P vs NP

Is everything easy?

No, some problems (halting, …) are uncomputable

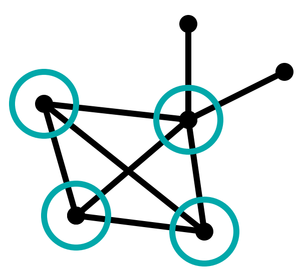
Is everything *computable* easy?

Sadly, no …

The Clique Problem

Given: a graph G=(V,E) and an integer k

Question: is there a subset U of V with
|U| ≥ k such that every pair of vertices in U is joined by an edge.

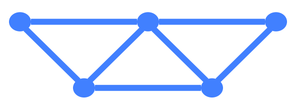


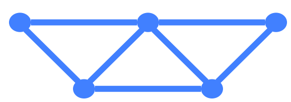
Some Convenient Technicalities

"Problem" – the general case

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

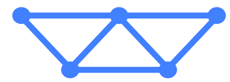
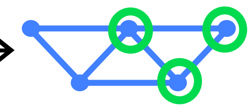
"Problem Instance" – the specific cases

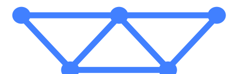
Ex: Does  contain a 4-clique? (no)

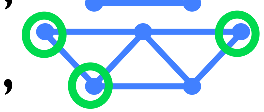
Ex: Does  contain a 3-clique? (yes)

Some Convenient Technicalities

Three kinds of problem:

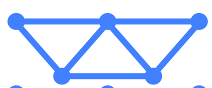
Search: *Find* a k-clique in G (3, ) → 

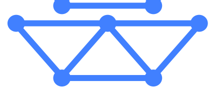
Decision: *Is there* a k-clique in G (3, ) → yes

Verification: Is *this* a k-clique in G (3, ) → no

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

But we'll sometimes be a little sloppy and use CLIQUE to mean the associated search problem

Difficulty/Utility

Computational Difficulty: verify ≤ decide ≤ search

Utility: ditto

In fact, decision and search are often equally difficult, but whether or not that holds for a particular problem, by the above, if we could show a *lower* bound on time for the decision problem, that implies a lower bound for the harder, more useful search versions as well, and the decision version is mathematically simpler, so the theory has emphasized the decision forms – another convenient technicality.

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?

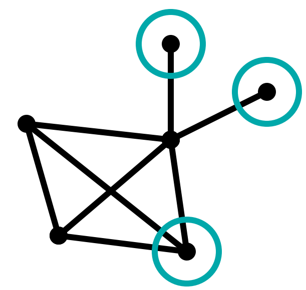
$$(x \lor y \lor z) \land (\neg x \lor y \lor \neg z) \land$$
$$(x \lor \neg y \lor z) \land (\neg x \lor \neg y \lor z) \land$$
$$(\neg x \lor \neg y \lor \neg z) \land (x \lor y \lor z) \land$$
$$(x \lor \neg y \lor z) \land (x \lor y \lor \neg z)$$

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

More Problems

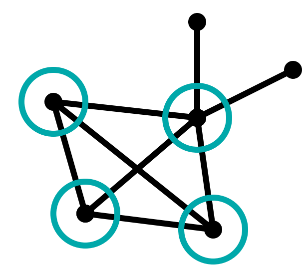
Independent-Set:

Pairs $\langle G,k \rangle$, where $G=(V,E)$ is a graph and k is an integer, for which there is a subset U of V with $|U| \geq k$ such that no two vertices in U are joined by an edge.



Clique:

Pairs $\langle G,k \rangle$, where $G=(V,E)$ is a graph and k is an integer k, for which there is a subset U of V with $|U| \geq k$ 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.

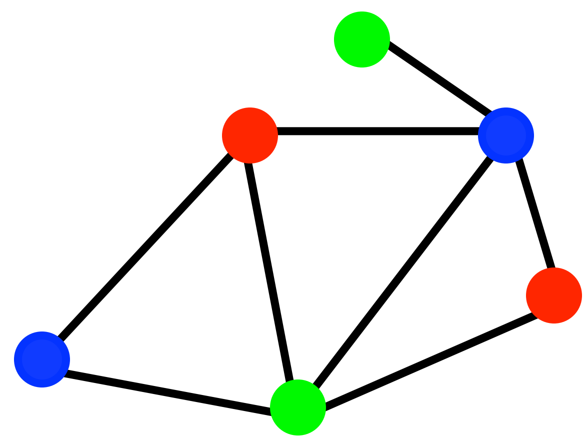
TSP:

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.

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:



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 $\geq k$

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 output of the search version would be.)

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)

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...”
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

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?

Solving NP problems without hints

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

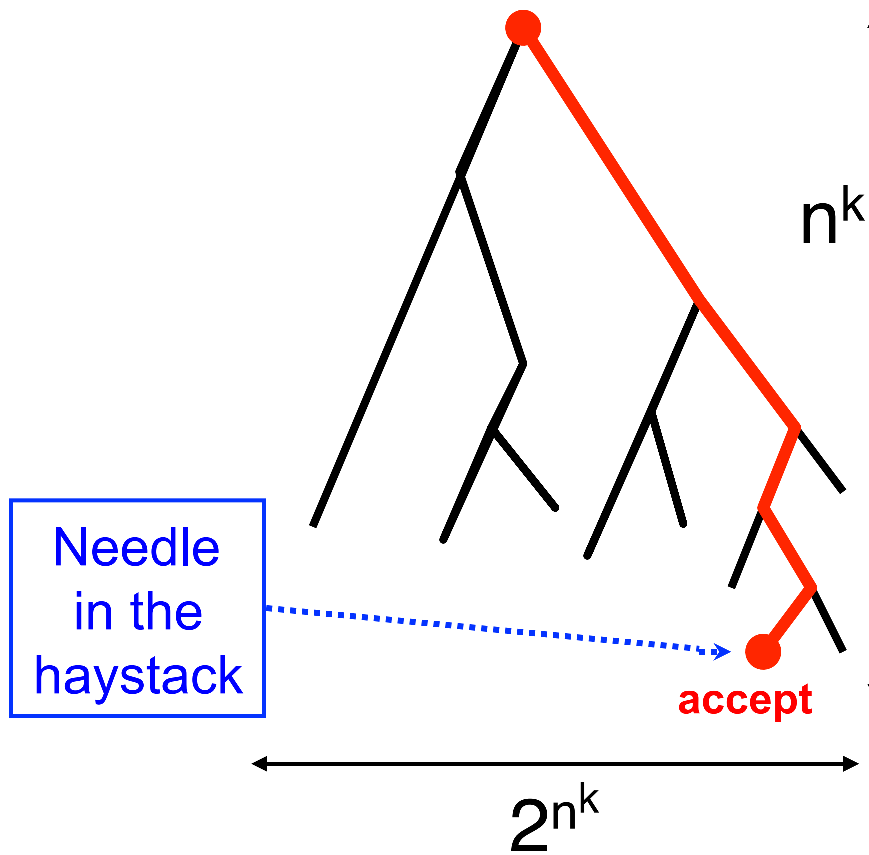
- try all possible hints; check each one to see if it works.
- Exponential time:
  - 2^n truth assignments for n variables
  - n! possible TSP tours of n vertices
  - $\binom{n}{k}$ possible k element subsets of n vertices
  - etc.

