CSE 417 Introduction to 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…

Some History

1930/40's

Gödel, Church, Turing, ...: What is (is not) computable

1960/70's and since

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 vs exponential time – qualitatively different

Polynomial Time

The class P

(defined later)

Definition: P = the 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., flows & matching – i.e.: *most of this quarter*

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

Why “Polynomial”?

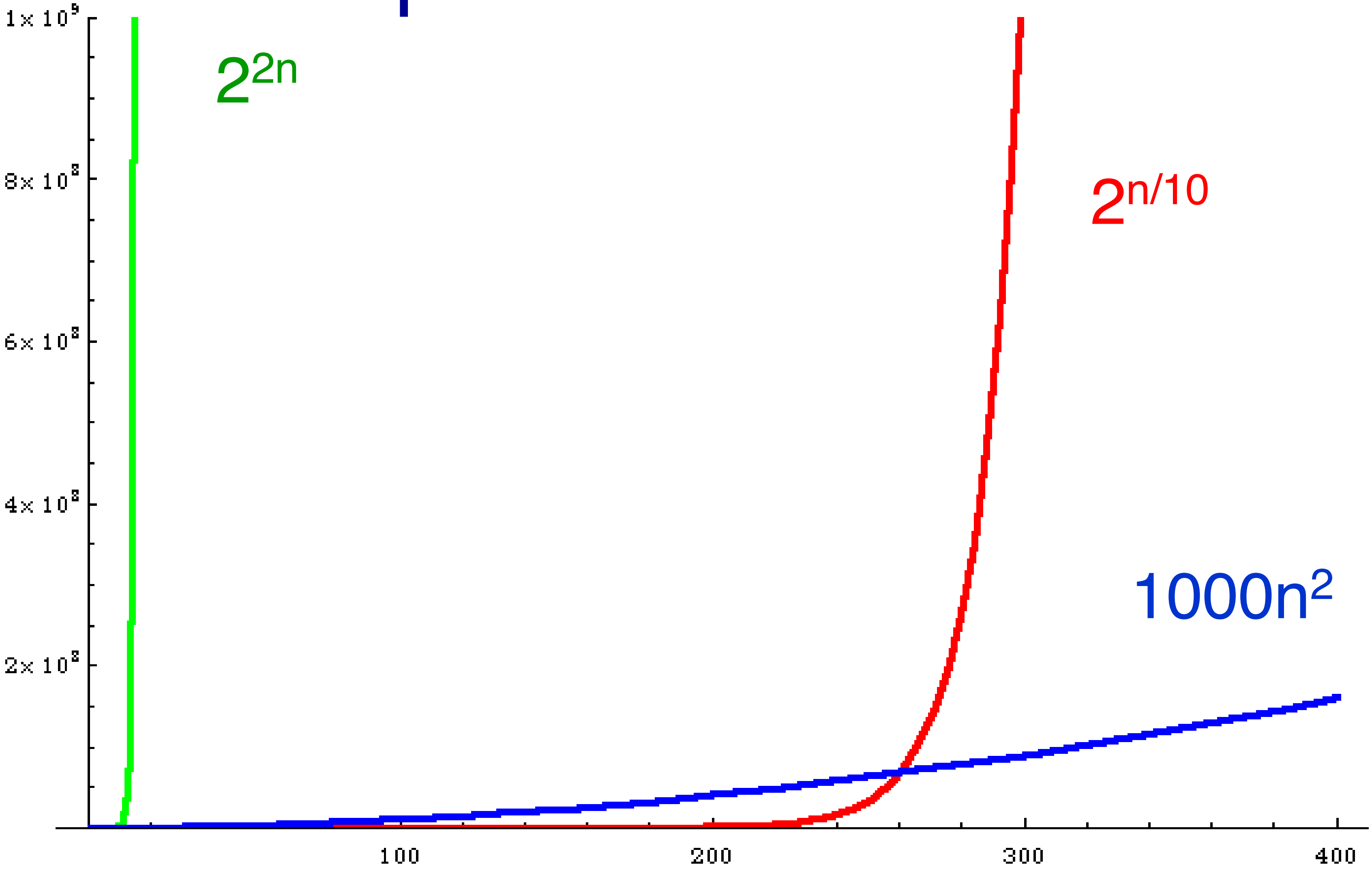
- n^{2000} is *not* a nice time bound
- differences among n, 2n and n^2 are *not* negligible.

But, simple theoretical tools don’t easily capture such differences, while exponential vs polynomial is a qualitative difference potentially more amenable to theoretical analysis.

- “Problem is in P”: starting point for more detailed analysis
- “Problem is not in P”: maybe you need to shift to a more tractable variant / lower your expectations

Reminder

Polynomial vs Exponential Growth



Reminder

Another view of Poly vs Exp

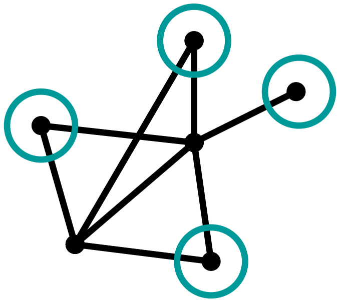
Next year's computer will be 2x faster. If I can solve problem of size n_0 today, how large a problem can I solve in the same time next year?

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

Two Problems

How hard are they? We don’t fully know…

The Independent Set Problem



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

Question: is there U ⊆ V with |U| ≥ k s.t. no pair of vertices in U is joined by an edge?

What's it good for?

E.g., if nodes = web pages, and edges join "similar" pages, then pages forming an independent set are likely to represent distinctly different topics

E.g., if nodes = courses, and edge = a student is co-enrolled, then an independent set is a set of courses whose finals could be scheduled simultaneously

How hard is it? Don't fully know. Exponential time is easily possible (try all 2^n subsets). But no poly time solution is known

The Boolean Satisfiability Problem

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

The Boolean Satisfiability Problem

CNF formula example

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

The formula is *satisfiable* if there's some assignment of 0's and 1's to the variables that makes it true

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?

AKA "SAT"; 3 literals per clause: "3SAT"

Satisfiable?

$$
\begin{array}{l}
(x \lor y \lor z) \land (\neg x \lor y \lor \neg z) \land \\
(x \lor \neg y \lor z) \land (\neg x \lor \neg y \lor z) \land \\
(\neg x \lor \neg y \lor \neg z) \land (x \lor y \lor z) \land \\
(x \lor \neg y \lor z) \land (x \lor y \lor \neg z)
\end{array}
$$

$$
\begin{array}{l}
(x \lor y \lor z) \land (\neg x \lor y \lor \neg z) \land \\
(x \lor \neg y \lor \neg z) \land (\neg x \lor \neg y \lor z) \land \\
(\neg x \lor \neg y \lor \neg z) \land (\neg x \lor y \lor z) \land \\
(x \lor \neg y \lor z) \land (x \lor y \lor \neg z)
\end{array}
$$

Satisfiability

What's it good for?

- Theorem provers
- Circuit validation
- Analysis of program logic
- Etc.

How hard is it?

- Don't know fully
- Exponential time is easily possible (try all 2^n assignments)
- But no poly time solution is known

Reduction, I

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 & Time

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

If setting up call, etc., is fast, then a *fast* algorithm for B implies (nearly as) *fast* an algorithm for A

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

poly-time, for our uses

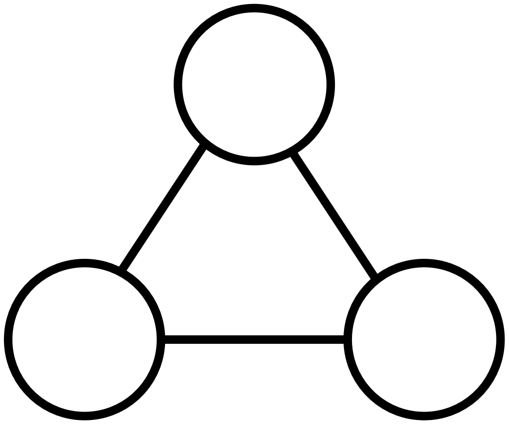
"complexity of A" ≤ *"complexity of B" + "complexity of reduction"*

SAT and Independent Set

They are superficially different problems, but are intimately related at a deep level

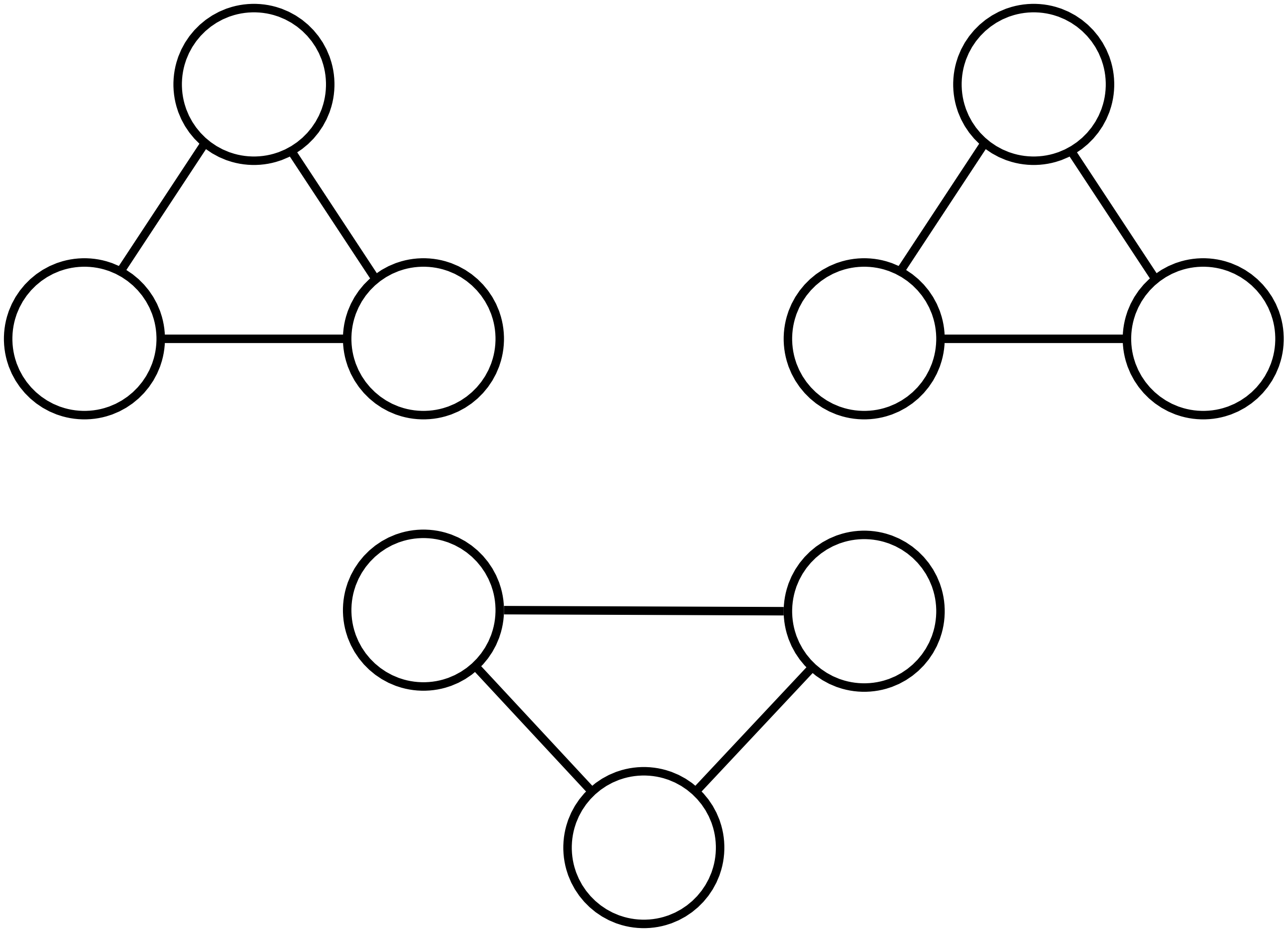
3SAT $\leq_p$ IndpSet

what indp sets?
how large?
how many?

