CSE 421
Introduction to Algorithms
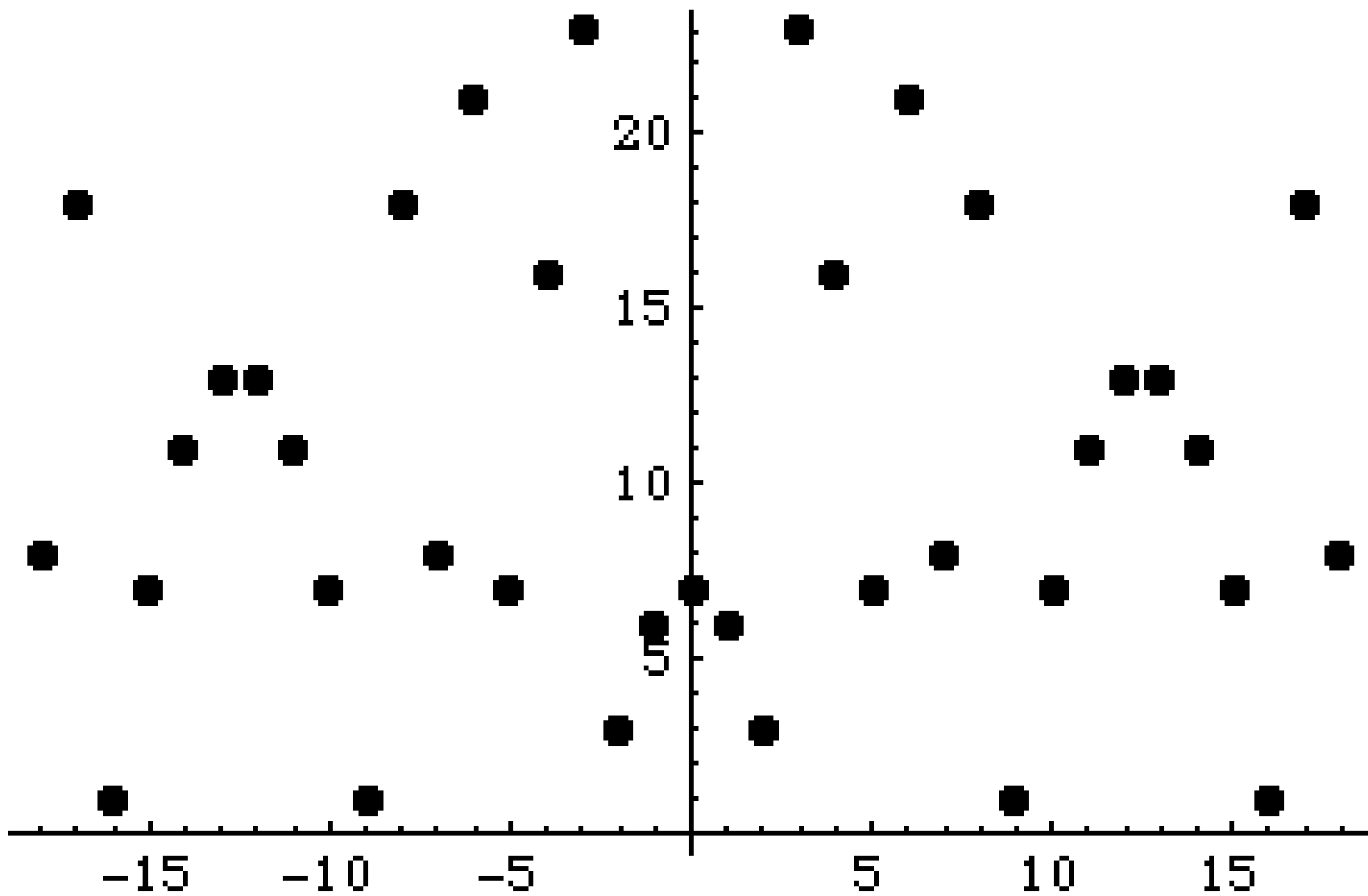# Summer 2007

NP-Completeness

(Chapter 8)

Some Algebra Problems (Algorithmic)
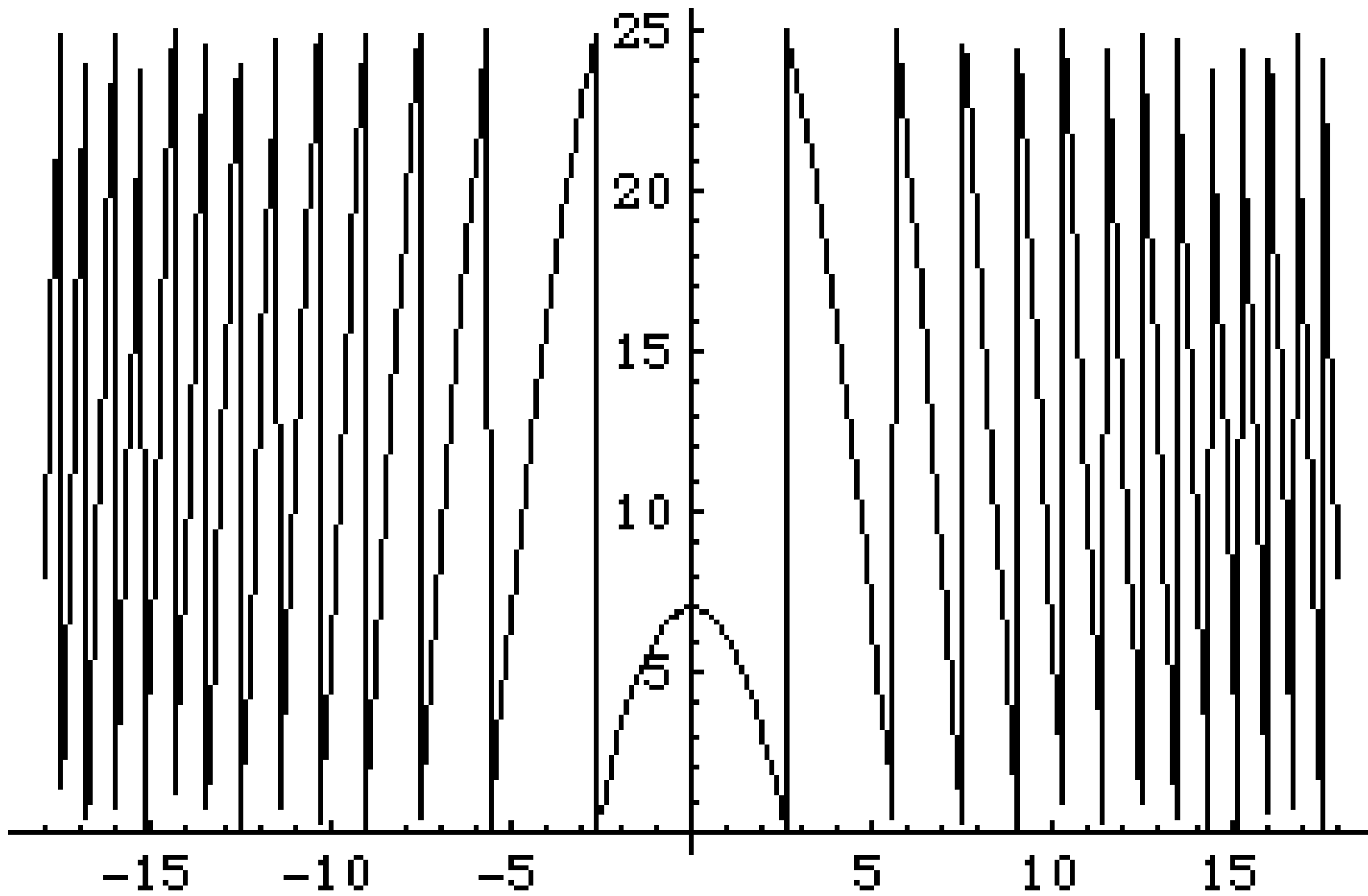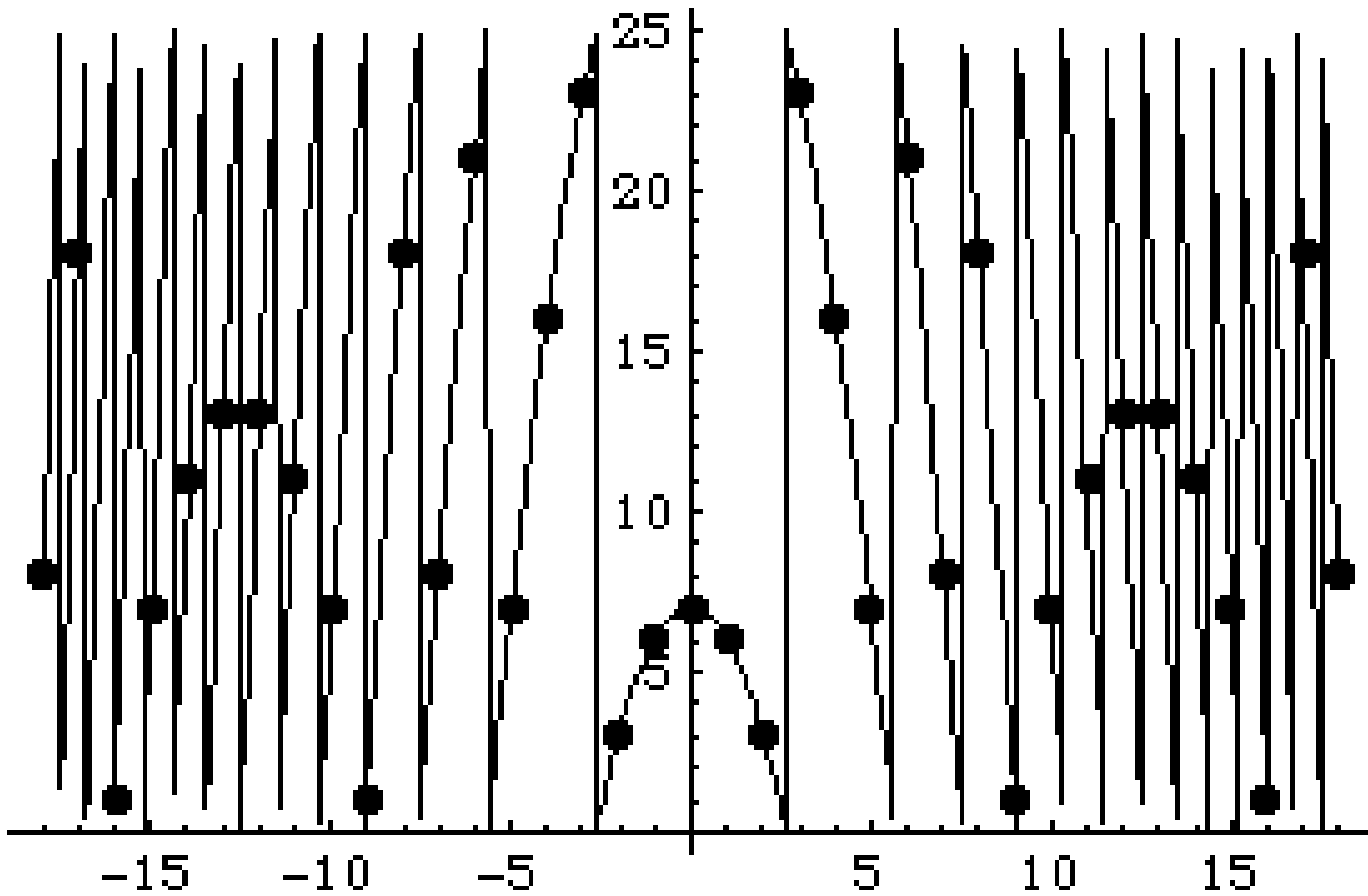
Given positive integers a, b, c

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

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

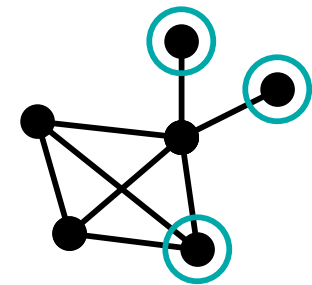Question 3: do there exist positive integers x and y such that $ax^2 + by = c$?
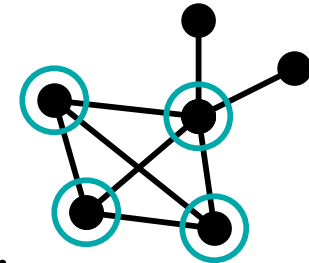
4

5

Some Problems

Independent-Set:

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

Clique:

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

8

A Brief History of Ideas

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

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

1900: David Hilbert's famous speech outlines program: mechanize all of mathematics?
http://mathworld.wolfram.com/HilbertsProblems.html

