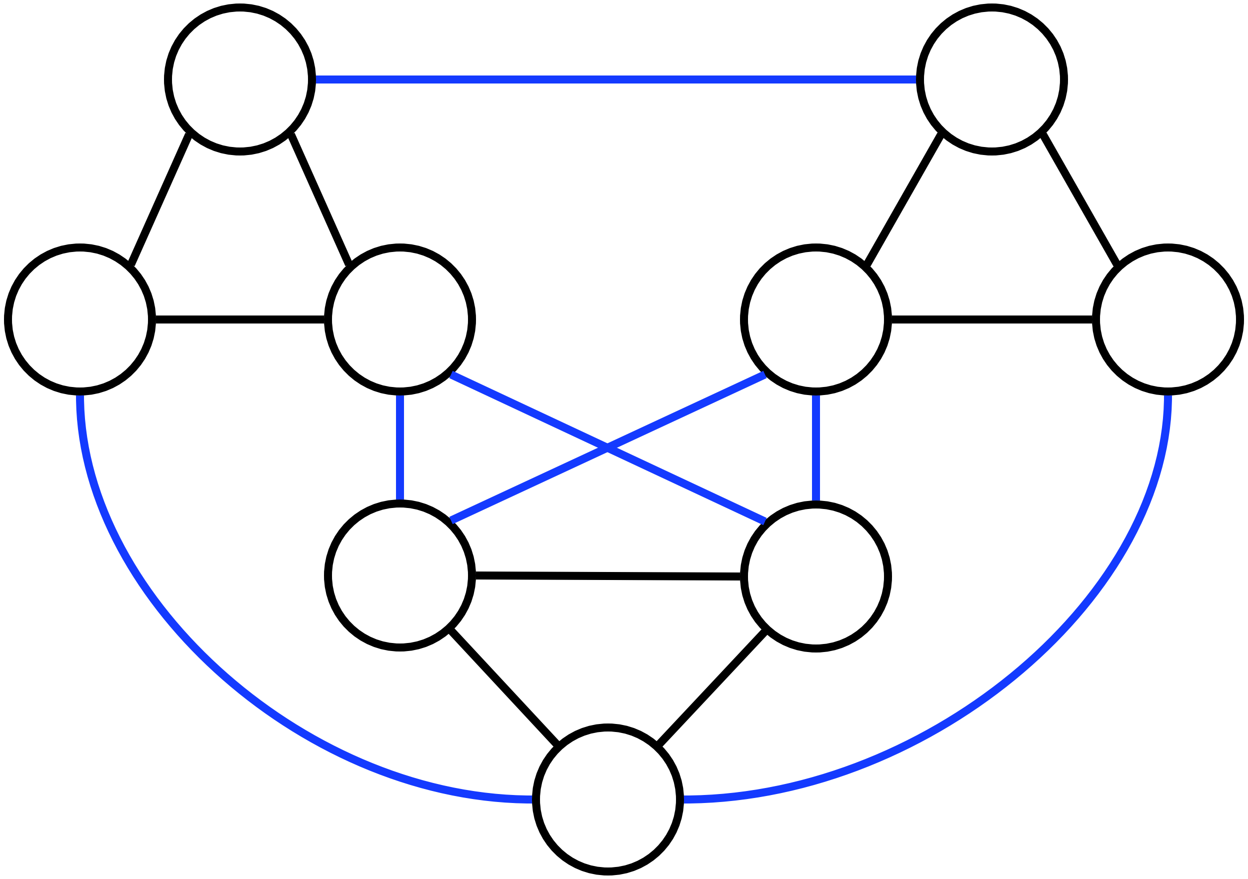
) =

VertexCover Instance:

- $k = 2q$
- $G = (V, E)$
- $V = \{ [i,j] \mid 1 \leq i \leq q, 1 \leq j \leq 3 \}$
- $E = \{ ([i,j], [k,l]) \mid i = k \text{ or } y_{ij} = \neg y_{kl} \}$

3SAT $\leq_p$ VertexCover

k=6



Correctness of "3SAT $\leq_p$ VertexCover"

Summary of reduction function f: Given formula, make graph G with one group per clause, one node per literal. Connect each to all nodes in same group, plus complementary literals (x, ¬x). Output graph G plus integer k = 2 * number of clauses. Note: *f does not know whether formula is satisfiable or not; does not know if G has k-cover; does not try to find satisfying assignment or cover.*

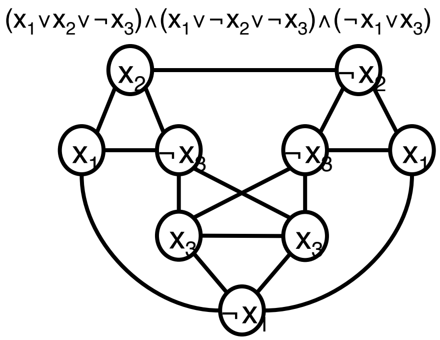
Correctness:

- Show f poly time computable: A key point is that graph size is polynomial in formula size; mapping basically straightforward.
- Show c in 3-SAT iff f(c)=(G,k) in VertexCover:

(⇒) Given an assignment satisfying c, pick one true literal per clause. Add other 2 nodes of each triangle to cover. Show it is a cover: 2 per triangle cover triangle edges; only true literals (but perhaps not all true literals) uncovered, so at least one end of every (x, ¬x) edge is covered.