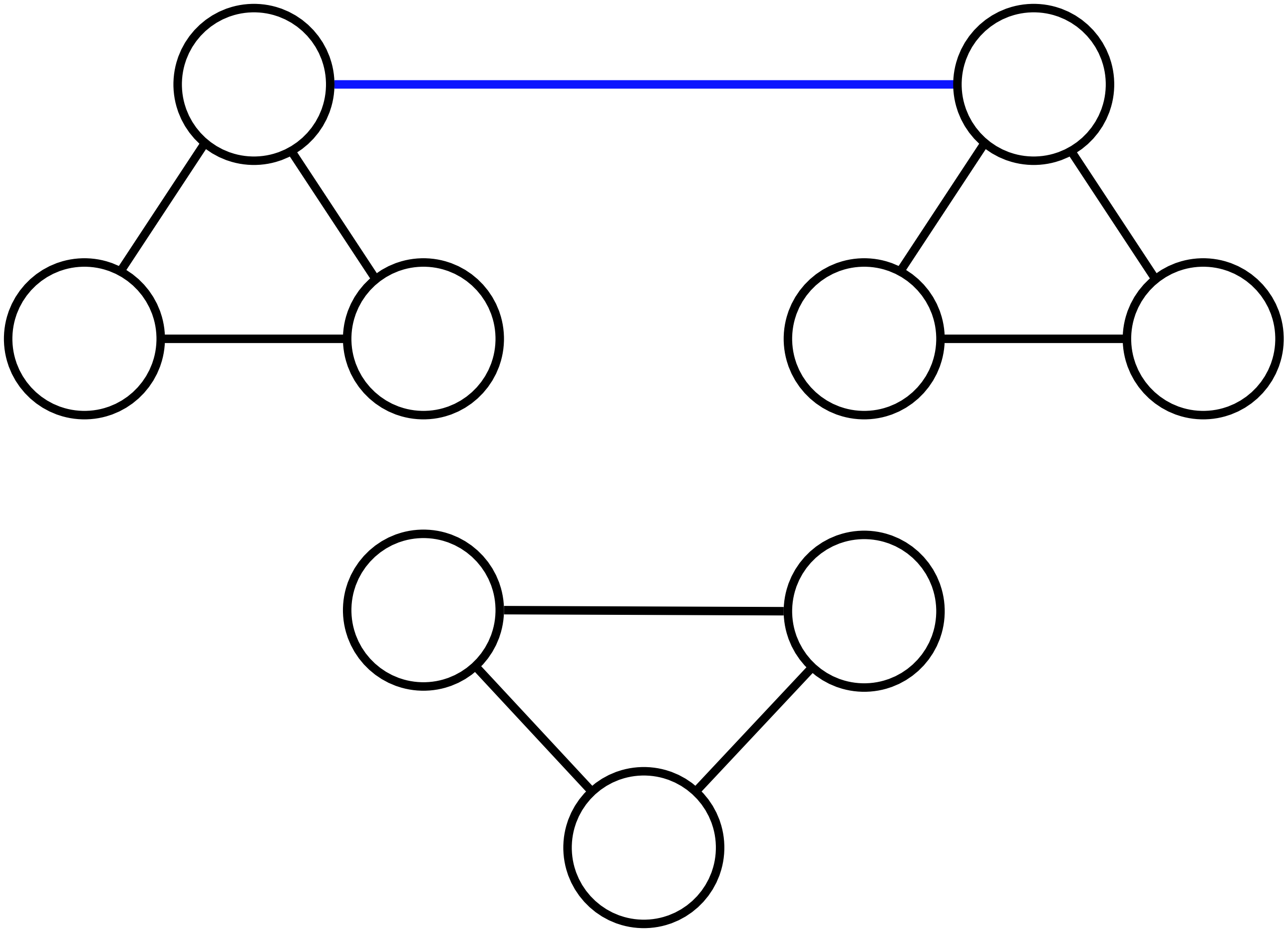


3SAT $\leq_p$ IndpSet

what indp sets?
how large?
how many?



3SAT $\leq_p$ IndpSet

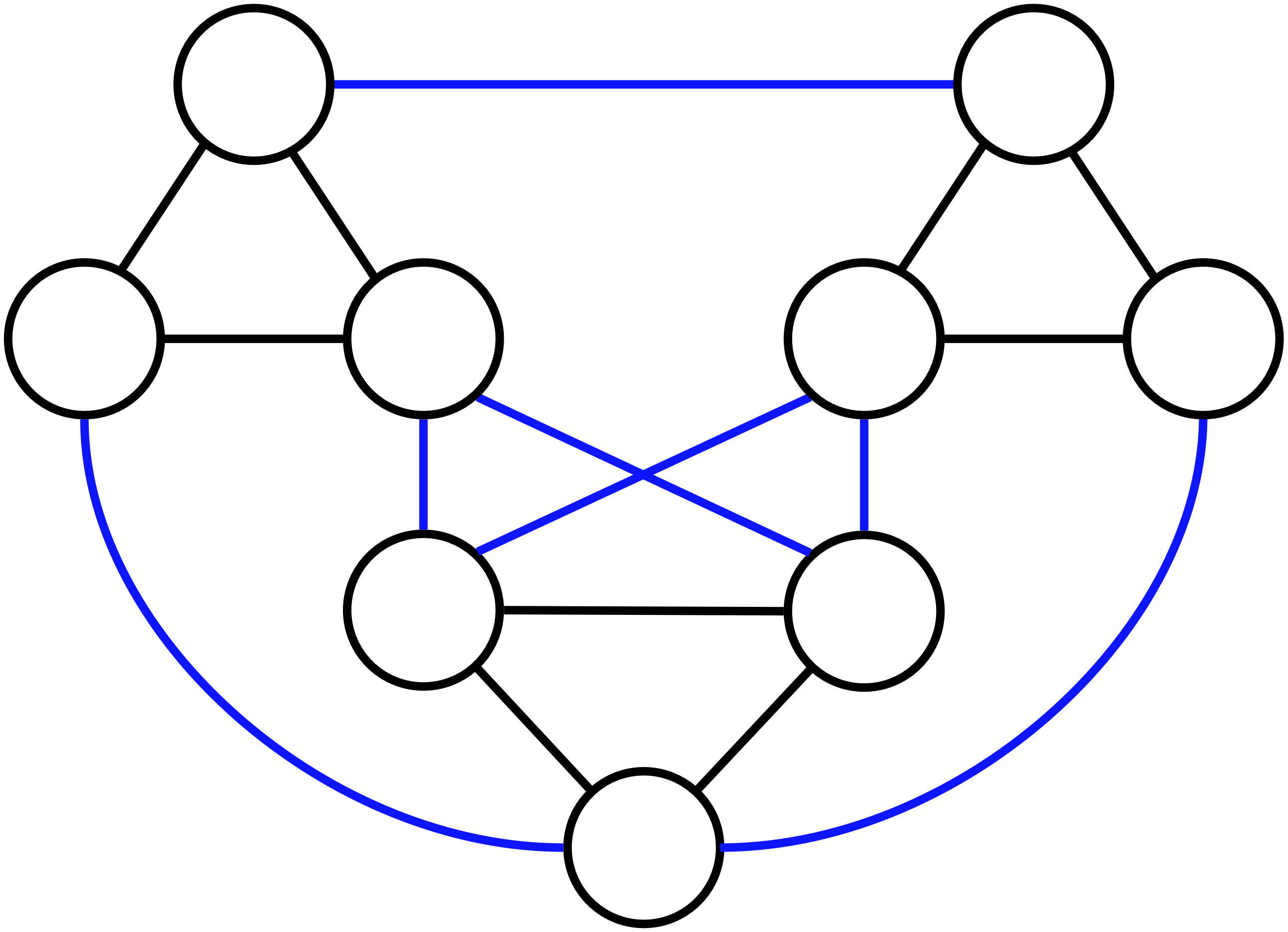


what indp sets?
how large?
how many?

3SAT $\leq_p$ IndpSet

what indp sets?
how large?
how many?

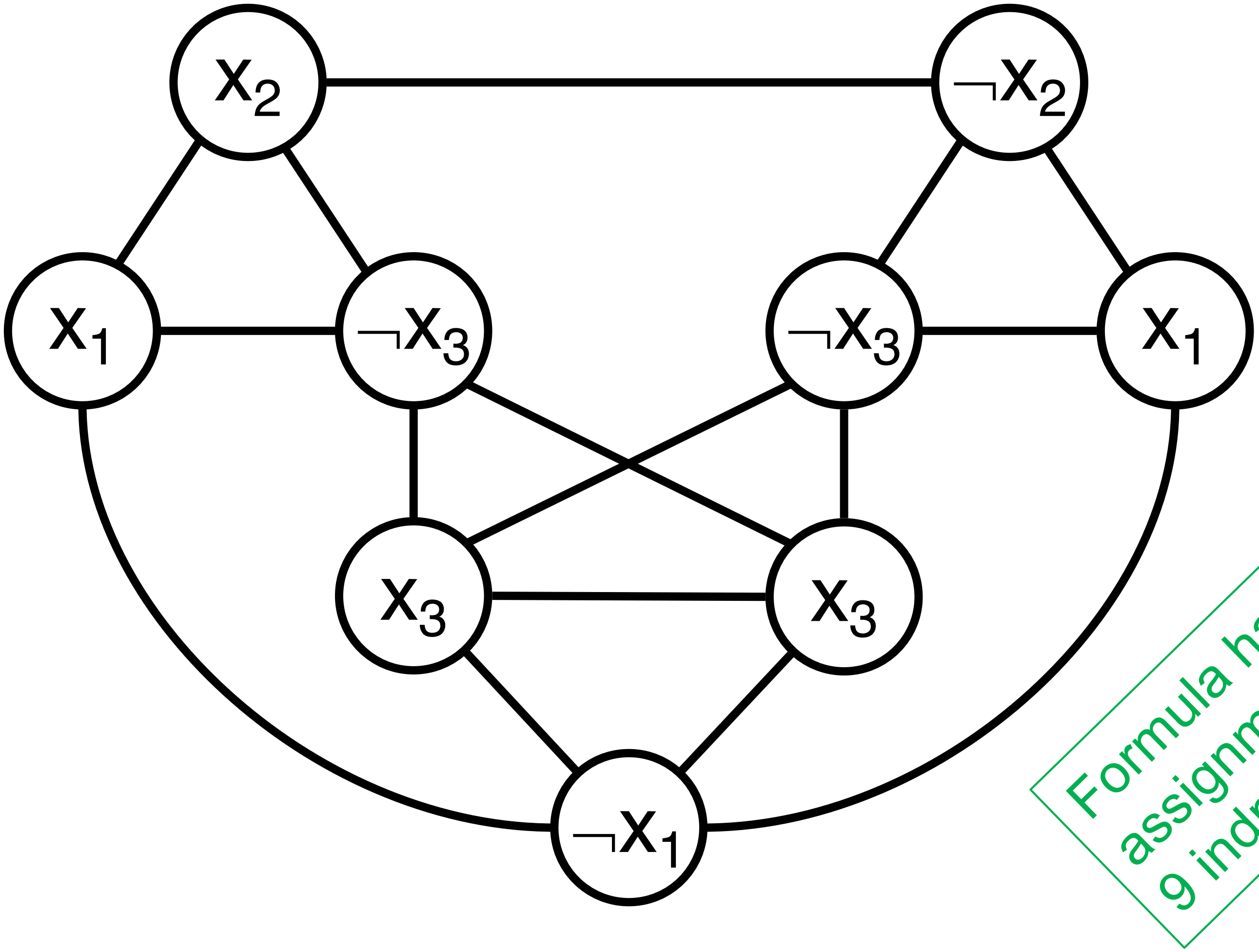
k=3



3SAT $\leq_p$ IndpSet

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

k=3



Formula has 4 satisfying assignments; Graph has 9 indp sets of size k=3

3SAT $\leq_p$ IndpSet

f (

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: $c = c_1 \wedge c_2 \wedge \ldots \wedge c_q$

) =

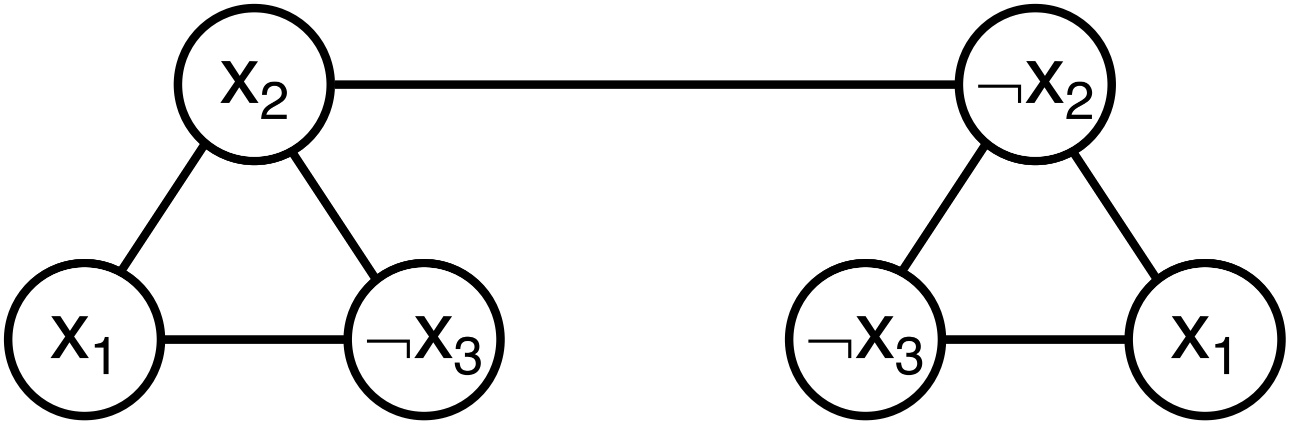
IndpSet Instance:

