CSE 421
Algorithms

NP-Completeness

(Chapter 8)

What can we feasibly compute?

Focus so far 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. Sadly, there are many…

History

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 Time

The class P

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

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

These problems are sometimes called *tractable* problems.

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

(exceptions: Change-Making/Stamps, Knapsack, TSP)

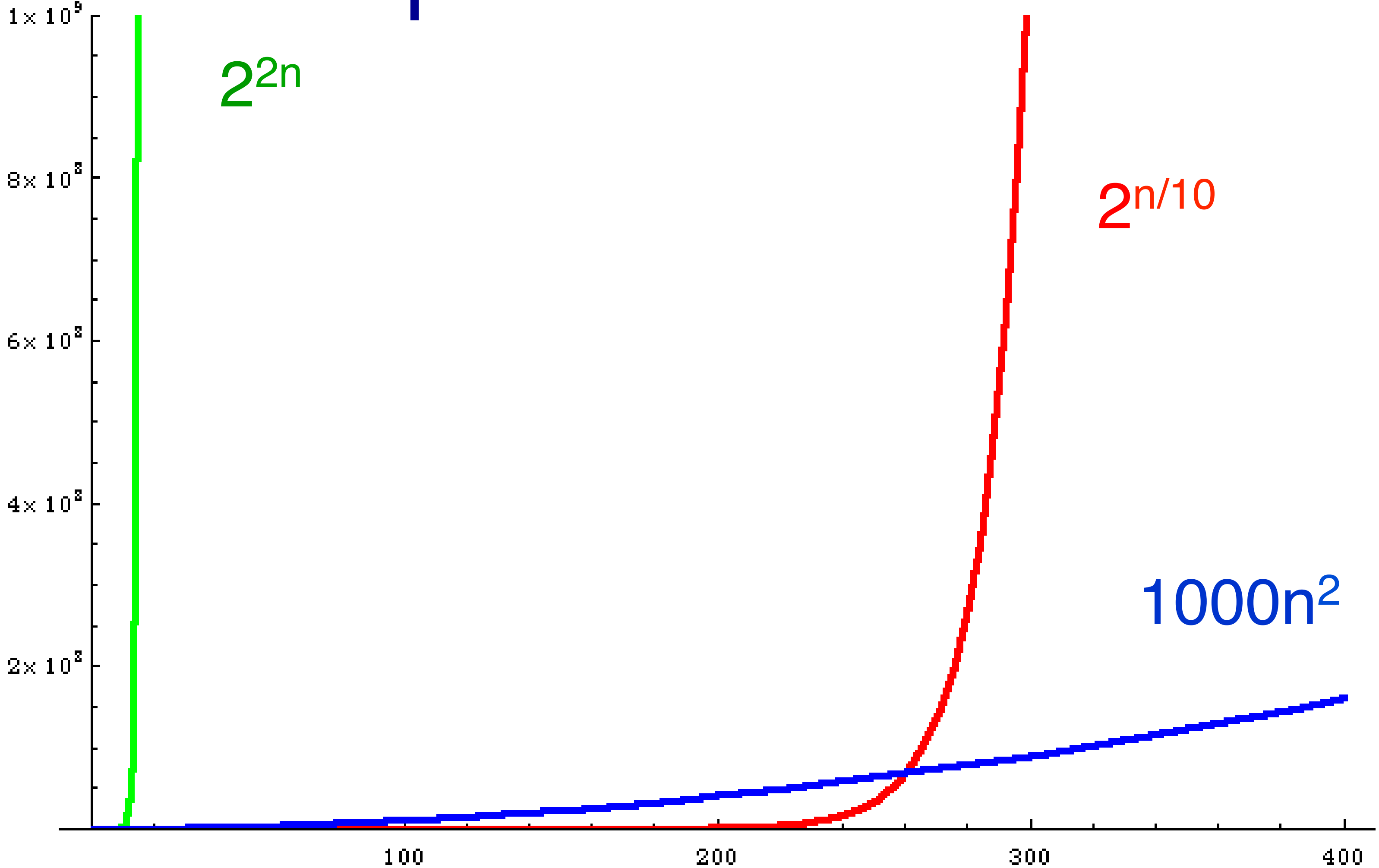
Why "Polynomial"?

Point is not that n^{2000} is a nice time bound, or that the differences among n and 2n and n^2 are negligible.

Rather, simple theoretical tools may not easily capture such differences, whereas exponentials are qualitatively different from polynomials and may be amenable to theoretical analysis.

- "My problem is in P" is a starting point for a more detailed analysis
- "My problem is not in P" may suggest that you need to shift to a more tractable variant

Polynomial vs Exponential Growth



Another view of Poly vs Exp

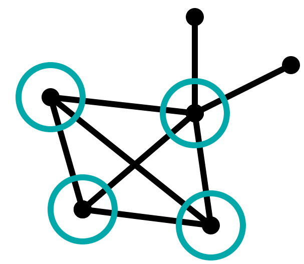
Next year's computer will be 2x faster. If I can solve problem of size n_0 today, how large a problem can I solve in the same time next year?

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

Decision vs Search Problems

Problem Types

A *clique* in an undirect graph G=(V,E) is a subset U of V such that every pair of vertices in U is joined by an edge.



E.g., mutual friends on facebook, genes that vary together

An *optimization* problem: *How large* is the largest clique in G

A *search* problem: *Find* the/a largest clique in G

A *search* problem: Given G and integer k, *find* a k-clique in G

A *decision* problem: Given G and k, *is there* a k-clique in G

Decision Problems

So far we have mostly considered *search* and *optimization* problems – "Find a…" or "How large is the largest…"

Below, we mainly restrict discussion to *decision problems* - problems that have an answer of either yes or no.

Loss of generality? Not really

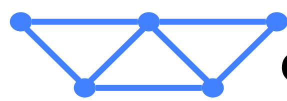
- Usually easy to convert to decision problem
- If we know how to solve the decision problem, then we can usually solve the original problem.
- Most importantly, decision problem is easier (at least, not harder), so a *lower bound* on the decision problem is a lower bound on the associated search/optimization problem.

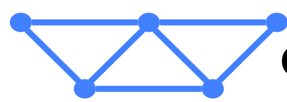
Some Convenient Technicalities

"Problem" – the general case

Ex: The Clique Problem: Given a graph G and an integer k, does G contain a k-clique?

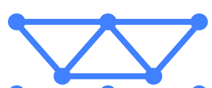
"Problem Instance" – the specific cases

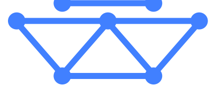
Ex: Does  contain a 4-clique? (no)

Ex: Does  contain a 3-clique? (yes)

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

Beyond P

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 $ax^2 + by = c$?

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

$(x_1 \vee \neg x_3 \vee x_7) \wedge (\neg x_1 \vee \neg x_4 \vee x_5 \vee \neg x_7)$

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

$x_1 \wedge (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge \neg x_3$

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

Satisfiable?

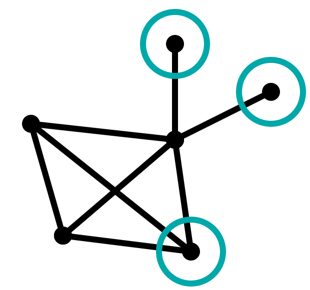
$$(x \lor y \lor z) \land (\neg x \lor y \lor \neg z) \land$$
$$(x \lor \neg y \lor z) \land (\neg x \lor \neg y \lor z) \land$$
$$(\neg x \lor \neg y \lor \neg z) \land (x \lor y \lor z) \land$$
$$(x \lor \neg y \lor z) \land (x \lor y \lor \neg z)$$

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

More Problems

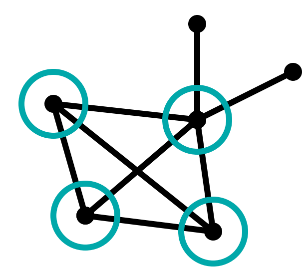
Independent-Set:

Pairs ⟨G,k⟩, where G=(V,E) is a graph and k is an integer, for which there is a subset U of V with |U| ≥ k such that *no* pair of vertices in U is joined by an edge.



Clique:

Pairs ⟨G,k⟩, where G=(V,E) is a graph and k is an integer k, for which there is a subset U of V with |U| ≥ k such that *every* pair of vertices in U is joined by an edge.



More Problems

Euler Tour:

Graphs G=(V,E) for which there is a cycle traversing each edge once.

Hamilton Tour:

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

TSP:

Pairs $\langle G,k \rangle$, where G=(V,E,w) is a a weighted graph and k is an integer, such that there is a Hamilton tour of G with total weight $\leq$ k.

More Problems

Short Path:

4-tuples $\langle G, s, t, k \rangle$, where $G=(V,E)$ is a digraph with vertices s, t, and an integer k, for which there is a path from s to t of length $\leq k$

Long Path:

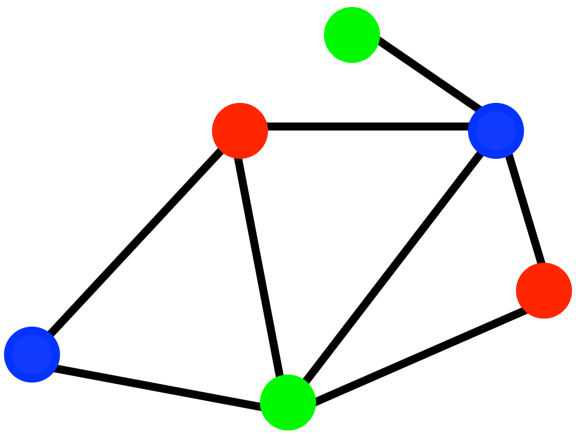
4-tuples $\langle G, s, t, k \rangle$, where $G=(V,E)$ is a digraph with vertices s, t, and an integer k, for which there is an acyclic path from s to t of length $\geq k$

More Problems

3-Coloring:

Graphs G=(V,E) for which 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:



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. $ax^2 + by = c$?

e.g., most of others just mentioned (excl: shortpath, Euler)

Lack of imagination or intrinsic barrier?

NP

Review: Some Problems

Quadratic Diophantine Equations
Clique
Independent Set
Euler Tour
Hamilton Tour
TSP
3-Coloring
Partition
Satisfiability
Short Paths
Long Paths

All of the form: Given input X Is there a Y with property Z

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

"Answer" to a decision problem is literally just yes/no, but there's always a somewhat more elaborate "solution" (aka "hint" or "certificate"; what the search version would report) 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

Defining NP

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

for every $x \in L$ there is a "hint" h with $|h| \leq |x|^k$ such that v(x,h) = YES and

for every $x \notin L$ there is *no* hint h with $|h| \leq |x|^k$ such that v(x,h) = YES

("Hints," sometimes called "certificates," or "witnesses", are just strings. Think of them as exactly what the search version would output.)

Example: Clique

"Is there a k-clique in this graph?"

any subset of k vertices *might* be a clique

there are *many* such subsets, but 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"

this can be *quickly checked*

And if there is *not* a k-clique, I wouldn't be fooled by a statement like "look at vertices 2,3,17,42,..."

More Formally: 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"

Important note: this answer does NOT mean $x \notin$ CLIQUE; just means this h isn't a k-clique (but some other might be).

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)

And $|h| < |x|$ and $v(x,h)$ takes time $\sim (|x|+|h|)^2$

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 this can be quickly checked

And if the formula is unsatisfiable, I wouldn't be fooled by , "x_1=T, x_2=F, ..., x_n=F"

More Formally: SAT $\in$ 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

Analysis: $|A| < |F|$, and time for v(F,A) ~ linear in $|F|+|A|$

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 do some algebra 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 (42,321) if there's no solution

Short 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 ... "

Long 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 ... "

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, i.e. allow verification in polynomial time? Is it polynomial length?

OK if some inputs need no hint

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

Two Final Points About "Hints"

1. Hints/verifiers aren't unique. The "… there is a …" framework often suggests their form, but many possibilities

   "is there a clique" could be verified from its vertices, or its edges, or all but 3 of each, or all non-vertices, or… Details of the hint string and the verifier and its time bound shift, but same bottom line

2. In NP doesn't prove its hard

   "Short Path" or "Small spanning tree" can be formulated as "…there is a…," but, due to very special structure of these problems, we can quickly find the solution even without a hint. The mystery is whether that's possible for the other problems, too.

Contrast: problems *not* in NP (probably)

Rather than "there is a…" maybe it's "*no*…" or "*for all*…"

E.g.

UNSAT: "*no* assignment satisfies formula," or
"*for all* assignments, formula is false"

Or

NOCLIQUE: "*every* subset of k vertices is not a k-clique"

It seems unlikely that a single, short hint is sufficiently informative to allow poly time verification of properties like these (but this is also an important open problem).