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

More History

1930/40's

What is (is not) computable

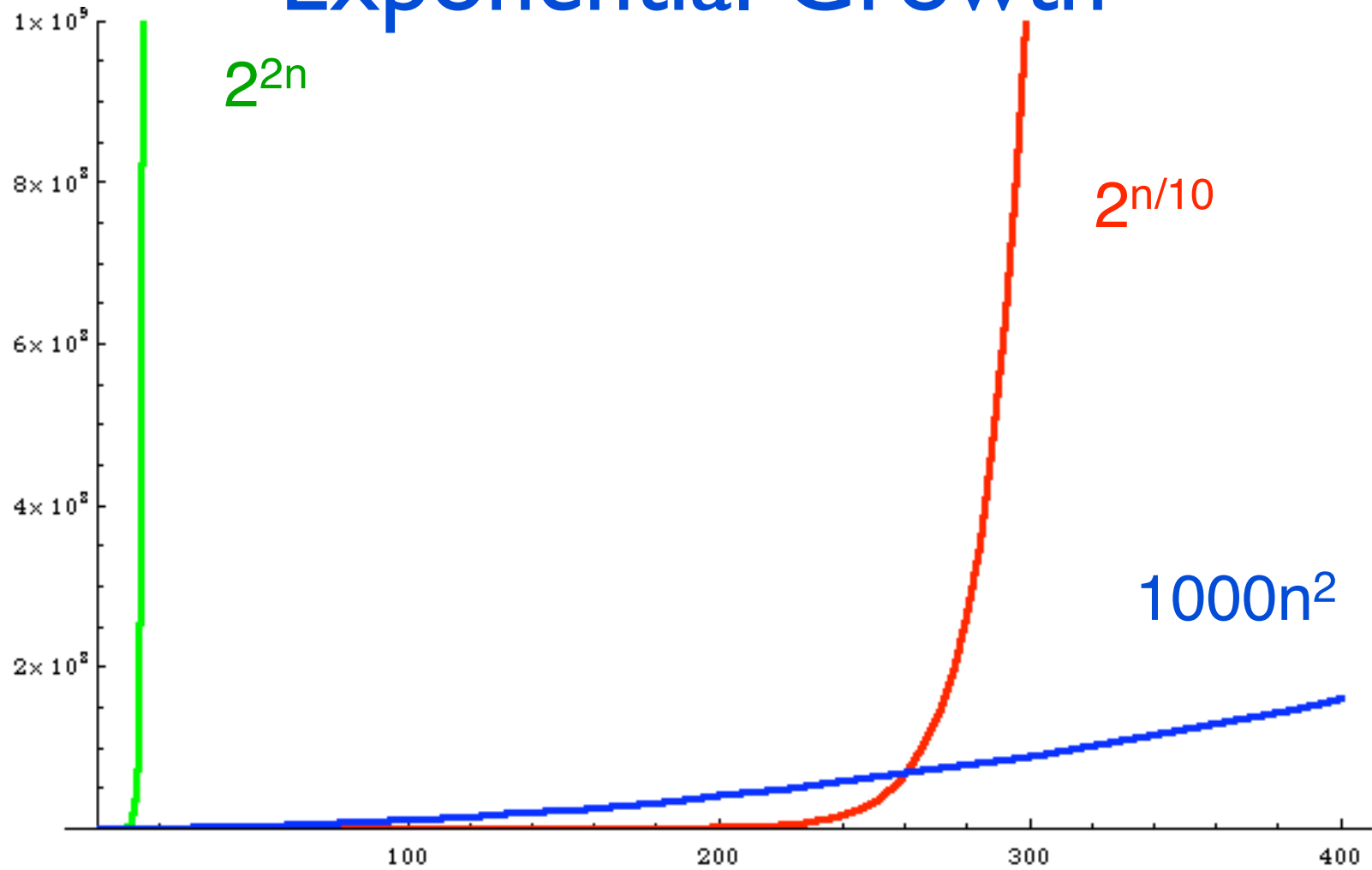**1960/70's**

What is (is not) feasibly computable

Goal – a (largely) technology-independent theory of time required by algorithms

Key modeling assumptions/approximations

Asymptotic (Big-O), worst case is revealing

Polynomial, exponential time – qualitatively different

Polynomial vs Exponential Growth

2^{2n}

$2^{n/10}$

$1000n^2$

Another view of Poly vs Exp

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

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

Polynomial versus exponential

We'll say any algorithm whose run-time is

    polynomial is good

    bigger than polynomial is bad

Note – of course there are exceptions:

    n^{100} is bigger than $(1.001)^n$ for most practical values of n but usually such run-times don't show up

    There are algorithms that have run-times like $O(2^{sqrt(n)/22})$ and these may be useful for small input sizes, but they're not too common either

Some Convenient Technicalities

"Problem" – the general case

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

"Problem Instance" – the specific cases

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

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

Decision Problems – Just Yes/No answer

Problems as Sets of "Yes" Instances

    Ex: CLIQUE = { (G,k) | G contains a k-clique }

      E.g., (, 4) $\notin$ CLIQUE

      E.g., (, 3) $\in$ CLIQUE

14

Decision problems

Computational complexity usually analyzed using decision problems

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

Why?

much simpler to deal with

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

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

The class P

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

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

These problems are sometimes called tractable problems.

Examples: sorting, shortest path, MST, connectivity, RNA folding & other dyn. prog. – most of 421

 (exceptions: Change-Making/Stamps, TSP)

Beyond P?

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

e.g. CLIQUE:

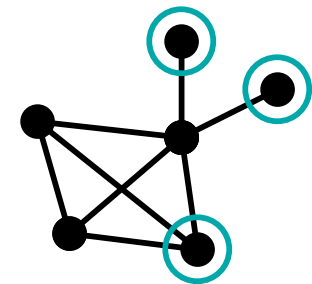Given an undirected graph G and an integer k, does G contain a k-clique?

e.g. quadratic Diophantine equations:

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

Some Problems

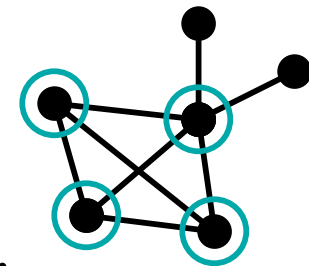## Independent-Set:

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

Clique:

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

Some More Problems

Euler Tour:

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

Hamilton Tour:

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

TSP:

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

Satisfiability

Boolean variables x_1, ..., x_n

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

Literals

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

Clause

  a logical OR of one or more literals

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

CNF formula

  a logical AND of a bunch of clauses

Satisfiability

CNF formula example

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

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

the one above is, the following isn't

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

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

Satisfiable?

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

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

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

2-SAT

Min Cut

Shortest Path

Hamilton Tour

3-SAT

Max Cut

Longest Path

Superficially different; similar computationally

Similar pairs; seemingly different computationally

24

Common property of these problems: Discrete Exponential Search

There is a special piece of information, a short hint proof or certificate, that allows you to efficiently (in polynomial-time) verify that the YES answer is correct. *BUT*, this hint might be *very hard to find – buried in an exponentially large search space*

e.g.

    TSP: one tour among many, "know it when you see it"
    Independent-Set, Clique: the vertex set U; ditto

    Satisfiability: an assignment that makes formula true; ditto

    Quadratic Diophantine eqns: the numbers x & y; ditto

The complexity class NP

NP consists of all decision problems where

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

one among exponentially many; know it when you see it

And

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

(implausible for all exponential time problems)

More Precise Definition of NP

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

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

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

"Hints" sometimes called "Certificates"

Example: CLIQUE is in NP

procedure v(x,h)

   if

      x is a well-formed representation of a graph
      G = (V, E) and an integer k,

   and

      h is a well-formed representation of a k-vertex
      subset U of V,

   and

      U is a clique in G,

   then output "YES"

   else output "I'm unconvinced"

Is it correct?

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

and

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

Another example: SAT ∈ NP

Hint: the satisfying assignment A

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

> Syntax: True iff F is a well-formed formula & A is a truth-assignment to its variables

> Satisfies: plug A into F and evaluate

Correctness:

> If F is satisfiable, it has some satisfying assignment A, and we'll recognize it

> If F is unsatisfiable, it doesn't, and we won't be fooled

Keys to showing that a problem is in NP

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

What's the input? Which are YES?

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

    OK if some inputs need no hint

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

Complexity Classes

NP = Polynomial-time
      verifiable

P   = Polynomial-time
      solvable
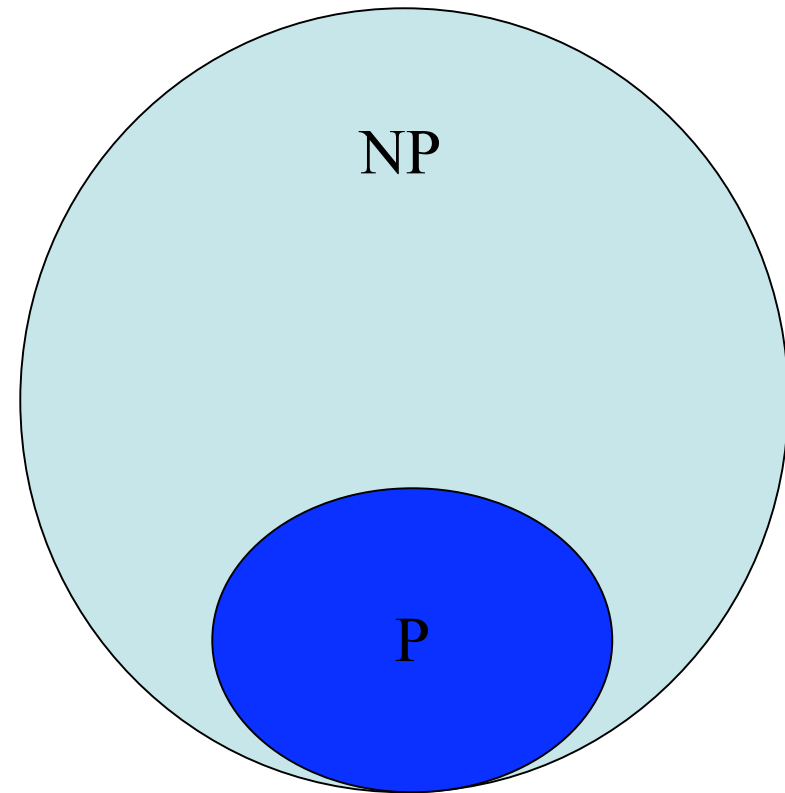
Solving NP problems without hints

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

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

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

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…)

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.

Needle in the haystack

n^k

accept

2^{n^k}
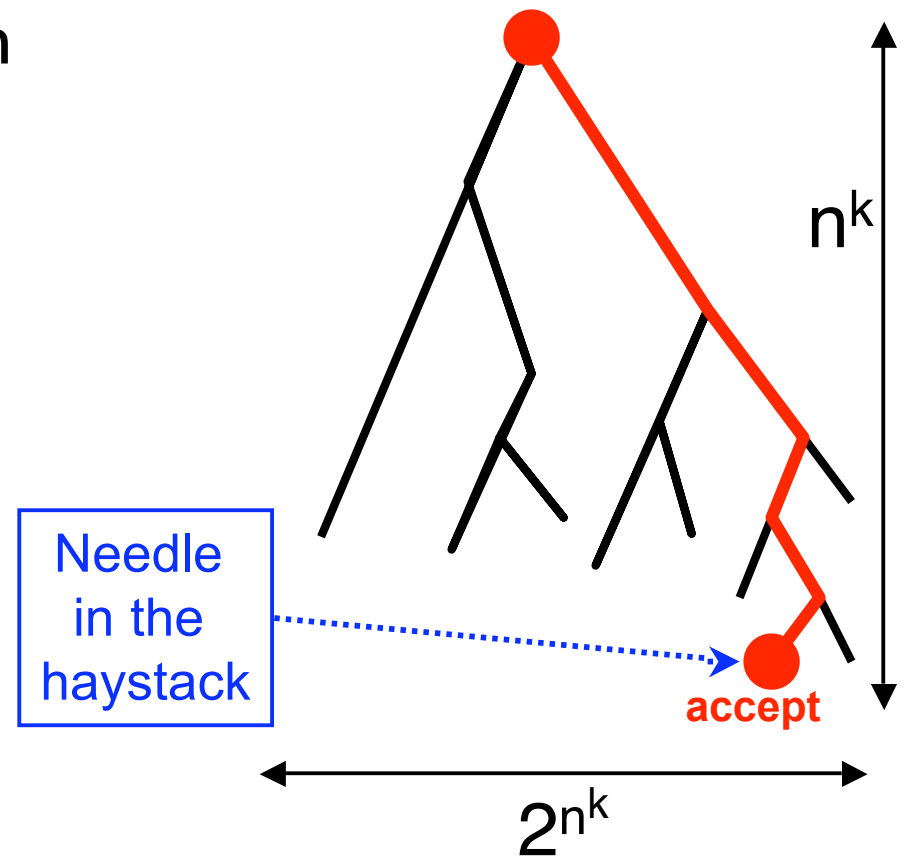
P and NP

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)

Worse…

Exp

NP

P
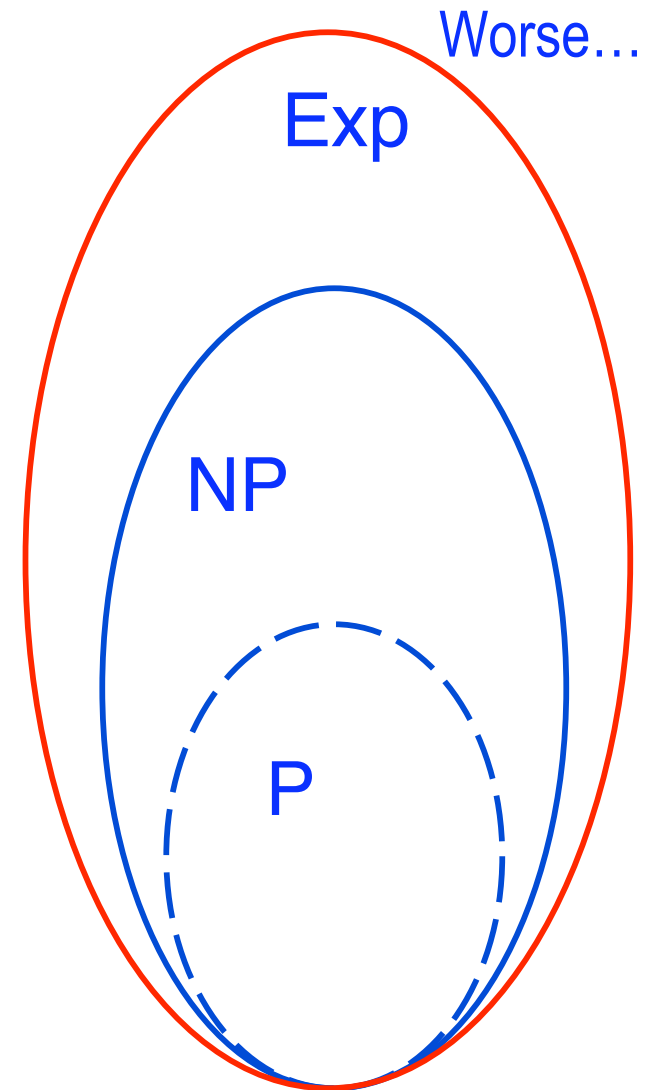
P vs NP

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 = "nondeterministic poly time solvable/verifiable"

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.