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

3SAT $\leq_p$ IndpSet

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

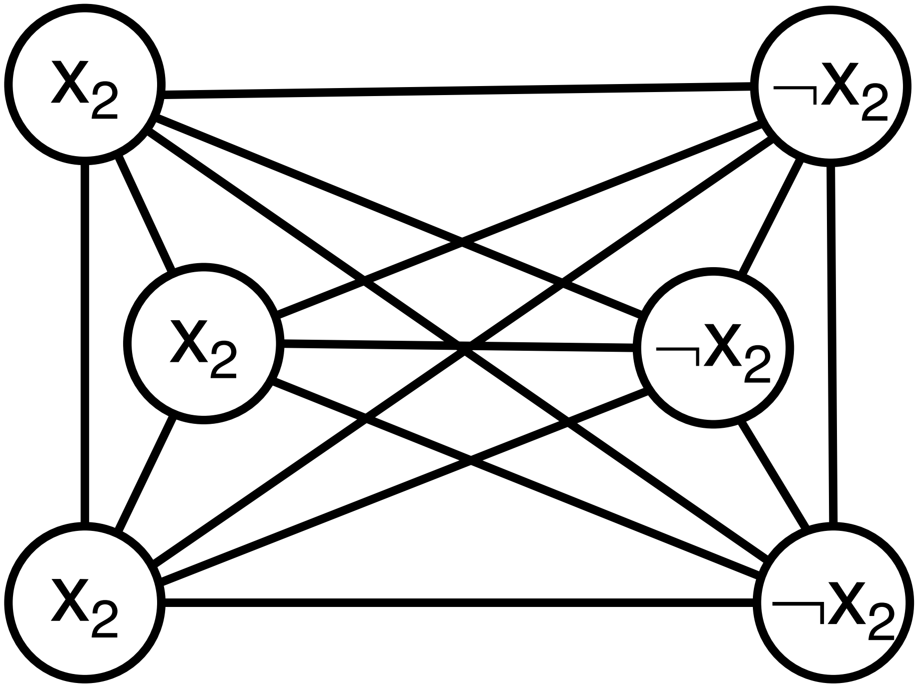
k=2



3SAT $\leq_p$ IndpSet

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

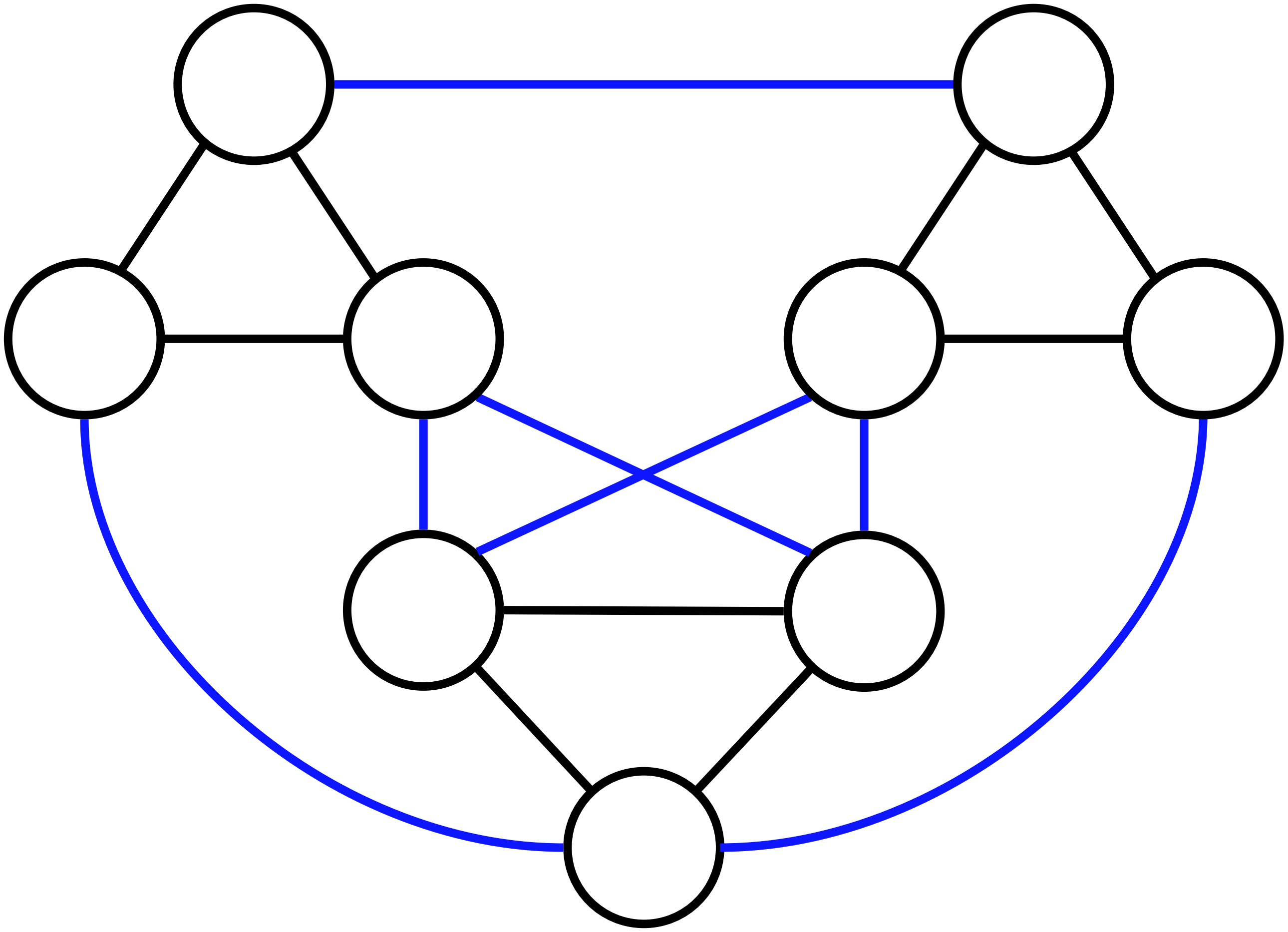
k=2



Satisfiable?   k-Independent set?

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; connect all complementary literal pairs (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:

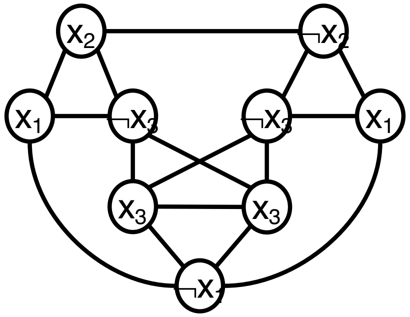
- Show f poly time computable: A key point is that graph size is polynomial in formula size; mapping basically straightforward.
- Show c in 3-SAT iff f(c)=(G,k) in IndpSet:

(⇒) 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"

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



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.

“3SAT $\leq_p$ IndpSet” 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 mapping reduction”), then properties like the above always hold.

Reduction, II

Polynomial time “mapping” reduction

Decision Problems & Notation

Most of NP theory is framed for *decision problems*, i.e., problems for which the desired answer is YES/NO, e.g.

- "Is there a satisfying assignment for formula *f*?" or
- "Does graph *G* have an independent set of size *k*?"

(As opposed to, say, "find such an assignment/set." Why? Because it's simpler.)

Notation: for a decision problem A, we view A as the set of YES instances: i.e., "$x \in A$" means "*x* is a YES instance of *A*". E.g., examples above become:

- "$f \in$ SAT ?" and
- "$(G, k) \in$ IndpSet ?"

Polynomial-Time Reductions

Definition: Let *A* and *B* be two decision problems.

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*

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

The notation "$A \leq_P B$" is meant to suggest "A is easier than B", or more precisely, "A is not more than polynomially harder than 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?

polynomial

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

Theorem:

(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)

Another Example Reduction

SAT to Subset Sum (Knapsack)

Defn is slightly different from versions in DP slides & KT ch 6

Subset-Sum, AKA Knapsack

KNAP = $\{ (\underbrace{w_1, w_2, \ldots, w_n, C}_{\text{Positive integers}}) \mid$ 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. See examples below.

3-SAT $\leq_p$ KNAP

Formula: (x)

		Variables	Clauses	
		x	(x)	
Literals	w_1 (x)	1	1	←(eleven)
	w_2 ($\neg$x)	1	0	←(ten)
Slack	w_7 (s_{11})		1	←(one)
	w_8 (s_{12})		1	←(one)
C		1	3	←(thirteen)

What/How Many Satisfying Assignments?

What/How Many KNAP solutions?

3-SAT $\leq_p$ KNAP

Formula: $(x) \wedge (\neg x)$

		Variables	Clauses	
		x	(x)	(¬x)
Literals	w_1 (x)	1	1	0
	w_2 (¬x)	1	0	1
Slack	w_7 (s_{11})		1	0
	w_8 (s_{12})		1	0
	w_9 (s_{21})			1
	w_{10} (s_{22})			1
C		1	3	3

What/How Many Satisfying Assignments?

What/How Many KNAP solutions?

3-SAT $\leq_P$ KNAP

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

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

What/How Many Satisfying Assignments?

What/How Many KNAP solutions?

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

What/How Many Satisfying Assignments/KNAP solutions?

3-SAT $\le_p$ KNAP

f (

3-SAT Instance:

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

) =

KNAP Instance:

- 2(p+q) wi's, each with (p+q) decimal digits, mostly 0
- For the 2p "literal" weights, a single 1 in H.O. p digits marks which variable; 1's in L.O. q digits mark each clause containing that literal.
- Two "slacks" per clause; single 1 marks the clause.
- Knapsack Capacity C = 11..133..3 (p 1's, q 3's)

Correctness

Poly time for reduction is routine; details omitted. 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 (1 ≤ k ≤ 3), also select (3 - k) of the "slack" weights for that clause. (note 0 ≤ 3-k ≤ 2, so this is possible.) Total = C.

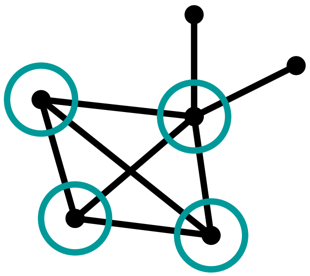
Conversely, suppose KNAP instance has a solution. Columns are decoupled since ≤ 5 one's per column, so no "carries" in sum (recall – weights are decimal). Since H.O. p digits of C are 1, exactly one of each pair of literal weights included in the subset, so it defines a valid assignment. Since L.O. q 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.

Decision vs Search Problems

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.



E.g., if nodes are web pages, and edges join “similar” pages, then pages forming a clique are likely to be about the same topic

Decision Problems

Computational complexity commonly 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

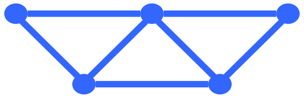
Less important, but if you have a good decider, you can often use it to get a good finder. (Ex.: does G still have a k-clique after I remove this vertex?)

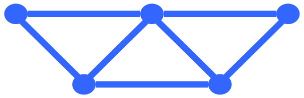
Some Convenient Technicalities

“Problem” – the general case

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

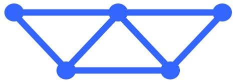
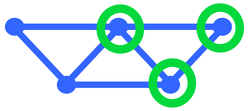
“Problem Instance” – one specific case

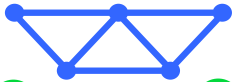
Ex: Does  contain a 4-clique? (no)

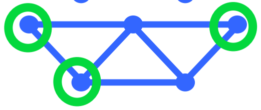
Ex: Does  contain a 3-clique? (yes)

Some Convenient Technicalities

Three kinds of problem:

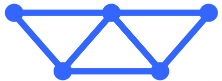
Search: *Find* a k-clique in G (3, ) → 

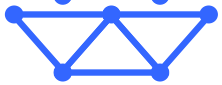
Decision: *Is there* a k-clique in G (3, ) → yes

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) ∉ CLIQUE

E.g., ( , 3) ∈ CLIQUE

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

Beyond P

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 $x=y=1$, $z=0$.)

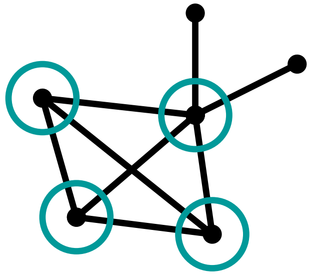
SAT = the set of satisfiable CNF formulas

3SAT = … having at most 3 literals per clause

More Problems

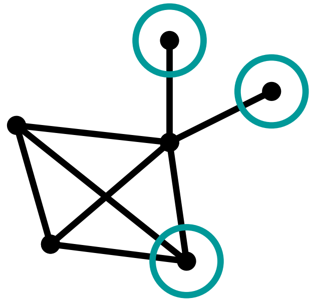
Clique:

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.