(⇐) Given a k-vertex cover in G, uncovered labels define a valid (perhaps partial) truth assignment since no (x, ¬x) pair uncovered. It satisfies c since there is one uncovered node in each clause triangle (else some other clause triangle has > 1 uncovered node, hence an uncovered edge.)

Utility of "3SAT $\leq_p$ VertexCover"



Suppose we had a fast algorithm for VertexCover, then we could get a fast algorithm for 3SAT:

Given 3-CNF formula w, build Vertex Cover instance y = f(w) as above, run the fast VC alg on y; say "YES, w is satisfiable" iff VC alg says "YES, y has a vertex cover of the given size"

On the other hand, suppose no fast alg is possible for 3SAT, then we know none is possible for VertexCover either.

“3SAT $\leq_p$ VertexCover” Retrospective

Previous slide: two suppositions

Somewhat clumsy to have to state things that way.

Alternative: abstract out the key elements, give it a name (“polynomial time reduction”), then properties like the above always hold.

Polynomial-Time Reductions

Definition: Let A and B be two problems.

We say that A is *polynomially reducible* to B ($A \leq_p B$) if there exists a polynomial-time algorithm f that converts each instance x of problem A to an instance f(x) of B such that:

x is a YES instance of A iff f(x) is a YES instance of B

$$x \in A \quad \Leftrightarrow \quad f(x) \in B$$

Polynomial-Time Reductions (cont.)

Define: $A \leq_p B$ "A is polynomial-time reducible to B", iff there is a polynomial-time computable function f such that: $x \in A \iff f(x) \in B$

Why the notation?

"complexity of A" ≤ "complexity of B" + "complexity of f"

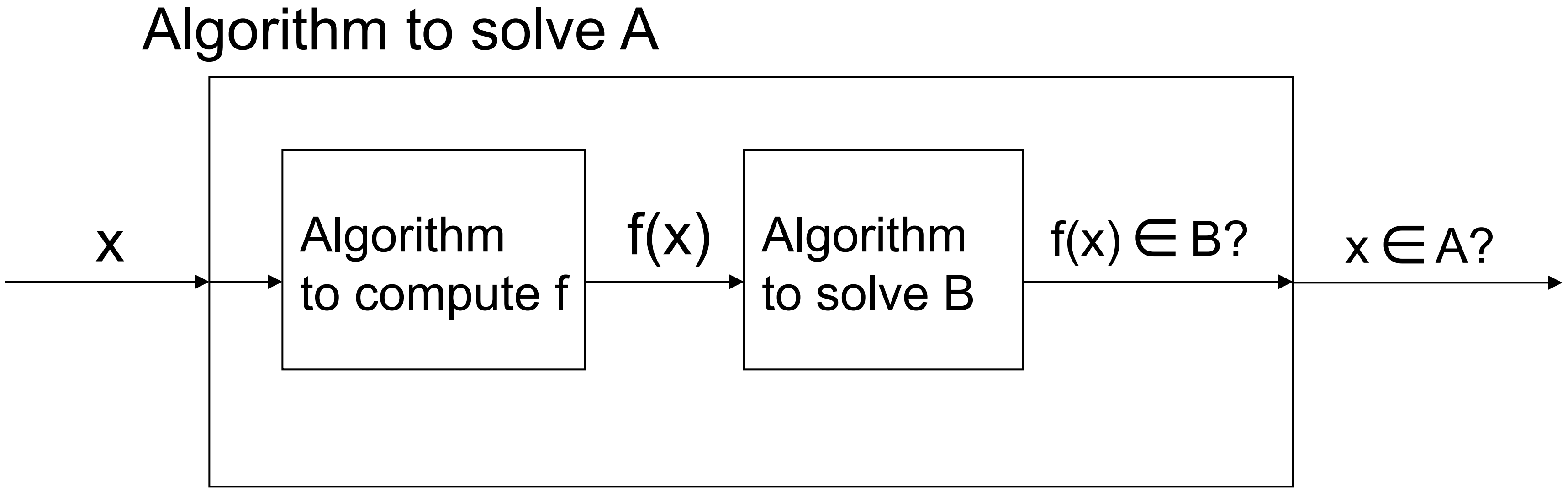
polynomial

(1) $A \leq_p B$ and $B \in P \Rightarrow A \in P$

(2) $A \leq_p B$ and $A \notin P \Rightarrow B \notin P$

(3) $A \leq_p B$ and $B \leq_p C \Rightarrow A \leq_p C$ (transitivity)

Using an Algorithm for ***B*** to Solve ***A***



"If $A \leq_p B$, and we can solve B in polynomial time, then we can solve A in polynomial time also."