NP: Yet to come

NP-Completeness: the "hardest" problems in NP.

Surprisingly, most know problems in NP are equivalent, in a strong sense, despite great superficial differences.

Reductions: key to showing those facts.

Does P = NP?

This is an open question.

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

No one has shown this yet.

(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 Connections

Some Examples in NP

      Satisfiability

      Independent-Set

      Clique

      Vertex Cover

Hints help on all, but all seem hard to solve without

Very surprising fact:

> Fast solution to any gives fast solution to all, & to every other problem in NP!

NP-complete Problems

Seems likely that there are problems in NP – P; if so, none can be solved in polynomial time.

Non-Definition: NP-complete = the hardest problems in the class NP. (Formal definition later.)

Interesting fact: If any one NP-complete problem could be solved in polynomial time, then all NP problems could be solved in polynomial time.

Complexity Classes

NP = Poly-time **verifiable**

P = Poly-time **solvable**

NP-Complete =
"**Hardest**" problems in
NP

44
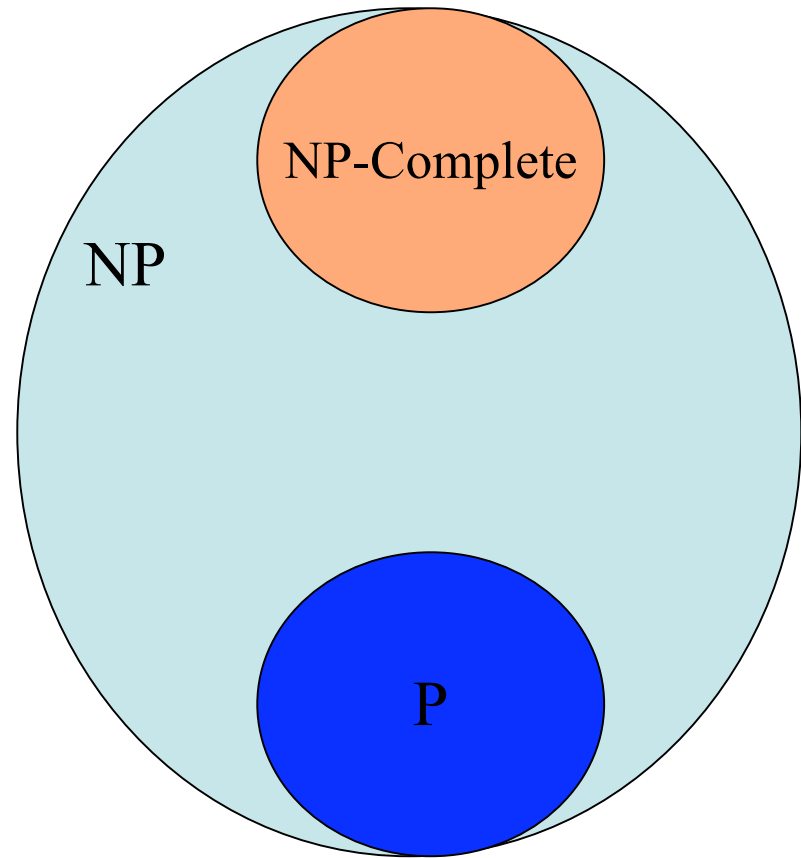
The class NP-complete (cont.)

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

Fact (Dogma): 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, Hamiltonian cycle, TSP, bin packing.

45
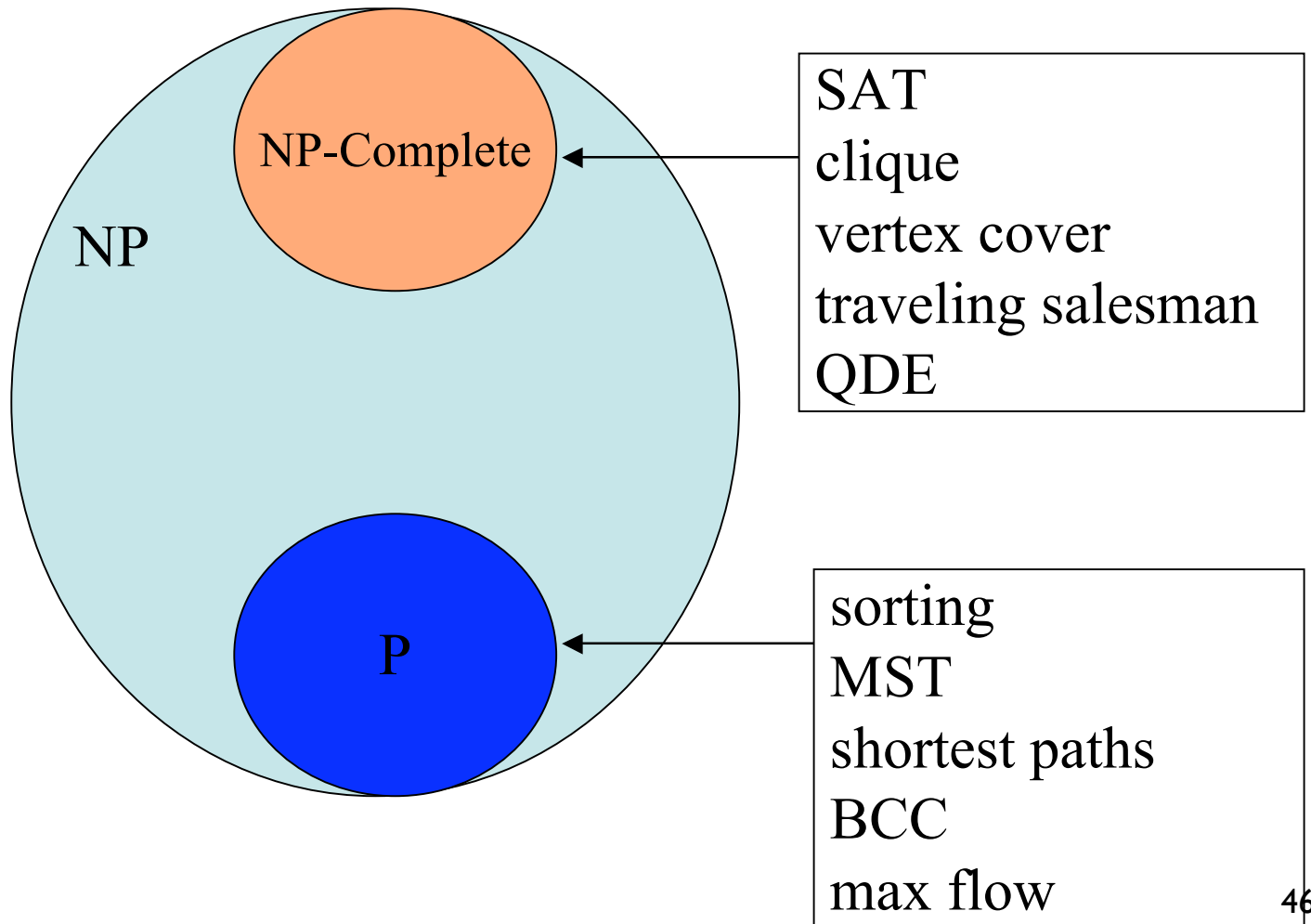
Is all of this useful for anything?

Earlier in this class we learned techniques for solving problems in P.

Question: Do we just throw up our hands if we come across a problem we suspect not to be in P?

Dealing with NP-complete Problems

What if I think my problem is not in P?

Here is what you might do:

1) Prove your problem is NP-hard or -complete
 (a common, but not guaranteed outcome)

2) Come up with an algorithm to solve the problem
usually or approximately.

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?
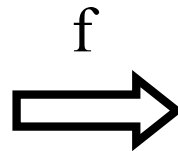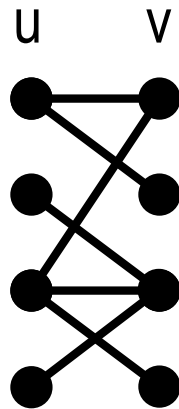
u   v

f

s   t
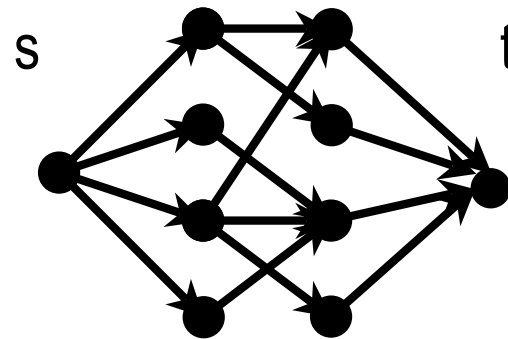
All capacities = 1

Reductions: Why useful

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

Fast algorithm for B implies fast algorithm for A
(nearly as fast; takes some time to set up call, etc.)

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

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

SAT is NP-complete

Cook's theorem: SAT is NP-complete

Satisfiability (SAT)

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

S=(x+y+¬z)•(¬x+y+z)•(¬x+¬y+¬z)

Example above is satisfiable. (We can see this by setting x=1, y=1 and z=0.)

NP-complete problem: Vertex Cover
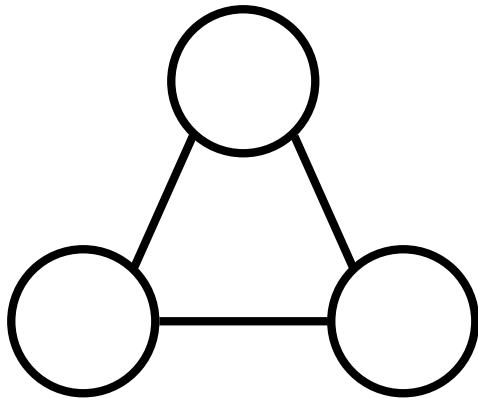
Input: Undirected graph G = (V, E), integer k.