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.



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 ⟨G,k⟩, 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 ≤ k.

More Problems

Short Path:

4-tuples ⟨G, s, t, k⟩, 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 ≤ k

Long Path:

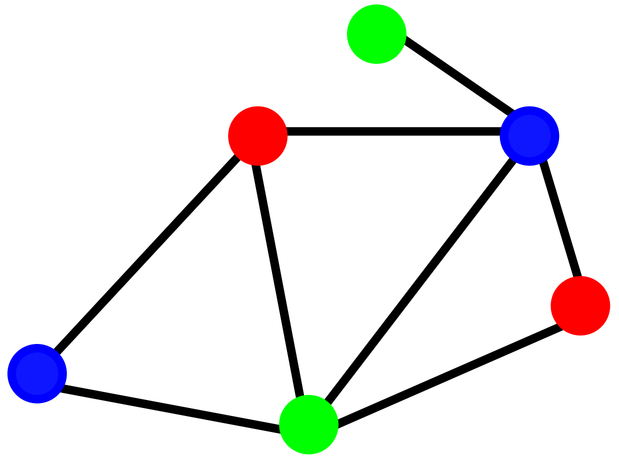
4-tuples ⟨G, s, t, k⟩, where G=(V,E) is a digraph with vertices s, t, and an integer k, for which there is an acyclic path from s to t of length ≥ k

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., SAT, IndpSet, CLIQUE, KNAP, TSP, …

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

Lack of imagination or intrinsic barrier?

And what, if anything, do they have in common?

NP

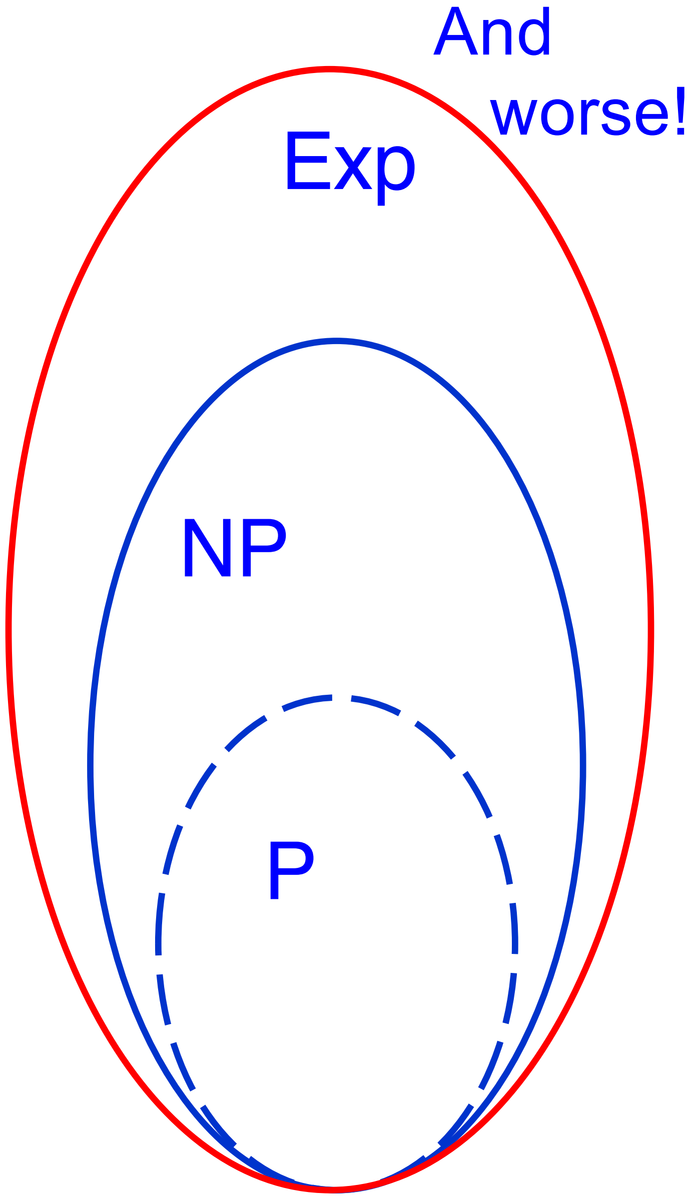
Roadmap

Not every problem is easy (in P)

Exponential time is bad

Worse things happen, too

There is a very commonly-seen class of problems, called *NP*, that *appear* to require exponential time (but unproven)



Review: Some Problems

Clique

Independent Set

Euler Tour

Hamilton Tour

TSP

3-Coloring

Partition

Satisfiability

Short Paths

Long Paths

All superficially different, but –

All of the form: Given input X, is there a Y with property Z? Furthermore, if I had a purported Y, I could quickly test whether it had 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: The Idea

NP consists of all decision problems where

Can verify 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

Defining NP: formally

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

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

and

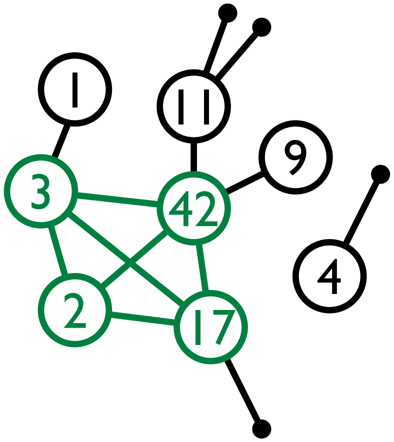
for every $x \notin L$ there is *no* hint h with $|h| \leq |x|^q$ such that $v(x,h)$ = YES

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

Note 1: a problem is "in NP" if it can be *posed* as an exponential search problem, even if there may be other ways to *solve* it.

Note 2: the defn is not quickly actionable without a way to find *h.*

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

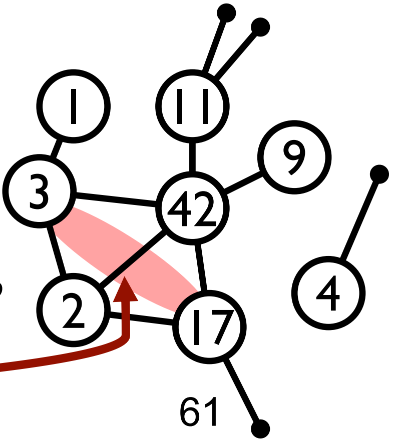
if I knew where it was, I could describe it succinctly, e.g. "look at the *k* vertices 2, 3, 17, 42, ...",

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 *no k*-clique, I wouldn't be fooled by a statement like "look at vertices 2, 3, 17, 42, ..."

("Ha! No edge from 3 to 17!")