Another Contrast: *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 kn... one: "yes, I

see an... A to 2 and from

2 to 42... total 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

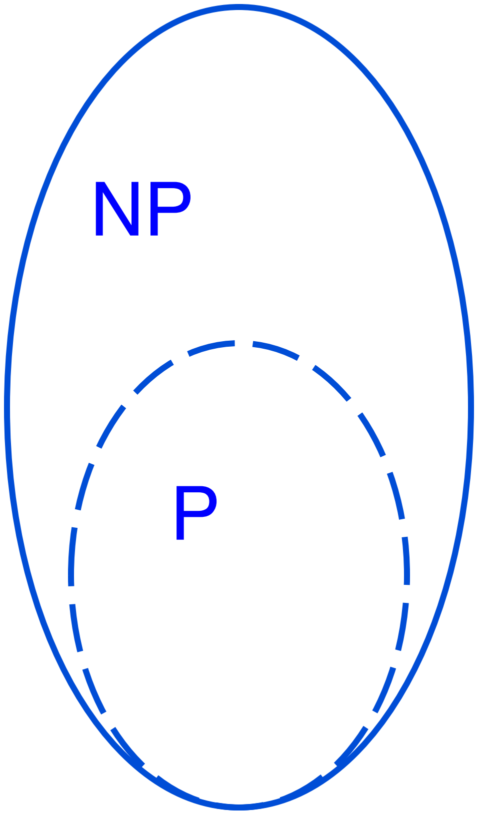
Relating P to NP

Complexity Classes

NP = Polynomial-time
verifiable

P = Polynomial-time
solvable

P $\subseteq$ NP: "verifier" is
just the P-time alg;
ignore "hint"



Solving NP problems without hints

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

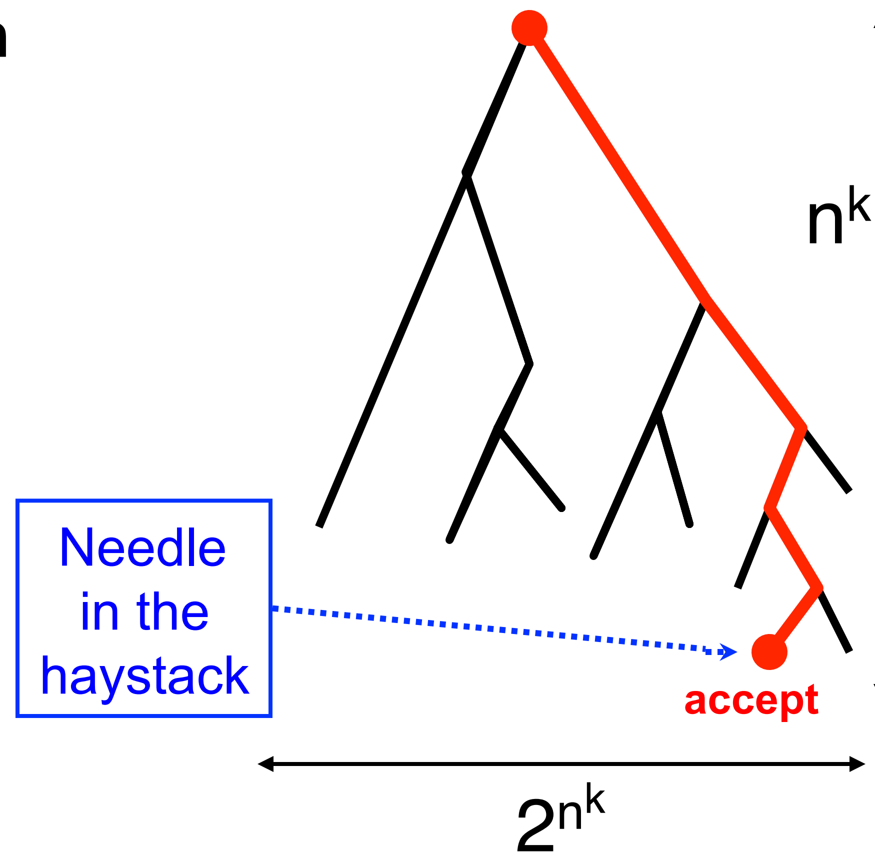
- try all possible hints; check each one to see if it works.
- Exponential time:
  - 2^n truth assignments for n variables
  - $n!$ possible TSP tours of n vertices
  - $\binom{n}{k}$ possible k element subsets of n vertices
  - etc.

…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

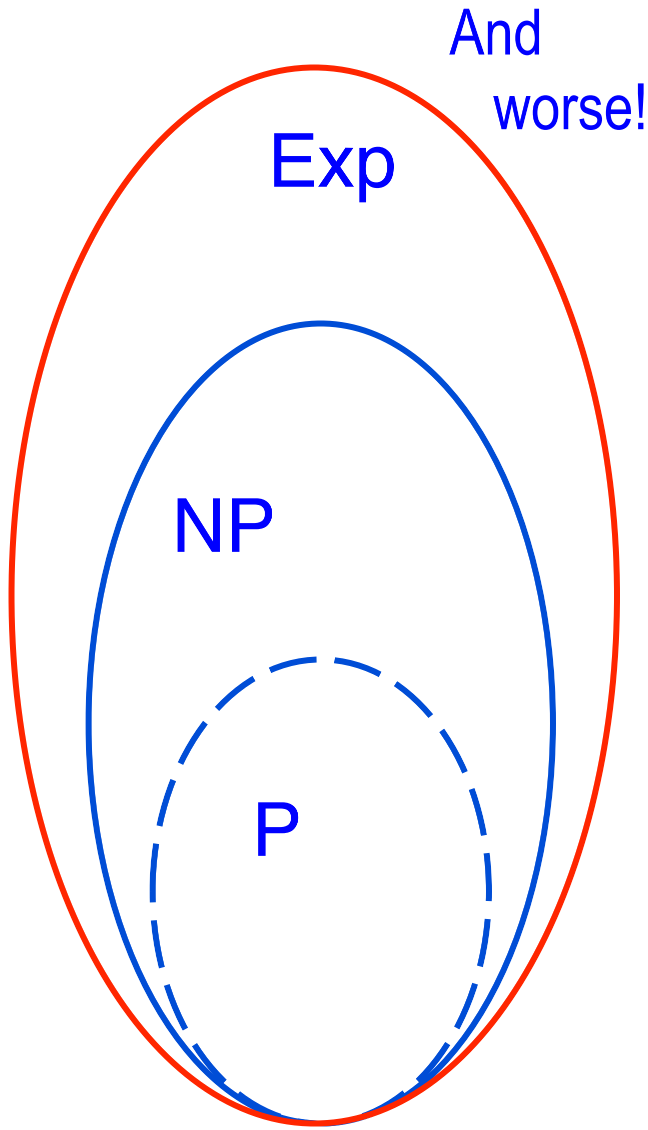
Every problem in P is in NP

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)



Does P = NP?

This is the big 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…?)

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
2-Coloring	3-Coloring
Min Cut	Max Cut
Shortest Path	Longest Path

Superficially different; similar computationally

Similar pairs; seemingly different computationally

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

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.

Reduction

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

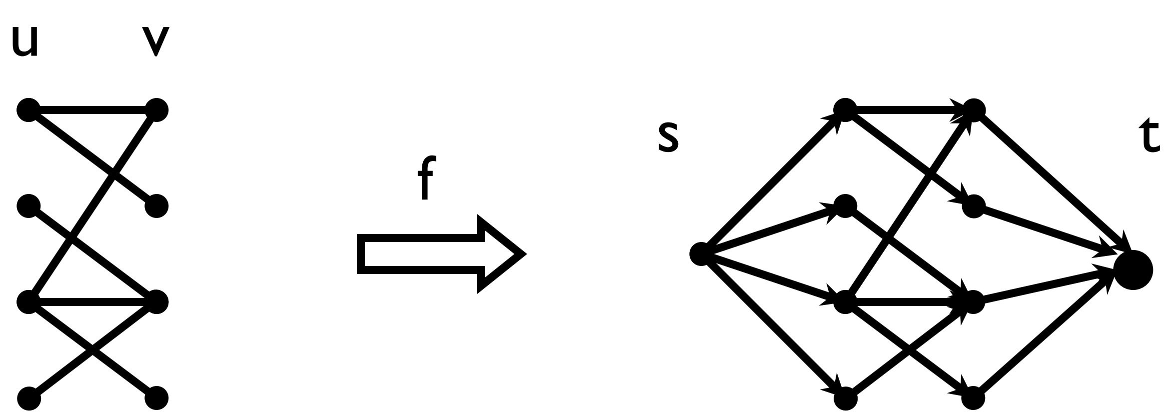
Example: reduce MEDIAN to FIND_MAX

Solution: transitivity: compose solutions above.

Another Example of Reduction

reduce BIPARTITE_MATCHING to MAX_FLOW

Is there a matching of size k? Is there a flow of size k?



All capacities = 1

P-time Reductions: What, Why

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"*

Polynomial-Time Reductions

Definition: Let A and B be two problems.

We say that A is *polynomially (mapping) 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

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

Polynomial-Time Reductions (cont.)

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

Why the notation?

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

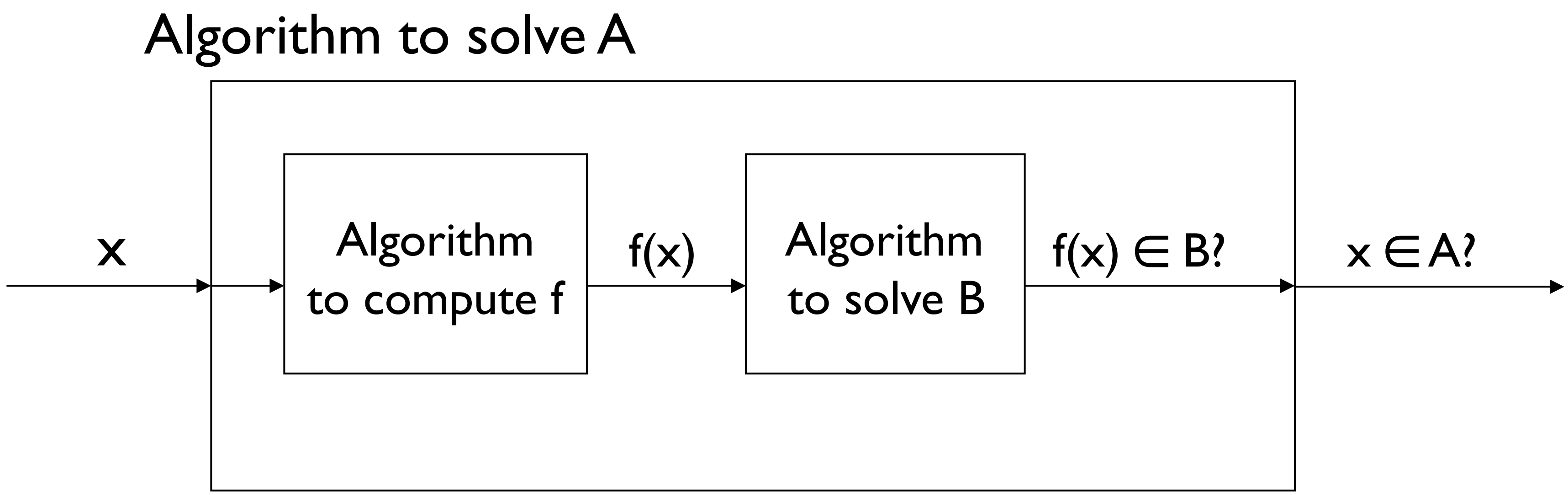
polynomial

(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 $\leq_p$ 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

Largely irrelevant for this course, but if you seem to need 1st defn, e.g. on HW, there's perhaps a simpler way…

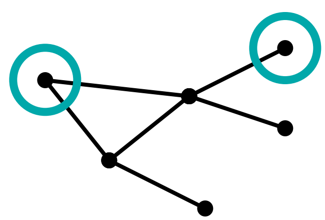
SAT and Independent Set

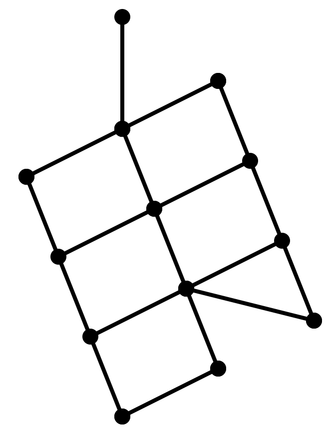
Another NP problem: Independent Set

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

Output: True iff there is a subset I of V of size ≥ k such that no edge in E has both end points in I.

Example: Independent Set of size ≥ 2.

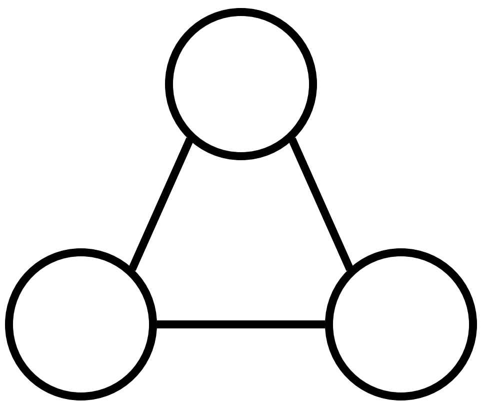




In NP? Exercise

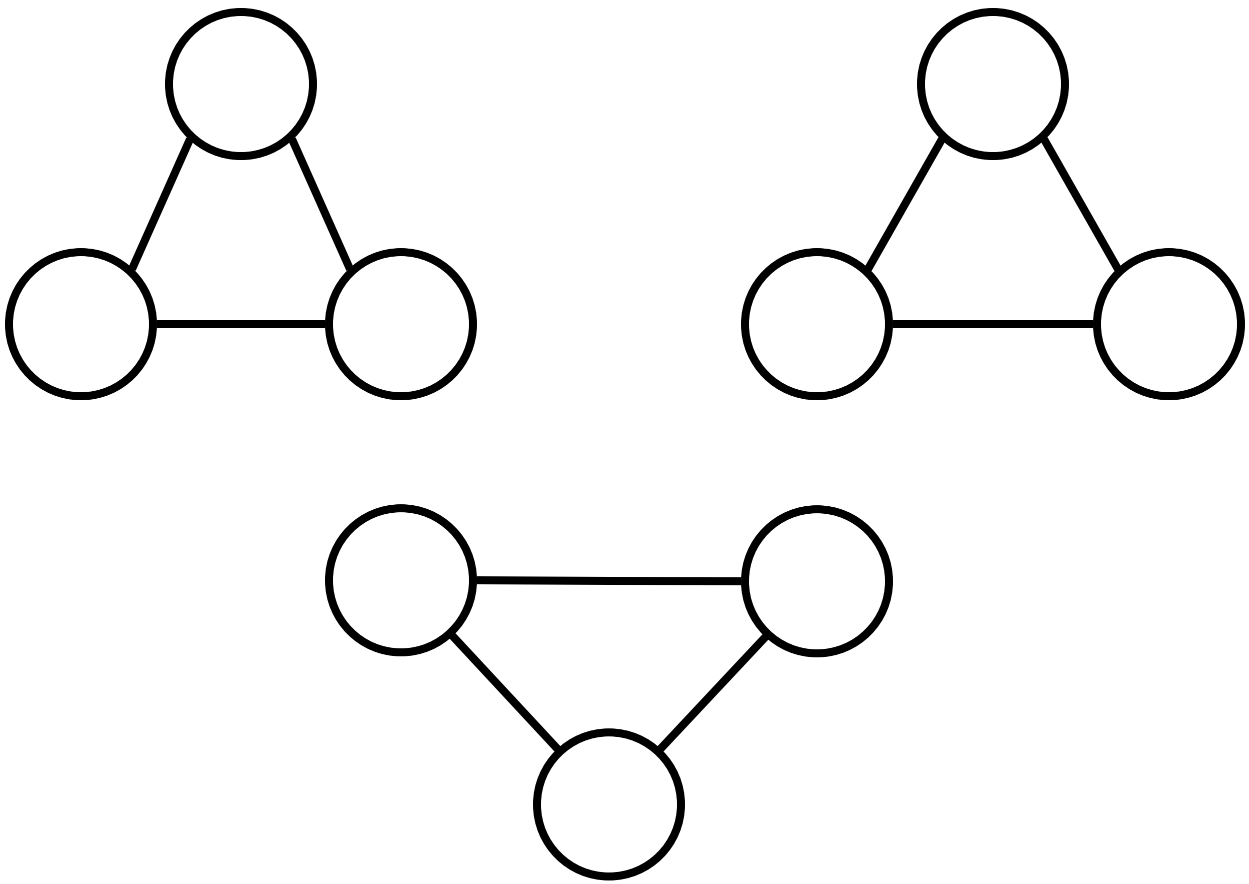
3SAT $\leq_p$ IndpSet

what indp sets?
how large?
how many?