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

P vs NP vs Exponential Time

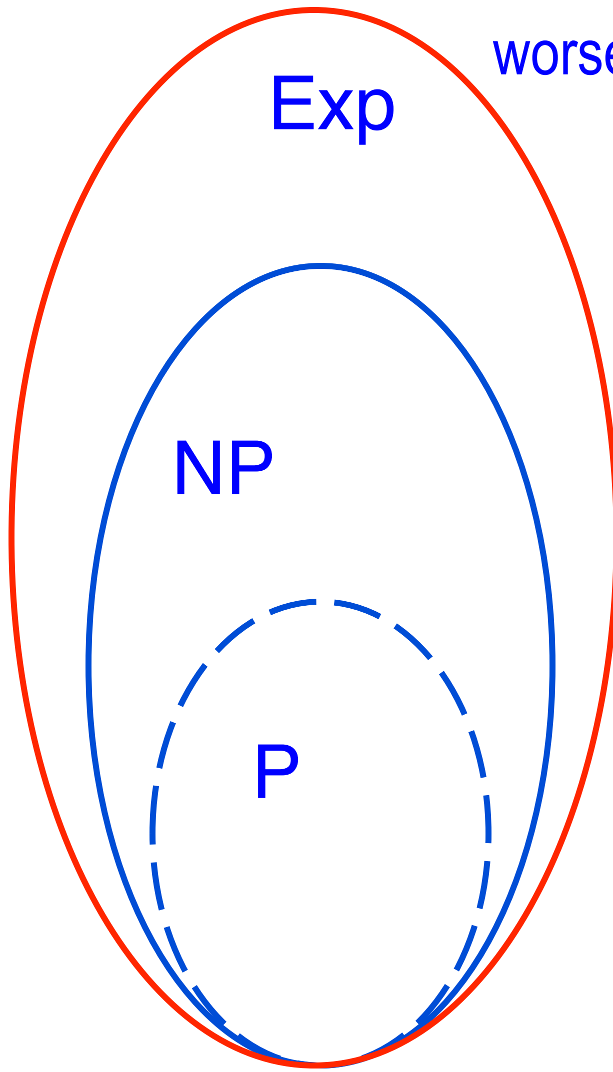
Theorem: Every problem in NP can be solved deterministically in exponential time

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

And worse!



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)

Review from previous lecture

Examples in NP:

SAT, short/long paths, Euler/Ham tours, clique, indp set…

Common feature/definition:

"… there is an X with property Y …" where the property is easy (P-time) to verify, *given X*, but there are exponentially many potential X's to search among.

P $\subseteq$ NP $\subseteq$ Exp (at least 1 containment is proper; likely both)

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

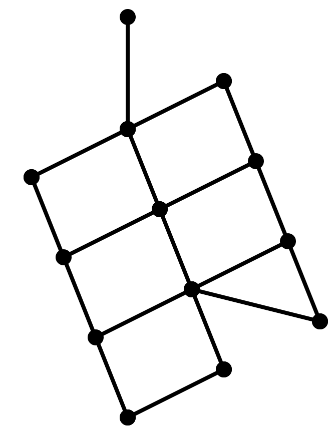
Many interesting, useful, natural, well-studied problems known to be NP-complete

With rare exceptions, no one routinely succeeds in finding exact solutions to large, arbitrary instances

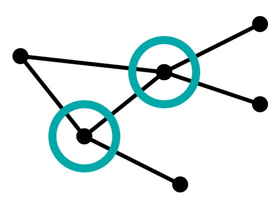
Another NP problem: Vertex Cover

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

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

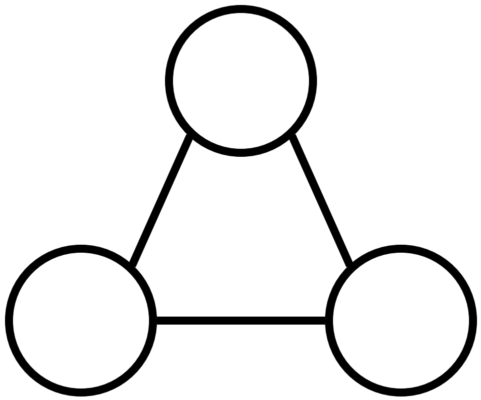


Example: Vertex cover of size ≤ 2.