More Formally: CLIQUE is in NP

Time $O(n^2)$

```
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”
```

Say, edge list & decimal number

Say, length $|V|$ 0-1 vector w/ k 1’s

$\forall x,y \in U$, is $(x,y) \in E$ …

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 ~ $(|x|+|h|)^2$

IndpSet 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 an Indp Set in G,
  then output “YES”
  else output “I’m unconvinced”
```

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

Is it correct?

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

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 Indp Set of size k (the interesting case)

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

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(C, A) = syntax(C, A) && satisfies(C, A)

- Syntax: True iff C is a well-formed CNF formula & A is a truth-assignment to its variables
- Satisfies: plug A into C; check that it evaluates to True

Correctness:

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

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

Short Path

“Is there a short path (< *k*) from *s* to *t* 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 *s* to node 2, then node 42, then ... ”

I’d know one if I saw one: “yes, I see there’s an edge from *s* 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 *s* to node 2, then node 42, then ... ”

Long Path

"Is there a long (acyclic) path (> *k*) from *s* to *t* 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 *s* to node 2, then node 42, then ... "

I'd know one if I saw one: "yes, I see there's an edge from *s* to 2 and from 2 to 42..., no dups, & total length is > *k*"

And if there isn't a long path, I wouldn't be fooled by, e.g., "go from *s* 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, the verifier and its time bound all shift, but same bottom line.

2. In NP doesn't prove its hard

   "Short Path" or "Small Spanning Tree" or "Large Flow" 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*…" or "*the smallest/largest*…"

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"

MAXCLIQUE: "the largest clique has size k"

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 *s* to *t* long (>*k*)?"

Any path might work

Yes! → There are lots of t'

I only need or

If I knew or I

succinct' to node

2, the

I'd one: "yes, I

see a to 2 and from

2 to 42... al length > k"

And if there sn'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-completeness

NP-Completeness

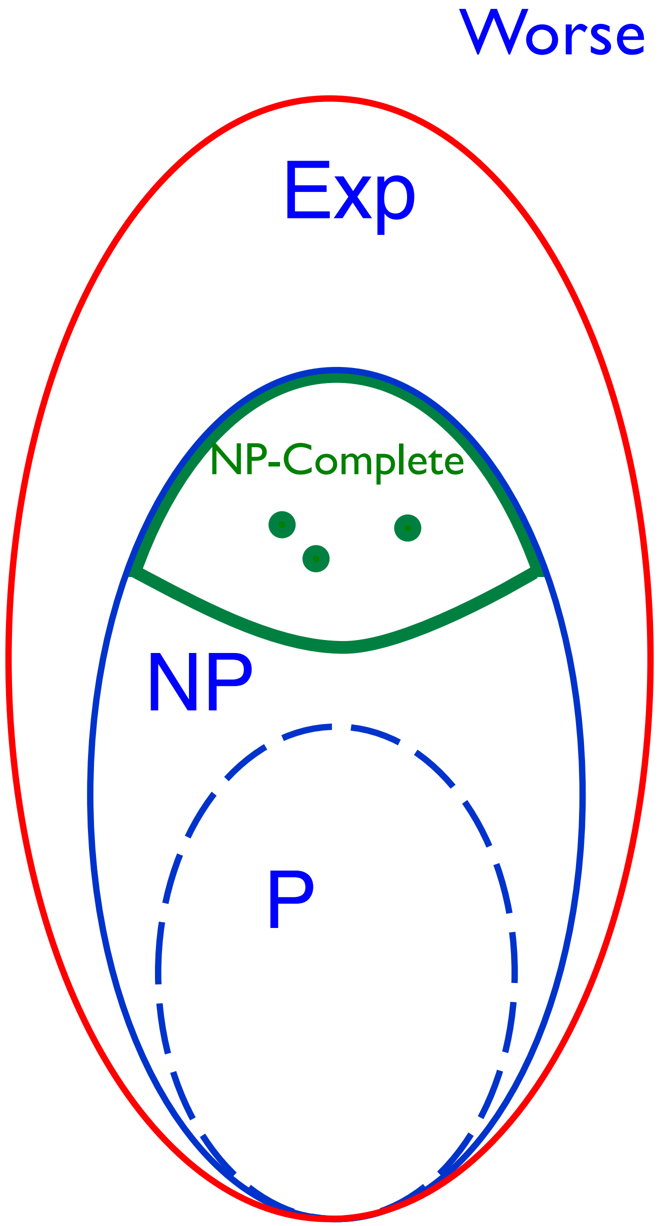
Definition: Problem B is *NP-complete* if:

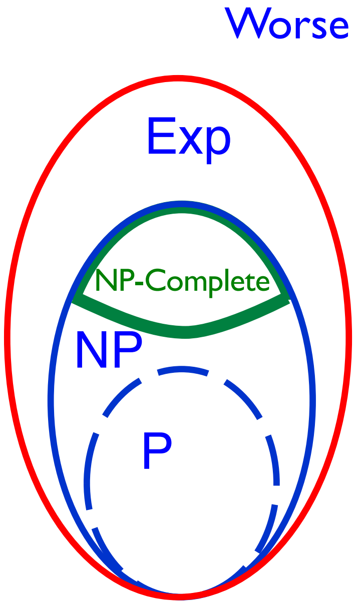
(1) B belongs to NP, and

(2) every problem in NP is polynomially reducible to B.

Intuitively, these are the "hardest problems" in NP

They are also all deeply related–solving any solves them all!





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

To prove NP-completeness