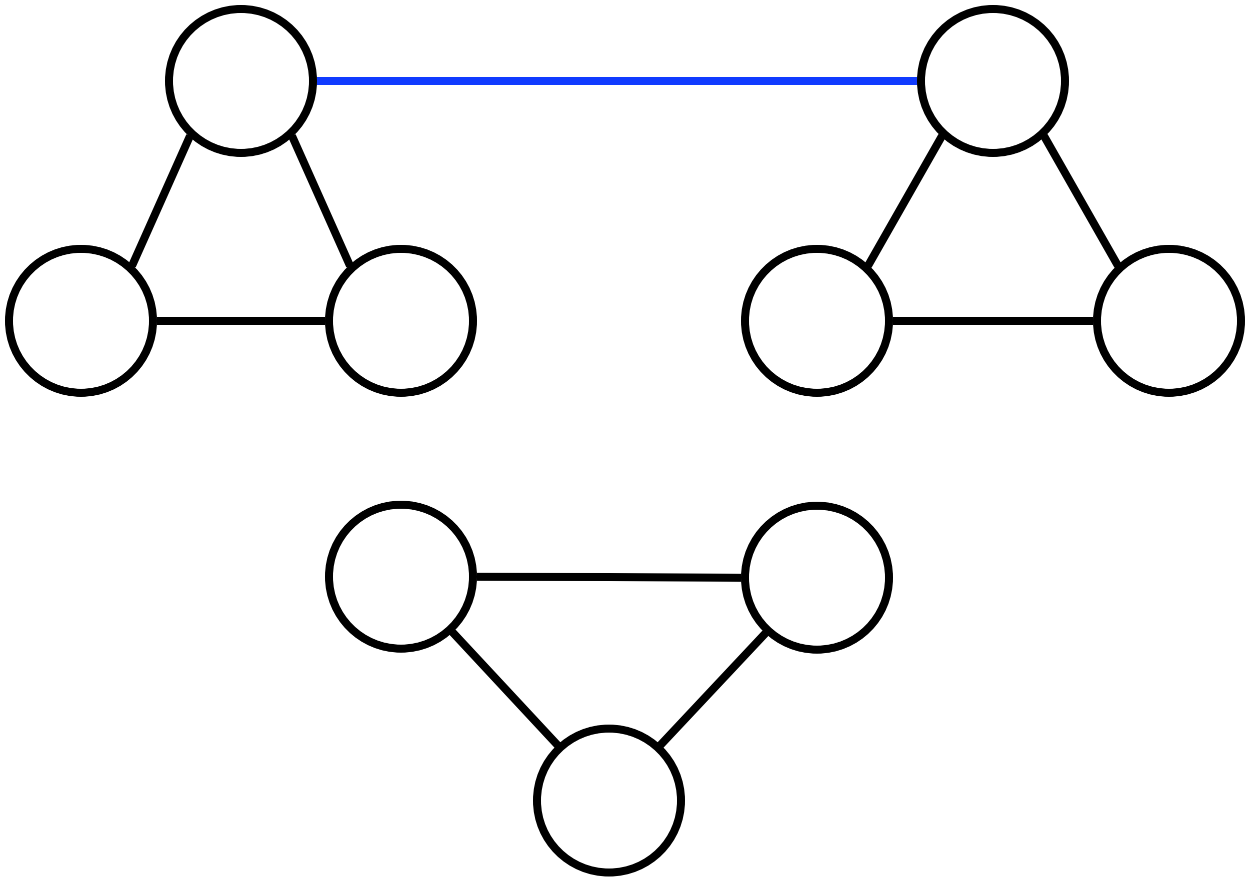
3SAT $\leq_p$ IndpSet

what indp sets?
how large?
how many?



3SAT $\leq_p$ IndpSet

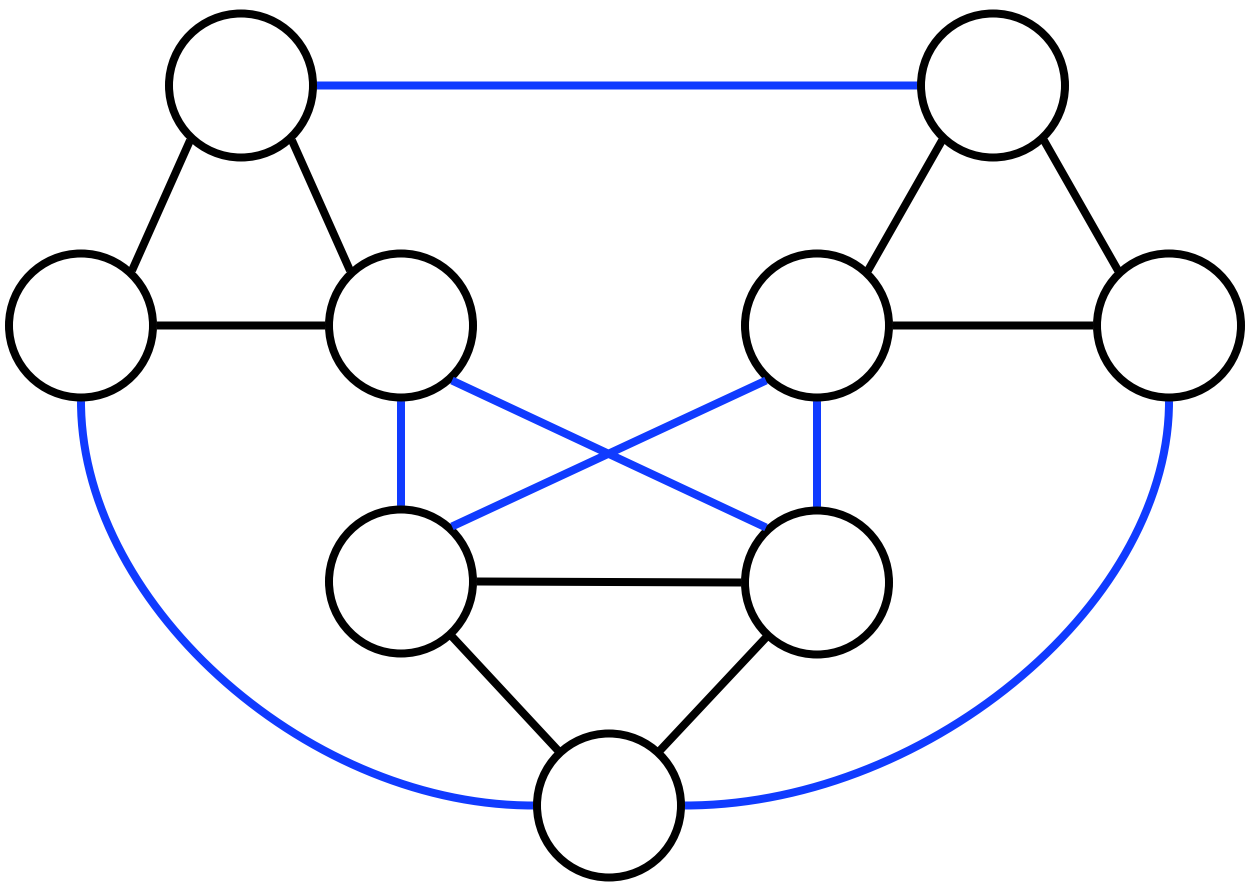
what indp sets?
how large?
how many?



3SAT $\leq_p$ IndpSet

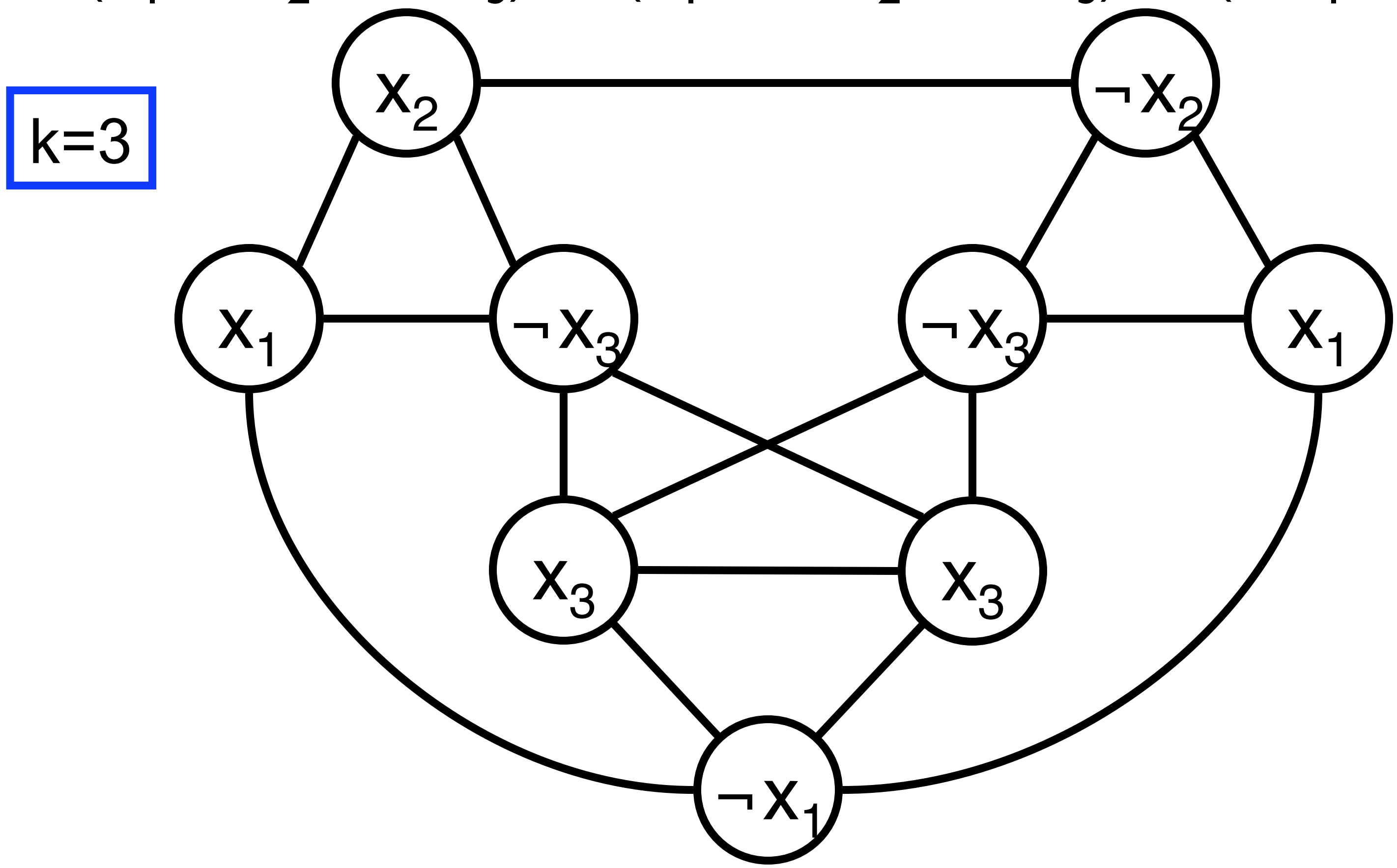
what indp sets?
how large?
how many?

k=3



$3SAT \leq_p \text{IndpSet}$

$$(x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3)$$



$3SAT \leq_p IndpSet$

$f\left(\right.$

3-SAT Instance:

- Variables: $x_1, x_2, \ldots$
- 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: $c = c_1 \wedge c_2 \wedge \ldots \wedge c_q$

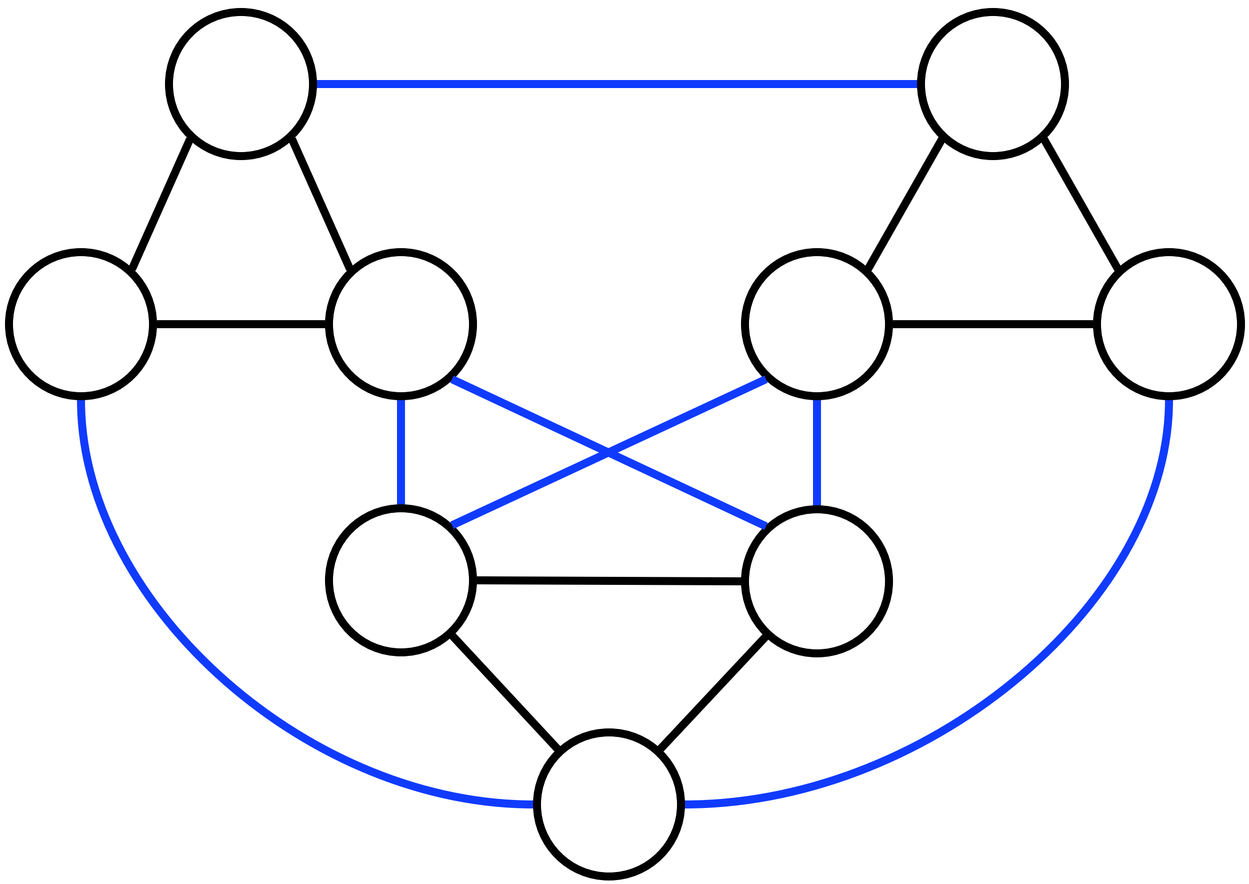
$\left.\right) =$

IndpSet Instance:

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

3SAT $\leq_p$ IndpSet

k=3



Correctness of "3SAT $\leq_p$ IndpSet"

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 = number of clauses. Note: *f does not know whether formula is satisfiable or not; does not know if G has k-IndpSet; does not try to find satisfying assignment or set.*

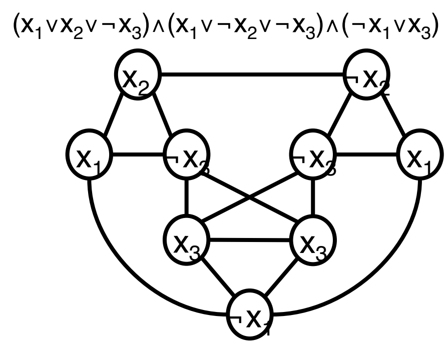
Correctness:

- 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 IndpSet:

(⇒) Given an assignment satisfying c, pick one true literal per clause. Add corresponding node of each triangle to set. Show it is an IndpSet: 1 per triangle never conflicts w/ another in same triangle; only true literals (but perhaps not all true literals) picked, so not both ends of any (x, ¬x) edge.