Output: True iff there is a subset C of V of size ≤ k such that every edge in E is incident to at least one vertex in C.
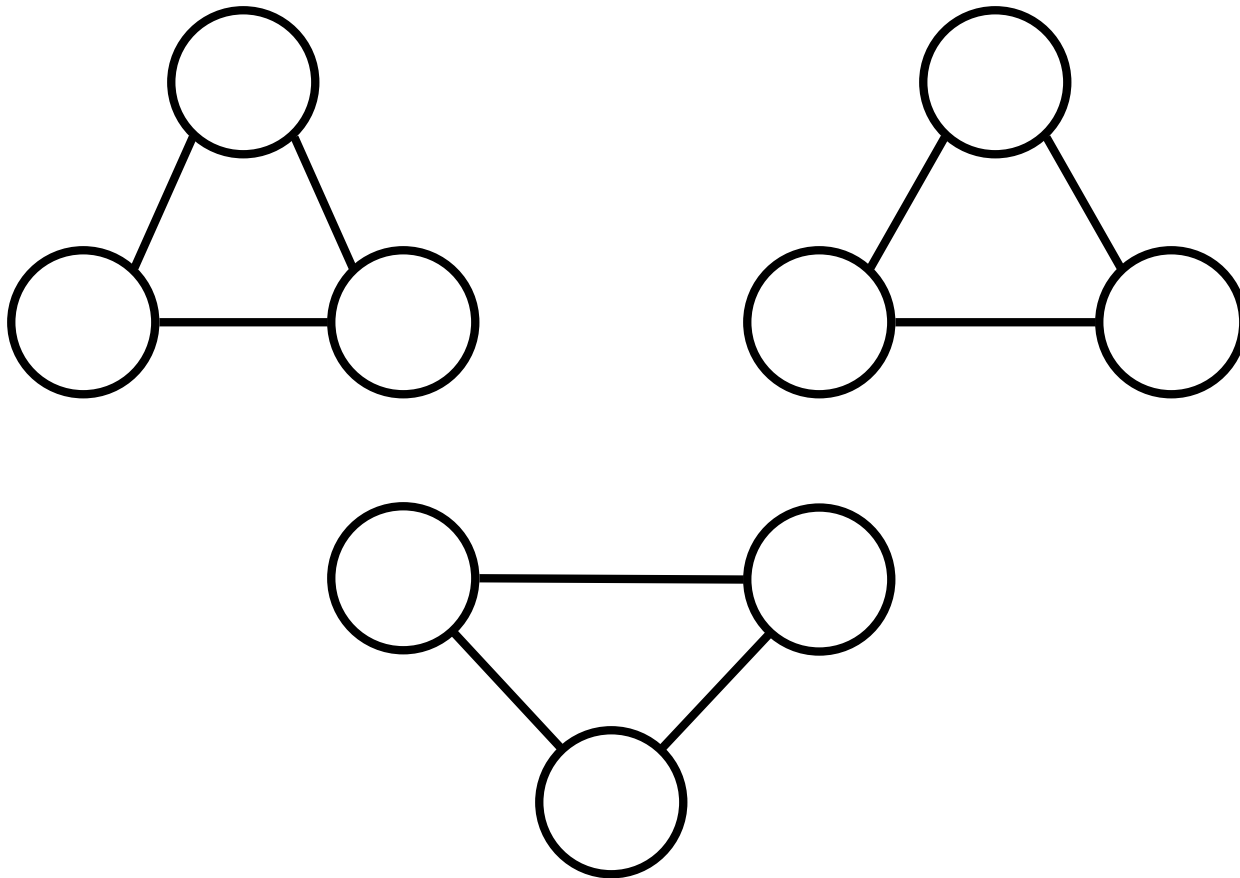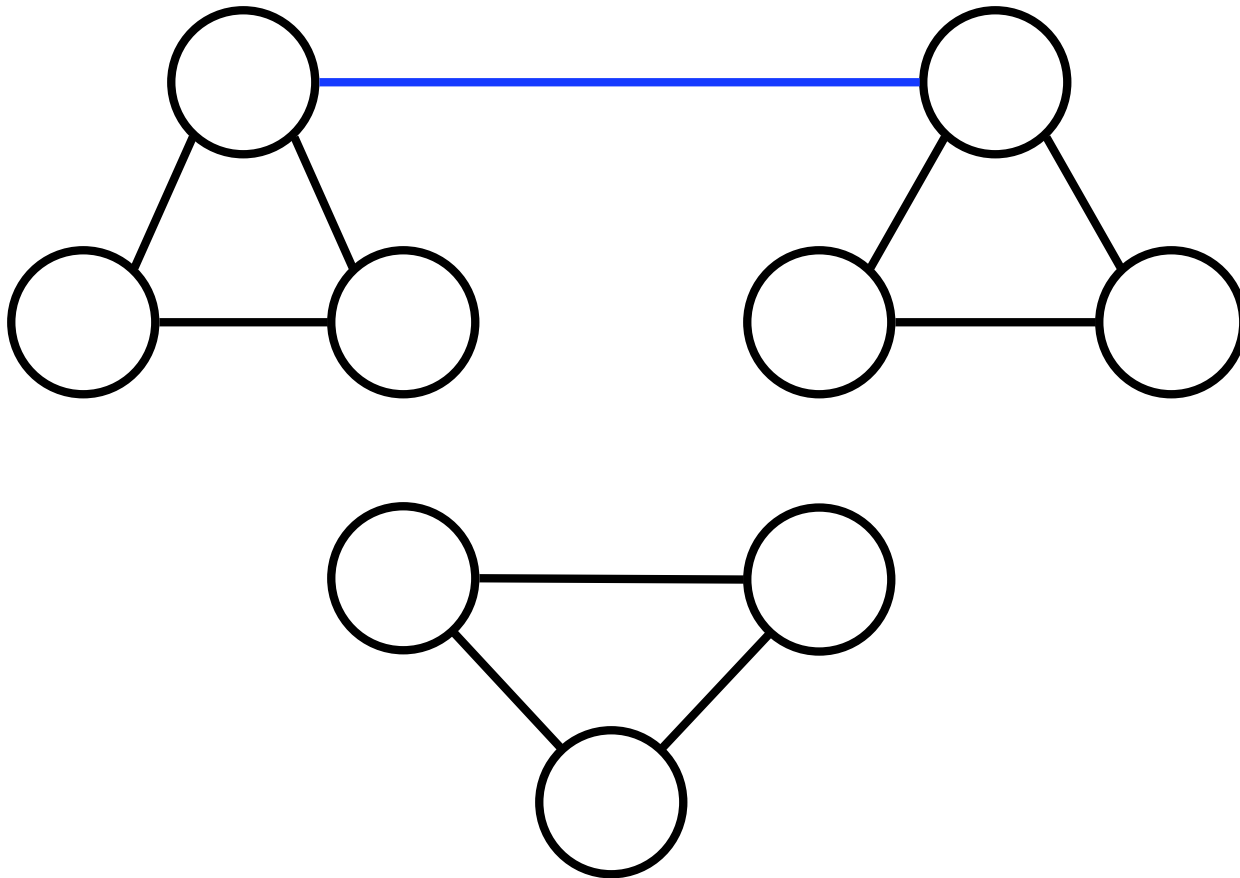
Example: Vertex cover of size ≤ 2.
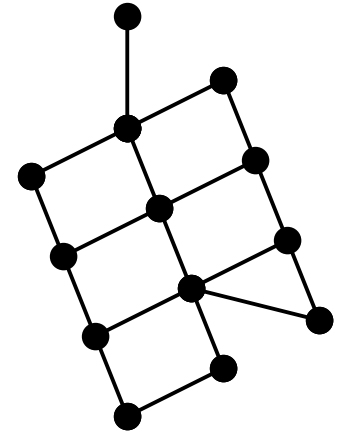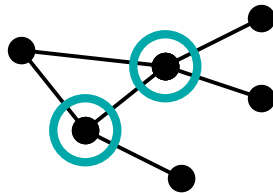
In NP? Exercise

3SAT ≤$_p$ VertexCover

3SAT ≤$_p$ VertexCover

3SAT ≤p VertexCover
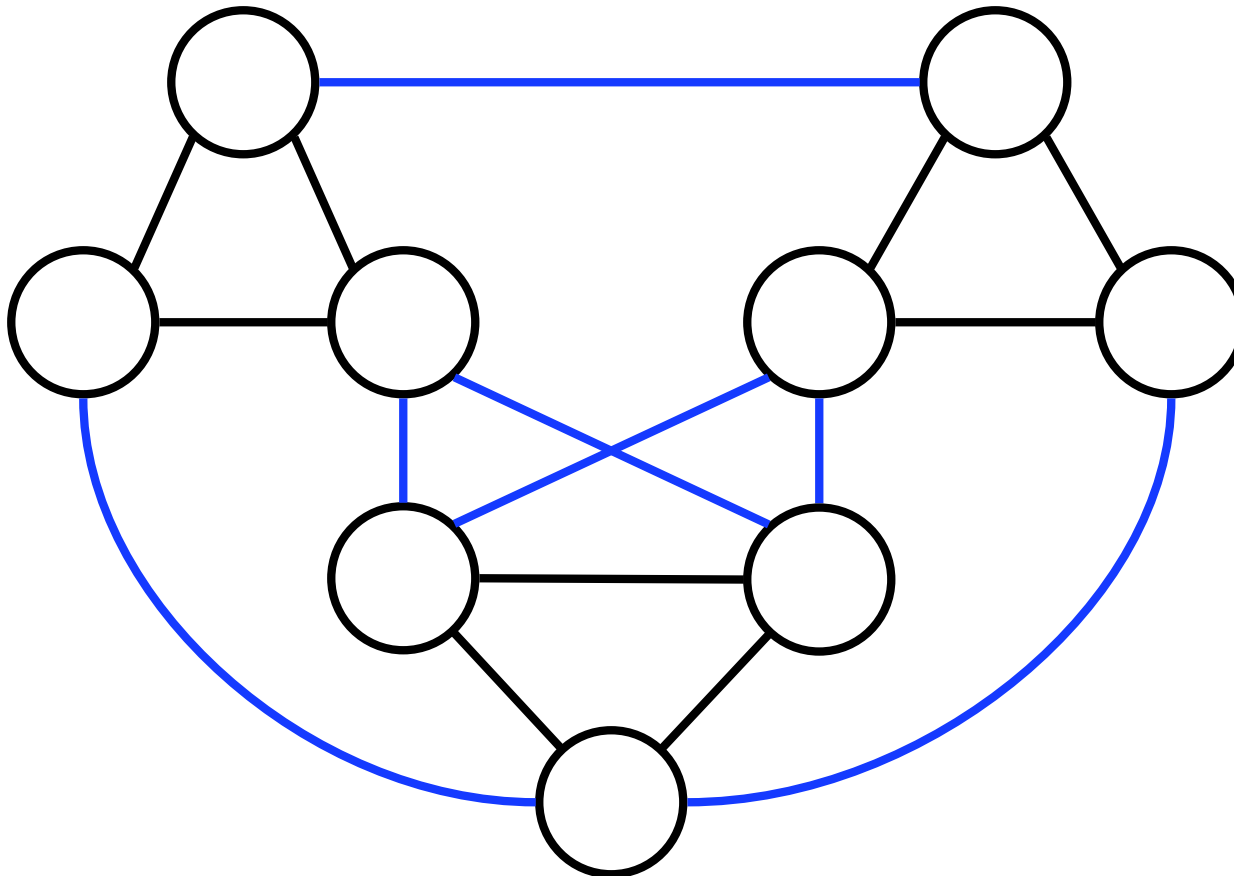
3SAT ≤$_p$ VertexCover