Ex: suppose f takes $O(n^3)$ and algorithm for B takes $O(n^2)$.
How long does the above algorithm for A take?

Two definitions of “$A \le_p B$”

Book uses more general definition: “could solve A in poly time, *if* I had a poly time subroutine for B.”

Cook

Defn on previous slides is special case where you only get to call the subroutine once, and must report its answer.

Karp

This special case is used in ~98% of all reductions

Definition of NP-Completeness

Definition: Problem B is *NP-hard* if every problem in NP is polynomially reducible to B.

Definition: Problem B is *NP-complete* if:

(1) B belongs to NP, and

(2) B is NP-hard.

“NP-completeness”

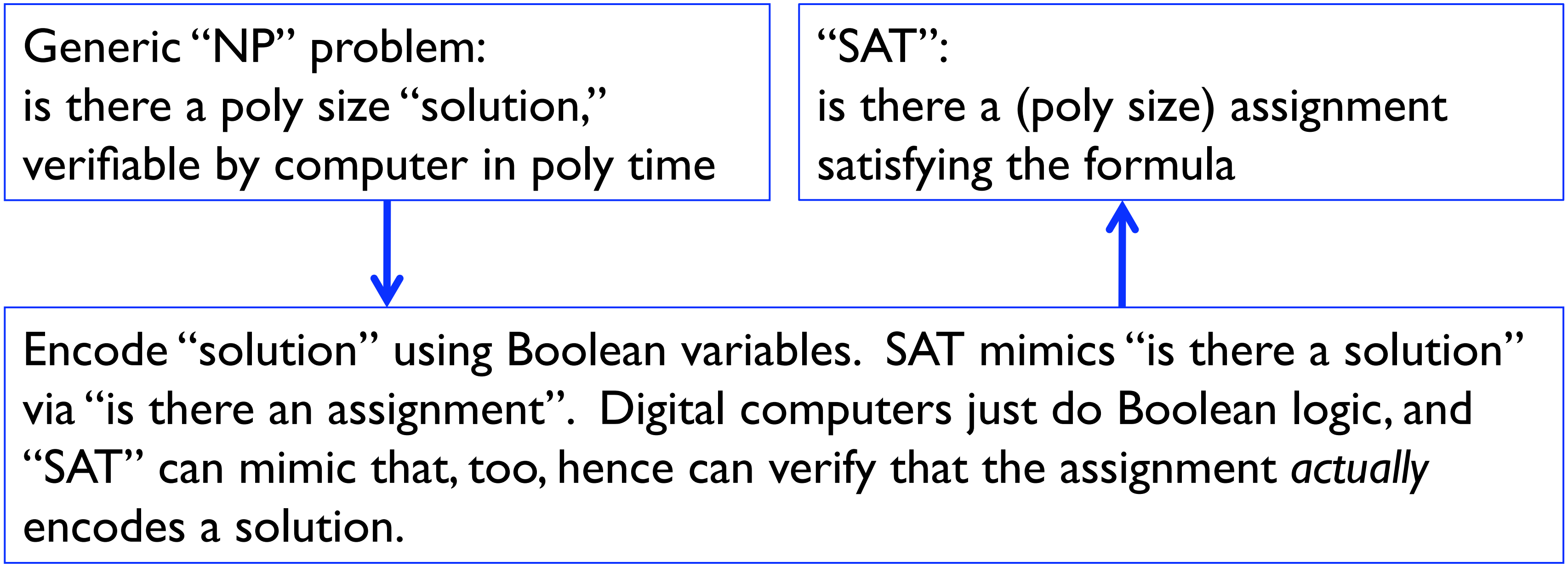
Cool concept, but are there any such problems?

Yes!

Cook’s theorem: SAT is NP-complete

Why is SAT NP-complete?

Cook’s proof is somewhat involved; I won’t show it.
But its essence is not so hard to grasp:



Generic “NP” problem:
is there a poly size “solution,”
verifiable by computer in poly time

“SAT”:
is there a (poly size) assignment
satisfying the formula

Encode “solution” using Boolean variables. SAT mimics “is there a solution” via “is there an assignment”. Digital computers just do Boolean logic, and “SAT” can mimic that, too, hence can verify that the assignment *actually* encodes a solution.

An Example

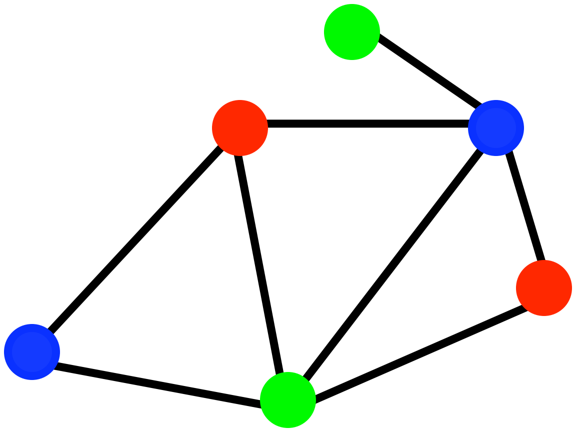
Again, Cook's theorem does this for generic NP problems, but you can get the flavor from a few specific examples

NP-complete problem: 3-Coloring

Input: An undirected graph G=(V,E).