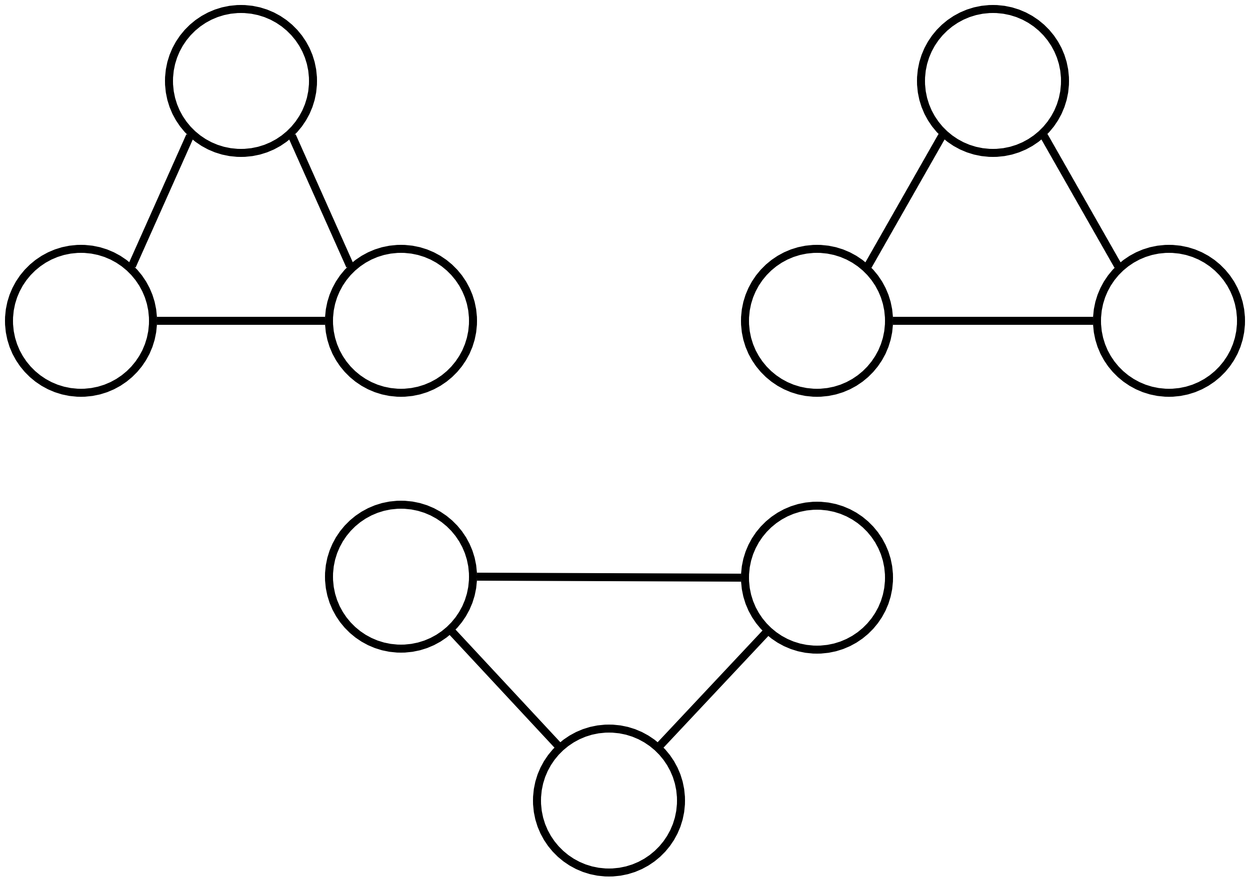


In NP? Exercise

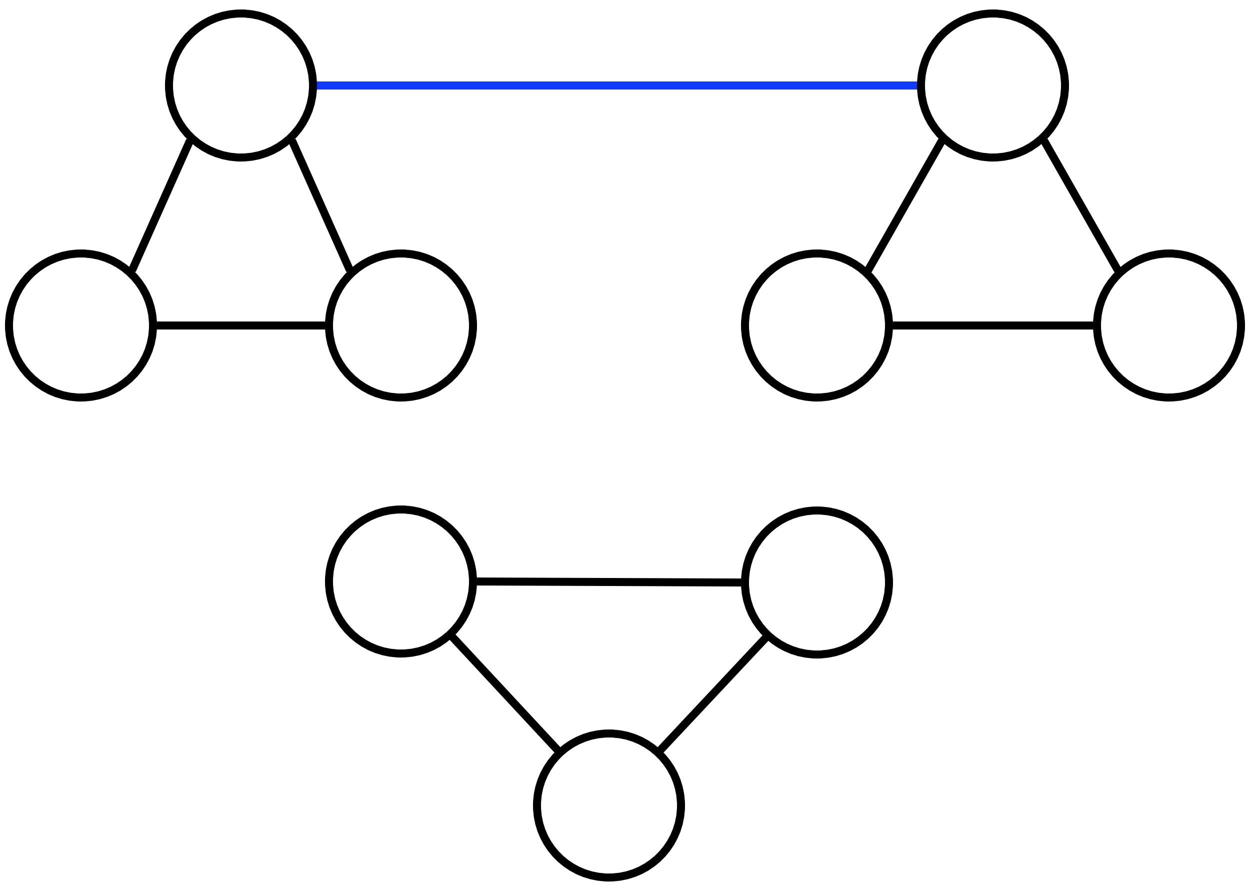
3SAT $\leq_p$ VertexCover



3SAT $\leq_p$ VertexCover

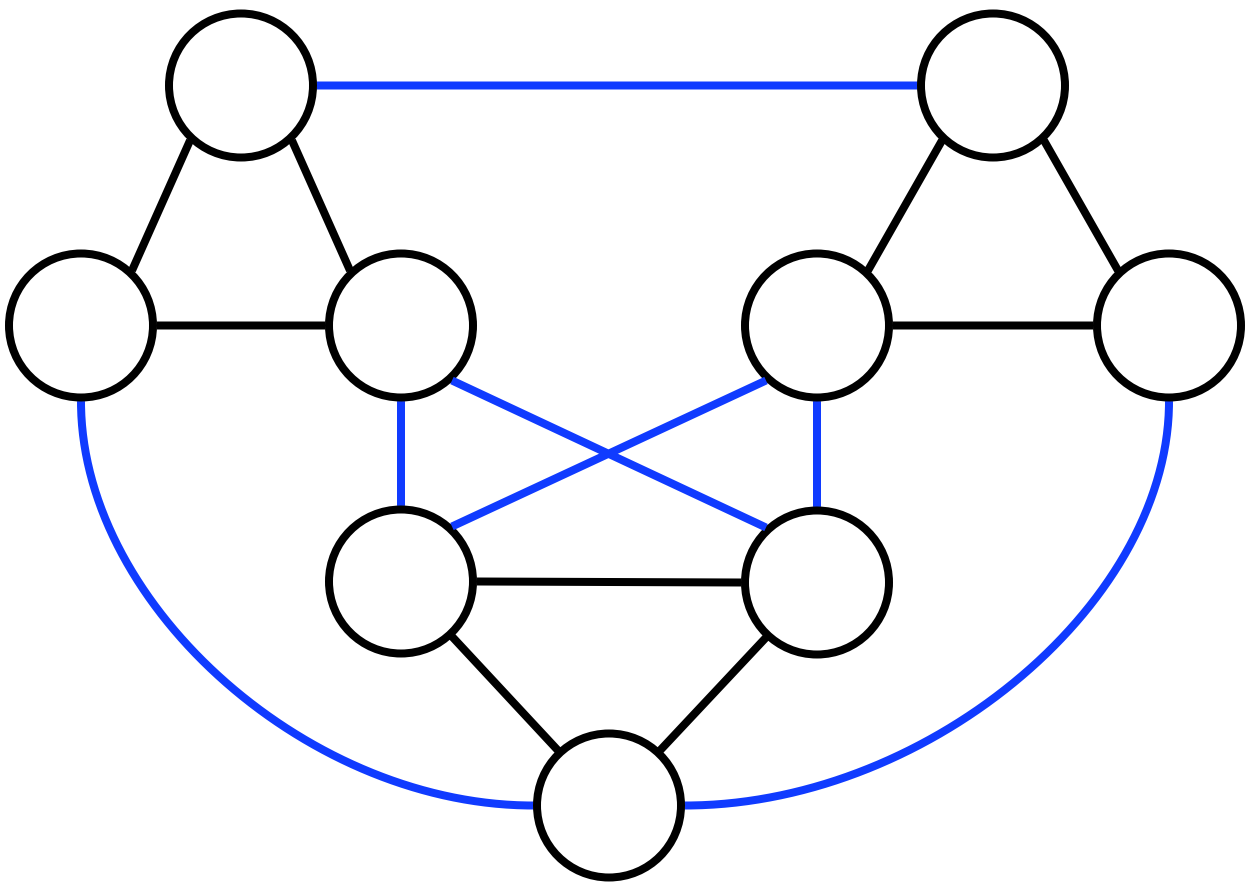


3SAT $\leq_p$ VertexCover



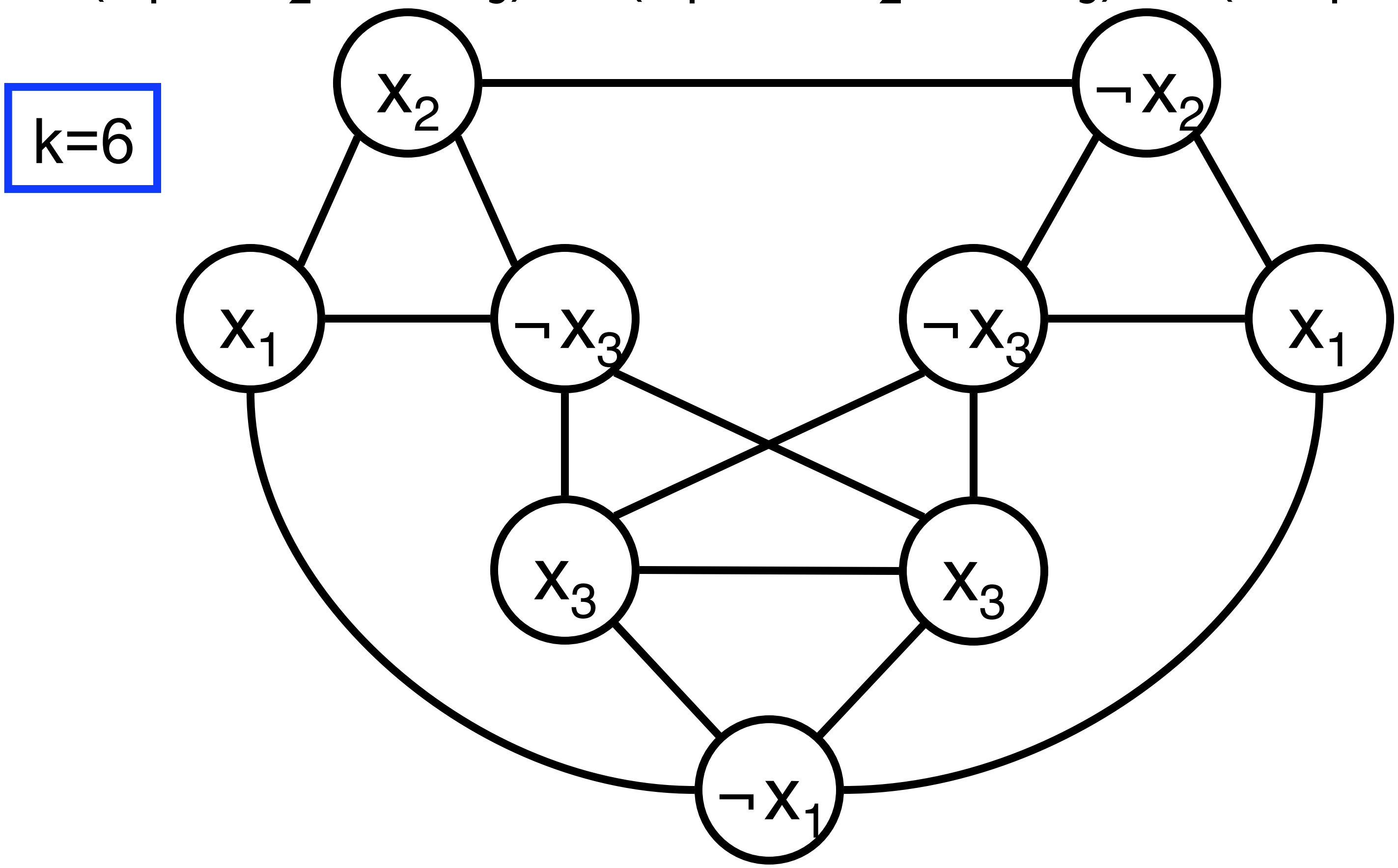
3SAT $\leq_p$ VertexCover

k=6



3SAT $\leq_p$ VertexCover

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



$3SAT \leq_p VertexCover$

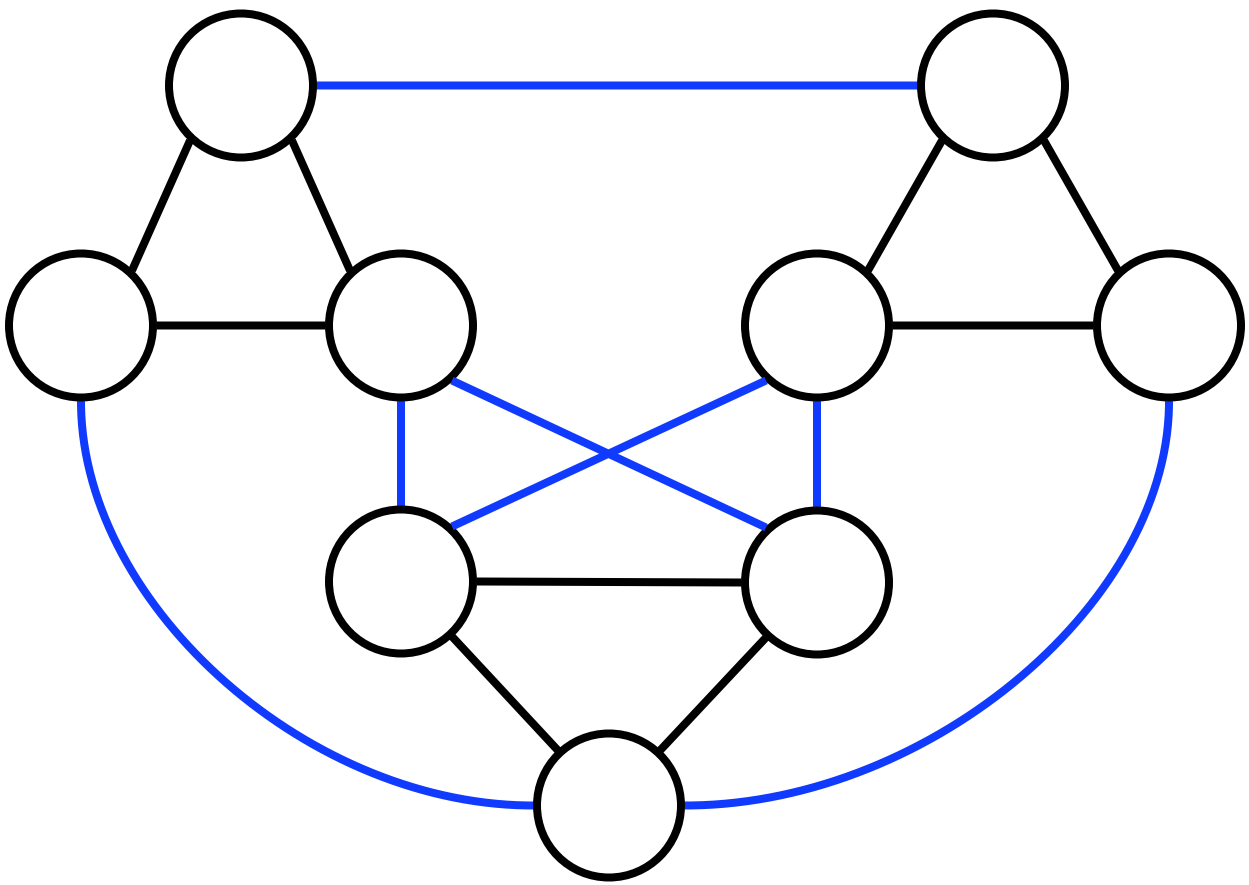
$$f\left(\boxed{\begin{array}{l}\text{3-SAT Instance:}\\ \quad - \text{Variables: } x_1, x_2, \ldots\\ \quad - \text{Literals: } y_{i,j},\ 1 \leq i \leq q,\ 1 \leq j \leq 3\\ \quad - \text{Clauses: } c_i = y_{i1} \vee y_{i2} \vee y_{i3},\ 1 \leq i \leq q\\ \quad - \text{Formula: } c = c_1 \wedge c_2 \wedge \ldots \wedge c_q\end{array}}\right) =$$

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 $\le_p$ VertexCover"

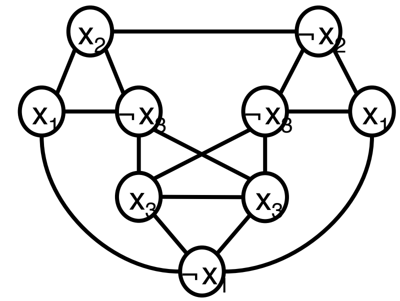
Summary of reduction function f: Given formula, make graph G with one group per clause, one node per literal. Connect each to all nodes in same group, plus complementary literals $(x, \neg x)$. Output graph G plus integer k = 2 * number of clauses. Note: *f does not know whether formula is satisfiable 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:
($\Rightarrow$) 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 $(x, \neg x)$ edge is covered.
($\Leftarrow$) Given a k-vertex cover in G, uncovered labels define a valid (perhaps partial) truth assignment since no $(x, \neg x)$ 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.