k=6

3SAT ≤$_p$ VertexCover

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

k=6
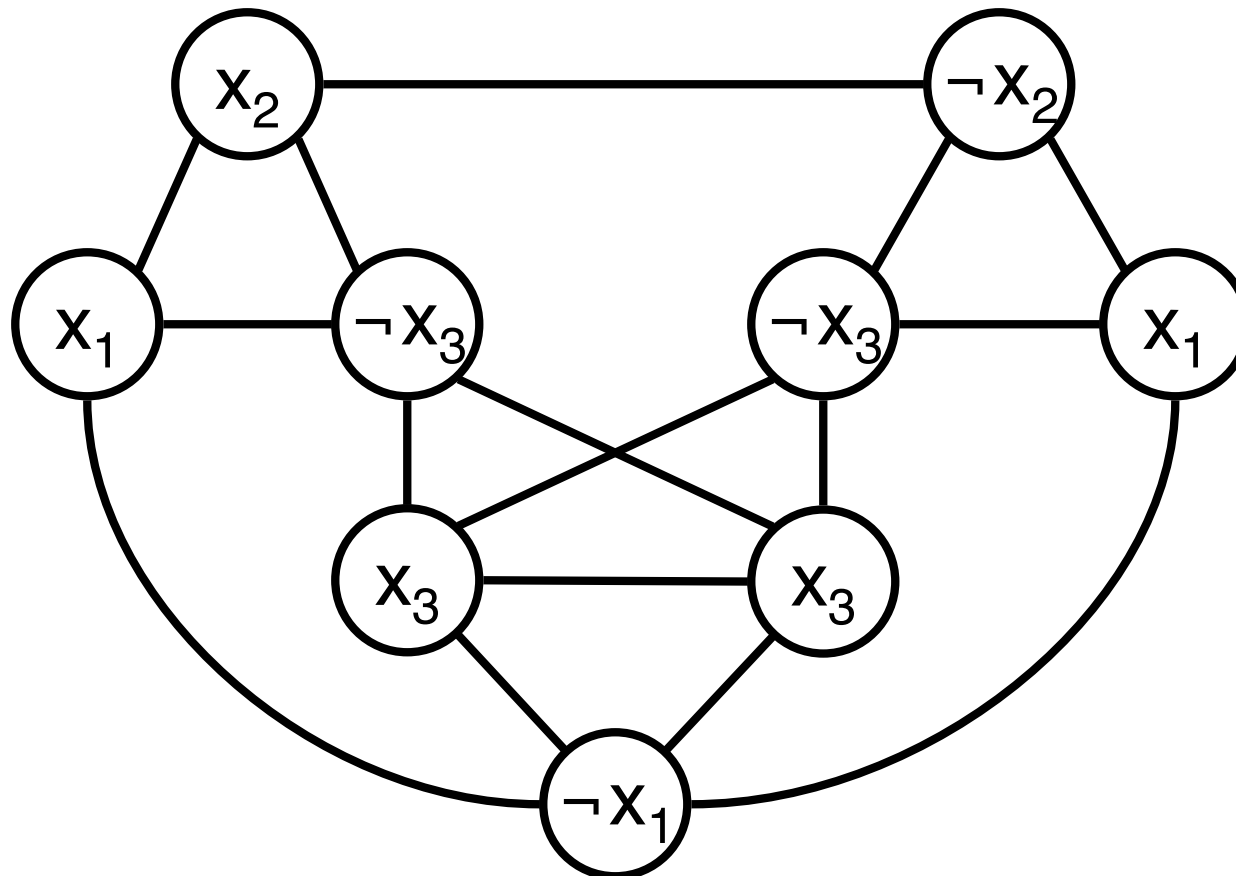
3SAT ≤$_p$ VertexCover

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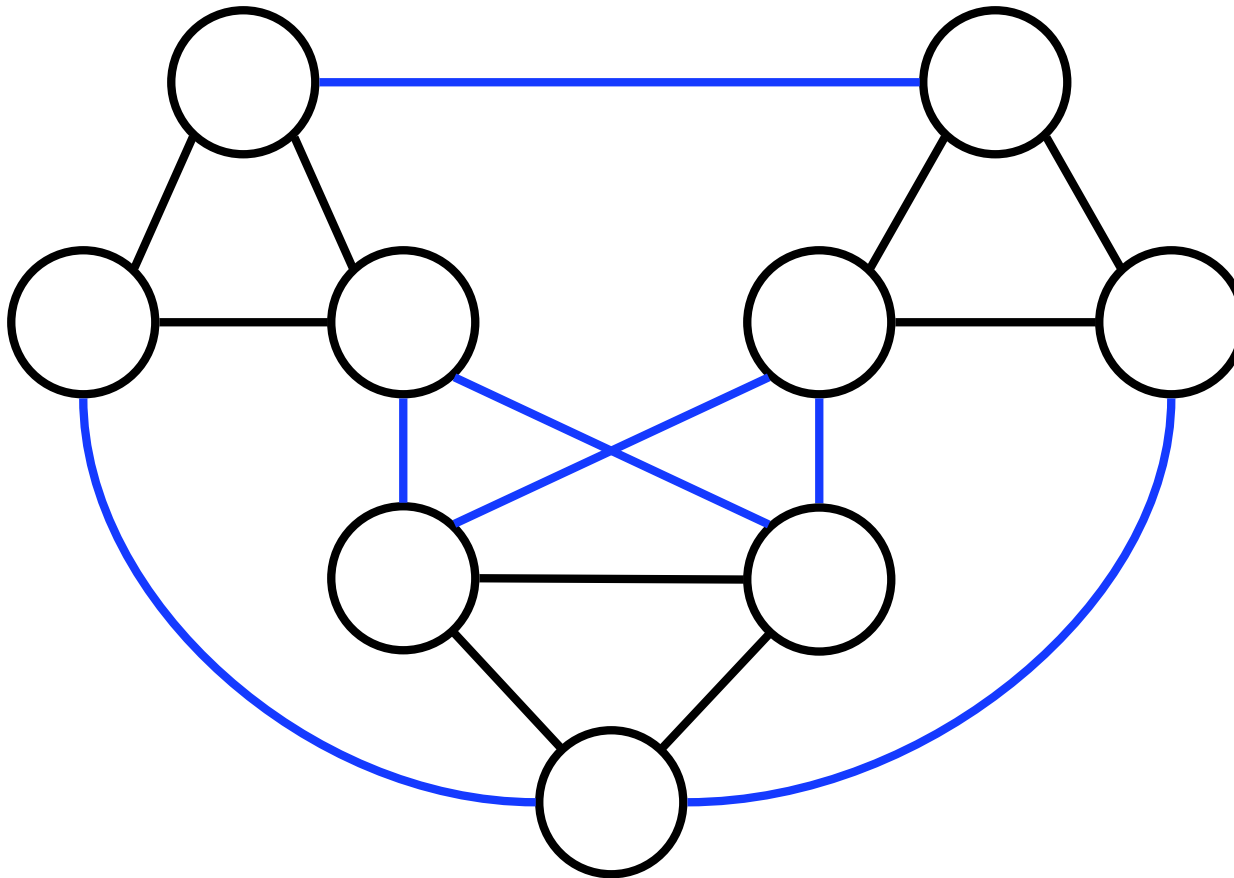- Formula: $c = c_1 \wedge c_2 \wedge \ldots \wedge c_q$
) =

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

3SAT ≤_p VertexCover

k=6

Correctness of "3SAT ≤$_p$ VertexCover"

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

<u>Correctness:</u>

 • Show f poly time computable: A key point is that graph size is polynomial in formula size; mapping basically straightforward.

 • Show c in 3-SAT iff f(c)=(G,k) in VertexCover:
(⟹) Given an assignment satisfying c, pick one true literal per clause. Add other 2 nodes of each triangle to cover. Show it is a cover: 2 per triangle cover triangle edges; only true literals (but perhaps not all true literals) uncovered, so at least one end of every (x, ¬x) edge is covered.
(⟸) Given a k-vertex cover in G, uncovered labels define a valid (perhaps partial) truth assignment since no (x, ¬x) pair uncovered. It satisfies c since there is one uncovered node in each clause triangle (else some other clause triangle has > 1 uncovered node, hence an uncovered edge.)

Utility of "3SAT ≤$_p$ VertexCover"

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

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

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

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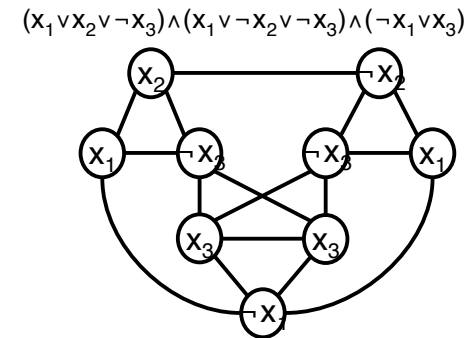
"3SAT ≤$_p$ VertexCover" Retrospective

Previous slide: two suppositions

Somewhat clumsy to have to state things that way.

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

Polynomial-Time Reductions

Definition: Let A and B be two problems.

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

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

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

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: $x \in A \iff f(x) \in B$

"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)

65

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

Algorithm to solve A
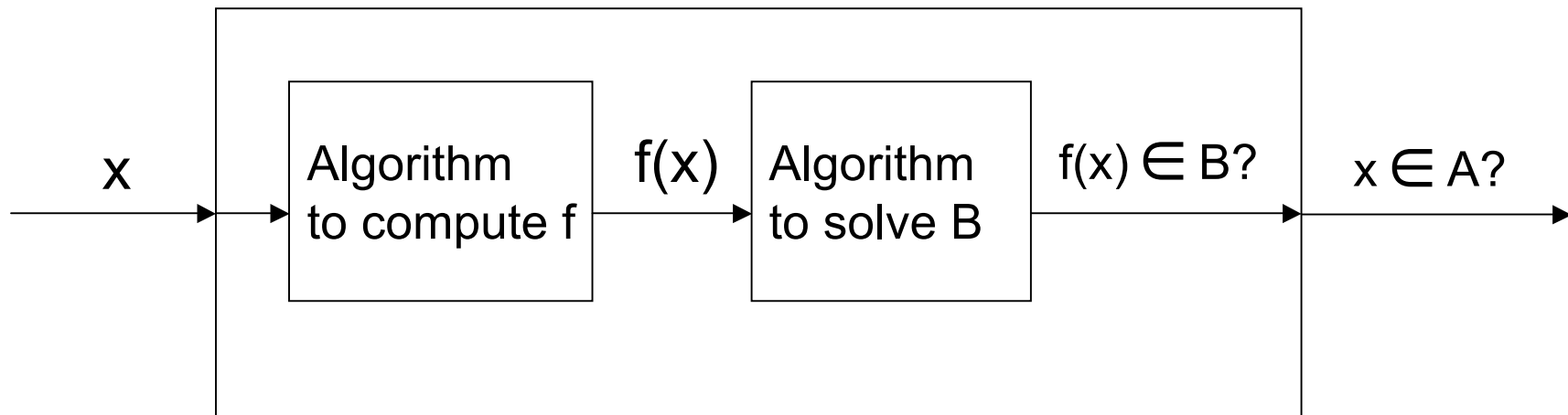
"*If A ≤$_p$ B, and we can solve B in polynomial time, then we can solve A in polynomial time also.*"

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

Definition of NP-Completeness

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

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

    (1) B belongs to NP, and

    (2) B is NP-hard.

Proving a problem is NP-complete

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

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

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

Why? Transitivity.

Re-stated Definition

Lemma: Problem B is NP-complete if:

(1) B belongs to NP, and

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

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

Usefulness of Transitivity

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

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

$$\forall L'' \in NP, \text{ we have } L'' \leq_p L'$$

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

Thus L is NP-hard.

Ex: VertexCover is NP-complete

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

3-SAT $\leq_p$ VertexCover

VertexCover is in NP (we showed this earlier)

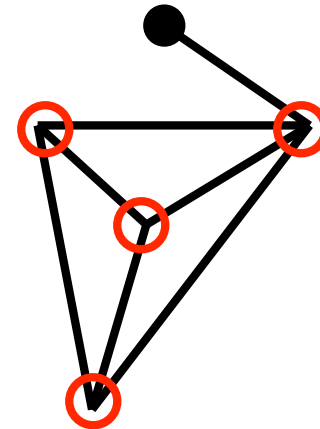Therefore VertexCover is also NP-complete