(⇐) Given a k-Independent Set in G, selected labels define a valid (perhaps partial) truth assignment since no (x, ¬x) pair picked. It satisfies c since there is one selected node in each clause triangle (else some other clause triangle has > 1 selected node, hence not an independent set.)

Utility of "3SAT $\leq_p$ IndpSet"



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

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

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

NP-completeness

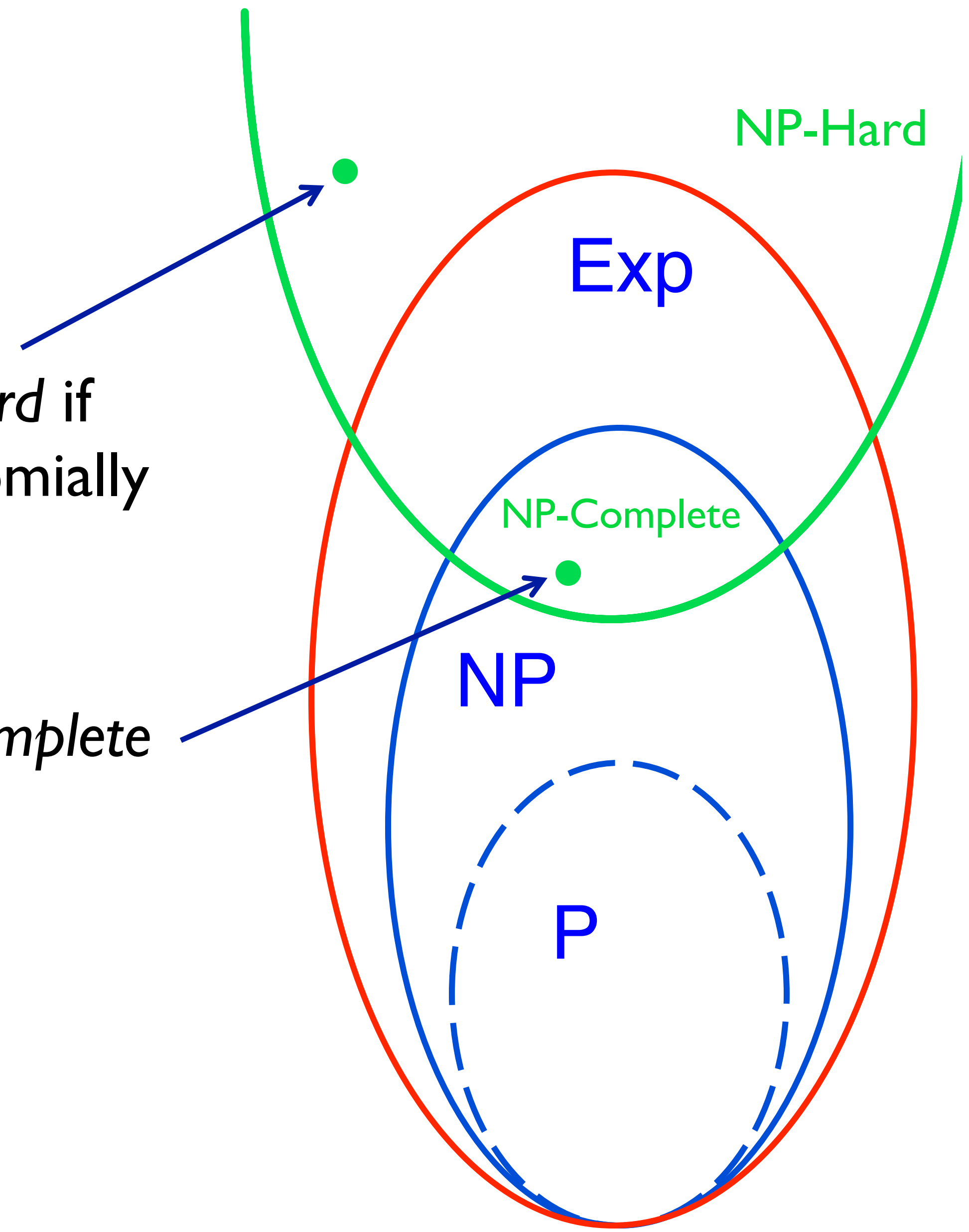
NP-Completeness

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 (cont.)

Thousands of important problems have been shown to be NP-complete.

The general belief is that there is no efficient algorithm for any NP-complete problem, but no proof of that belief is known.

Examples: SAT, clique, vertex cover, IndpSet, Ham tour, TSP, bin packing… Basically, everything we've seen that's in NP but not known to be in P

Proving a problem is NP-complete

Technically, for condition (2) we have to show that every problem in NP is reducible to B.
(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.

Alt way to prove NP-completeness

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 NP-completeness of a new problem B in NP, it suffices to show that SAT or any other NP-complete problem is polynomial-time reducible to B.

Ex: IndpSet is NP-complete

3-SAT is NP-complete (S. Cook; see below)

3-SAT $\leq_p$ IndpSet

IndpSet is in NP

we showed this earlier

Therefore IndpSet is also NP-complete

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

More Reductions

SAT to Subset Sum (Knapsack)

Subset-Sum, AKA Knapsack

KNAP = { (w_1, w_2, …, w_n, C) | a subset of the w_i sums to C }

w_i's and C encoded in radix $r \geq 2$. (Decimal used in following example.)

Theorem: 3-SAT $\leq_p$ KNAP

Pf: given formula with p variables & q clauses, build KNAP instance with 2(p+q) w_i's, each with (p+q) decimal digits. For the 2p "literal" weights, H.O. p digits mark which variable; L.O. q digits show which clauses contain it. Two "slack" weights per clause mark that clause. See example below.

3-SAT $\leq_p$ KNAP

Formula: $(x \vee y \vee z) \wedge (\neg x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee z)$

		Variables			Clauses		
		x	y	z	$(x \vee y \vee z)$	$(\neg x \vee y \vee \neg z)$	$(\neg x \vee \neg y \vee z)$
Literals	w_1 (x)	1	0	0	1	0	0
	w_2 $(\neg x)$	1	0	0	0	1	1
	w_3 (y)		1	0	1	1	0
	w_4 $(\neg y)$		1	0	0	0	1
	w_5 (z)			1	1	0	1
	w_6 $(\neg z)$			1	0	1	0
Slack	w_7 (s_{11})				1	0	0
	w_8 (s_{12})				1	0	0
	w_9 (s_{21})					1	0
	w_{10} (s_{22})					1	0
	w_{11} (s_{31})						1
	w_{12} (s_{32})						1
	C	1	1	1	3	3	3

Correctness

Poly time for reduction is routine; details omitted. Again note that it does *not* look at satisfying assignment(s), if any, nor at subset sums, but the problem instance it builds captures one via the other...

If formula is satisfiable, select the literal weights corresponding to the true literals in a satisfying assignment. If that assignment satisfies k literals in a clause, also select (3 - k) of the "slack" weights for that clause. Total = C.

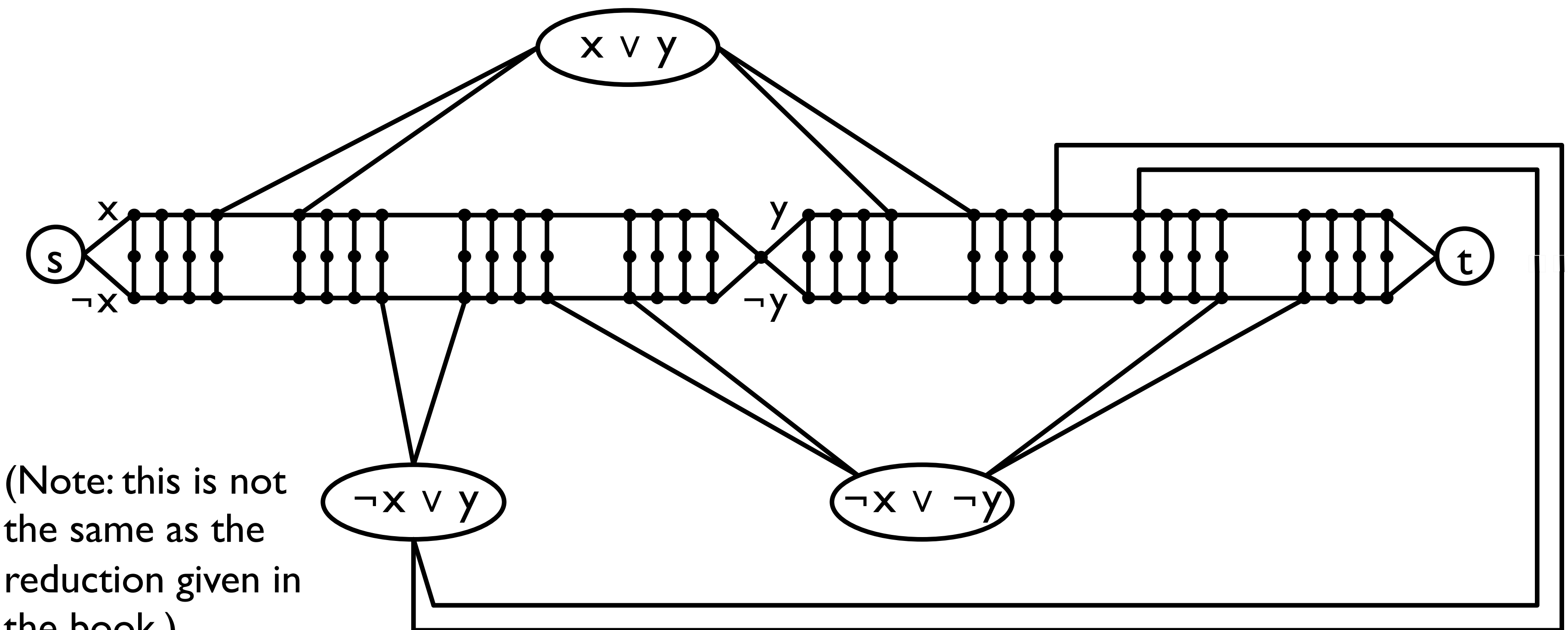
Conversely, suppose KNAP instance has a solution. Columns are decoupled since ≤ 5 one's per column, so no "carries" in sum (recall – weights are decimal). Since H.O. p digits of C are 1, exactly one of each pair of literal weights included in the subset, so it defines a valid assignment. Since L.O. q digits of C are 3, but at most 2 "slack" weights contribute to each, at least one of the selected literal weights must be 1 in that clause, hence the assignment satisfies the formula.

More Reductions

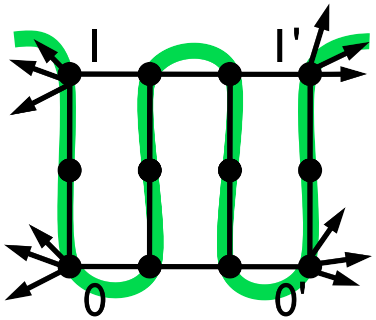
SAT to Undirected Hamilton Path

3-SAT $\leq_p$ UndirectedHamPath

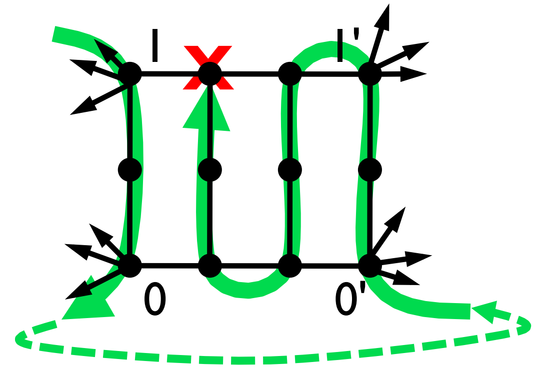
Example: $(x \vee y) \wedge (\neg x \vee y) \wedge (\neg x \vee \neg y)$



(Note: this is not the same as the reduction given in the book.)



Ham Path Gadget



Many copies of this 12-node gadget, each with one or more edges connecting each of the 4 corners to other nodes or gadgets (but no other edges to the 8 "internal" nodes).

Claim: There are only 2 Ham paths – one entering at 1, exiting at 1' (as shown); the other (by symmetry) 0→0'

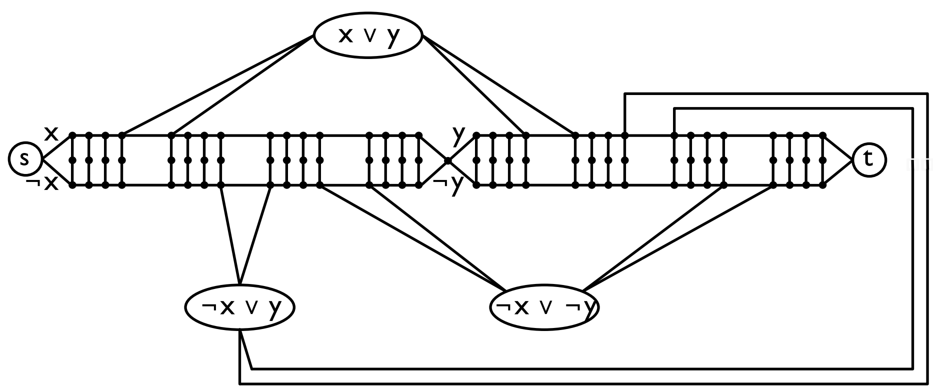
Pf: Note *: at 1st visit to any column, must next go to *middle* node in column, else it will subsequently become an untraversable "dead end."
WLOG, suppose enter at 1. By *, must then go down to 0. 2 cases:

Case a: (top left) If next move is to right, then * forces path up, left is blocked, so right again, * forces down, etc; out at 1'.