Problem B is NP-complete iff:

(1) B belongs to NP, and either (2) or (2'):

(2) For every problem A in NP, $A \leq_p B$

(2') For some NP-complete problem A, $A \leq_p B$

Thanks, S. Cook

Transitivity of $\leq_p$

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 these earlier

Therefore IndpSet is also NP-complete

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

Ditto for KNAP, 3COLOR, …

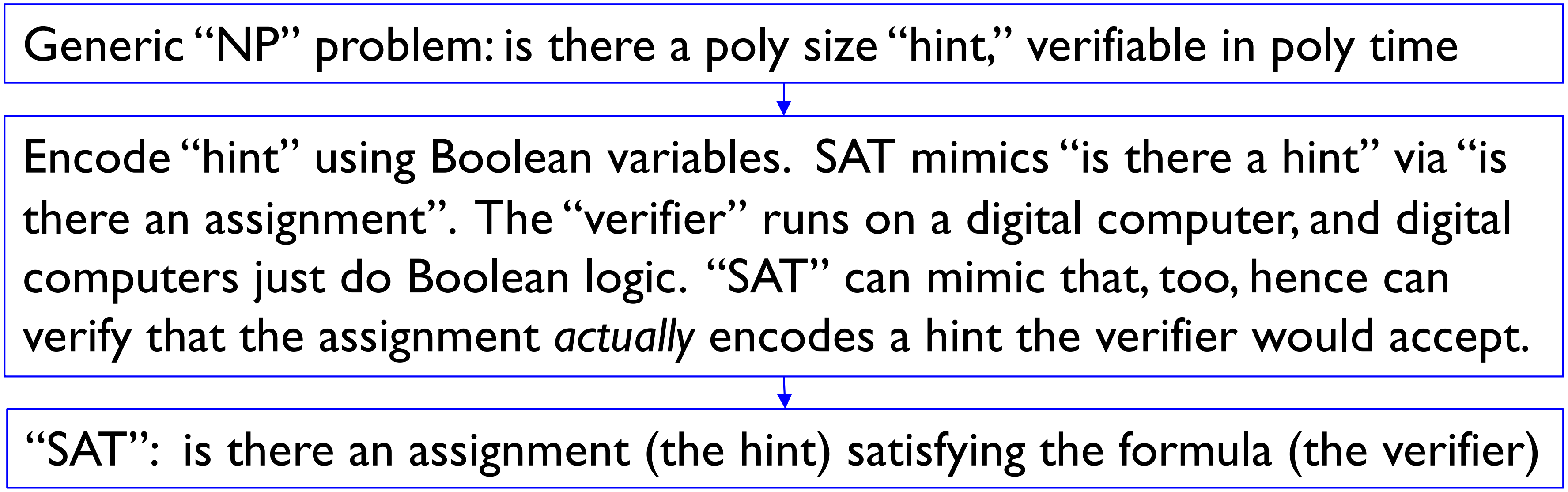
Cook's Theorem

SAT is NP-Complete

Cook’s Theorem

Theorem: Every problem in NP is reducible to SAT

Proof Sketch: SAT assignment = hint; formula = verifier.



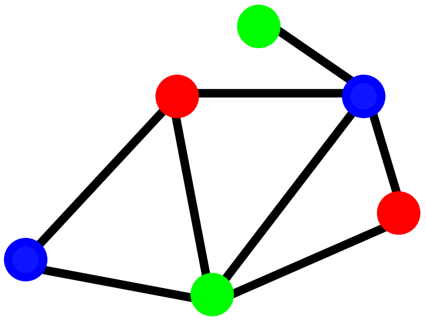
Generic “NP” problem: is there a poly size “hint,” verifiable in poly time

Encode “hint” using Boolean variables. SAT mimics “is there a hint” 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 hint the verifier would accept.

“SAT”: is there an assignment (the hint) satisfying the formula (the verifier)

Pf uses *generic* NP problems, but a few specific examples will give the flavor

3-Coloring $\leq_p$ SAT



Given G = (V, E)

∀ i in V, variables r_i, g_i, b_i encode color of i ← hint

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

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

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

← verifier

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

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

Independent Set $\leq_p$ SAT

Given G = (V, E) and k

$\forall$ i in V, variable x_i encodes inclusion of i in IS $\leftarrow$ hint

$$\underbrace{\bigwedge_{(i,j) \in E} (\neg x_i \vee \neg x_j)}_{\substack{\text{every edge has one end}\\ \text{or other not in IS}\\ \text{(no edge connects 2 in IS)}}} \wedge \underbrace{\text{“number of True } x_i \text{ is } \geq k\text{”}}_{\substack{\text{possible in 3 CNF, but technically}\\ \text{messy, so details omitted;}\\ \text{basically, count 1's}}} \leftarrow \text{verifier}$$

Knapsack $\le_p$ SAT

Given weights w_1, w_2, …, $w_{n,}$ and C

$1 \le i \le n$, variable x_i encodes inclusion of w_i in sum ← hint

Plus b*n variables $y_{i,j}$, where b = # of bits in w_i's

$$\underbrace{\text{“}\Lambda_{(i,j)}\ (y_{i,j} = ((\text{j-th bit of } w_i) \wedge x_i))\text{”}}_{y_{i,j}\text{'s encode bits of selected weights}} \ \wedge\ \underbrace{\text{“ }\Sigma y_{i,-} = C\text{ ”}}_{\text{adds n binary numbers}} \leftarrow \text{verifier}$$

possible in 3 CNF, but technically messy, so details omitted

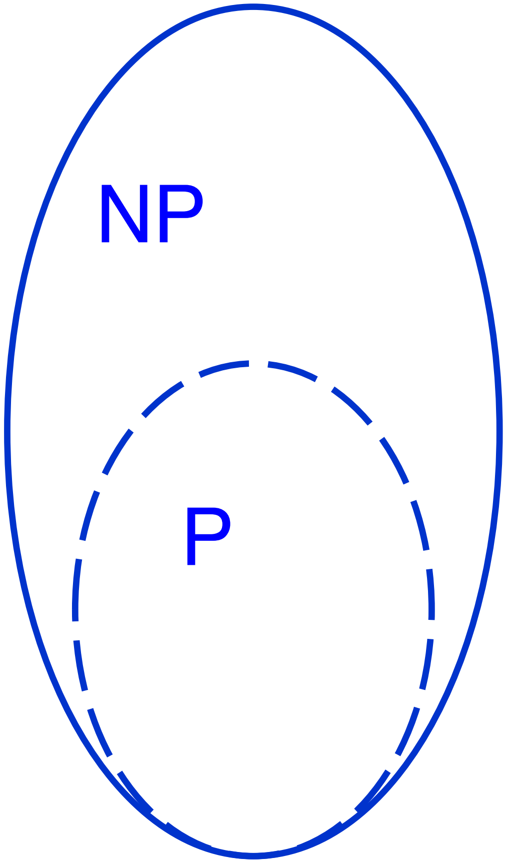
Relating P to NP

Complexity Classes

NP = Polynomial-time verifiable

P = Polynomial-time solvable

P ⊆ 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:

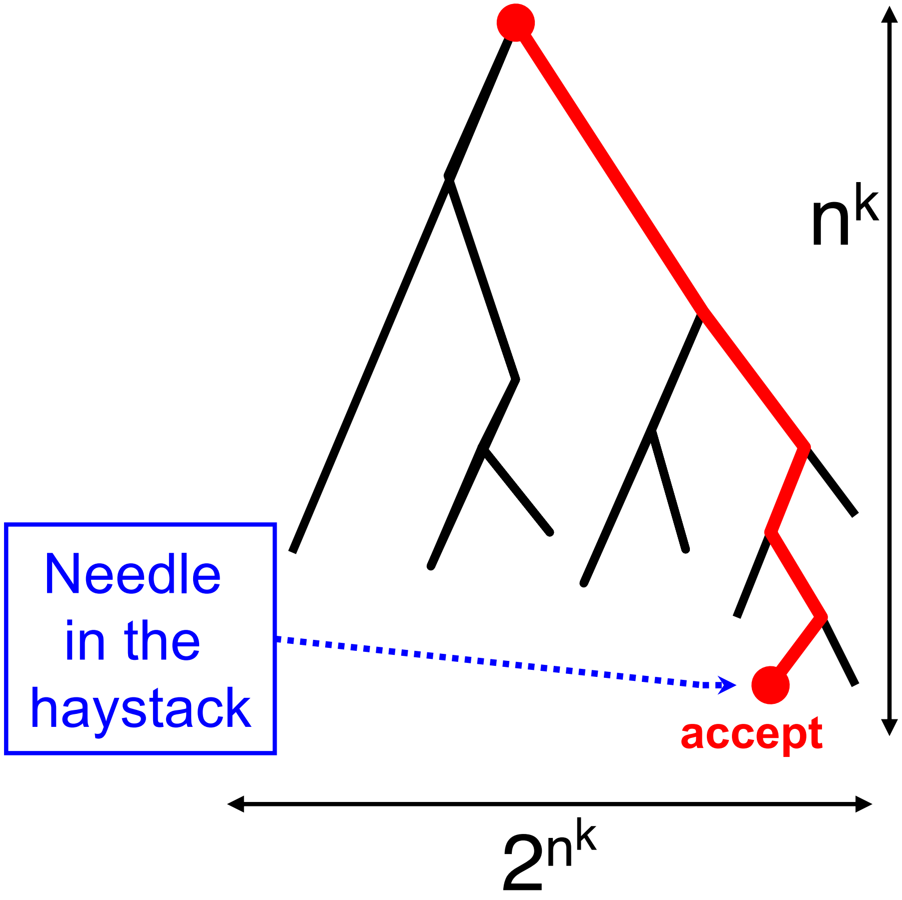
- try all possible hints; check each one to see if it works.
- Exponential time:
  - 2^n truth assignments for n variables
  - n! possible TSP tours of n vertices