Subset-Sum, AKA Knapsack

KNAP= { $(w_1, w_2, \ldots, w_n, C)$ | 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: $(x \lor y) \land (\neg x \lor y) \land (\neg x \lor \neg y)$

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

Conversely, suppose KNAP instance has a solution. Note ≤ 5 one's per column, so no "carries" in sum (recall – weights are decimal); i.e., columns are decoupled. 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 it, at least one of the selected literal weights must be 1 in that clause, hence the assignment satisfies the formula.

Notes on final

Coverage: comprehensive, perhaps slight emphasis post-midterm

Format: similar to midterm:

T/F, multiple choice, problem-solving, explain, …

Closed book, but 1 page of notes

Review in sections tomorrow and class Friday – bring questions!

NP

Examples:

VC: given a set if vertices, is size $\leq$ k & all edges covered?

KNAP: given subset of weights, does sum = C?

Graph 3-Coloring: given a coloring, are all nodes different from their neighbors in color?

SAT: given an assignment, does it satisfy the formula?

Definition:

A problem L is in *NP* iff there is a poly time procedure v(-,-), (the "verifier") and an integer k such that for every $x \in L$ (but no $x \notin L$) $\exists$ h, $|h| \leq |x|^k$ such that v(x,h) = YES

SAT has a (superficially) special role

Cook's Theorem: *Every* problem in NP can be reduced to SAT

Why?

Intuitively, "solutions" are just bit strings,

"There exists a solution" → "there exists an assignment"

Computers are just big, dumb piles of Boolean logic, so "the verifier says YES" → "That assignment satisfies this formula.

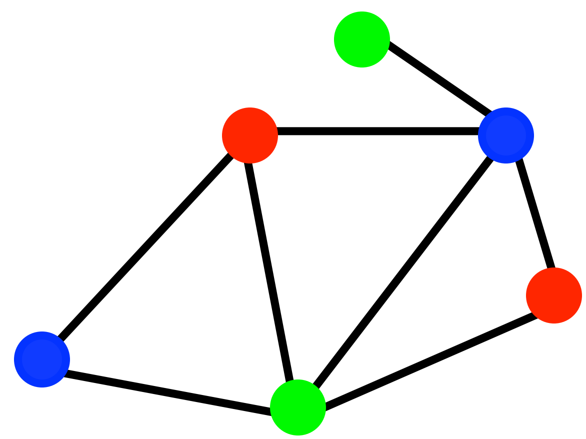
I won't prove Cook's theorem, but will give a few examples.

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 $\leq_p$ SAT

Given G = (V, E) and k

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

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

Hamilton Circuit $\leq_p$ SAT

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

variables x_{ij}, for each i,j in V encode "j follows i in the tour"

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

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.

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:

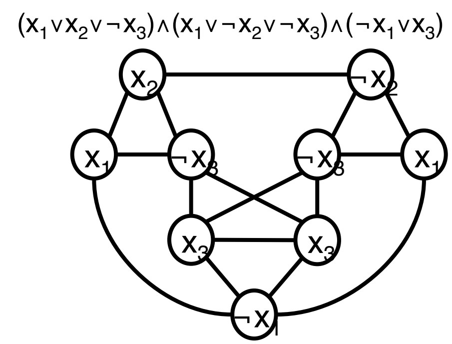
Generic "NP" problems: expo. search–
is there a poly size "solution," verifiable
by computer in poly time

"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.

Reductions

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.

Utility of “3SAT $\leq_p$ KNAP”

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

Given 3-CNF formula w, build Knap instance y = f(w) as above, run the fast Knap alg on y; say “YES, w is satisfiable” iff Knap alg says “YES, a subset sums to C”

	Variables x	Variables y	Clauses (x v y)	Clauses (¬x v y)	Clauses (¬x v ¬y)
w_1 (x)	1	0	1	0	0
w_2 (¬x)	1	0	0	1	1
w_3 (y)		1	1	1	0
w_4 (¬y)		1	0	0	1
w_5 (s_{11})			1	0	0
w_6 (s_{12})			1	0	0
w_7 (s_{21})				1	0
w_8 (s_{22})				1	0
w_9 (s_{31})					1
w_{10} (s_{32})					1
C	1	1	3	3	3

If, on the other hand, no fast alg is possible for 3SAT, then we know none is possible for KNAP either.

"3SAT $\leq_p$ VC/KNAP" Retrospective

Previous slides: 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)

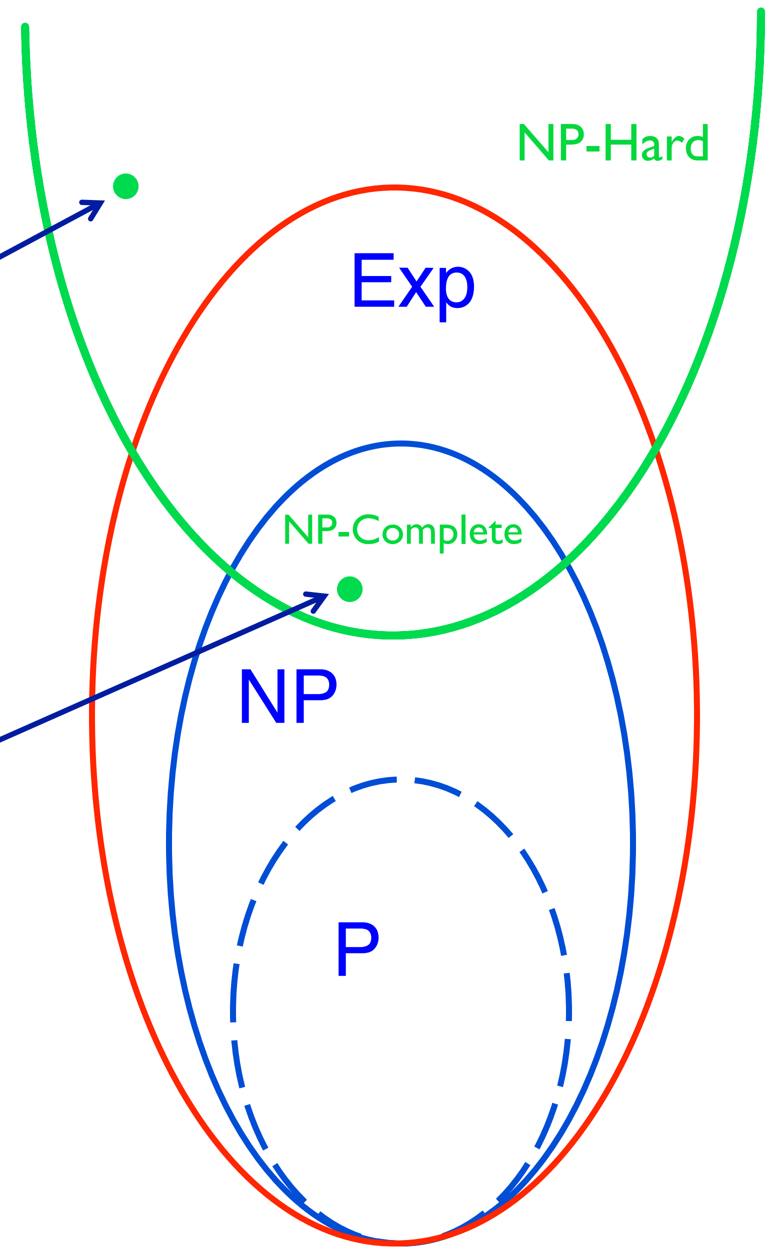
NP-Completeness

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.