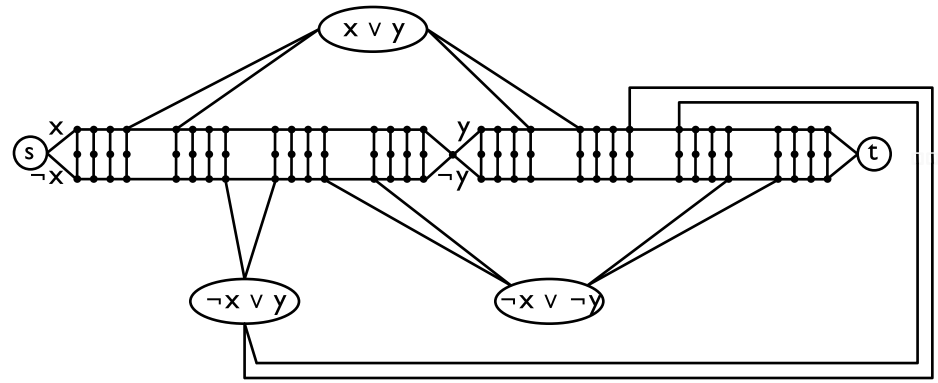
Case b: (top rt) if exit at 0, then path must eventually reenter at 0' or 1'. * forces next move to be up/down to the other of 0'/1'. Must then go left to reach the 2 middle columns, but there's *no exit* from them. So case b is impossible.

3-SAT $\leq_p$ UndirectedHamPath

Time for the reduction: to be computable in poly time it is necessary (but not sufficient) that G's size is polynomial in n, the length of the formula. Easy to see this is true, since G has q + 12 (p + m) + 1 = O(n) vertices, where q is the number of clauses, p is the number of instances of literals, and m is the number of variables. Furthermore, the structure is simple and regular, given the formula, so easily / quickly computable, but details are omitted. (More detail expected in your homeworks, e.g.) Again, reduction *builds* G, doesn't solve it.

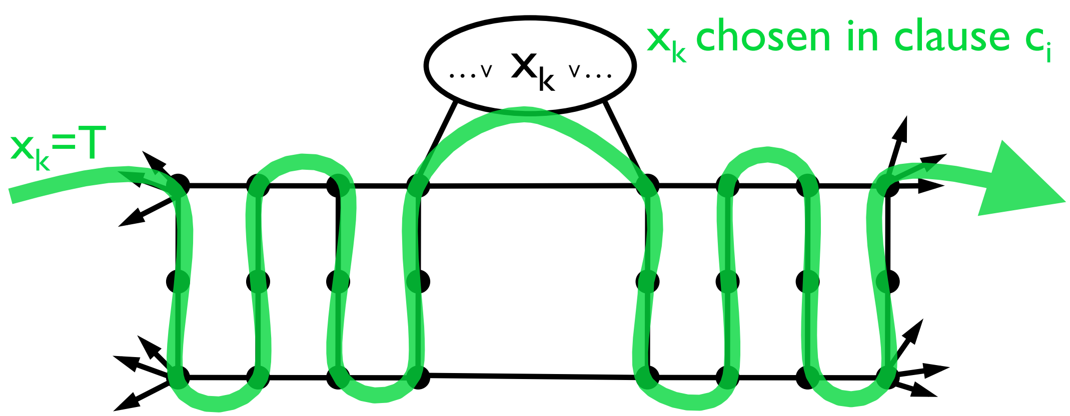


Correctness, I

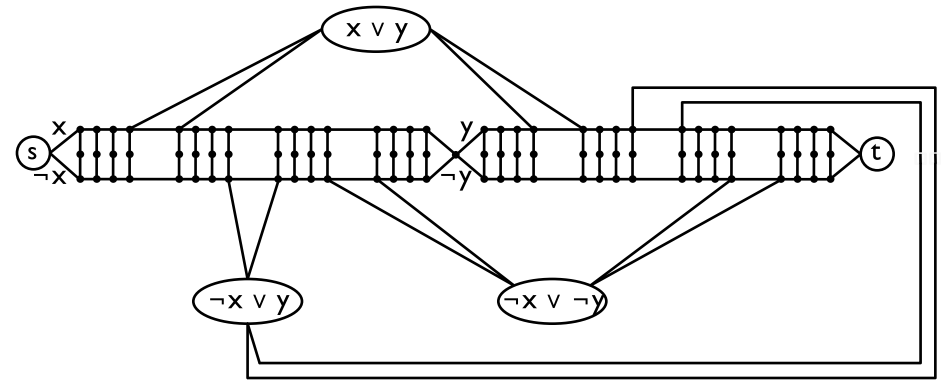


Ignoring the clause nodes, there are 2^m s-t paths along the "main chain," one for each of 2^m assignments to m variables.

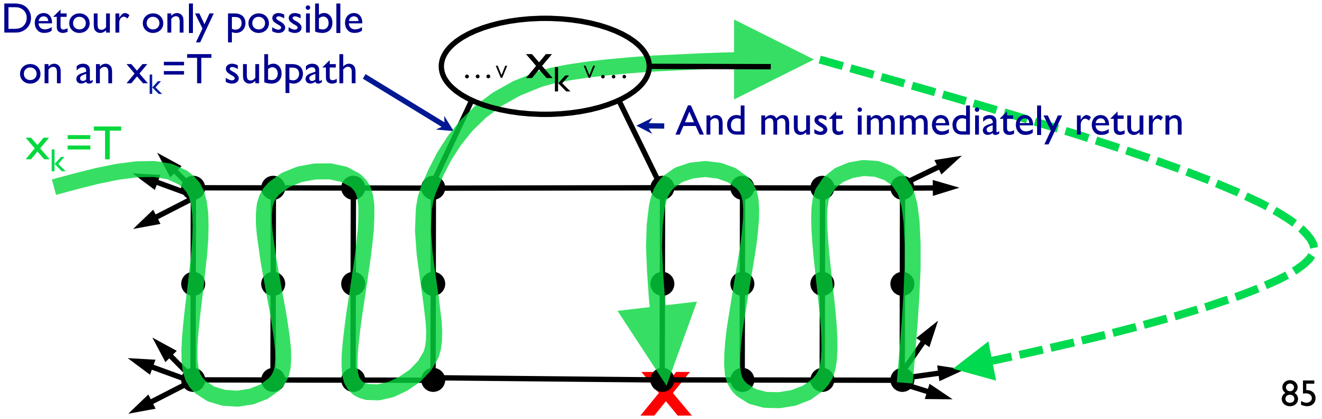
If f is satisfiable, pick a satisfying assignment, and pick a true literal in each clause. Take the corresponding "main chain" path; add a detour to/from c_i for the true literal chosen from clause i. Result is a Hamilton path.



Correctness, II



Conversely, suppose G has a Ham path. Obviously, the path must detour from the main chain to each clause node c_i. If it does not return *immediately* to the next gadget on main chain, then (by gadget properties on earlier slide), that gadget cannot be traversed. Thus, the Ham path must consistently use "top chain" or consistently "bottom chain" exits to clause nodes from each variable gadget. If top chain, set that variable True; else set it False. Result is a satisfying assignment, since each clause is visited from a "true" literal.



More Reductions

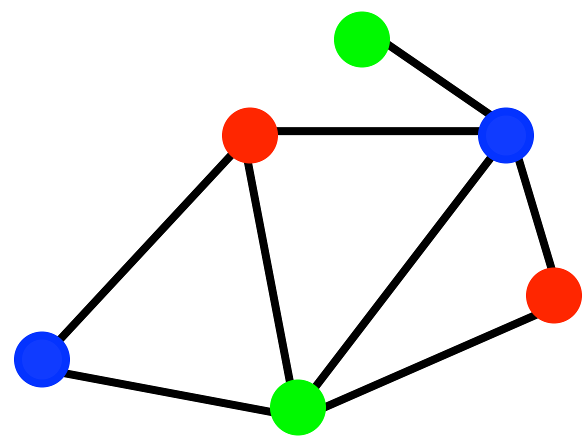
SAT to Coloring

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.

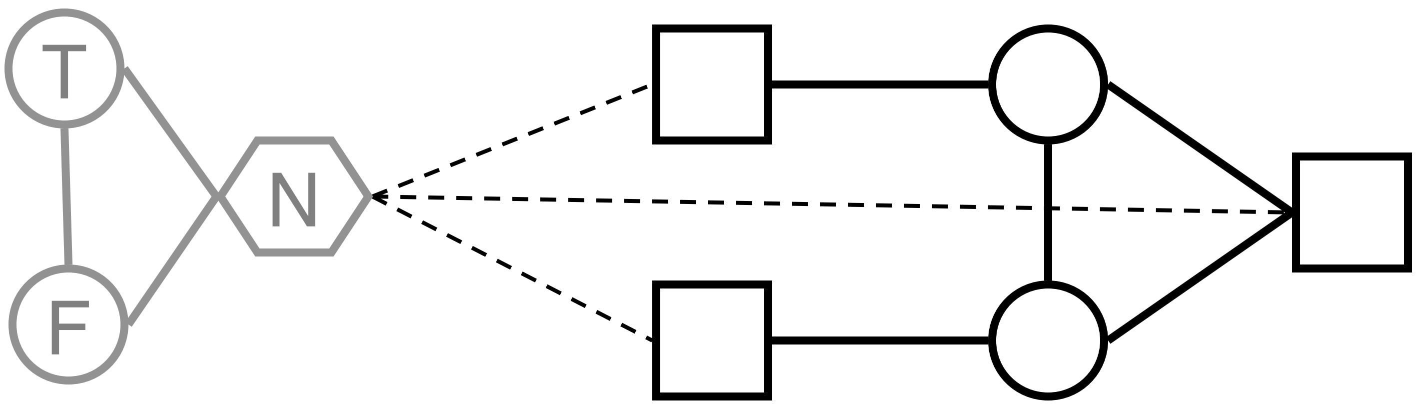
Example:



In NP? Exercise

A 3-Coloring Gadget:

In what ways can this be 3-colored?

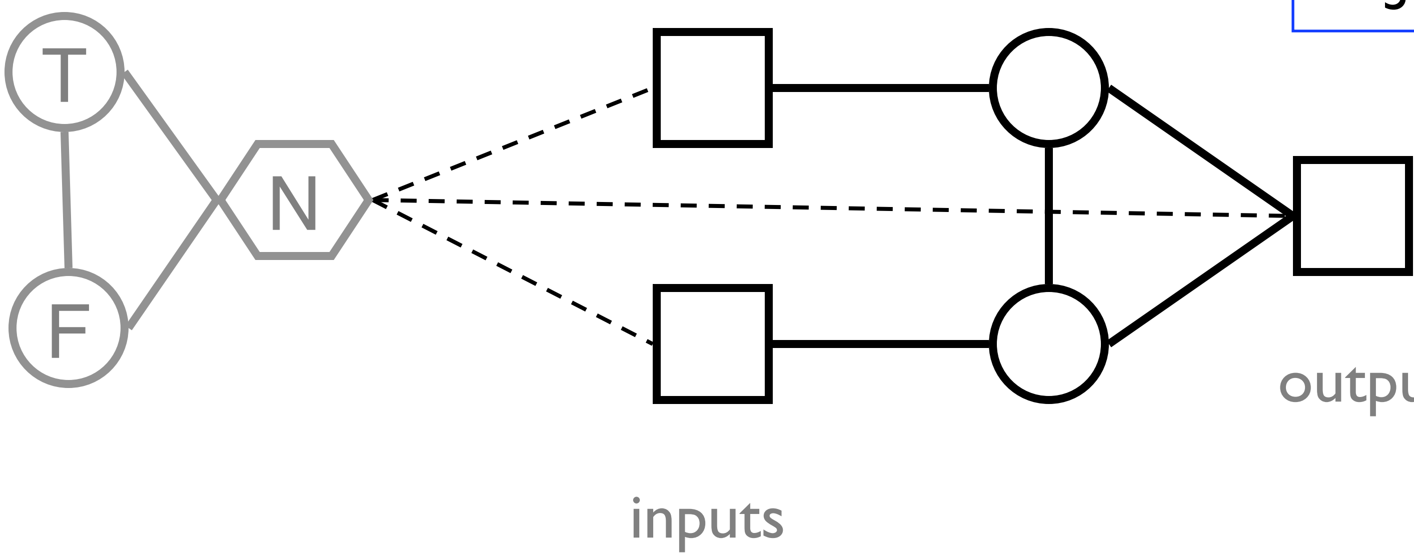


A 3-Coloring Gadget: "Sort of an OR gate"

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



NB: this is *not* the same gadget as used in KT 8.7

3SAT $\leq_p$ 3Color

$f\left(\right.$

3-SAT Instance:

- Variables: $x_1, x_2, \ldots$
- 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: $c = c_1 \wedge c_2 \wedge \ldots \wedge c_q$