Output: True iff there is an assignment of at most 3 colors to the vertices in G such that no two adjacent vertices have the same color.

Example:



In NP? Exercise

3-Coloring $\leq_p$ SAT

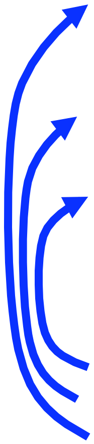
Given $G = (V, E)$

variables r_i, g_i, b_i for each i in V encode color

$$\bigwedge_{i \in V} [(r_i \vee g_i \vee b_i) \wedge$$

$$(\neg r_i \vee \neg g_i) \wedge (\neg g_i \vee \neg b_i) \wedge (\neg b_i \vee \neg r_i)] \wedge$$

$$\bigwedge_{(i,j) \in E} [(\neg r_i \vee \neg r_j) \wedge (\neg g_i \vee \neg g_j) \wedge (\neg b_i \vee \neg b_j)]$$



adj nodes ⇔ diff colors
no node gets 2
every node gets a color

Vertex cover $\le_p$ SAT

Given G = (V, E) and k

variables x_i, for each i in V encode inclusion of i in cover

$$\underbrace{\bigwedge_{(i,j) \in E} (x_i \vee x_j)}_{\substack{\text{every edge covered}\\ \text{by one end or other}}} \wedge \underbrace{\text{“number of True } x_i \text{ is } \le k\text{”}}_{\substack{\text{possible in 3 CNF, but}\\ \text{technically messy}}}$$

Proving a problem is NP-complete

Technically, for condition (2) we have to show that every problem in NP is reducible to B.
(Yikes! Sounds like a lot of work.)

For the very first NP-complete problem (SAT) this had to be proved directly.

However, once we have one NP-complete problem, then we don't have to do this every time.

Why? Transitivity.

Re-stated Definition

Lemma: Problem B is NP-complete if:

(1) B belongs to NP, and

(2') A is polynomial-time reducible to B, for some problem A that is NP-complete.

That is, to show (2') given a new problem B, it is sufficient to show that SAT or any other NP-complete problem is polynomial-time reducible to B.

Usefulness of Transitivity

Now we only have to show $L' \leq_p L$, for some NP-complete problem L', in order to show that L is NP-hard. Why is this equivalent?

1) Since L' is NP-complete, we know that L' is NP-hard. That is:

   $\forall L'' \in NP$, we have $L'' \leq_p L'$

2) If we show $L' \leq_p L$, then by transitivity we know that: $\forall L'' \in NP$, we have $L'' \leq_p L$.

Thus L is NP-hard.

Ex: VertexCover is NP-complete

3-SAT is NP-complete (shown by S. Cook)

3-SAT $\leq_p$ VertexCover

VertexCover is in NP (we showed this earlier)

Therefore VertexCover is also NP-complete

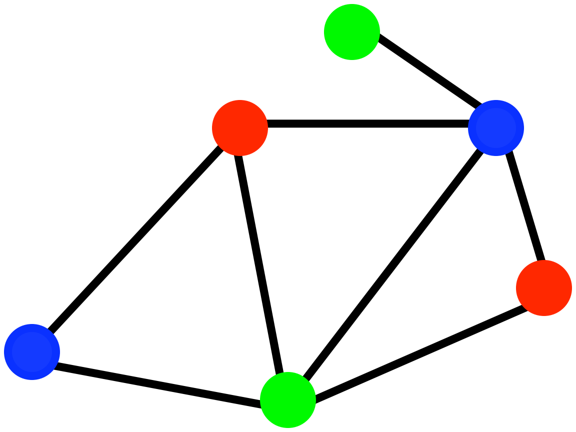
So, poly-time algorithm for VertexCover would give poly-time algs for everything in NP

NP-complete problem: 3-Coloring

Input: An undirected graph G=(V,E).

Output: True iff there is an assignment of at most 3 colors to the vertices in G such that no two adjacent vertices have the same color.