  - $\binom{n}{k}$ possible k element subsets of n vertices, perhaps k = log n or n/3
  - etc.

…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, answer YES; if all fail, answer NO.



P and NP

Every problem in P is in NP

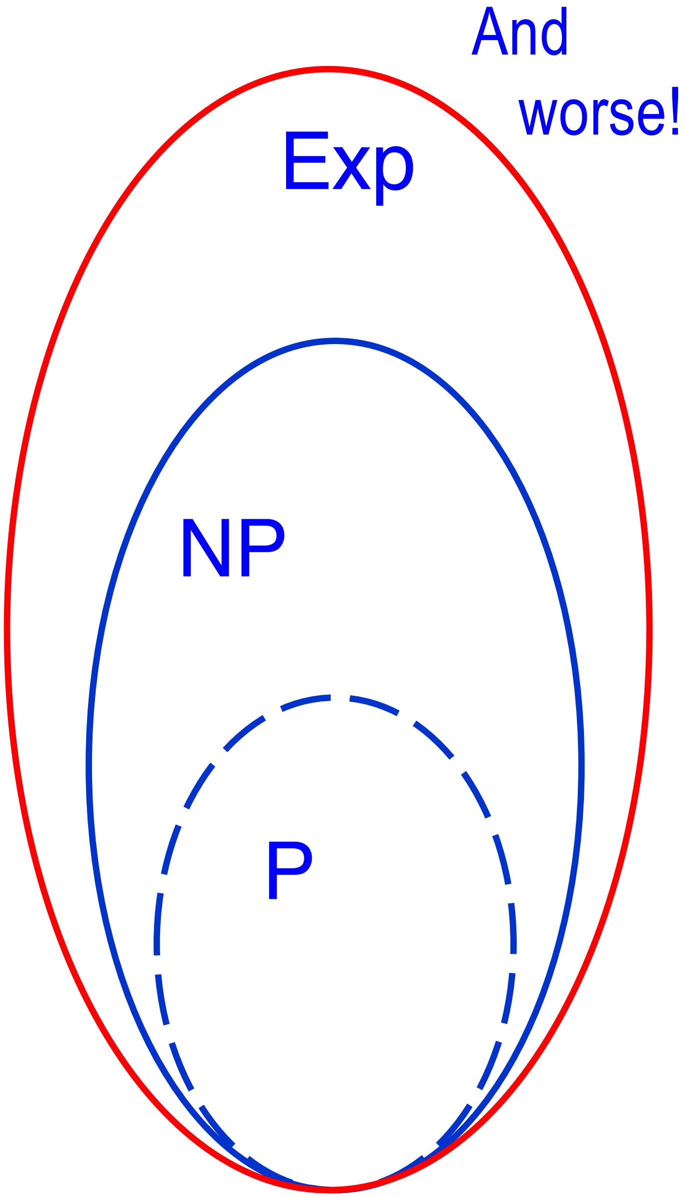
one doesn’t even need a hint for problems in P so just ignore any hint you are given

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)

E.g., see CSE 431



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.)

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

Polynomial Time Reduction, III

Two definitions of “$A \leq_p B$”

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

Cook

Examples on previous slides are special case:

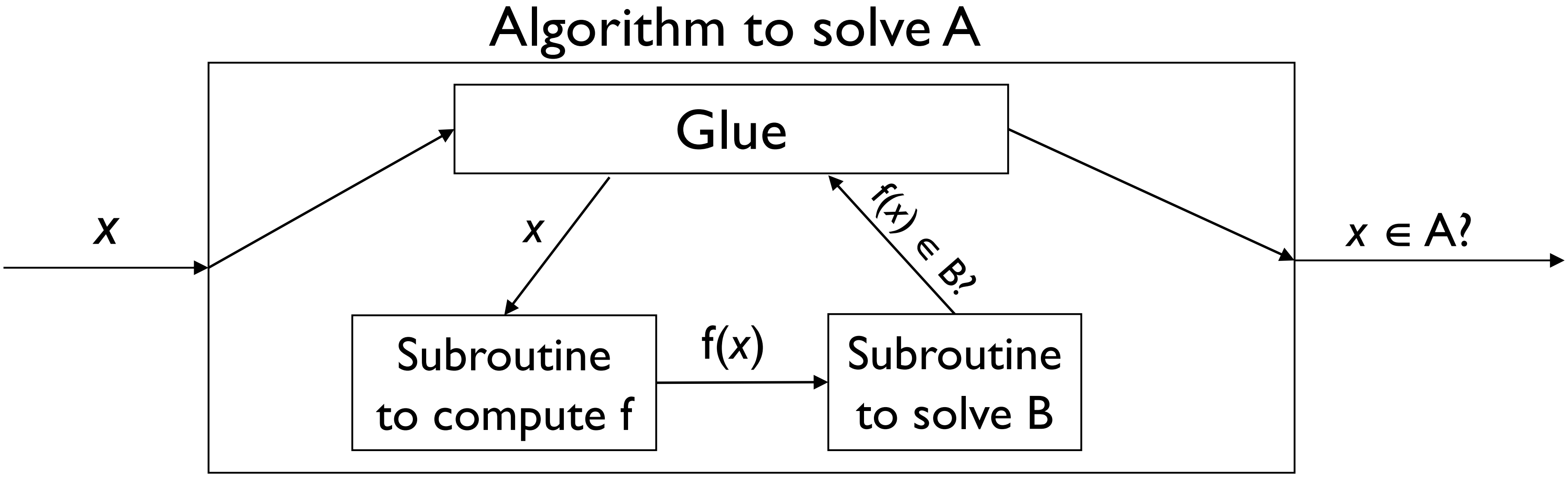
- call the subroutine *once*, report *its* answer.

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, fine, but there’s perhaps a simpler way…

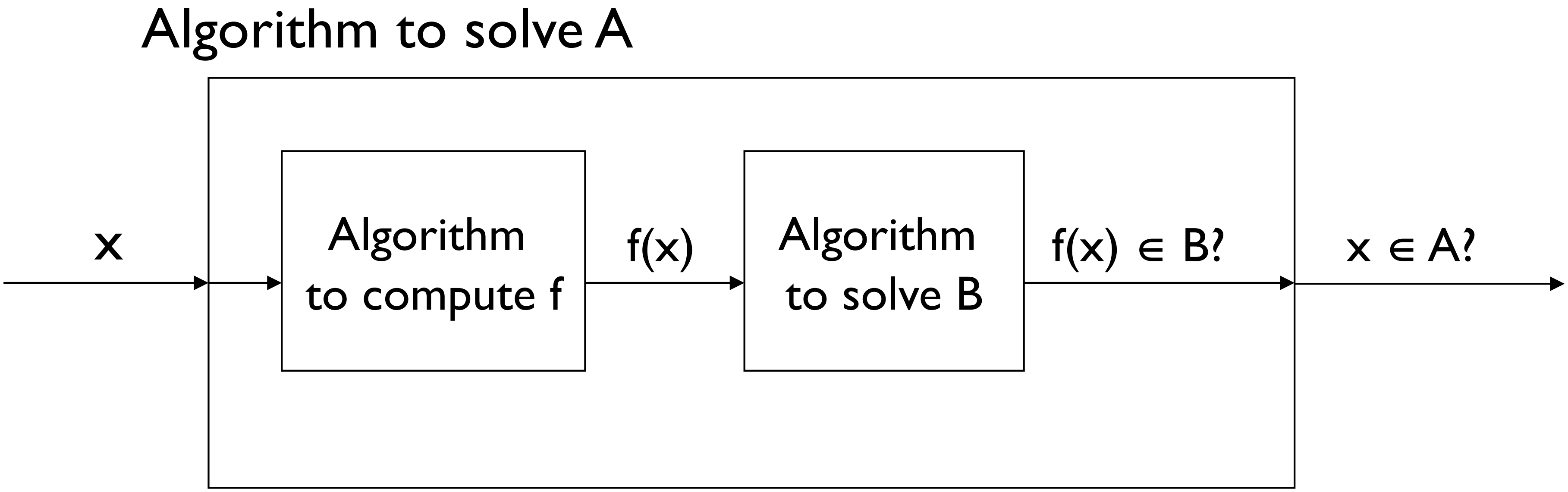
Using an Algorithm for *B* to Solve *A*



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

Key issue: Can we (quickly) turn an A-instance x into one (or more) B-instance(s) f(x) so that answer(s) to "f(x) ∈ B" help us decide x ∈ A"?

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?

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 finds exact solutions to large, arbitrary instances

P vs 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 could *always* do it quickly.

More Reductions

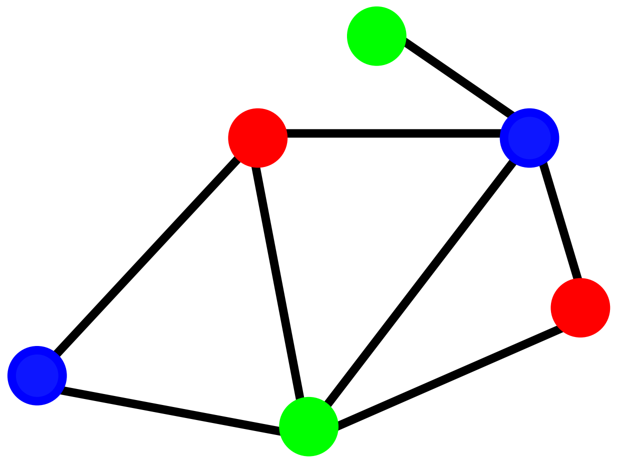
SAT to Coloring

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.

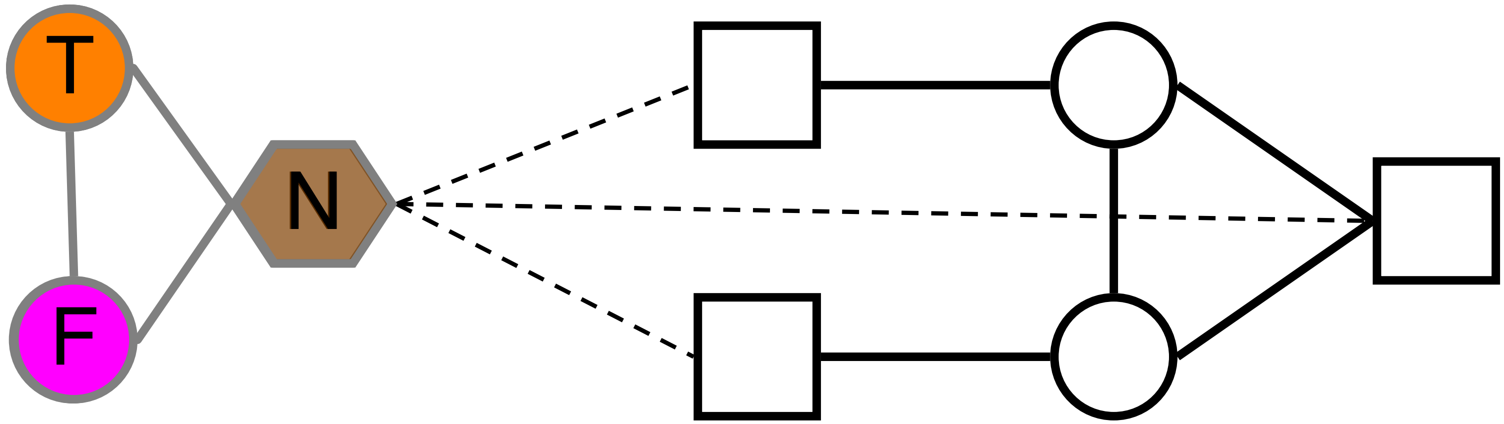
Example:



In NP? Exercise

A 3-Coloring Gadget:

In what ways can this be 3-colored?

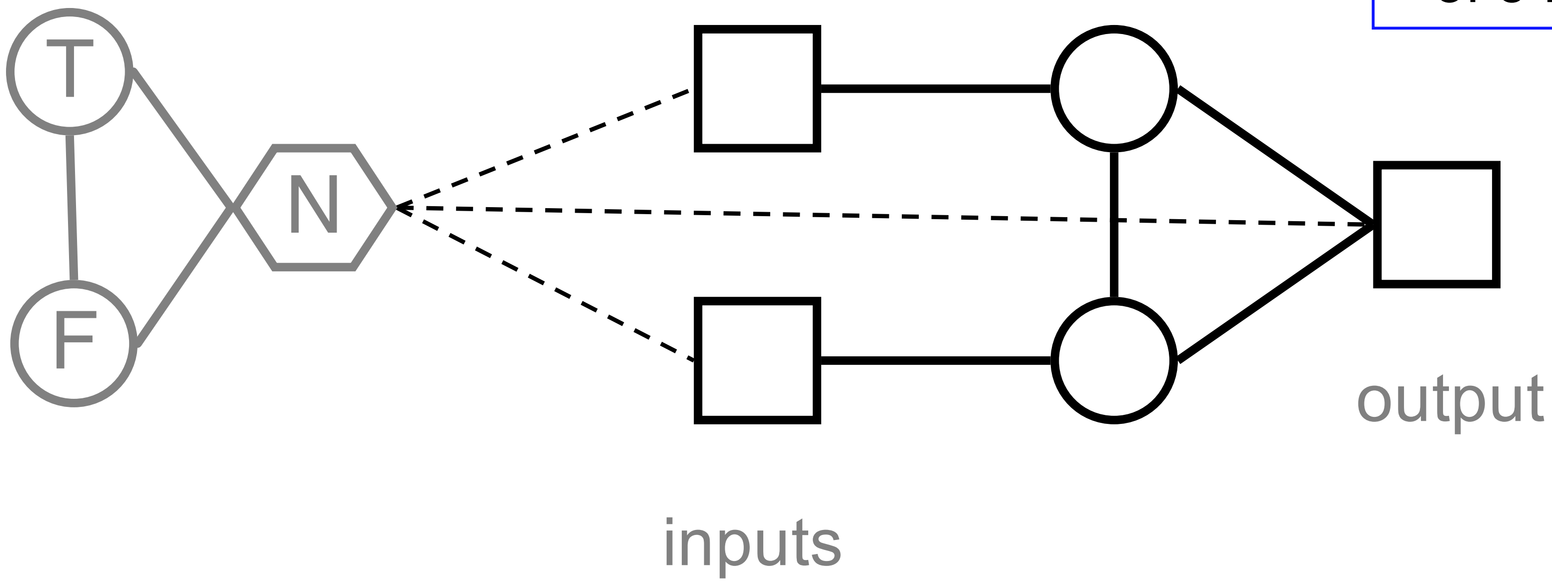


A 3-Coloring Gadget: “Sort of an OR gate“

if output is T, some input must be T

if some input is T, output may 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 (

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: $c = c_1 \wedge c_2 \wedge \ldots \wedge c_q$

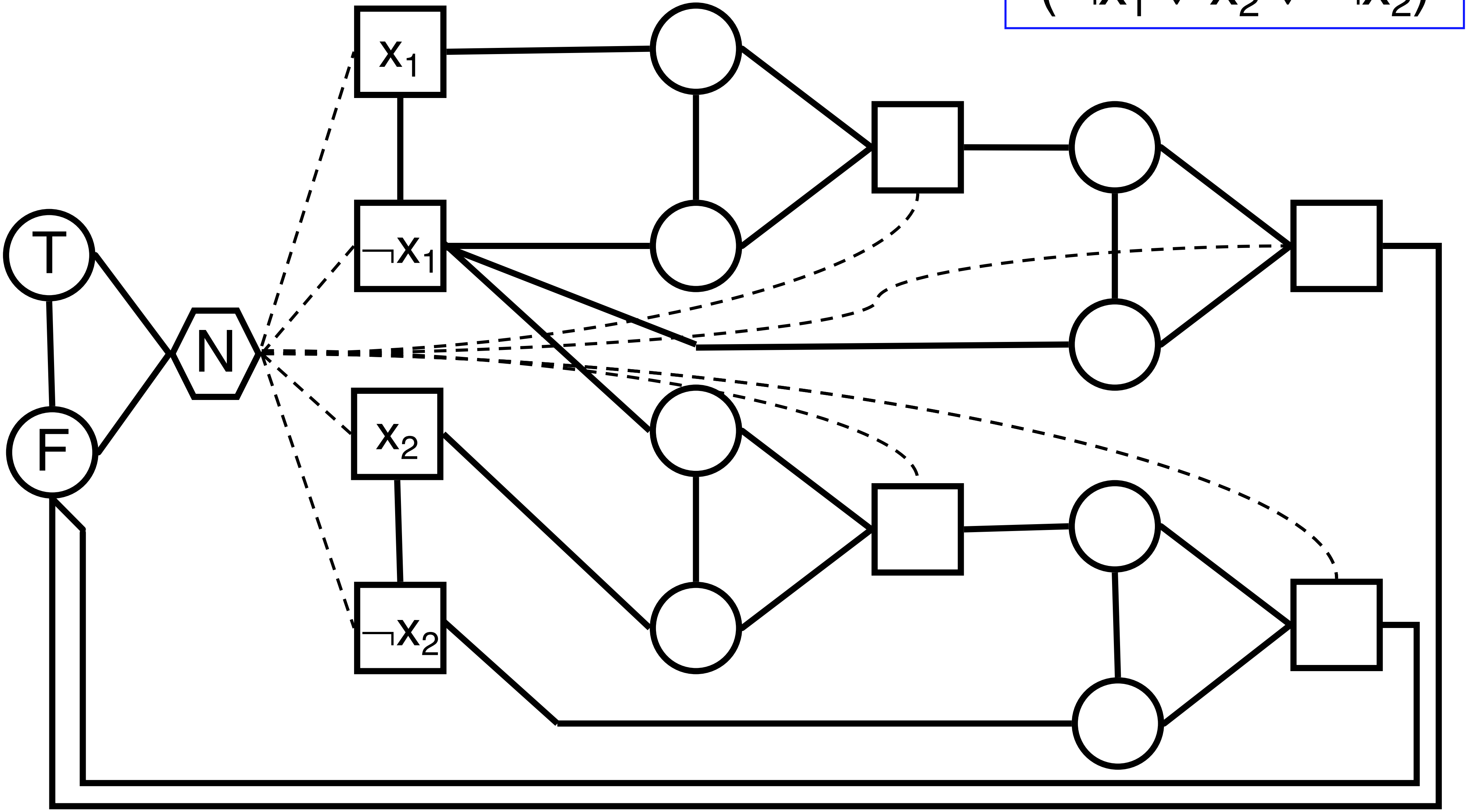
) =

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

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



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.

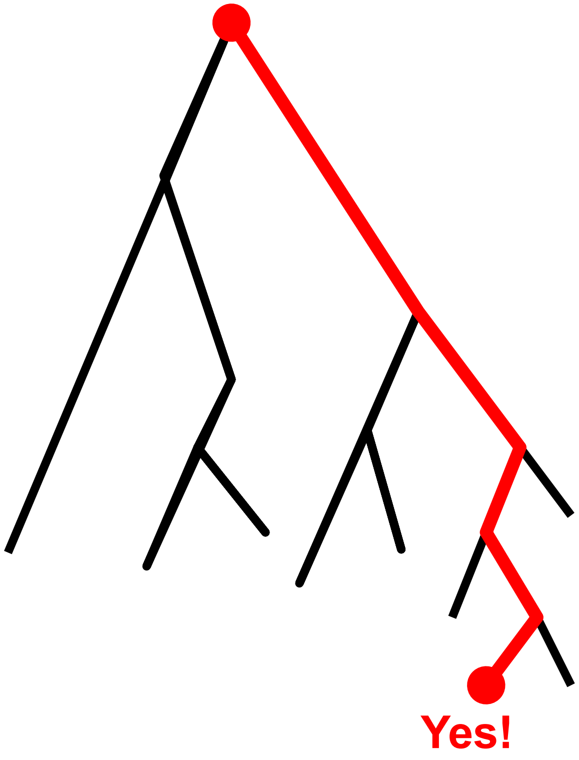
Coping with NP-hardness

Coping with NP-Hardness

- 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. only need planar-/interval-/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

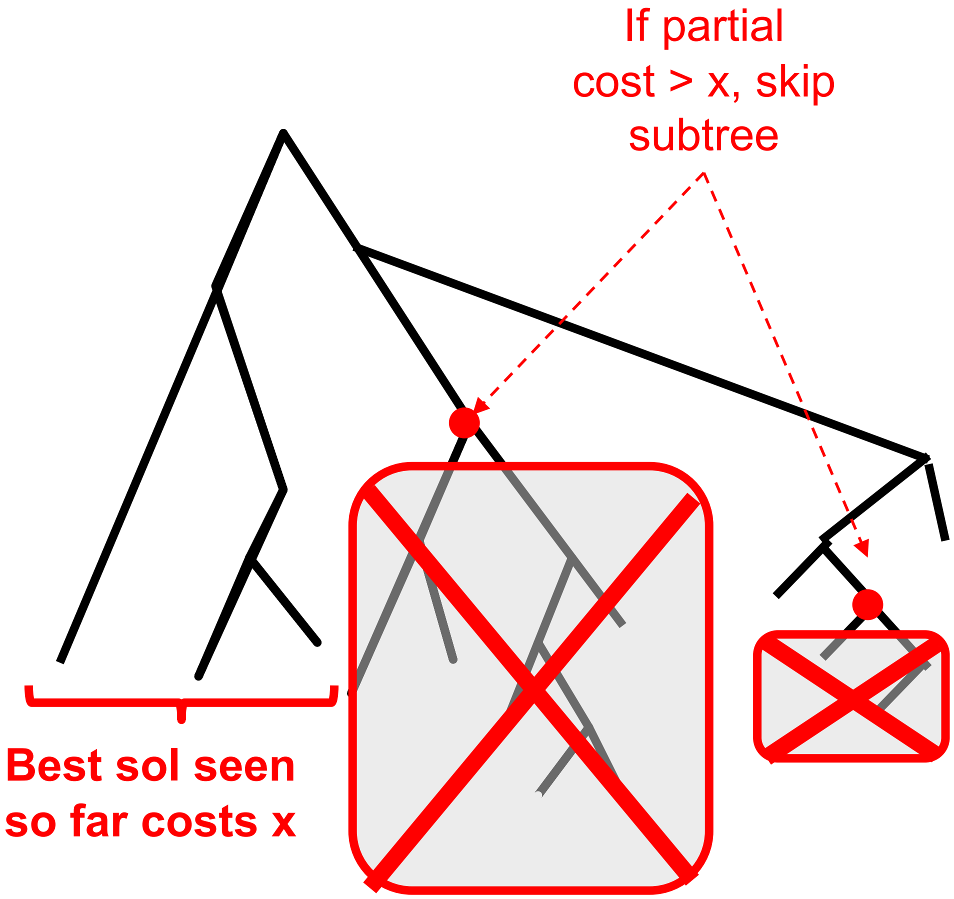
Backtracking

Search problems often naturally described as making a series of choices ("x_i=T or F?"; "Vertex j next in path?" …). Systematically try one after another, "backtracking" to try next alternative after exhausting subsequent possibilities. If any succeed, answer YES; if all fail, answer NO.



Branch-and-Bound

For many minimization problems, during backtrack search, you can lower-bound the cost of all potential continuations of the search based on "partial" solution so far. If that exceeds best known solution, you can cut out entire subtrees.