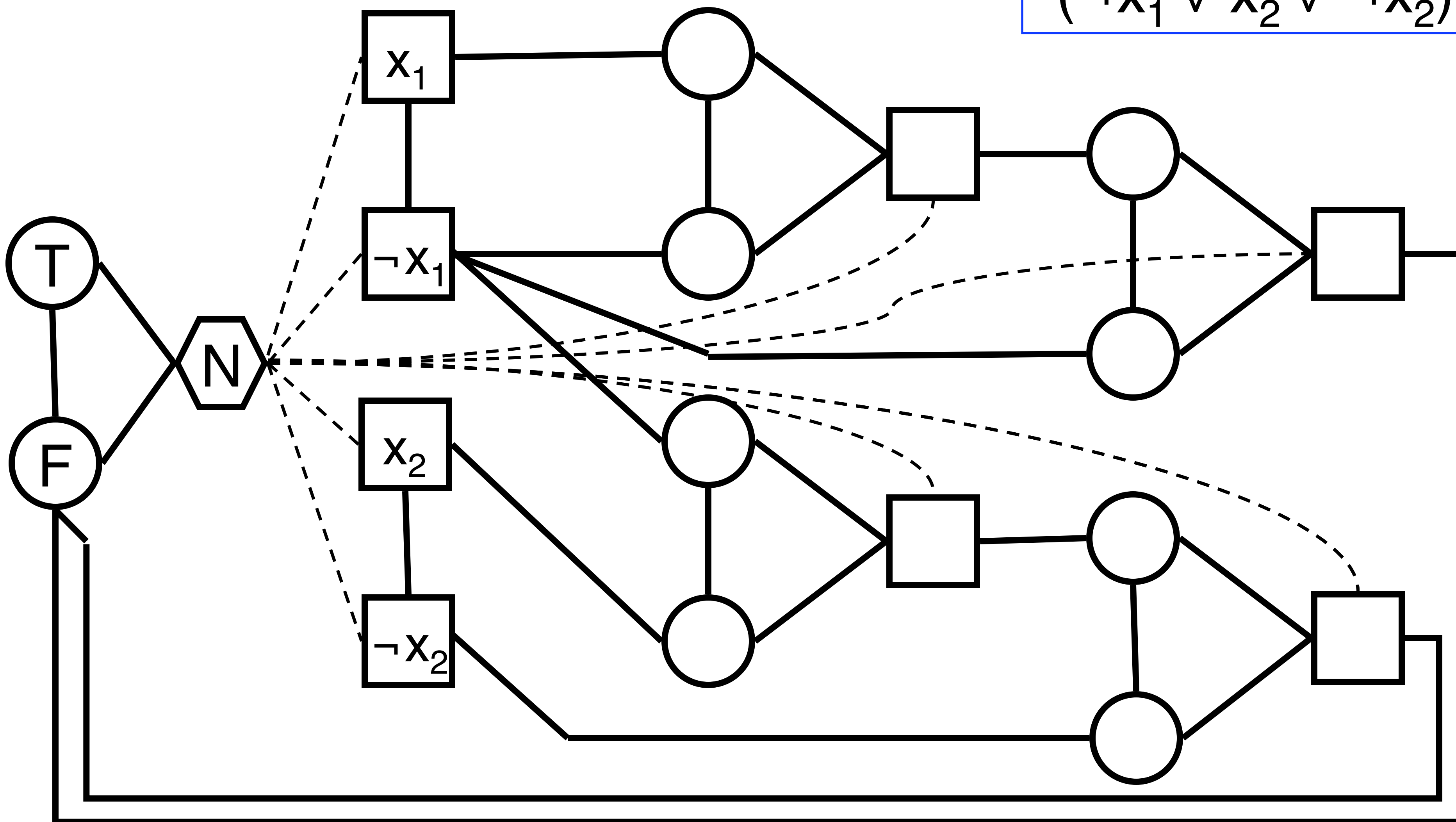
$\left.\right) =$

3Color Instance:

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

3SAT $\leq_p$ 3Color Example

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



6 q + 2 n + 3 vertices 13 q + 3 n + 3 edges

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

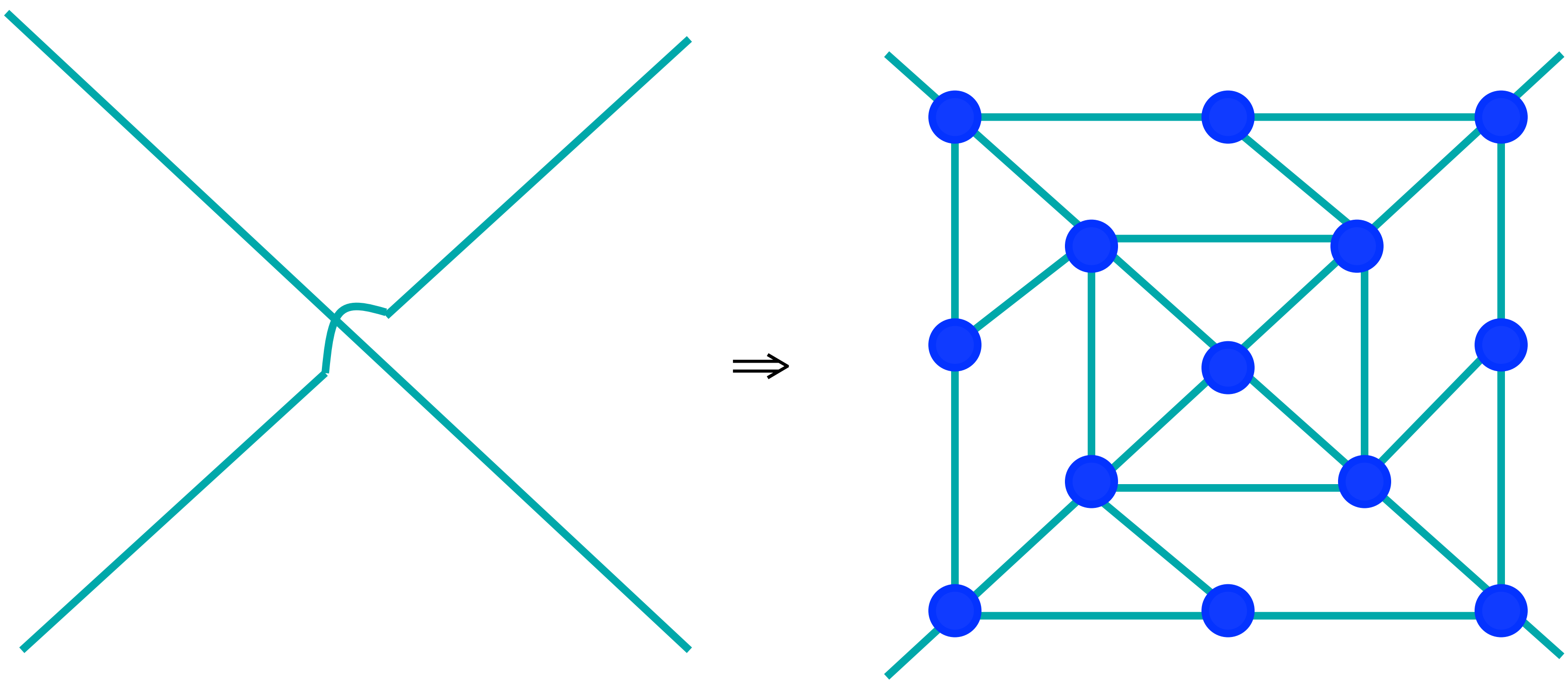
Summary of reduction function f:
Given formula, make G with T-F-N triangle, 1 pair of literal nodes per variable, 2 "or" gadgets per clause, connected as in example.
Note: *again, f does not know or construct satisfying assignment or coloring.*

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:

($\Rightarrow$) 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.
($\Leftarrow$) 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



Cook's Theorem

SAT is NP-Complete

"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'll sketch it below. But its essence is not so hard to grasp:

Generic "NP" problems: expo. search– is there a poly size "solution," verifiable by computer in poly time

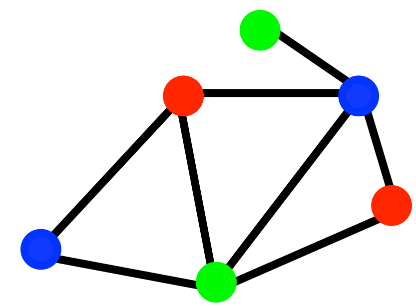
"SAT": is there a poly size assignment (the hint) satisfying the formula (the verifier)

Encode "solution" using Boolean variables. SAT mimics "is there a solution" via "is there an assignment". The "verifier" runs on a digital computer, and digital computers just do Boolean logic. "SAT" can mimic that, too, hence can verify that the assignment *actually* encodes a solution.

Examples

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

3-Coloring $\leq_p$ SAT



Given G = (V, E)

$\forall$ i in V, variables r_i, g_i, b_i encode color of i ← hint

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

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

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

← verifier

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

Vertex cover $\leq_p$ SAT

Given G = (V, E) and k

$\forall$ i in V, variable x_i encodes inclusion of i in cover ← hint

$$\underbrace{\wedge_{(i,j) \in E} (x_i \vee x_j)}_{\text{every edge covered by one end or other}} \wedge \underbrace{\text{"number of True } x_i \text{ is} \leq k\text{"}}_{\text{possible in 3 CNF, but technically messy; basically, count 1's}}$$ ← verifier

Hamilton Circuit $\leq_p$ SAT

Given $G = (V, E)$ [encode, e.g.: $e_{ij} = 1 \Leftrightarrow$ edge (i,j)]

$\forall$ i,j in V, variables x_{ij}, encode "j follows i in the tour" $\leftarrow$ hint

$$\underbrace{\wedge_{(i,j)} (x_{ij} \Rightarrow e_{ij})}_{\text{the path follows actual edges}} \wedge \underbrace{\text{"it's a permutation"}}_{\text{every row/column has exacty 1 one bit}} \wedge \underbrace{\text{"cycle length = n"}}_{X^n = I\text{, no smaller power k has } X^k ii=1}$$

verifier

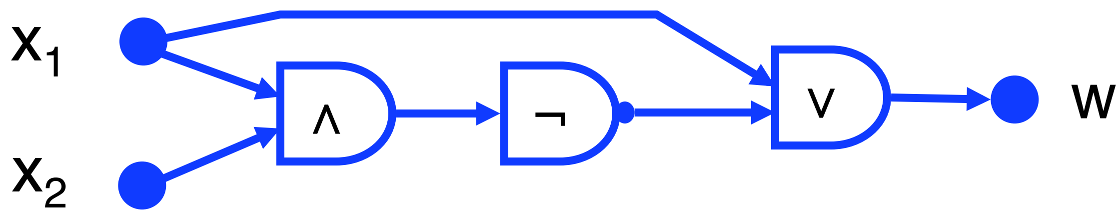
Cook's Theorem

Every problem in NP is reducible to SAT

Idea of proof is extension of above examples, but done in a general way, based on the definition of NP – show how the SAT formula can simulate whatever (polynomial time) computation the verifier does.

Cook proved it directly, but easier to see via an intermediate problem – Satisfiability of *Circuits* rather than Formulas

Boolean Circuits



Directed *acyclic* graph *(yes, "circuit" is a misnomer…)*

Vertices = Boolean logic gates (∧, ∨, ¬, …)

Multiple input bits (x_1, x_2, …)

Single output bit (w)

Gate values as expected (e.g. propagate vals by depth to x_i's)

Boolean Circuits and Complexity

Two Problems:

Circuit Value: given a circuit *and an assignment* of values to its inputs, is its output = 1?

Circuit SAT: given a circuit, *is there* an assignment of values to its inputs such that output =1?

Complexity:

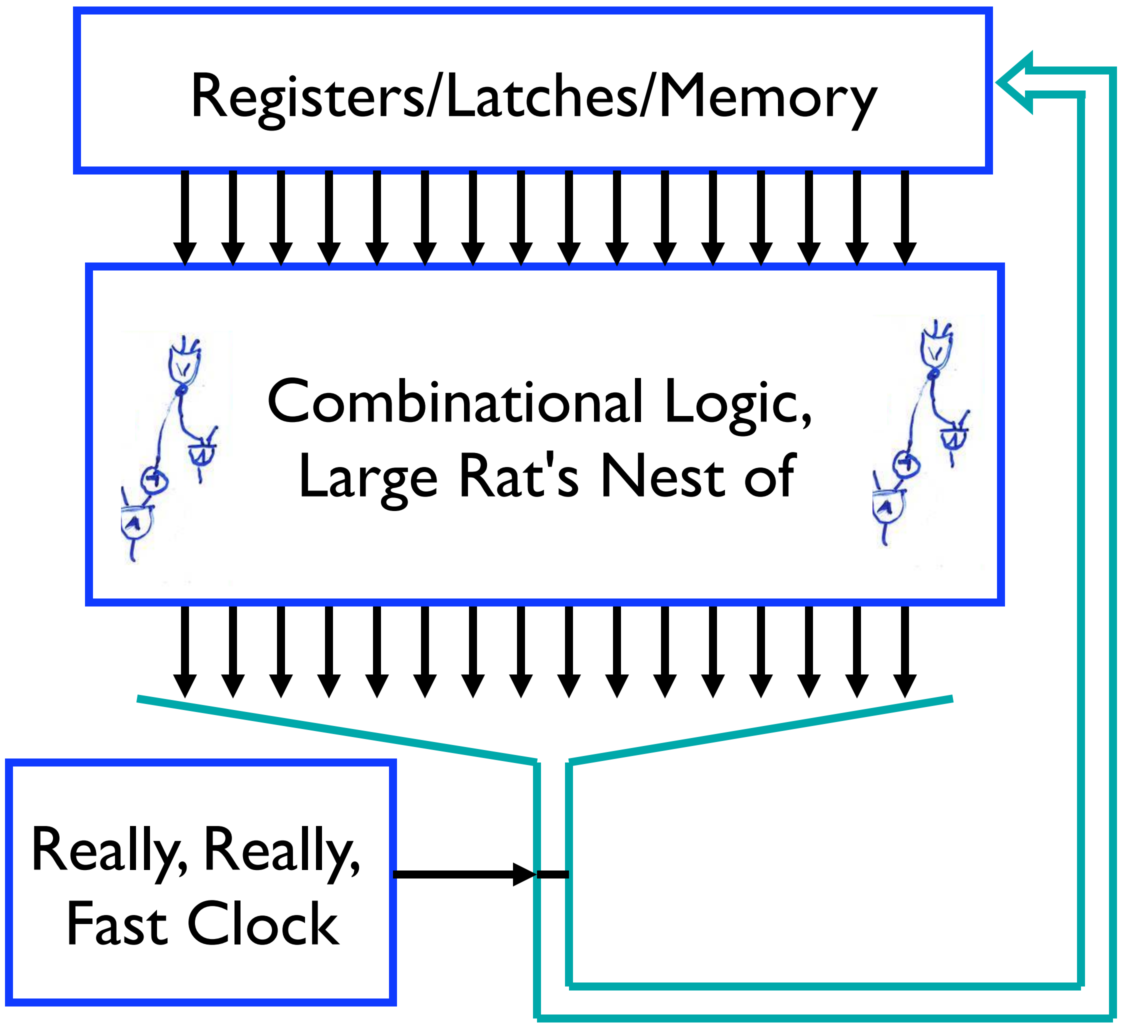
Circuit Value Problem is in P

Circuit SAT Problem is in NP

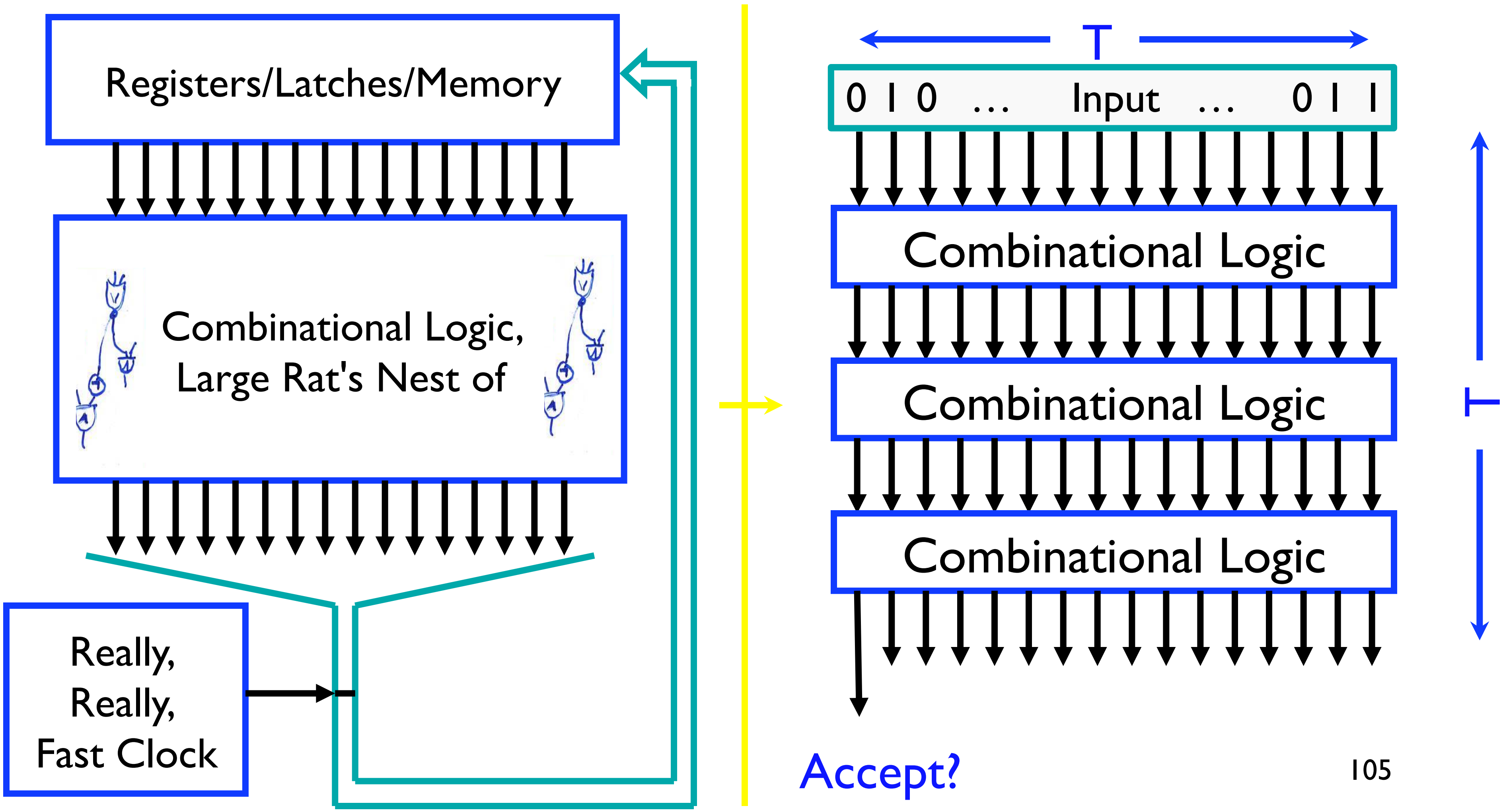
Sketched below

Given implementation of computers via Boolean circuits, it may be unsurprising that they are *complete* in P/NP, resp.

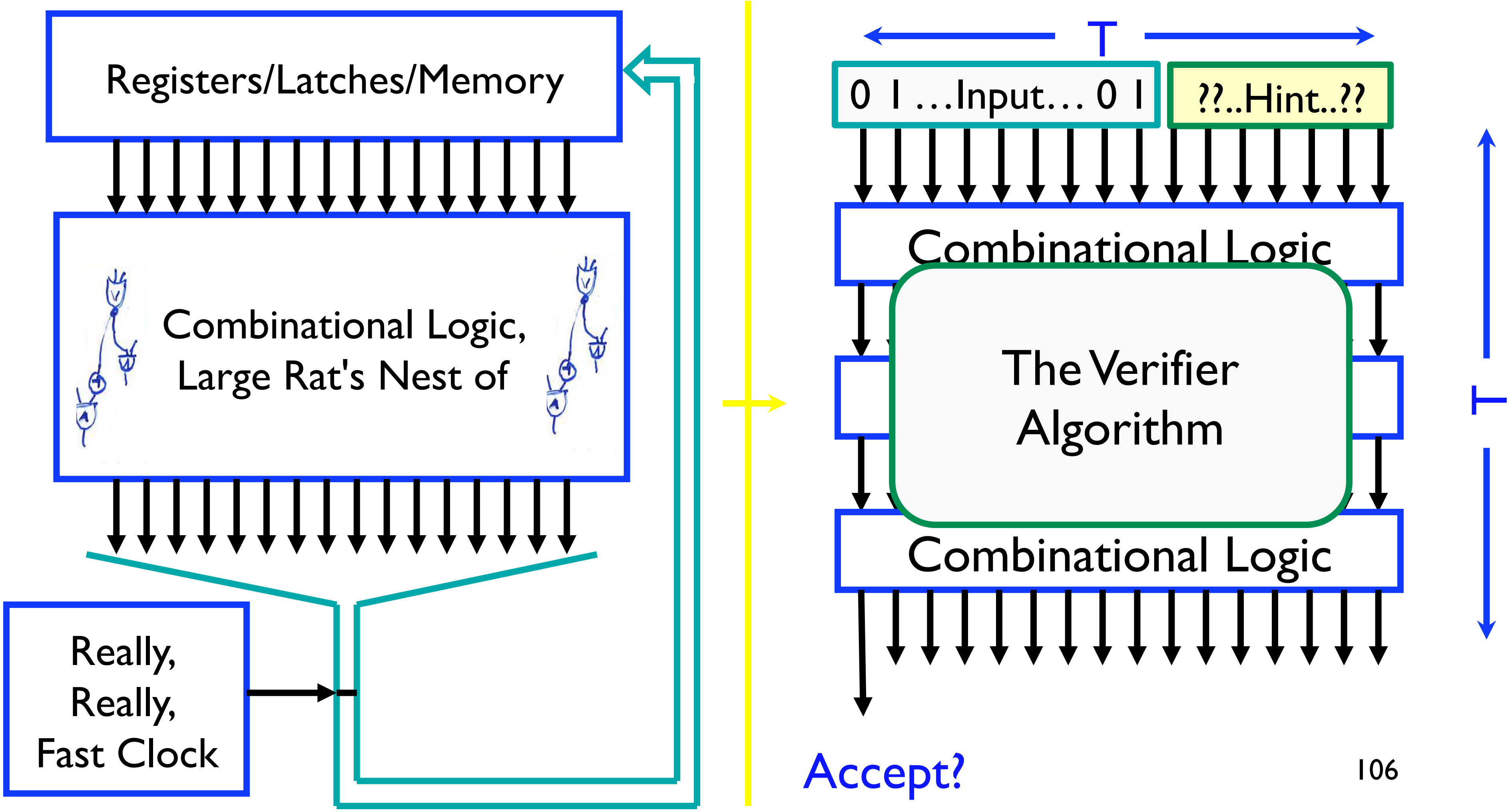
Detailed Logic Diagram, Intelorola Pentathlon® 6600



P Is Reducible To The Circuit *Value* Problem



NP Is Reducible To The Circuit *Satisfiability* Problem



Correctness of NP $\leq_p$ CircuitSAT

?

Correctness of NP $\leq_p$ CircuitSAT

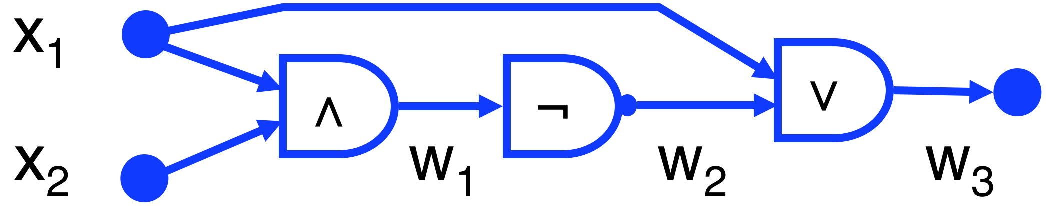
Fix an arbitrary NP-problem, a verifier alg V(x,h) for it, and a bound n^k on hint length/run time of V, show:

1) In poly time, given x, can output a circuit C as above,
2) ∃ h s.t. V(x,h)="yes" ⇒ C is satisfiable (namely by h), and
3) C is satisfiable (say, by h) ⇒ ∃ h s.t. V(x,h)="yes"

1) is perhaps very tedious, but mechanical–you are "compiling" the verifier's code into hardware (just enough hardware to handle inputs of length |x|)

2) & 3) exploit the fact that C simulates V, with C's "hint bit" inputs exactly corresponding to V's input h.

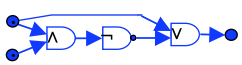
Circuit-SAT $\leq_p$ 3-SAT



$$\underbrace{(w_1 \Leftrightarrow (x_1 \wedge x_2))}_{\text{Replace with 3-CNF Equivalent:}} \wedge (w_2 \Leftrightarrow (\neg w_1)) \wedge (w_3 \Leftrightarrow (w_2 \vee x_1)) \wedge w_3$$

¬clause
↓
Truth Table
↓
DNF
↓
DeMorgan
↓
CNF

x_1	x_2	w_1	$x_1 \wedge x_2$	$\neg(w_1 \Leftrightarrow (x_1 \wedge x_2))$	
0	0	0	0	0	
0	0	1	0	1	$\leftarrow \neg x_1 \wedge \neg x_2 \wedge w_1$
0	1	0	0	0	
0	1	1	0	1	$\leftarrow \neg x_1 \wedge x_2 \wedge w_1$
1	0	0	0	0	
1	0	1	0	1	$\leftarrow x_1 \wedge \neg x_2 \wedge w_1$
1	1	0	1	1	$\leftarrow x_1 \wedge x_2 \wedge \neg w_1$
1	1	1	1	0	

f() $= (x_1 \vee x_2 \vee \neg w_1) \wedge (x_1 \vee \neg x_2 \vee \neg w_1) \wedge (\neg x_1 \vee x_2 \vee \neg w_1) \wedge (\neg x_1 \vee \neg x_2 \vee w_1) \ldots$

Build truth table clause-by-clause vs whole formula, so $n*2^3$ vs 2^n rows

Correctness of "Circuit-SAT $\leq_p$ 3-SAT"

Summary of reduction function f: Given circuit, add variable for every gate's value, build clause for each gate, satisfiable iff gate value variable is appropriate logical function of its input variables, convert each to CNF via standard truth-table construction. Output conjunction of all, plus output variable. *Note: as usual, does not know whether circuit or formula are satisfiable or not; does not try to find satisfying assignment.*

Correctness:

Show f is poly time computable: A key point is that formula size is linear in circuit size; mapping basically straightforward; details omitted.

Show c in Circuit-SAT iff f(c) in SAT:
($\Rightarrow$) Given an assignment to x_i's satisfying c, extend it to w_i's by evaluating the circuit on x_i's gate by gate. Show this satisfies f(c).
($\Leftarrow$) Given an assignment to x_i's & w_i's satisfying f(c), show x_i's satisfy c (with gate values given by w_i's).

Thus, 3-SAT is NP-complete.

Coping with NP-hardness

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, degree 3 graphs, trees,…?

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 dynamic programming, backtrack, branch & bound, pruning

Heuristics – usually a good approx and/or fast

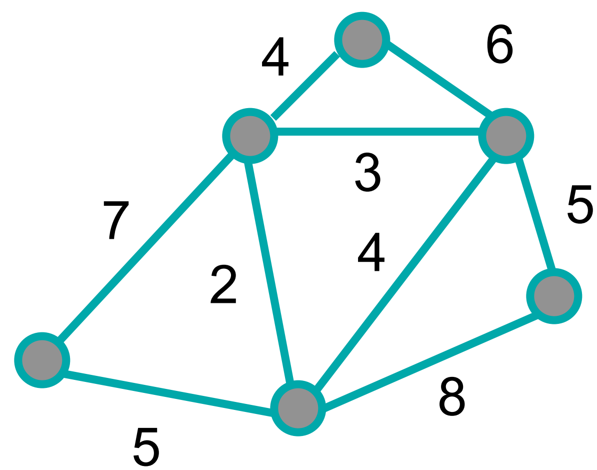
NP-complete problem: TSP

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.

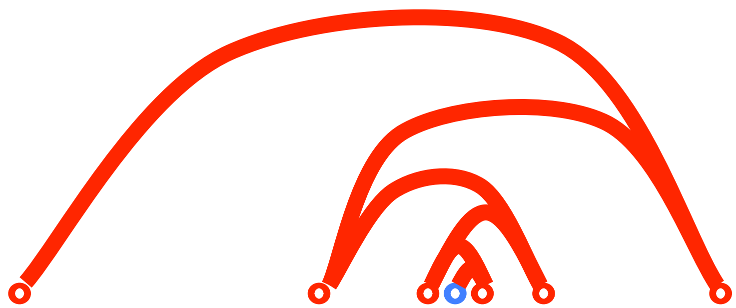
Example:

b = 34



TSP - Nearest Neighbor Heuristic

Recall NN Heuristic–go to nearest unvisited vertex



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

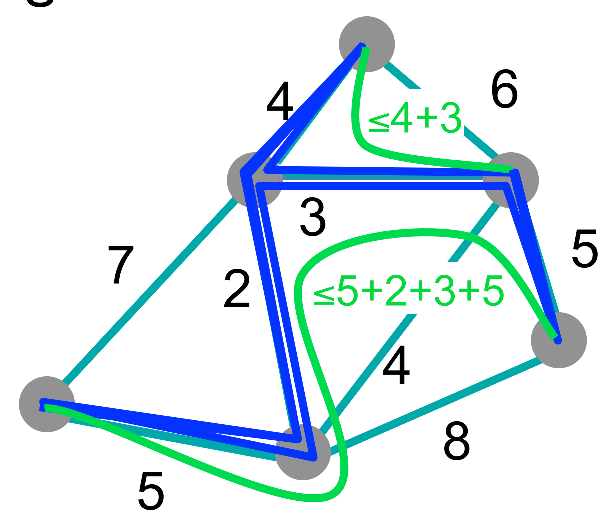
$$\lim_{n \to \infty} \frac{NN}{OPT} \to \infty$$

(above example is not that bad)