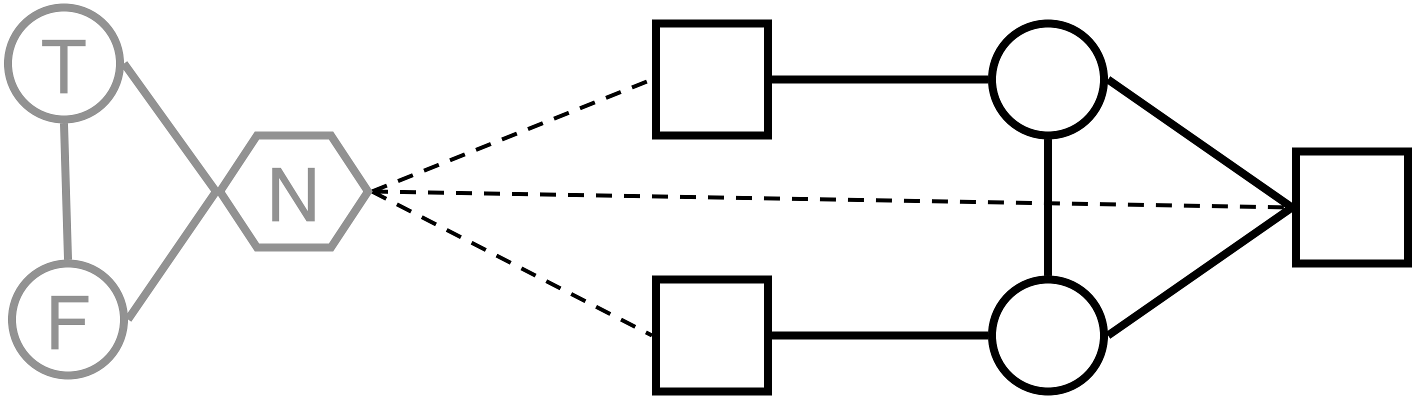
Example:



In NP? Exercise

A 3-Coloring Gadget:

In what ways can this be 3-colored?

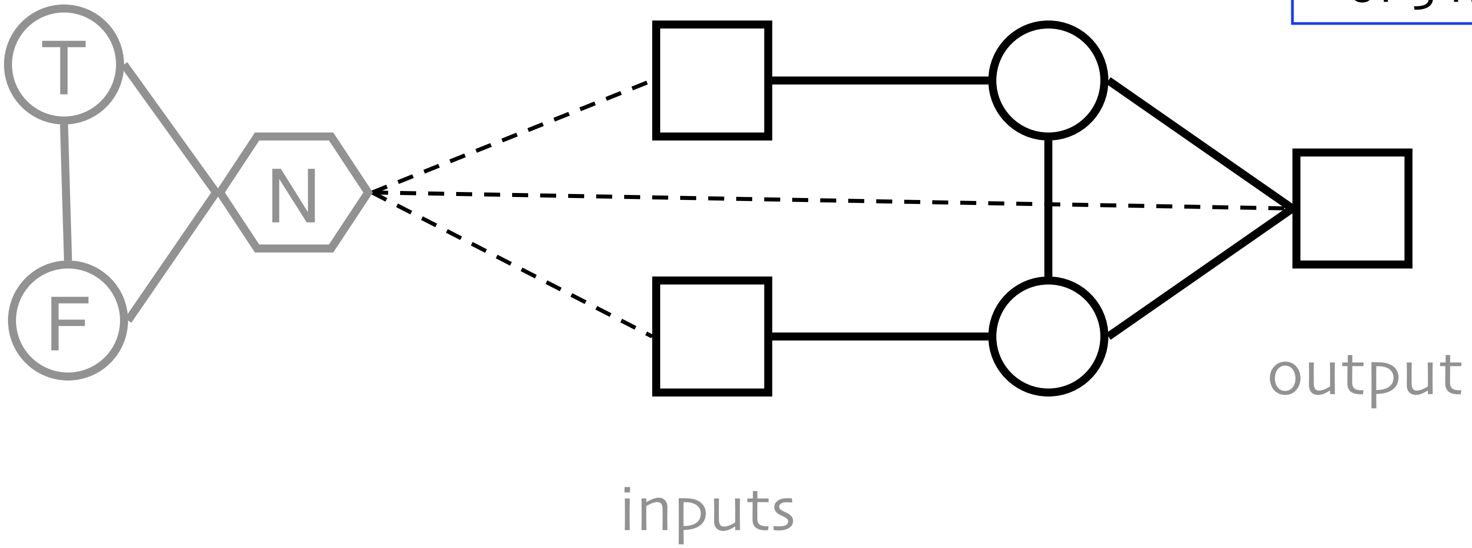


A 3-Coloring Gadget: "Sort of an OR gate"

if any input is T, the output can be T

if output is T, some input must be T

Exercise: find all colorings of 5 nodes



3SAT $\leq_p$ 3Color

f (

3-SAT Instance:

- Variables: x_1, x_2, …
- Literals: $y_{i,j}$, $1 \leq i \leq q$, $1 \leq j \leq 3$
- Clauses: $c_i = y_{i1} \vee y_{i2} \vee y_{i3}$, $1 \leq l \leq q$
- Formula: $c = c_1 \wedge c_2 \wedge \ldots \wedge c_q$