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

NP-complete problem: Clique

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

Output: True iff there is a subset C of V of size ≥ k such that all vertices in C are connected to all other vertices in C.
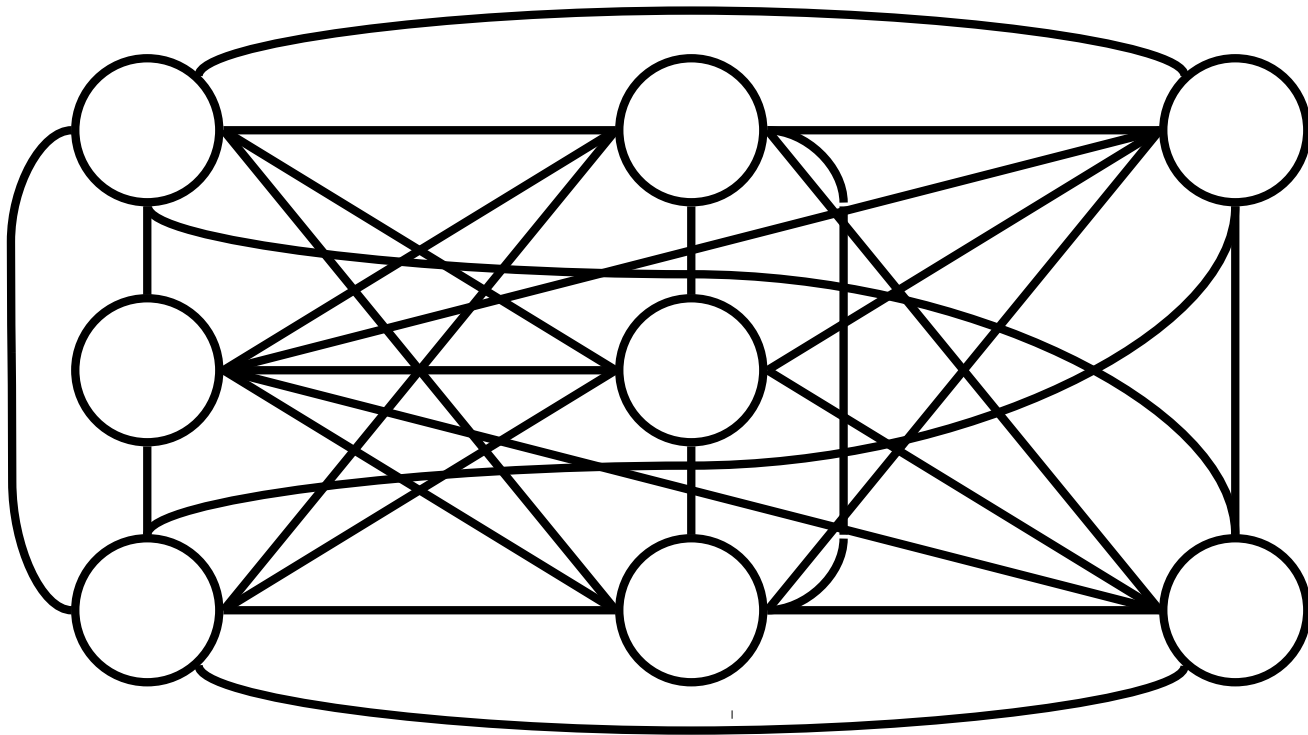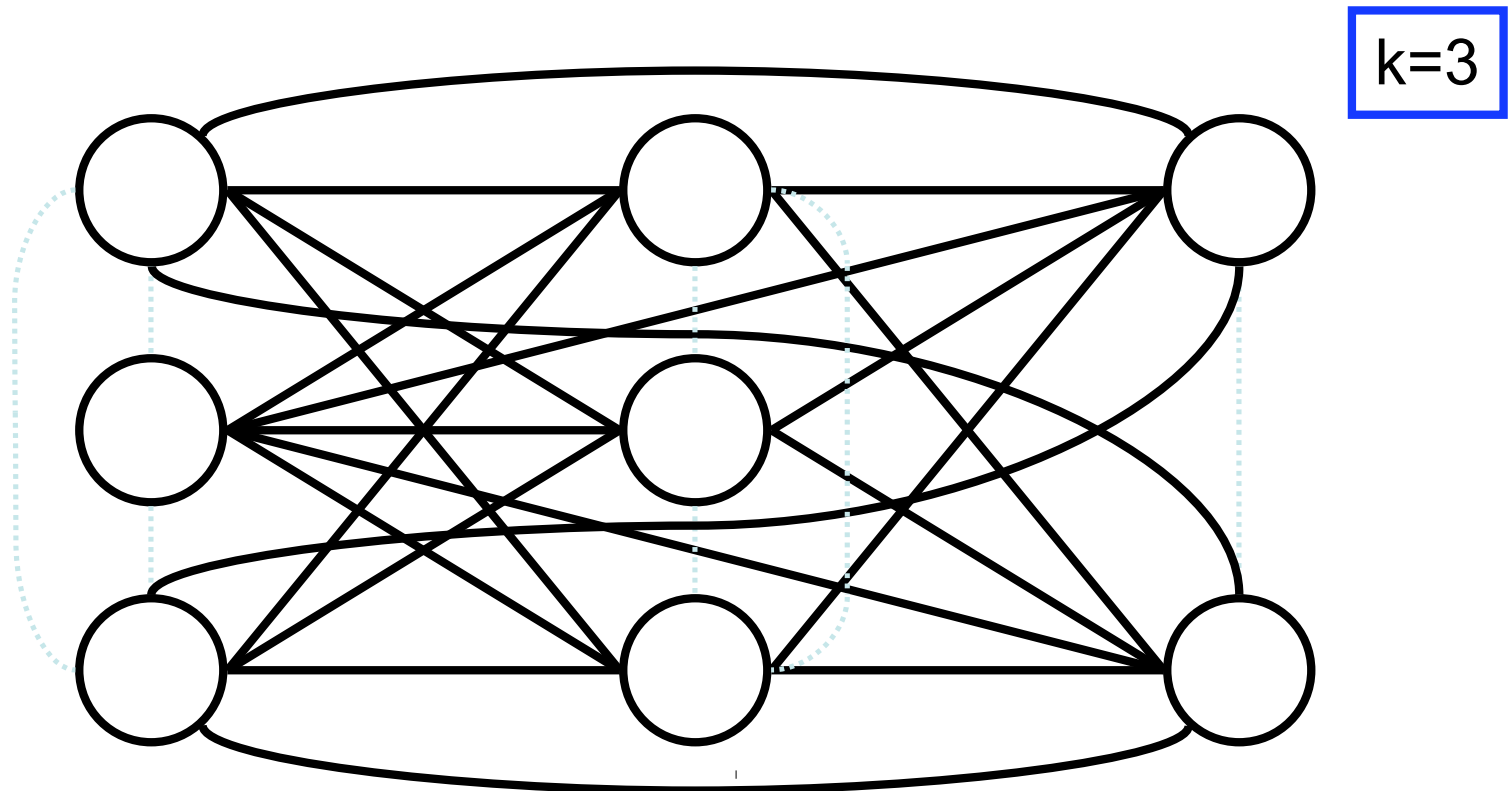
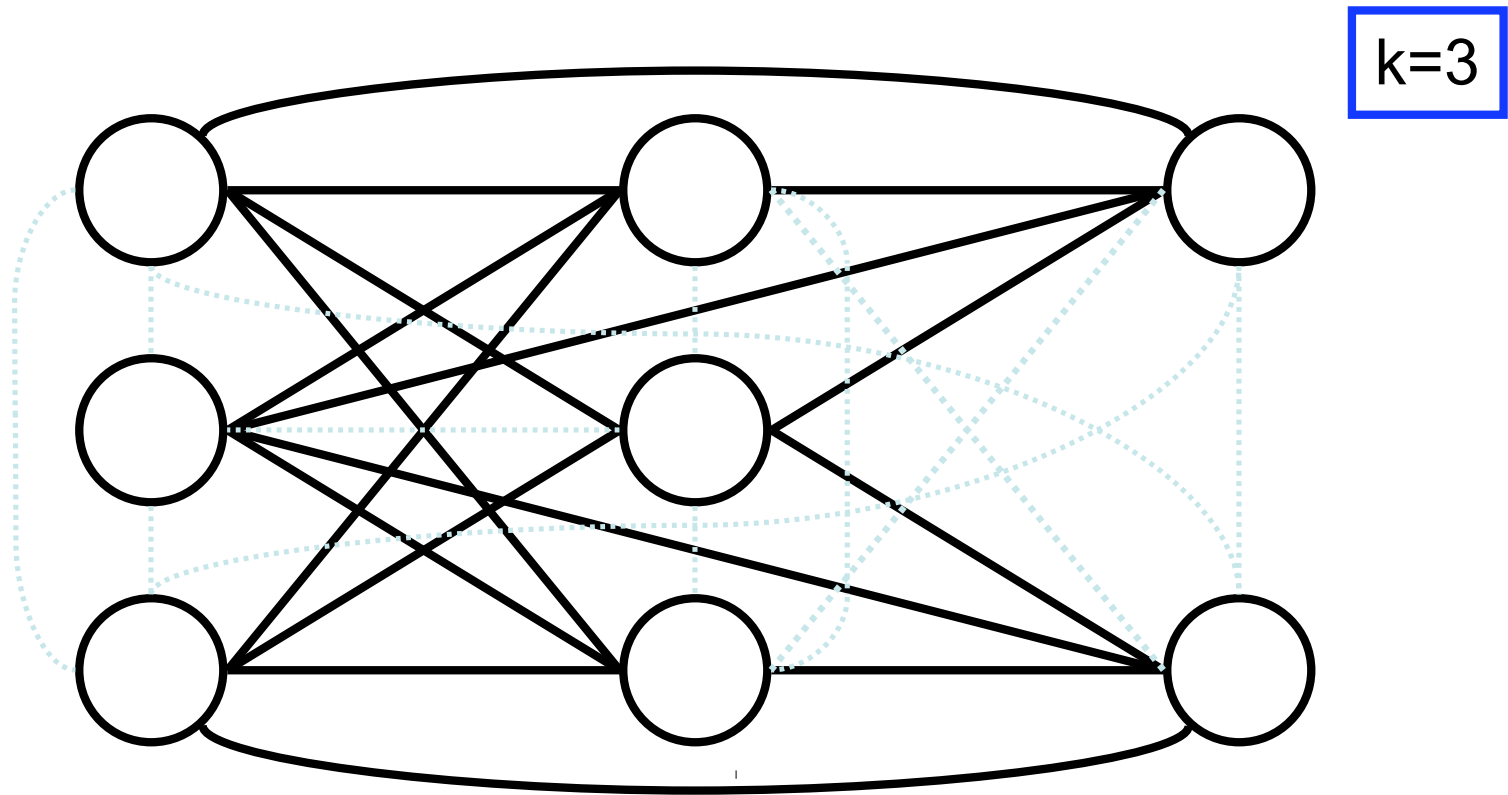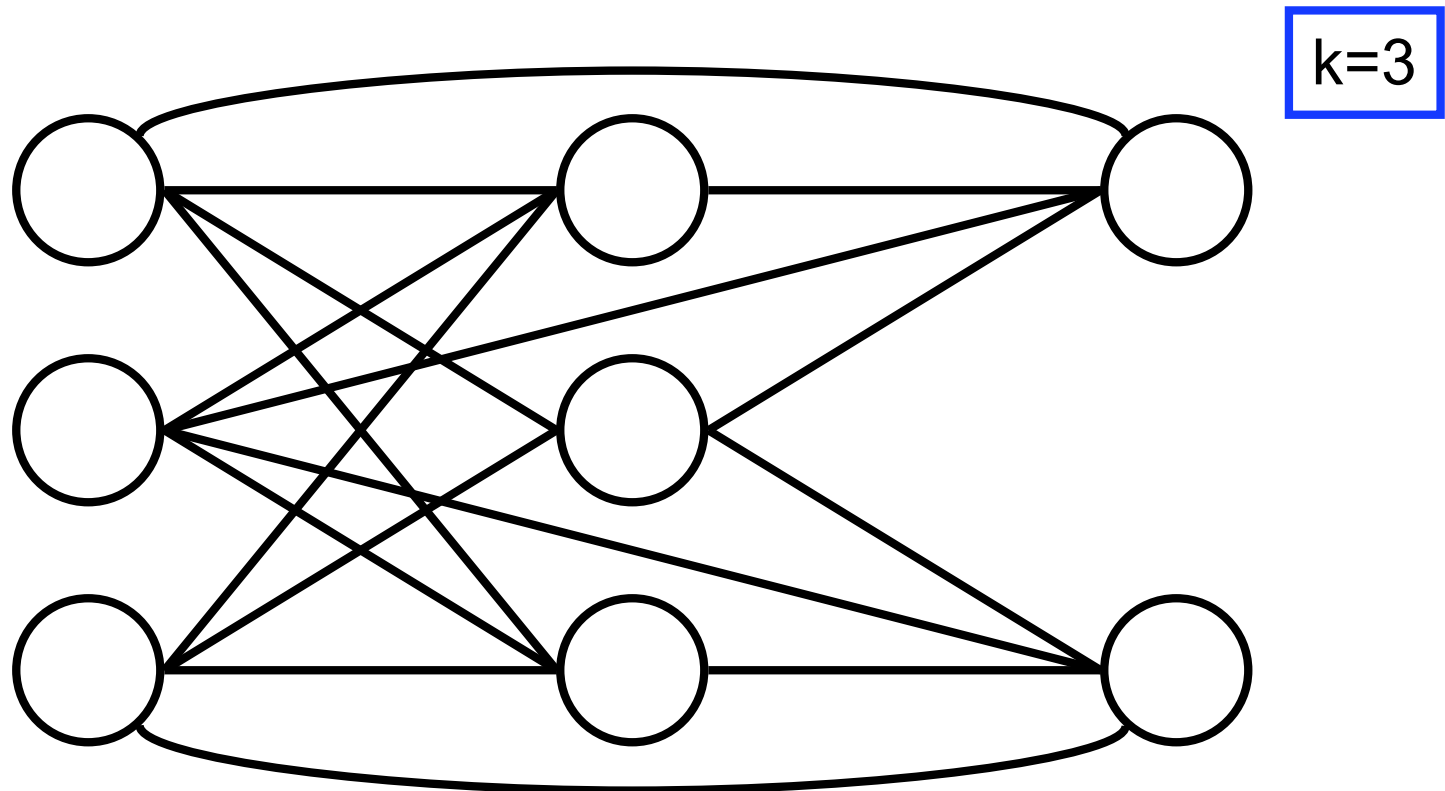Example: Clique of size ≥ 4

In NP? Exercise

3SAT ≤p Clique

k=3

74

3SAT ≤p Clique

k=3

75

3SAT ≤p Clique

k=3

3SAT ≤p Clique

k=3

3SAT ≤p Clique

$(x_1 \lor x_2 \lor \neg x_3) \land (x_1 \lor \neg x_2 \lor \neg x_3) \land (\neg x_1 \lor x_3)$
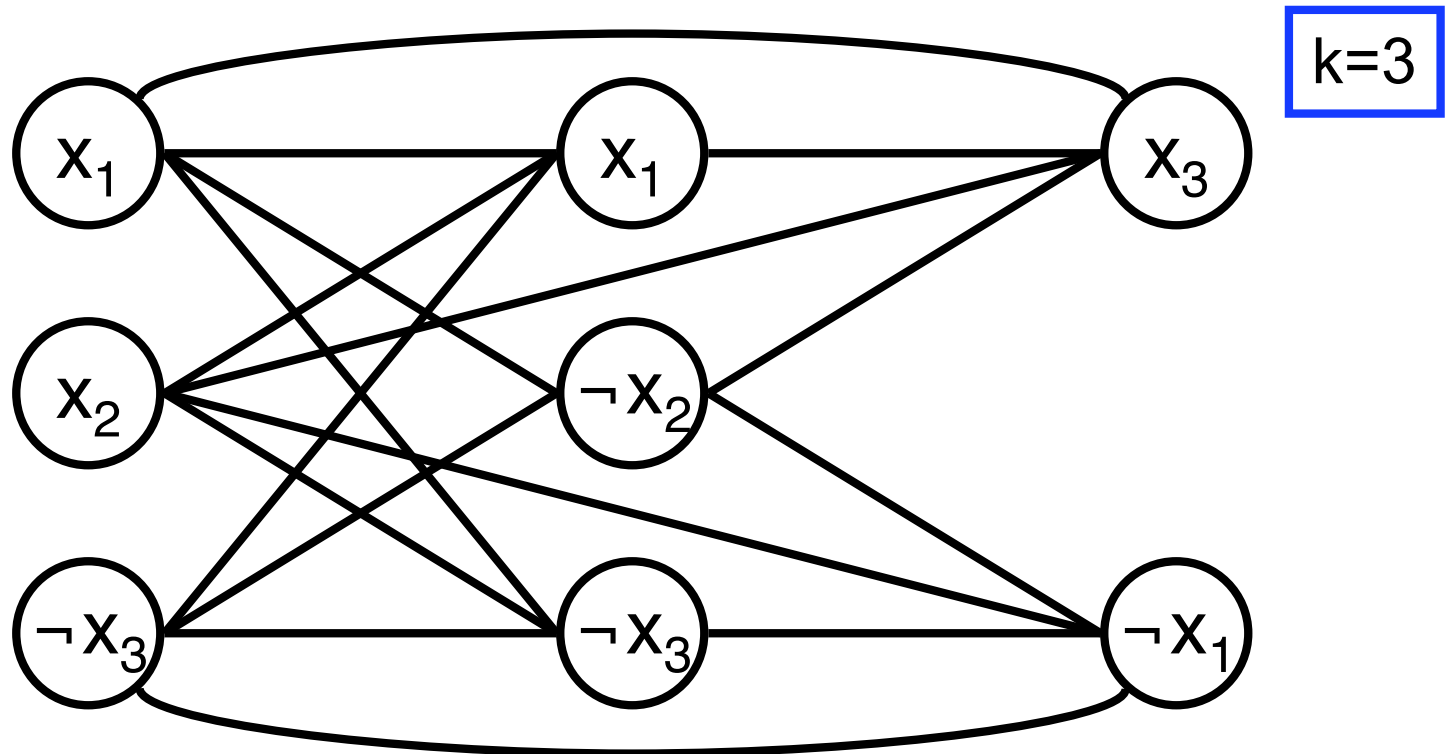
k=3