2x Approximation to EuclideanTSP

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

1.5x Approximation to EuclideanTSP

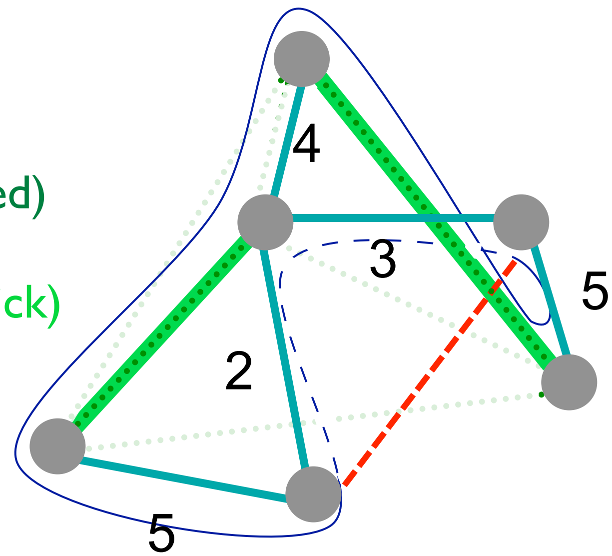
Find MST (solid edges)

Connect odd-degree tree vertices (dotted)

Find min cost matching among them (thick)

Find *Euler* Tour (thin)

Shortcut (dashed)



Shortcut $\leq$ ET $\leq$ MST + TSP/2 < 1.5* TSP

Cost of matching $\leq$ TSP/2
(next slide)

Matching ≤ TSP/2

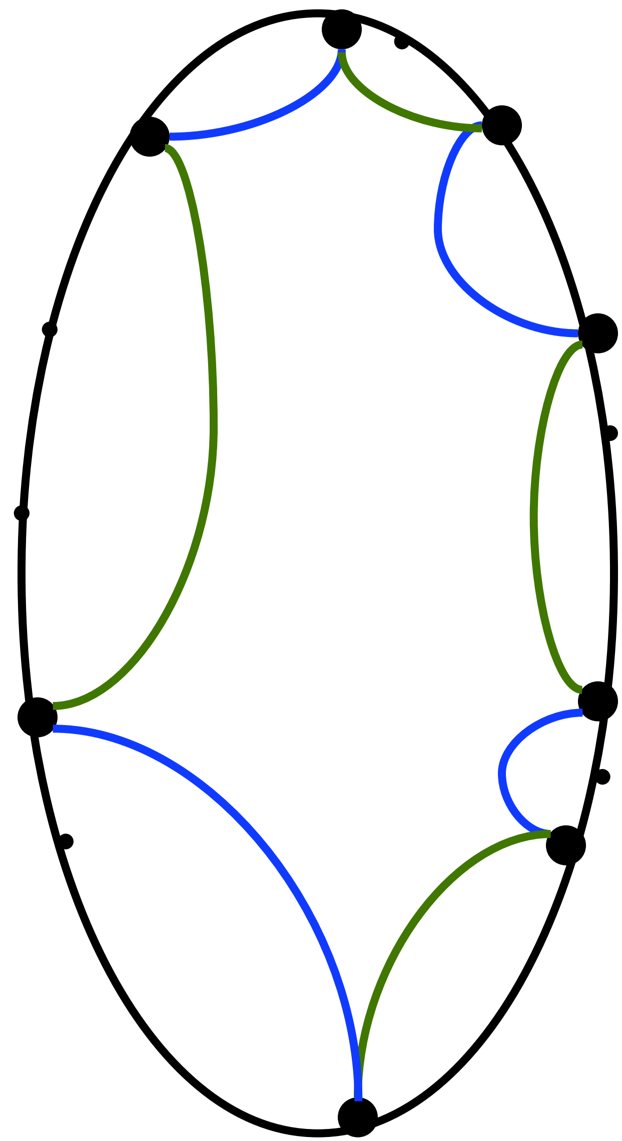
Oval = TSP

Big dots = odd tree nodes
(Exercise: show every graph has an even number of odd degree vertices)

Blue, Green = 2 matchings

Blue + Green ≤ TSP (triangle inequality)

So min matching ≤ TSP/2



P / NP Summary

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

E.g., delete clause nodes in HAM reduction. It's still true that "satisfiable $\Rightarrow$ G has a Ham path", but path doesn't necessarily give a satisfying assignment.

P

Many important problems are in P: solvable in deterministic polynomial time

- Details are the fodder of algorithms courses. We've seen a few examples here, plus many other examples in other courses

Few problems *not* in P are routinely solved;

- For those that are, practice is usually restricted to small instances, or we're forced to settle for approximate, suboptimal, or heuristic "solutions"

A major goal of complexity theory is to delineate the boundaries of what we can feasibly solve

NP

The tip-of-the-iceberg in terms of problems conjectured not to be in P, but a very important tip, because

a) they're very commonly encountered, probably because

b) they arise naturally from basic "search" and "optimization" questions.

Definition: poly time verifiable; "guess and check", "is there a…" – also useful

NP-completeness

Defn & Properties of $\leq_p$

A is NP-hard: everything in NP reducible to A

A is NP-complete: NP-hard and *in* NP

"the hardest problems in NP"

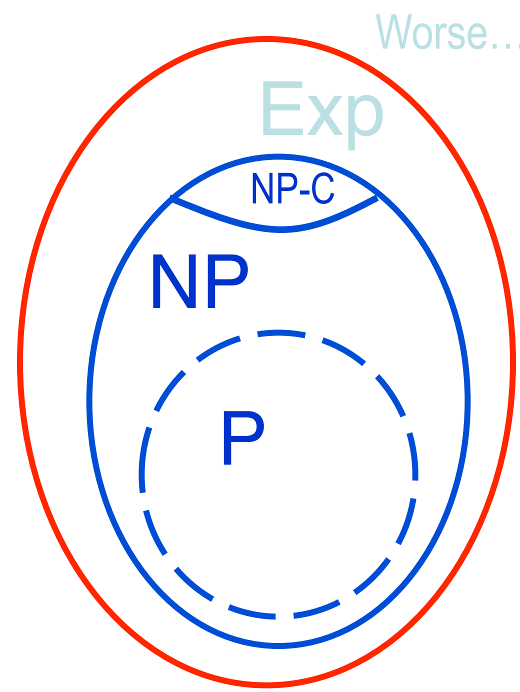
"All alike under the skin"

Most known natural problems in NP are complete

#1: 3CNF-SAT

Many others: Clique, VertexCover, HamPath, Circuit-SAT,…

Summary



Big-O – good

P – good

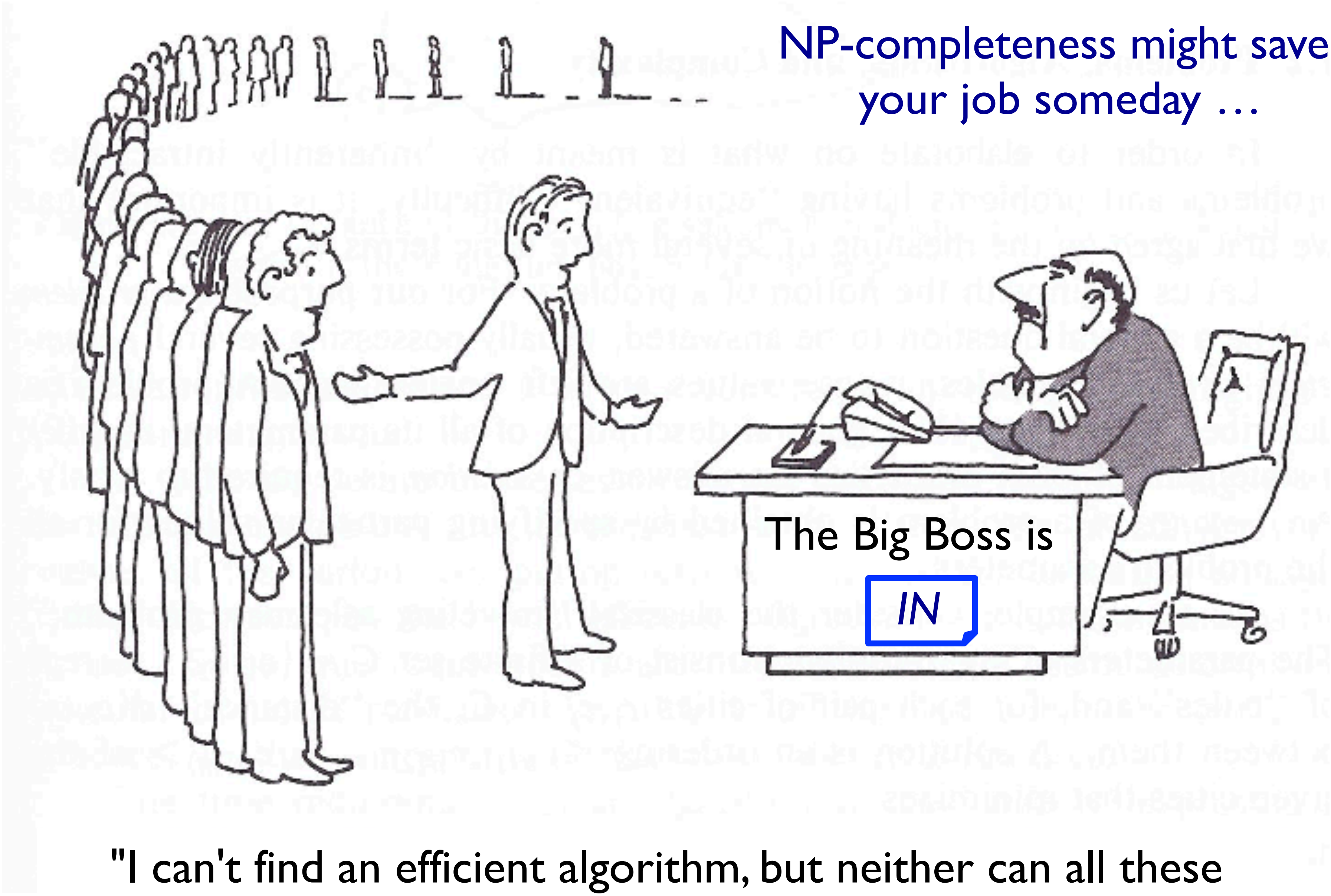
Exp – bad

Exp, but hints help? NP

NP-hard, NP-complete – bad (I bet)

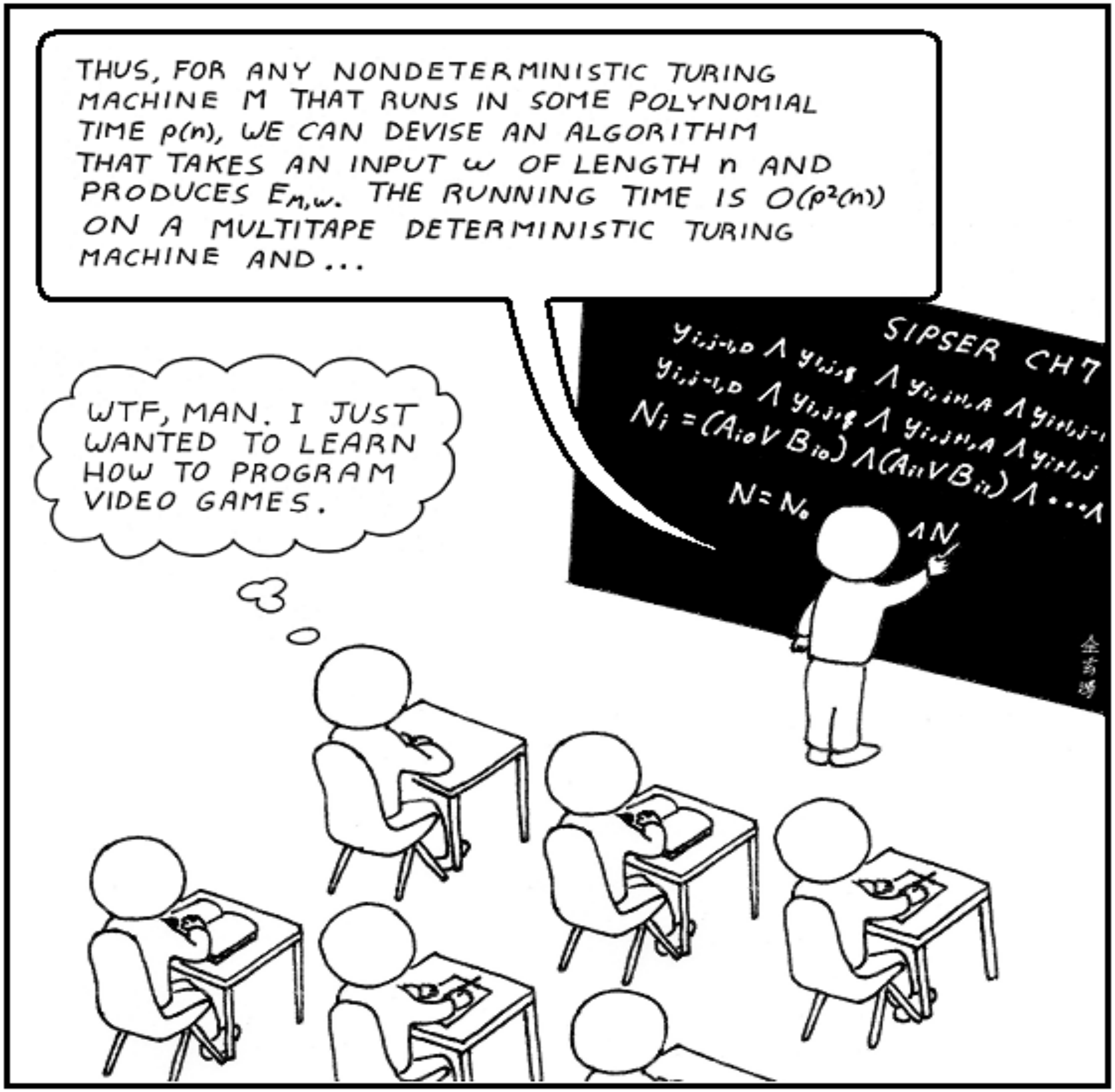
To show NP-complete – reductions

NP-complete = hopeless? – no, but you need to lower your expectations: heuristics, approximations and/or small instances.



"I can't find an efficient algorithm, but neither can all these famous people."

[Garey & Johnson, 1979]


THUS, FOR ANY NONDETERMINISTIC TURING MACHINE M THAT RUNS IN SOME POLYNOMIAL TIME $p(n)$, WE CAN DEVISE AN ALGORITHM THAT TAKES AN INPUT w OF LENGTH n AND PRODUCES $E_{M,w}$. THE RUNNING TIME IS $O(p^2(n))$ ON A MULTITAPE DETERMINISTIC TURING MACHINE AND...
WTF, MAN. I JUST WANTED TO LEARN HOW TO PROGRAM VIDEO GAMES.
SIPSER CH7