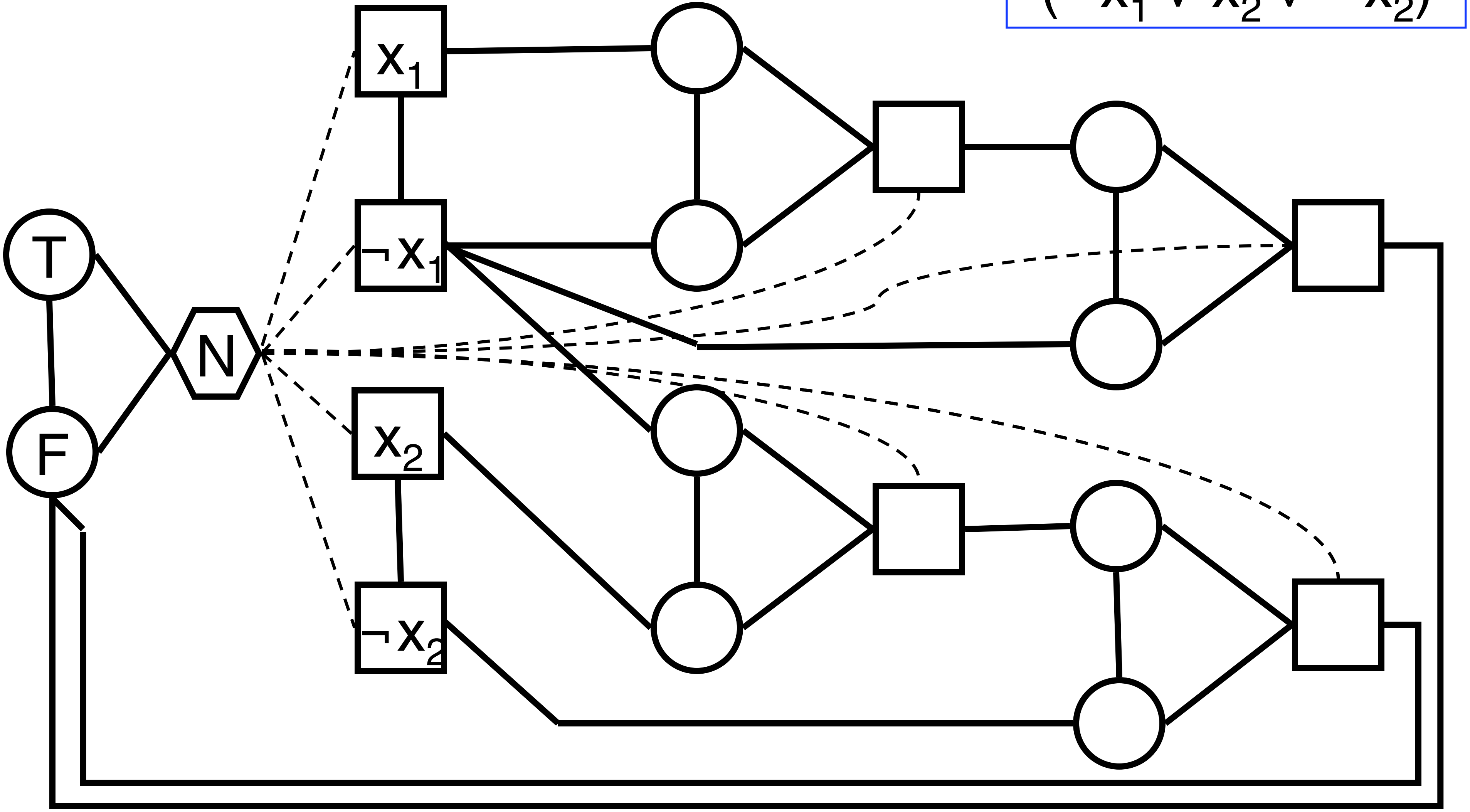
) =

3Color Instance:

- G = (V, E)
- 6 q + 2 n + 3 vertices
- 13 q + 3 n + 3 edges
- (See Example for details)

3SAT $\leq_p$ 3Color Example

$$(x_1 \vee \neg x_1 \vee \neg x_1)$$
$$\wedge$$
$$(\neg x_1 \vee x_2 \vee \neg x_2)$$



$6q + 2n + 3$ vertices $\qquad$ $13q + 3n + 3$ edges

Correctness of "3SAT $\leq_p$ 3Coloring"