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 (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.

Ex: VertexCover is NP-complete

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

3-SAT $\leq_p$ VertexCover

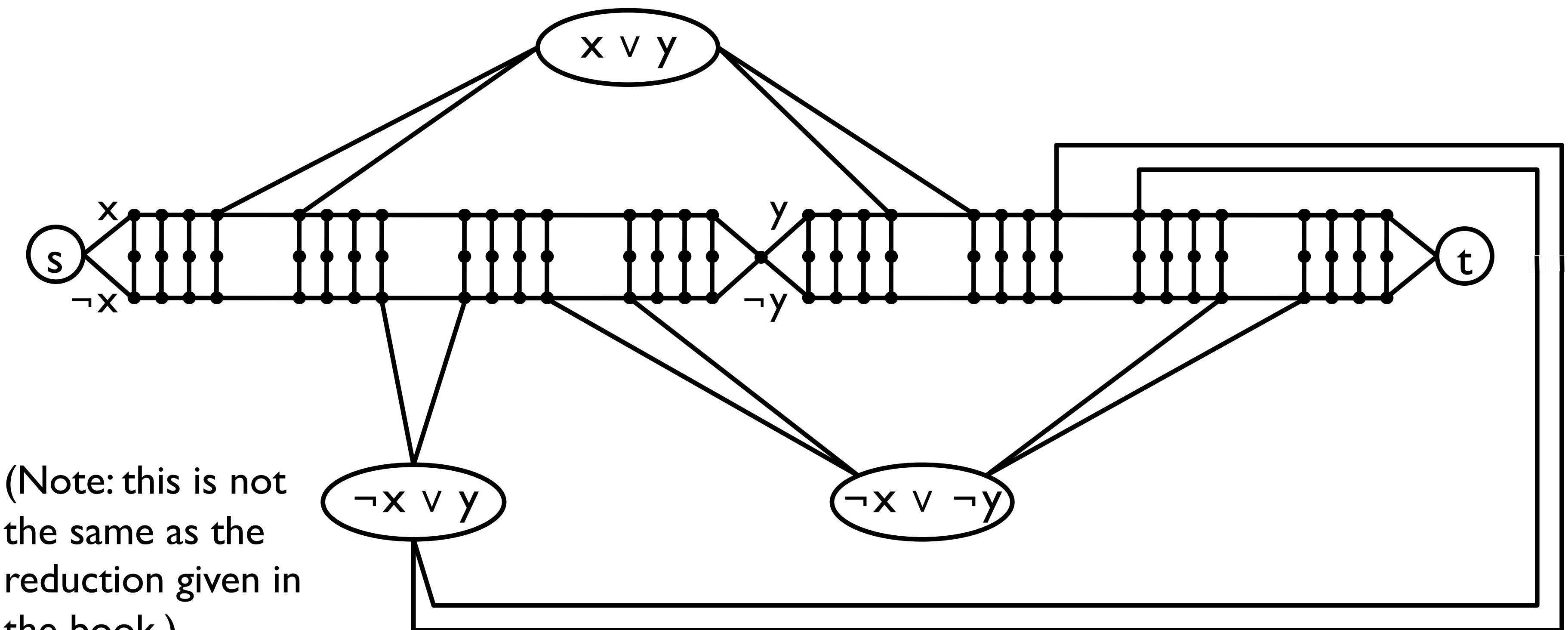
VertexCover is in NP (we showed this earlier)

Therefore VertexCover is also NP-complete

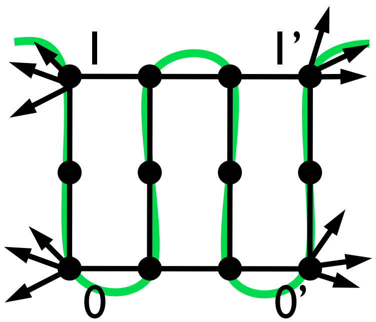
So, poly-time algorithm for VertexCover would give poly-time algs for everything in NP

3-SAT $\leq_p$ UndirectedHamPath

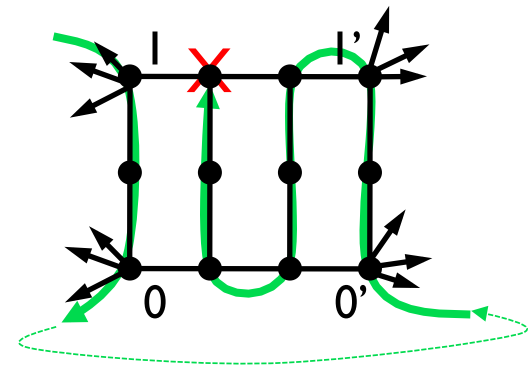
Example: $(x \vee y) \wedge (\neg x \vee y) \wedge (\neg x \vee \neg y)$



(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'

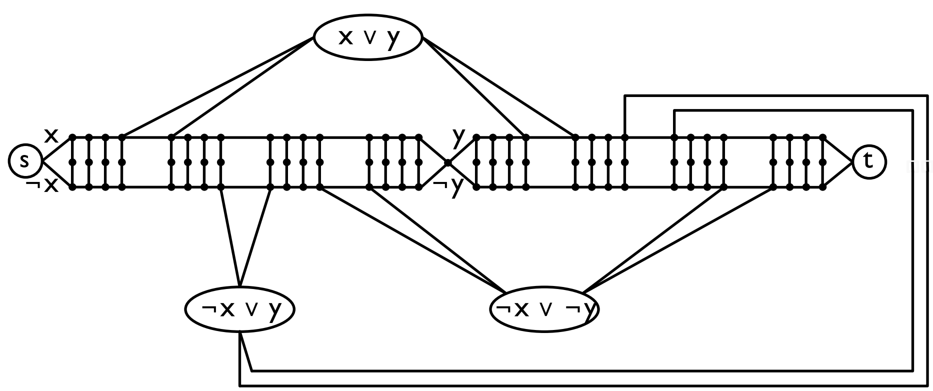
Pf: Note *: at 1st visit to any column, must next go to *middle* node in column, else it will subsequently become an untraversable "dead end."
WLOG, suppose enter at 1. By *, must then go down to 0. 2 cases:

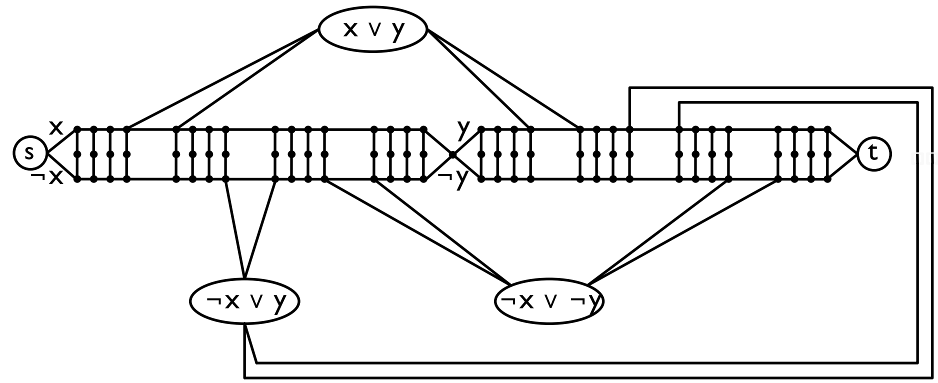
Case a: (top left) If next move is to right, then * forces path up, left is blocked, so right again, * forces down, etc; out at 1'.