3SAT ≤p Clique

$$f \left(\begin{array}{l} \textbf{3-SAT Instance:} \\[4pt] \quad \text{– Variables: } x_1, x_2, \ldots \\ \quad \text{– Literals: } y_{i,j},\ 1 \le i \le q,\ 1 \le j \le 3 \\ \quad \text{– Clauses: } c_i = y_{i1} \vee y_{i2} \vee y_{i3},\ 1 \le l \le q \\ \quad \text{– Formula: } c = c_1 \wedge c_2 \wedge \ldots \wedge c_q \end{array} \right) = $$

Clique Instance:
  – $K = q$
  – $G = (V, E)$
  – $V = \{\ [i,j]\ |\ 1 \le i \le q,\ 1 \le j \le 3\ \}$
  – $E = \{\ (\ [i,j],\ [k,l]\ )\ |\ i \ne k \text{ and } y_{ij} \ne \neg y_{kl}\ \}$

Correctness of "3-SAT ≤p Clique"

Summary of reduction function f:
Given formula, make graph G with column of nodes per clause, one node per literal. Connect each to all nodes in other columns, except 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-clique; does not try to find satisfying assignment or clique.

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 Clique:
(⟹) Given an assignment satisfying c, pick one true literal per clause. Show corresponding nodes in G are k-clique.
(⟸) Given a k-clique in G, clique labels define a truth assignment; show it satisfies c. Note: literals in a clique are a valid truth assignment [no "(x, ¬x)" edges] & k nodes must be 1 per column, [no edges within columns].

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.
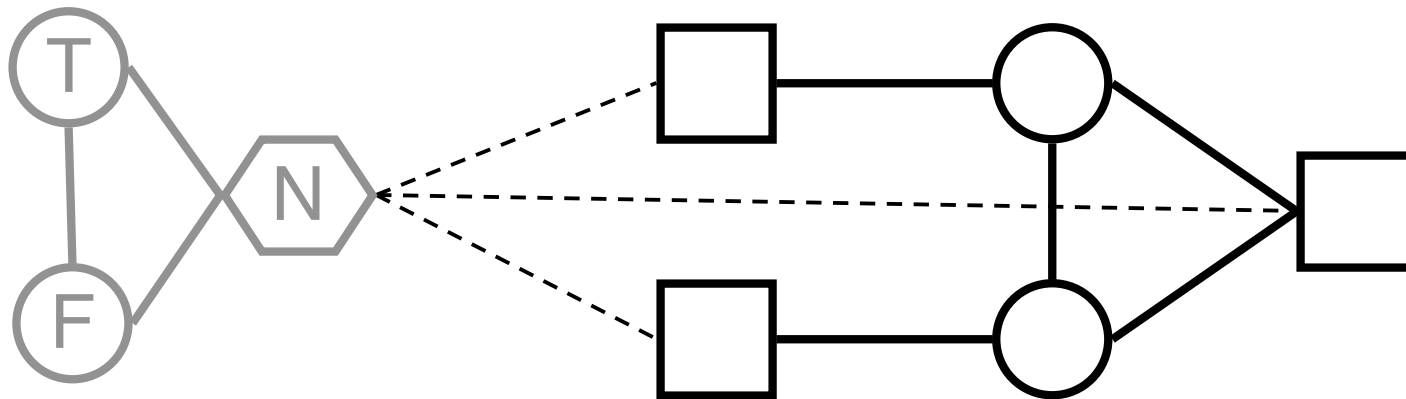
Example:
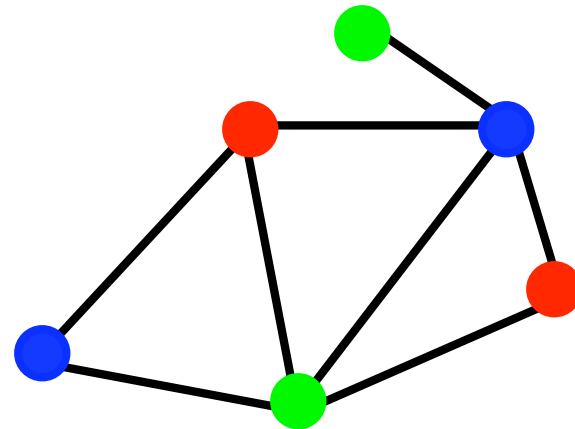
In NP? Exercise
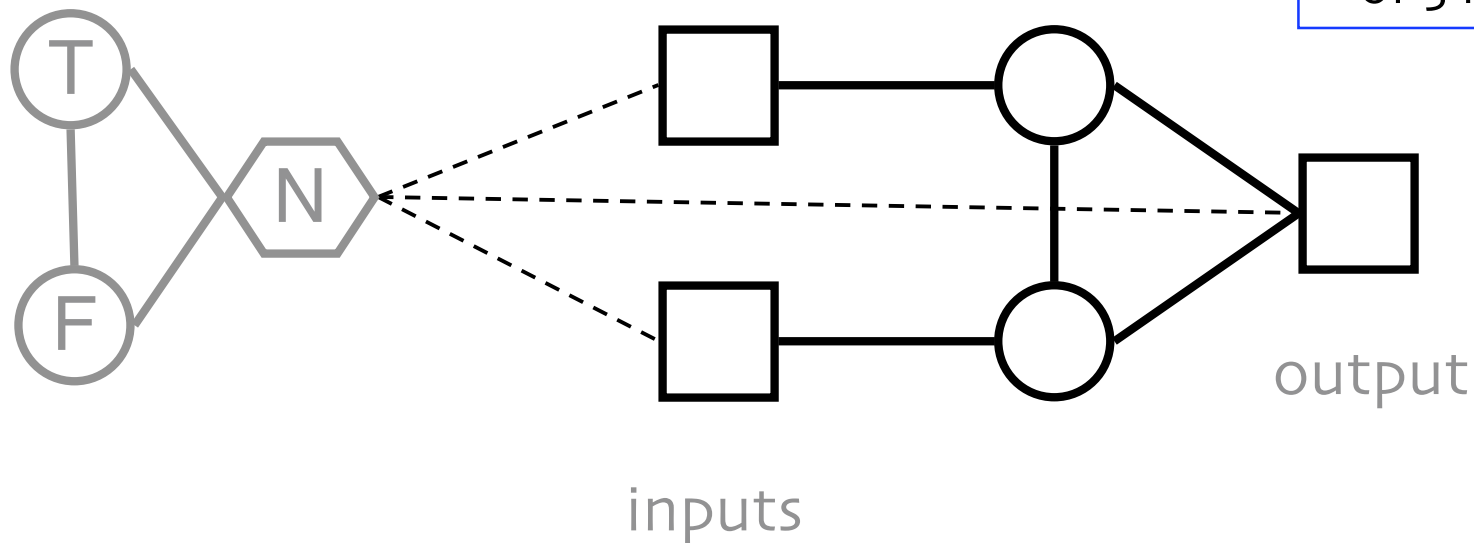
A 3-Coloring Gadget:

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

inputs

output

3SAT $\leq_p$ 3Color

$$f \left(\begin{array}{l} \text{3-SAT Instance:} \\ \quad - \text{Variables: } x_1, x_2, \ldots \\ \quad - \text{Literals: } y_{i,j}, \ 1 \leq i \leq q, \ 1 \leq j \leq 3 \\ \quad - \text{Clauses: } c_i = y_{i1} \vee y_{i2} \vee y_{i3}, \ 1 \leq l \leq q \\ \quad - \text{Formula: } c = c_1 \wedge c_2 \wedge \ldots \wedge c_q \end{array} \right) = $$
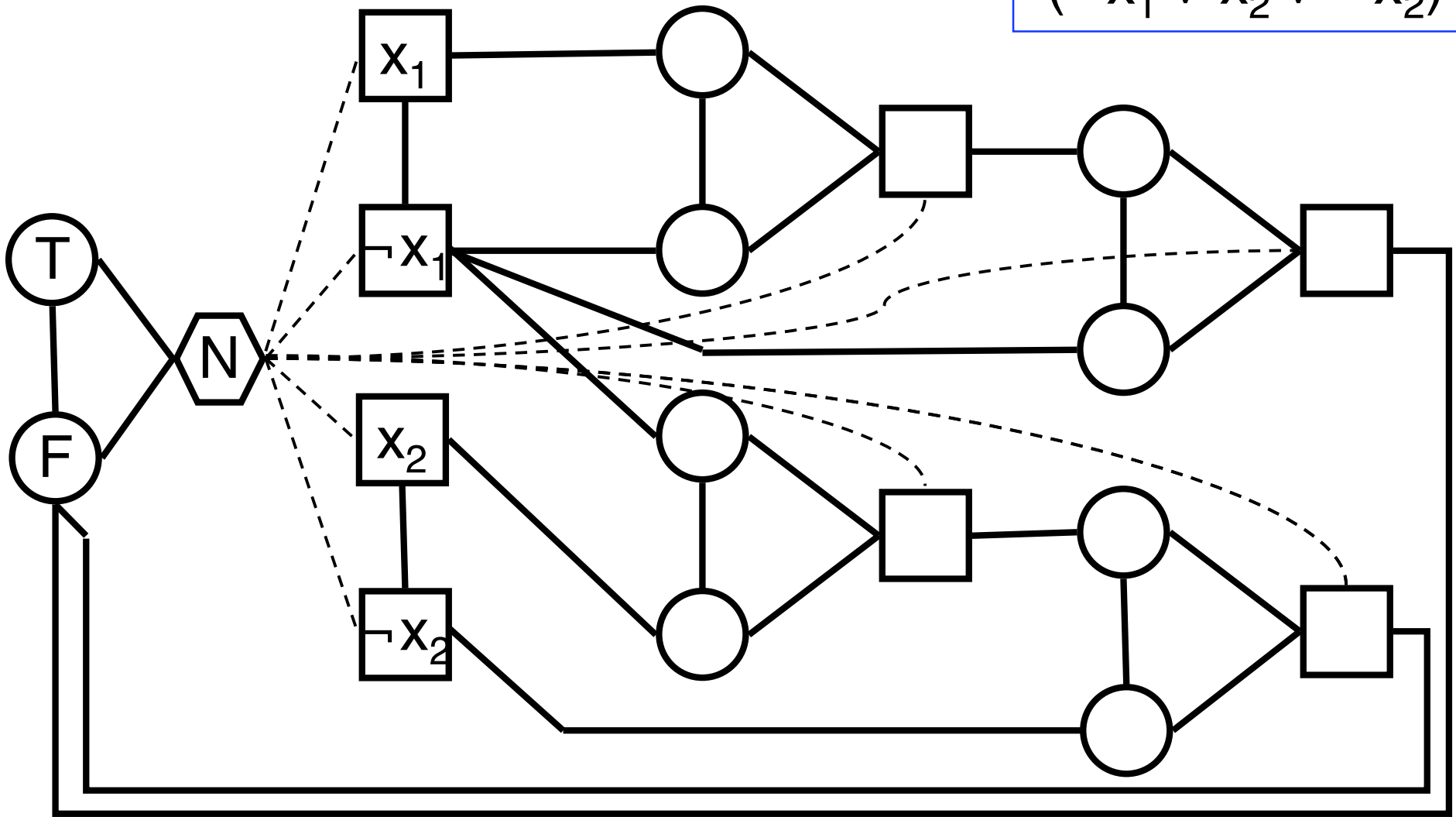
3Color Instance:
- $G = (V, E)$
- $6\,q + 2\,n + 3$ vertices
- $13\,q + 3\,n + 3$ edges
- (See Example for details)

3SAT ≤$_P$ 3Color Example

$$(x_1 \lor \lnot x_1 \lor \lnot x_1)$$
$$\land$$
$$(\lnot x_1 \lor x_2 \lor \lnot x_2)$$

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

Correctness of "3SAT ≤$_p$ 3Coloring"

<u>Summary of reduction function f:</u>
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.*