If partial cost > x, skip subtree
Best sol seen so far costs x

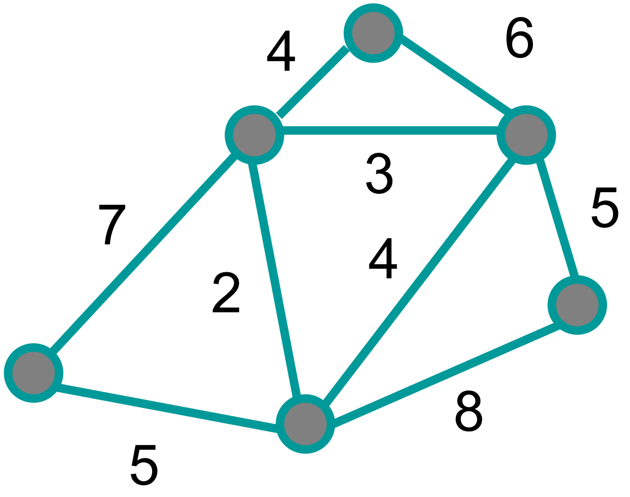
NP-complete problem: TSP

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.

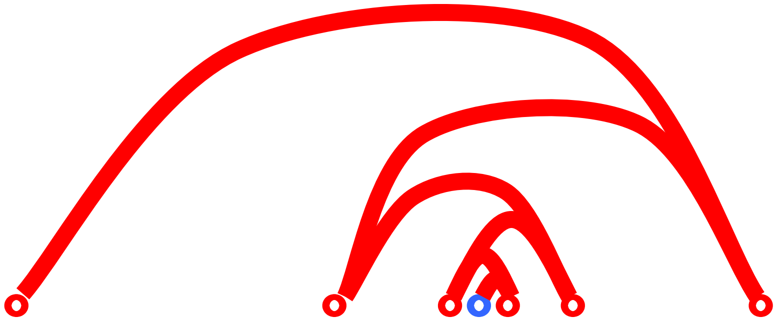
Example:

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

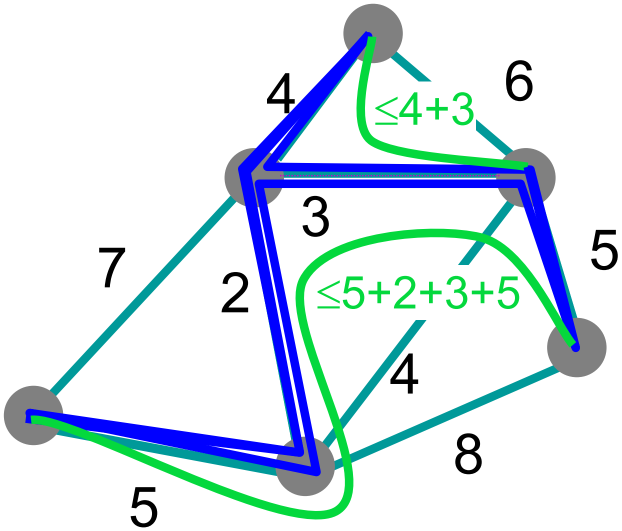
n points in space, Euclidean distance, all possible edges; example omits edges for clarity

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

Find MST

Find “DFS” Tour

Shortcut



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

1.5x Approximation to EuclideanTSP

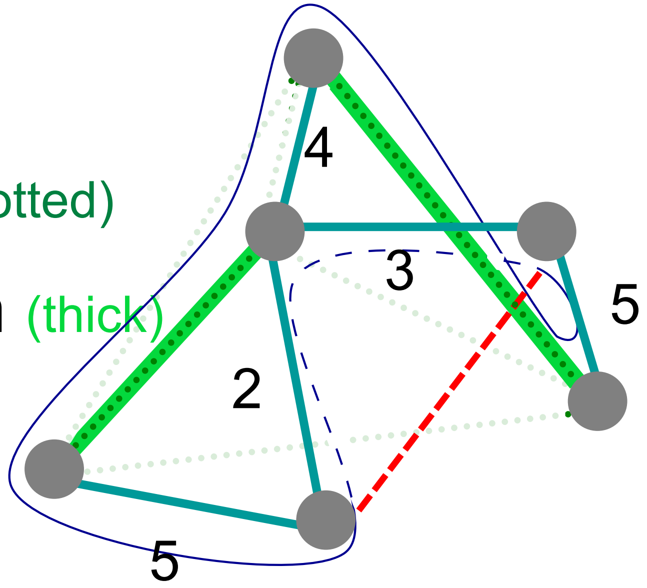
Find MST (solid edges)

Connect odd-degree tree vertices (dotted)

Find min cost matching among them (thick)

Find *Euler* Tour (thin)

Shortcut (dashed)



Shortcut ≤ ET ≤ MST + TSP/2 < 1.5* TSP

↑

Cost of matching ≤
TSP/2 (next slide)

Min Matching ≤ TSP/2

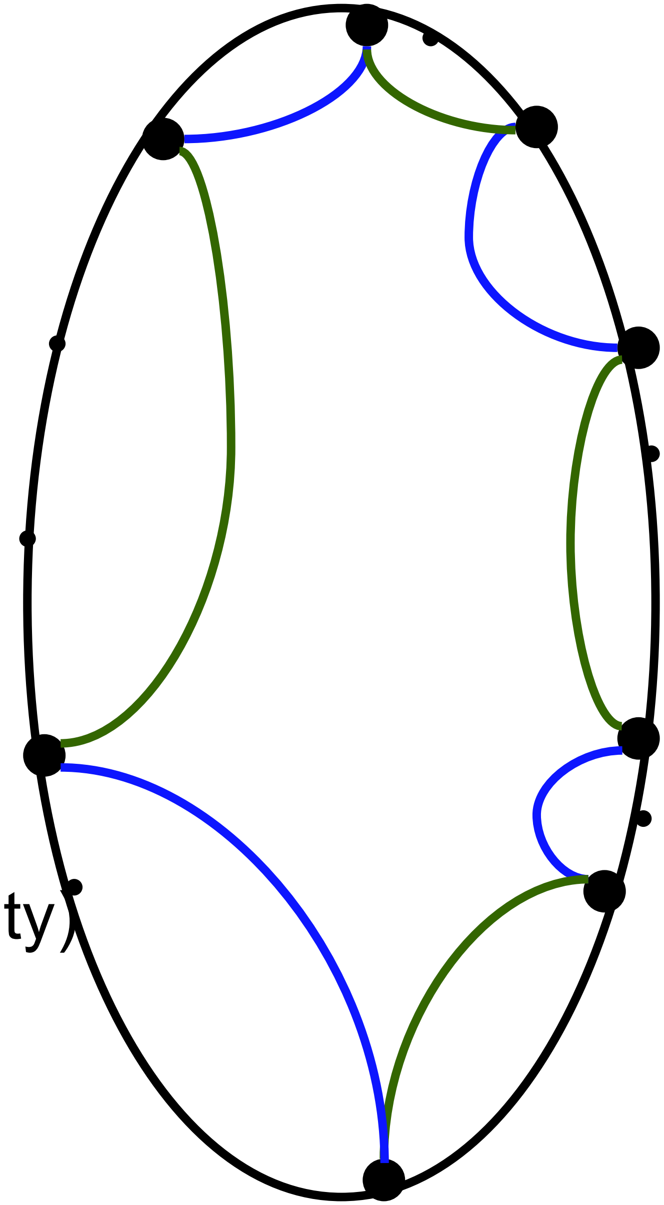
Oval = TSP

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



Progress on TSP approximation

This 1.5x approximation was the best know for ≈ 35 years

CSE faculty member Shayan Oveis Gharan with collaborators Saberi and Singh improved on this a few years ago; you might enjoy watching the recording of the colloquium he gave on this in April, 2013:

New Approximation Algorithms for
the Traveling Salesman Problem

(http://www.cs.washington.edu/events/colloquia/search/details?id=2360)

P / NP Summary

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

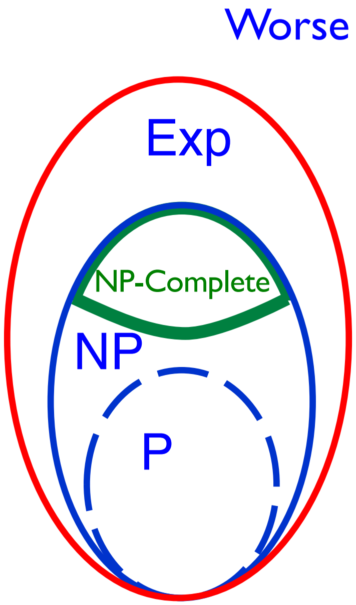
NP

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…" – are also useful views



NP-completeness

Defn & Properties of $\leq_p$

A is NP-complete: in NP & everything in NP reducible to A

"the hardest problems in NP"

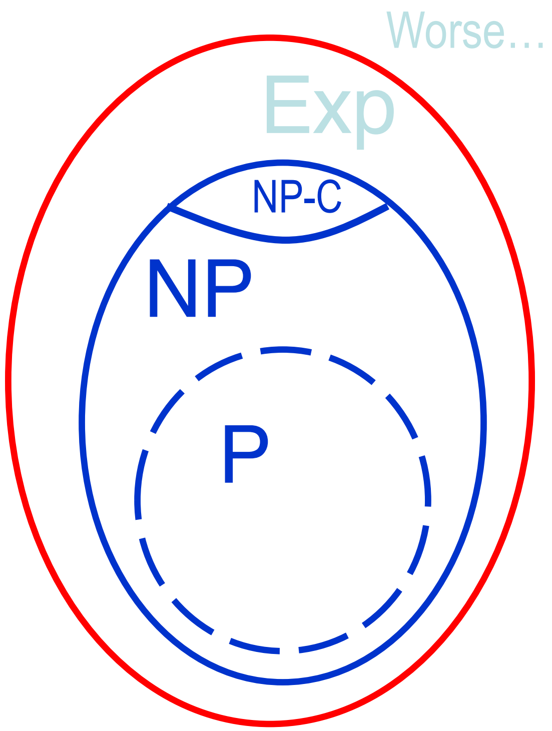
"All alike under the skin"

Most known natural problems in NP are complete

#1: 3CNF-SAT

Many others: Clique, IndpSet, 3Color, KNAP, HamPath, TSP, …

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.

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

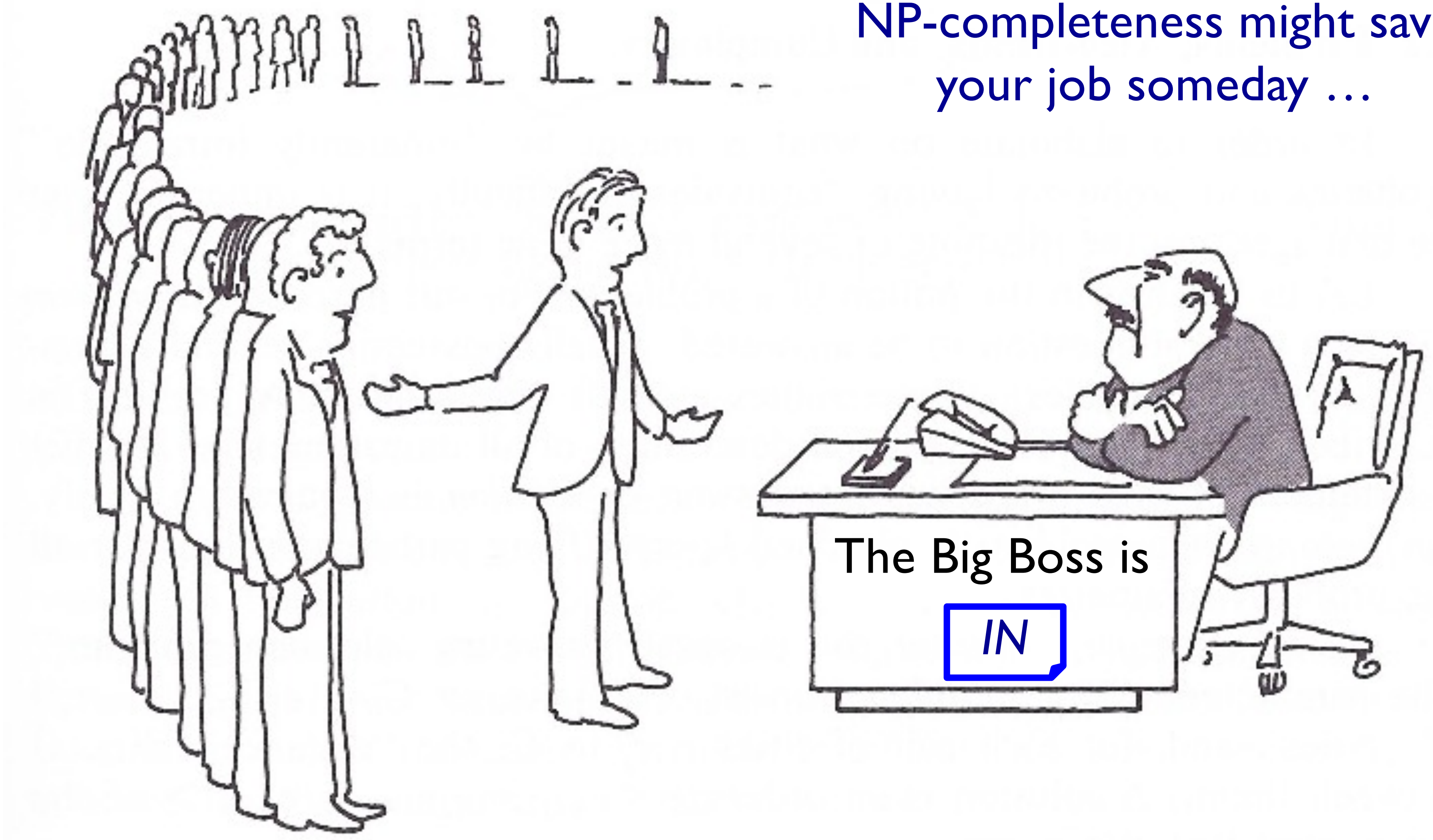
"Find a satisfying assignment, then output…"

Half Reductions

E.g., delete dashed edges in 3Color reduction. It's still true that "c satisfiable $\Rightarrow$ G is 3 colorable", but 3-colorings don't necessarily give satisfying- (or valid) assignments.

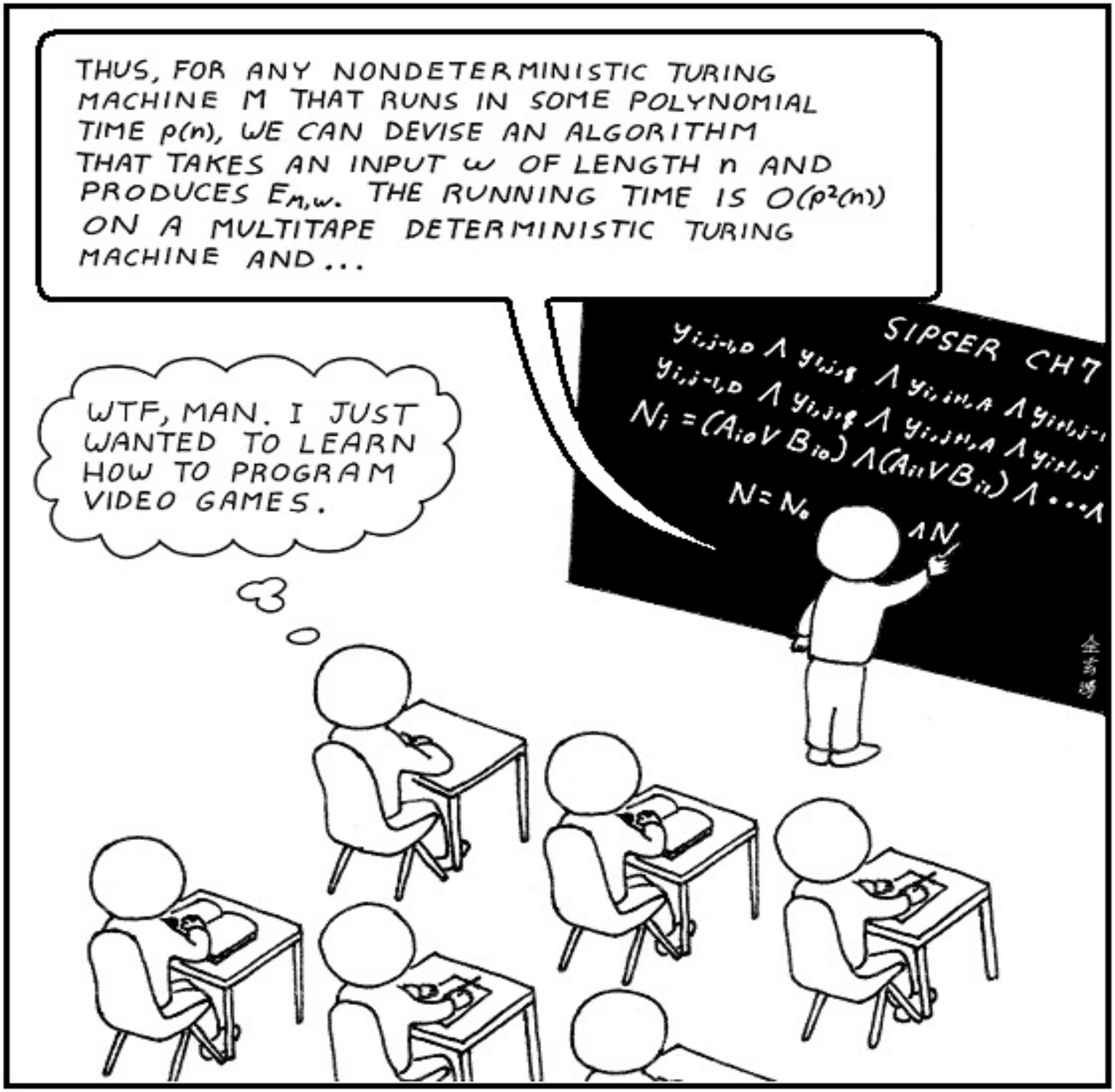
E.g., add or delete slacks in KNAP: similar troubles

NP-completeness might save your job someday …



"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
$y_{i,j-1,b} \wedge y_{i,j,q} \wedge y_{i,j+1,a} \wedge y_{i+1,j-1}$
$y_{i,j-1,b} \wedge y_{i,j,q} \wedge y_{i,j+1,a} \wedge y_{i+1,j}$
$N_i = (A_{i0} \vee B_{i0}) \wedge (A_{i1} \vee B_{i1}) \wedge \cdots \wedge$
$N = N_0$
$\wedge N$