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.

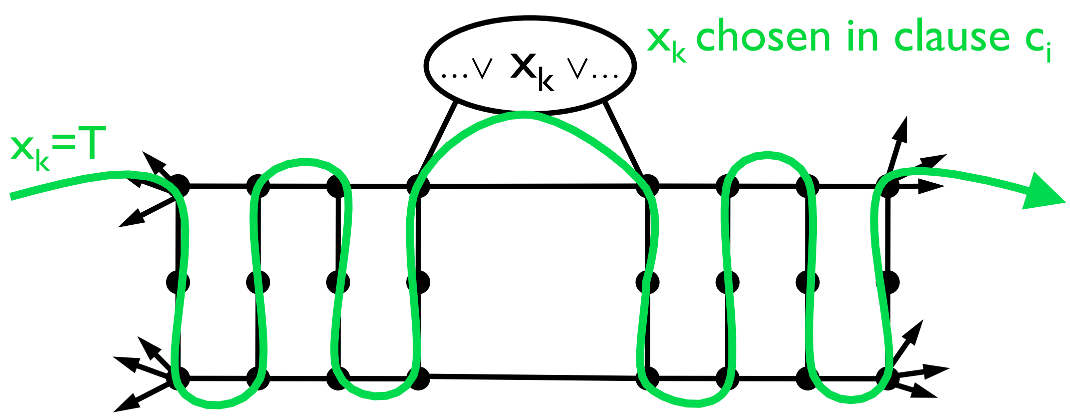




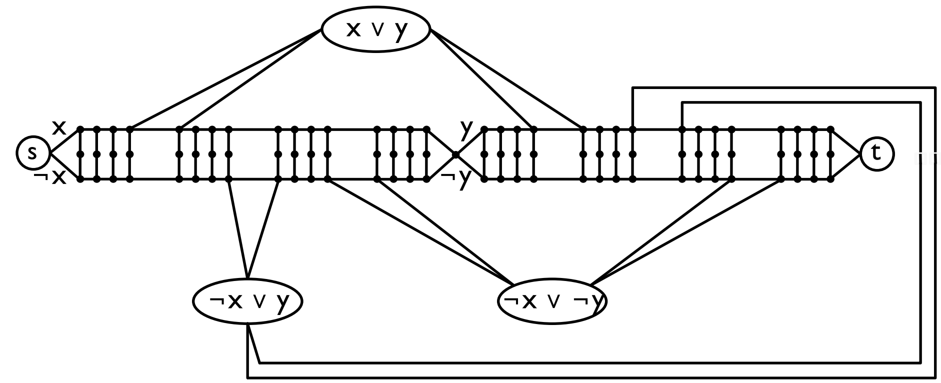
Correctness, I

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.

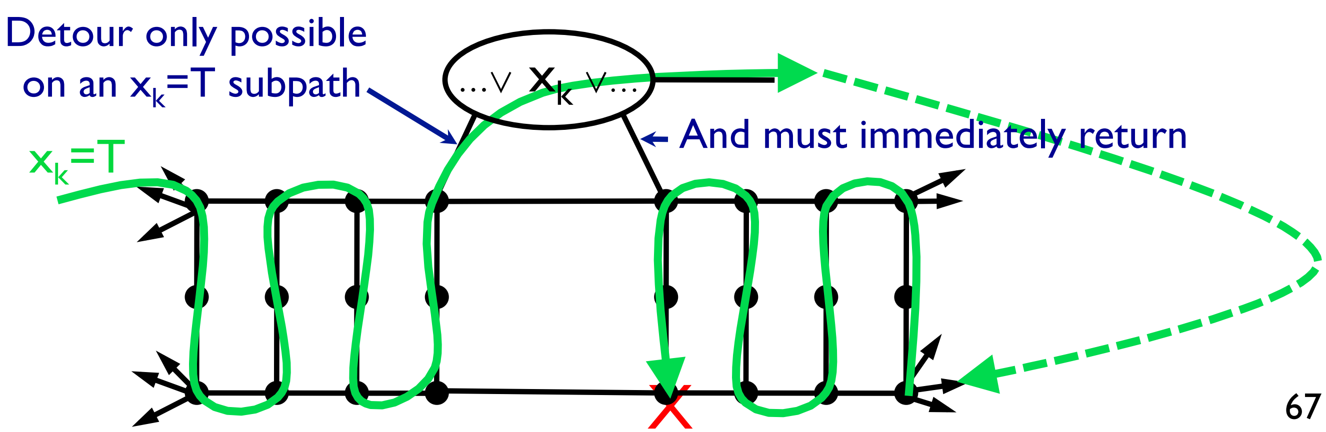
If f is satisfiable, pick a satisfying assignment, and pick a true literal in each clause. Take the corresponding "main chain" path; add a detour to/from c_i for the true literal chosen from clause i. Result is a Hamilton path.