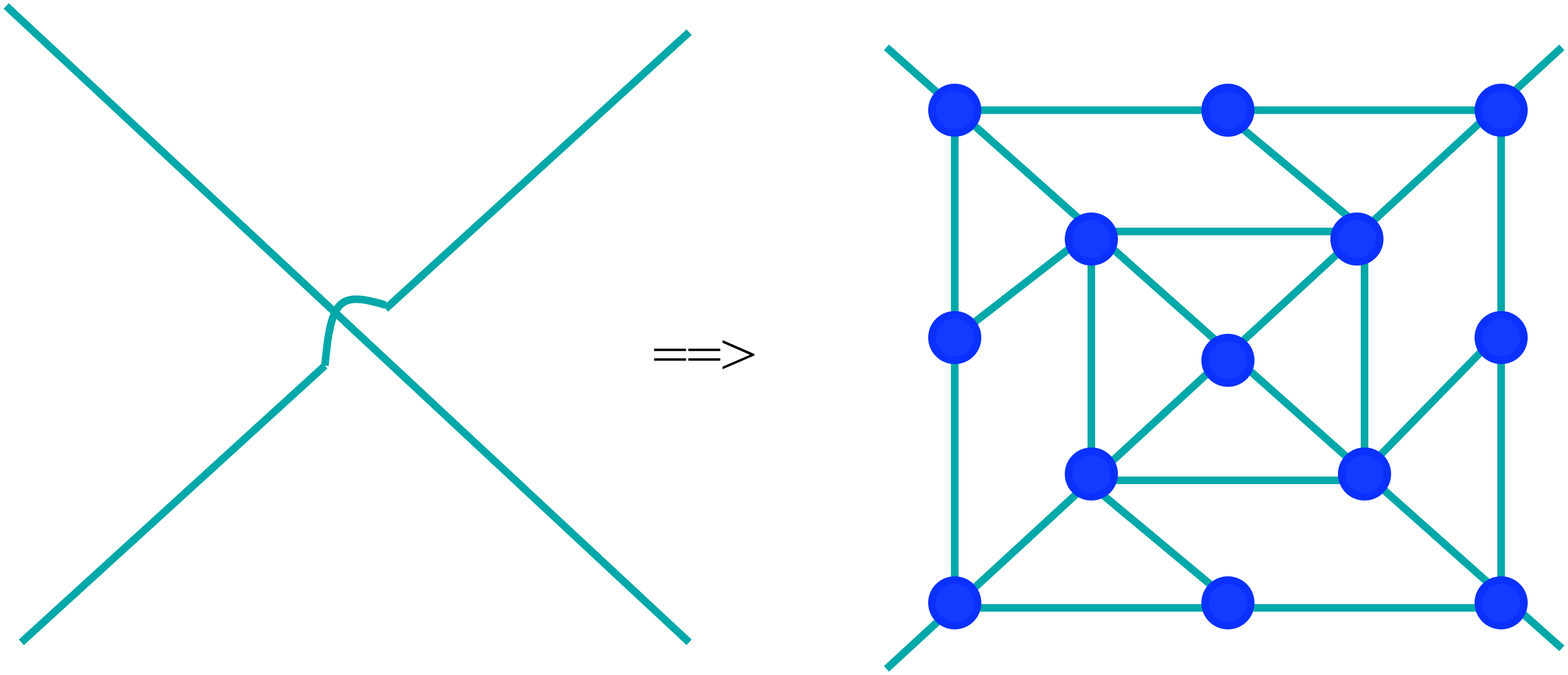
Summary of reduction function f:
Given formula, make G with T-F-N triangle, 1 pair of literal nodes per variable, 2 "or" gadgets per clause, connected as in example.
Note: *again, f does not know or construct satisfying assignment or coloring.*

Correctness:

- Show f poly time computable: A key point is that graph size is polynomial in formula size; graph looks messy, but pattern is basically straightforward.
- Show c in 3-SAT iff f(c) is 3-colorable:

(⇒) Given an assignment satisfying c, color literals T/F as per assignment; can color "or" gadgets so output nodes are T since each clause is satisfied.
(⇐) Given a 3-coloring of f(c), name colors T-N-F as in example. All square nodes are T or F (since all adjacent to N). Each variable pair (x_i, $\neg x_i$) must have complementary labels since they're adjacent. Define assignment based on colors of x_i's. Clause "output" nodes must be colored T since they're adjacent to both N & F. By fact noted earlier, output can be T only if at least one input is T, hence it is a satisfying assignment.

Planar 3-Coloring is also NP-Complete



Common Errors in NP-completeness Proofs

Backwards reductions

Bipartiteness $\leq_p$ SAT is true, but not so useful.
(XYZ $\leq_p$ SAT shows XYZ in NP, doesn't show it's hard.)

Sloooow Reductions

"Find a satisfying assignment, then output…"

Half Reductions

Delete dashed edges in 3Color reduction. It's still true that "c satisfiable ⇒ G is 3 colorable", but 3-colorings don't necessarily give good assignments.

Coping with NP-Completeness

- Is your real problem a special subcase?
  - E.g. 3-SAT is NP-complete, but 2-SAT is not; ditto 3- vs 2-coloring
  - E.g. you only need planar graphs, or degree 3 graphs, …?
- Guaranteed approximation good enough?
  - E.g. Euclidean TSP within 1.5 * Opt in poly time
- Fast enough in practice (esp. if n is small),
  - E.g. clever exhaustive search like backtrack, branch & bound, pruning
- Heuristics – usually a good approximation and/or usually fast