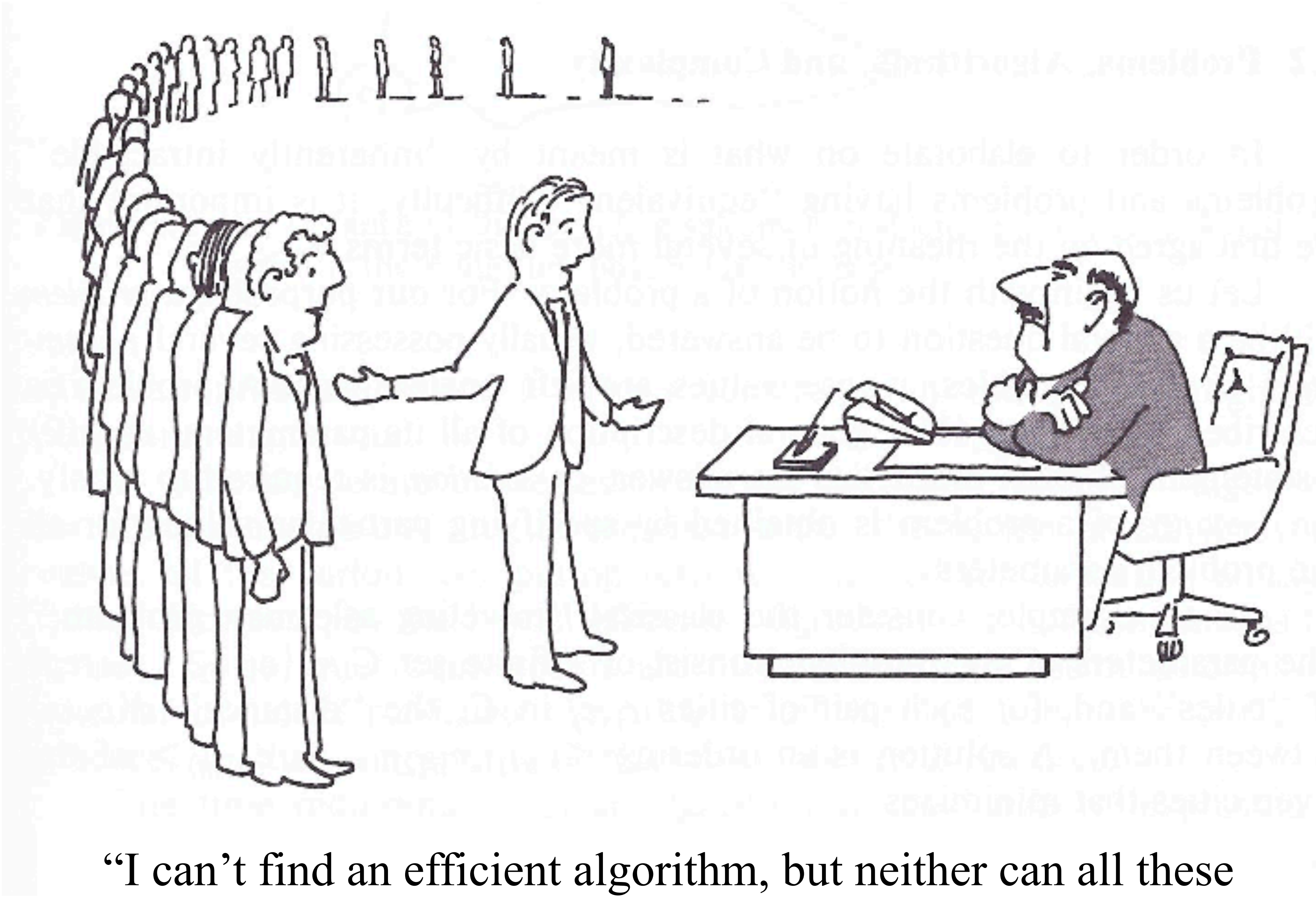


Correctness, II



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.





“I can’t find an efficient algorithm, but neither can all these famous people.” [Garey & Johnson, 1979]

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?

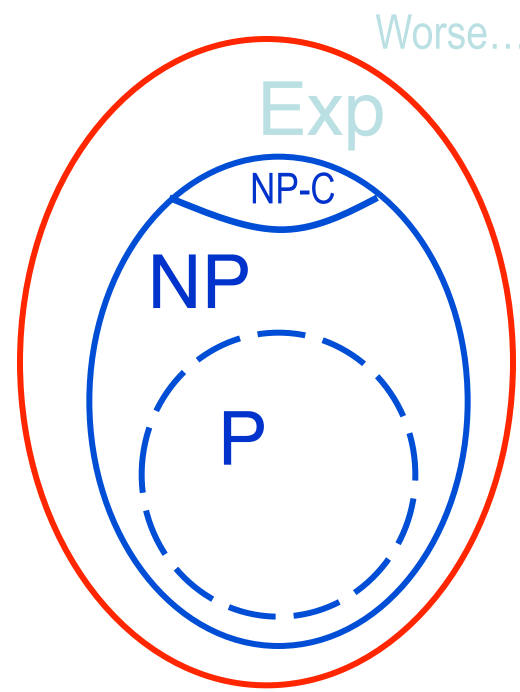
- E.g. Euclidean TSP within 2 * Opt in poly time

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

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.

P

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

- Details are more the fodder of algorithms courses, but 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…" – all 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

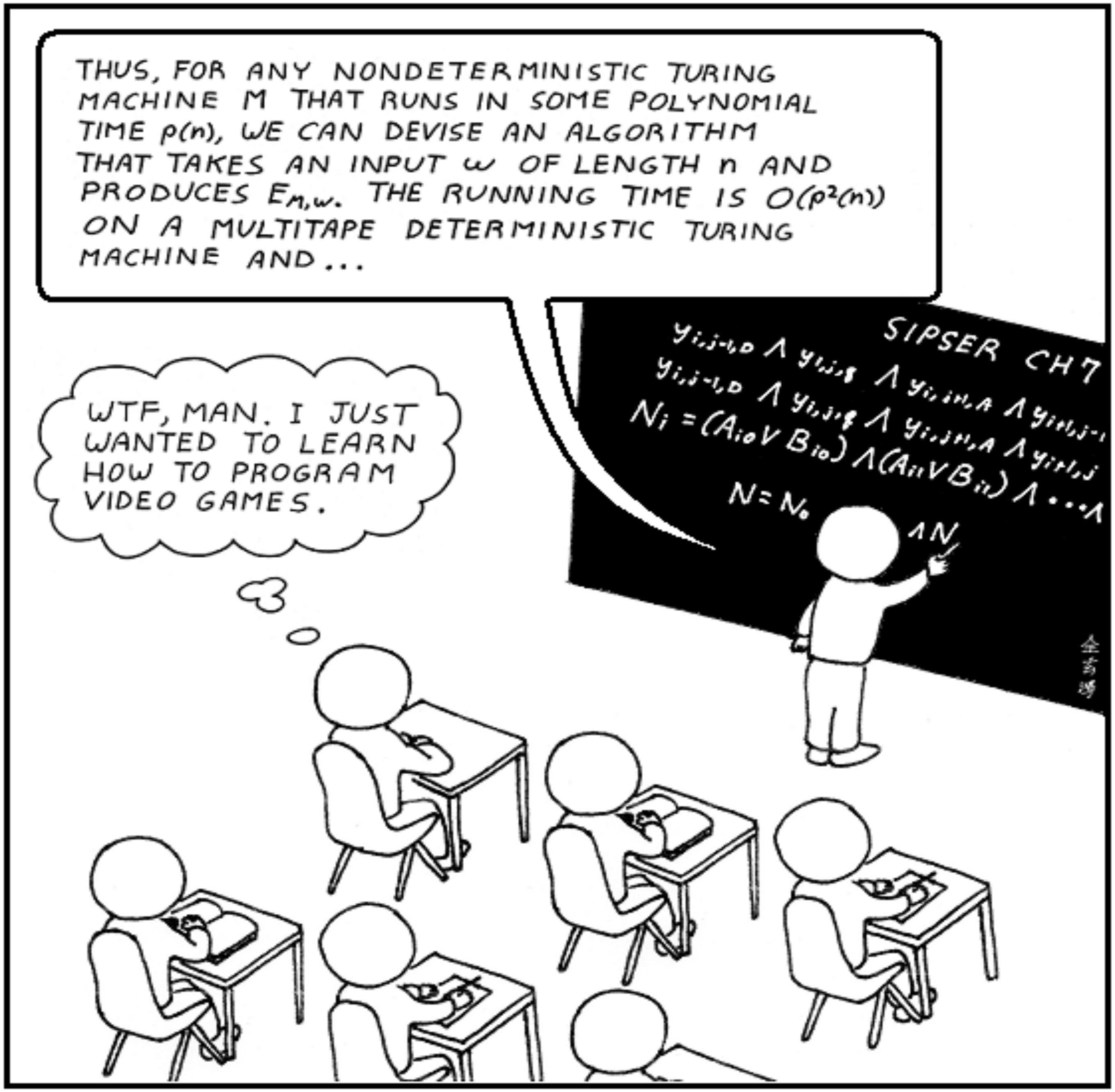
"the hardest problems in NP"

"All alike under the skin"

Most known natural problems in NP are complete

#1: 3CNF-SAT

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


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(p²(n)) ON A MULTITAPE DETERMINISTIC TURING MACHINE AND...
WTF, MAN. I JUST WANTED TO LEARN HOW TO PROGRAM VIDEO GAMES.
SIPSER CH7