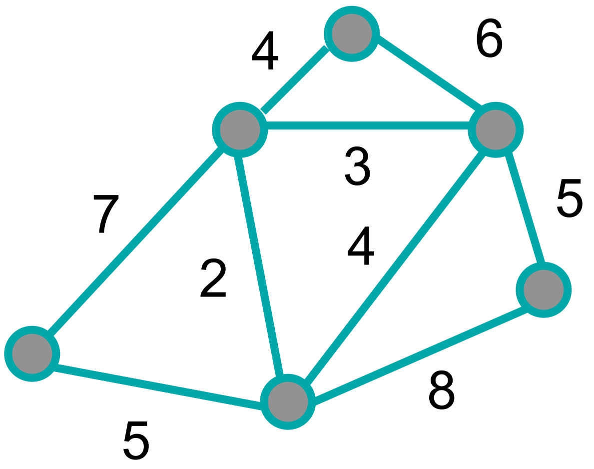
NP-complete problem: TSP

Input: An undirected graph G=(V,E) with integer edge weights, and an integer b.

Output: YES iff there is a simple cycle in G passing through all vertices (once), with total cost ≤ b.

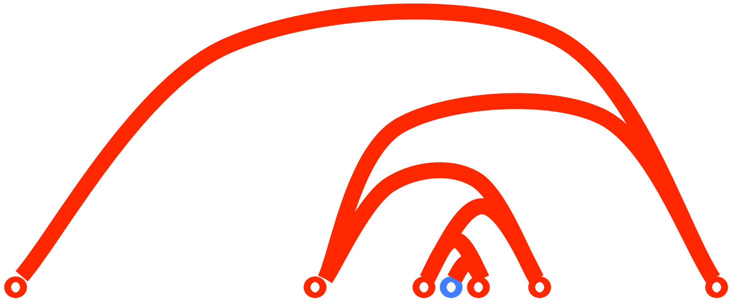
Example:

b = 34



TSP - Nearest Neighbor Heuristic

Recall NN Heuristic



Fact: NN tour can be about (log n) x opt, i.e.

$$\lim_{n\to\infty} \frac{NN}{OPT} \to \infty$$

(above example is not that bad)

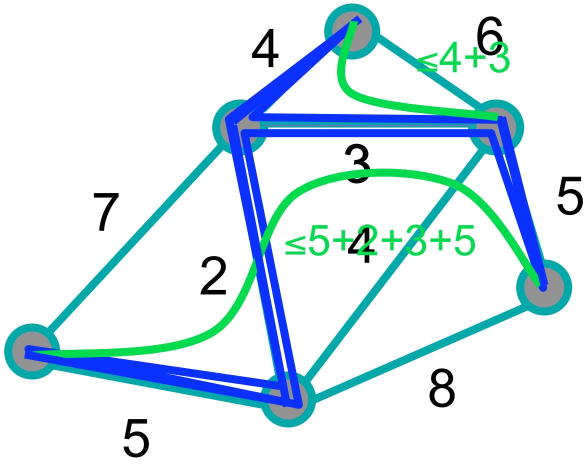
2x Approximation to Euclidean TSP

A TSP tour visits all vertices, so contains a spanning tree, so TSP cost is > cost of min spanning tree.

Find MST

Find “DFS” Tour

Shortcut



TSP ≤ shortcut < DFST = 2 * MST < 2 * TSP

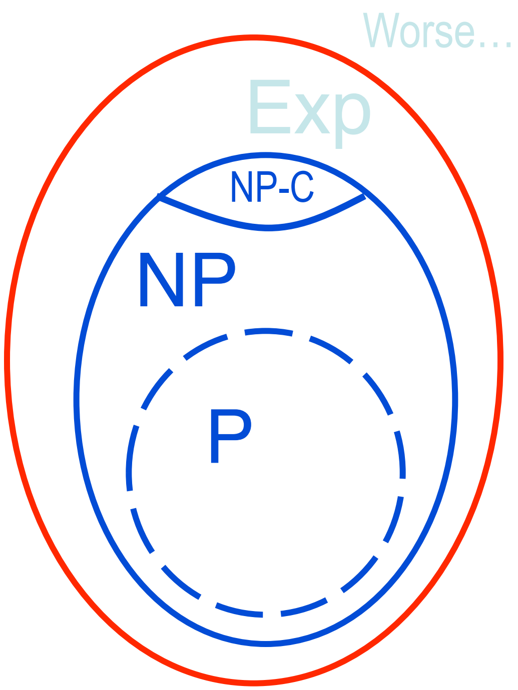
A problem NOT in NP;
A bogus “proof” to the contrary

EEXP = $\{(p,x) \mid \text{prog } p \text{ accepts input } x \text{ in } < 2^{2^{|x|}} \text{ steps } \}$

NON Theorem: EEXP in NP

“Proof” I: Hint = step-by-step trace of the computation of p on x; verify step-by-step

“Proof” II: Hint = a bit; accept iff it’s 1



Summary

Big-O – good
P – good
Exp – bad
Exp, but hints help? NP
NP-hard, NP-complete – bad (I bet)
To show NP-complete – reductions
NP-complete = hopeless? – no, but you need to lower your expectations: heuristics & approximations.