<u>Correctness:</u>

• 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
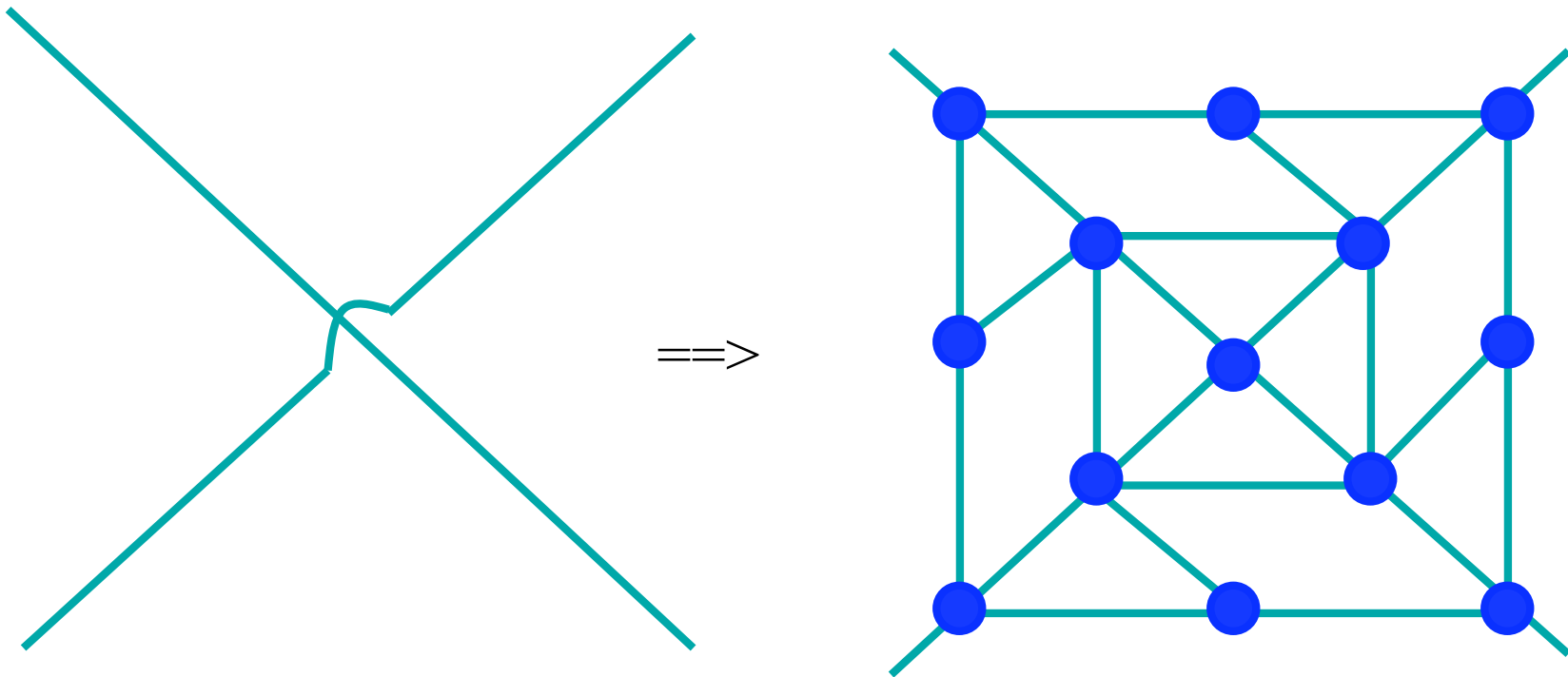
SAT is NP-complete

Rough idea of proof:

(1) SAT is in NP because we can guess a truth assignment and check that it satisfies the expression in polynomial time.
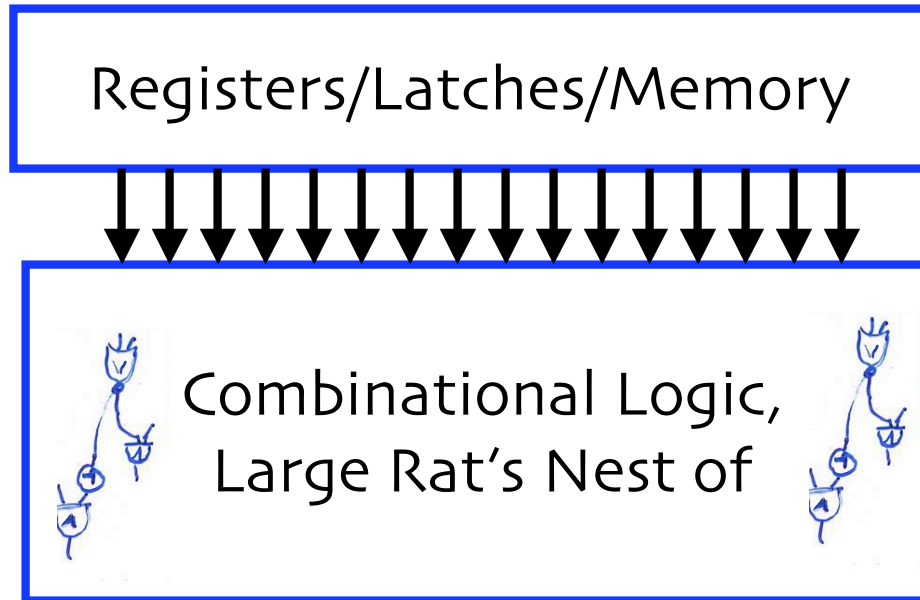
(2) SAT is NP-hard because …..

Cook proved it directly, but easier to see via an intermediate problem – Circuit-SAT

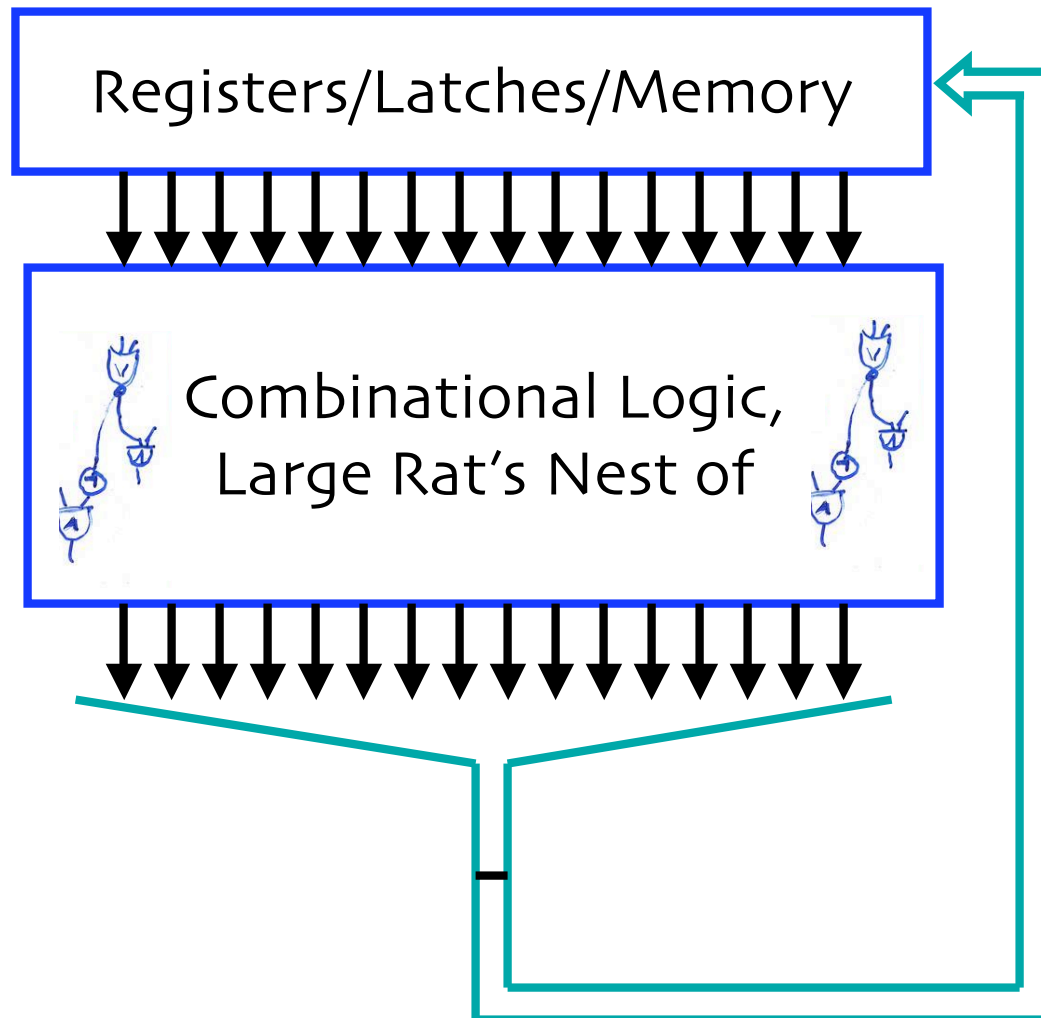# Detailed Logic Diagram, Intelorola Pentathlon® 6600

Detailed Logic Diagram, Intelorola Pentathlon® 6600

Registers/Latches/Memory
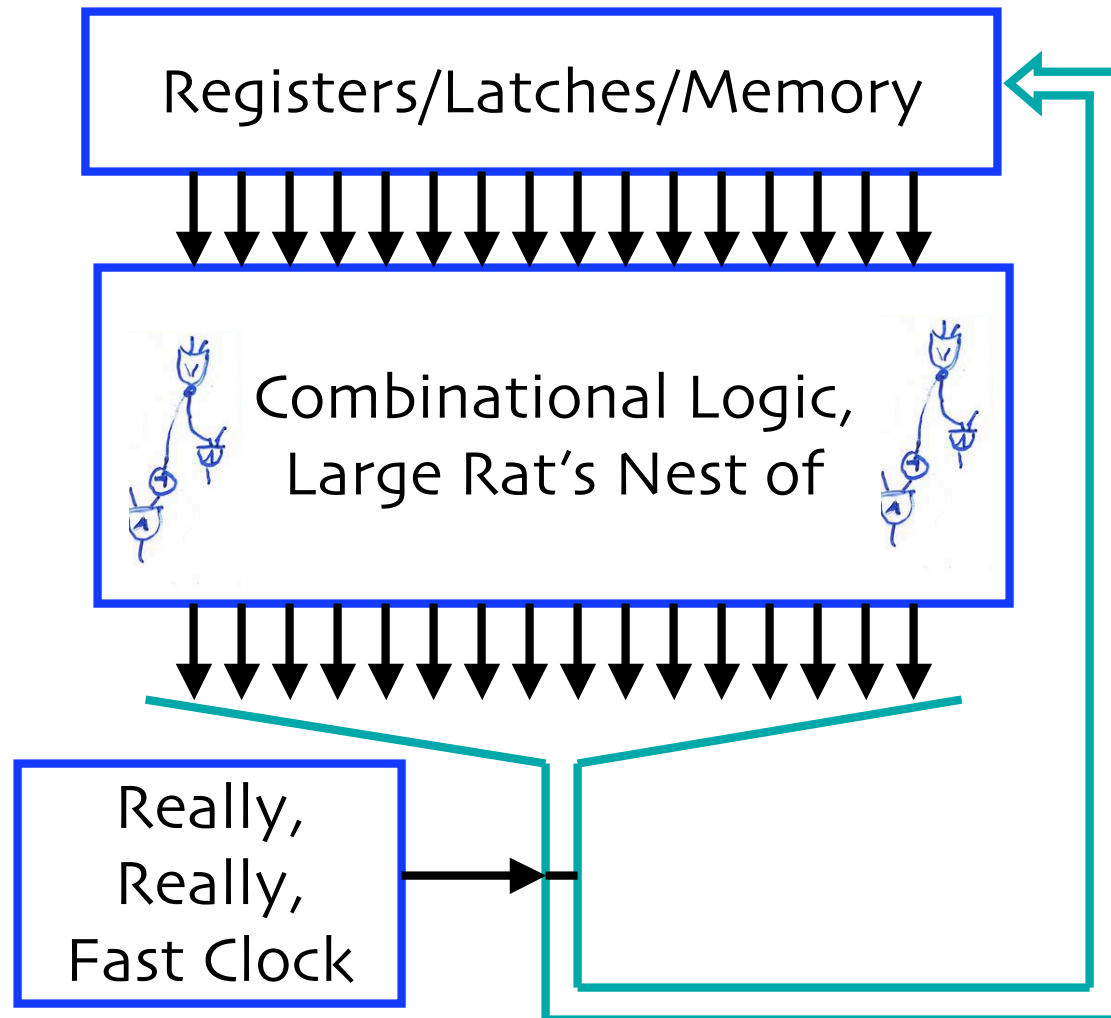
Detailed Logic Diagram, Intelorola Pentathlon® 6600

Registers/Latches/Memory

Combinational Logic,
Large Rat's Nest of

Detailed Logic Diagram, Intelorola Pentathlon® 6600

Registers/Latches/Memory

Combinational Logic,
Large Rat's Nest of

Detailed Logic Diagram, Intelorola Pentathlon® 6600

Registers/Latches/Memory

Combinational Logic, Large Rat's Nest of

Really, Really, Fast Clock

P Is Reducible To The Circuit Value Problem

Registers/Latches/Memory

Combinational Logic
Large Rat's Nest of

Really,
Really,
Fast Clock

T

Input

Combinational Logic

Combinational Logic

Combinational Logic

T

Accept?

95
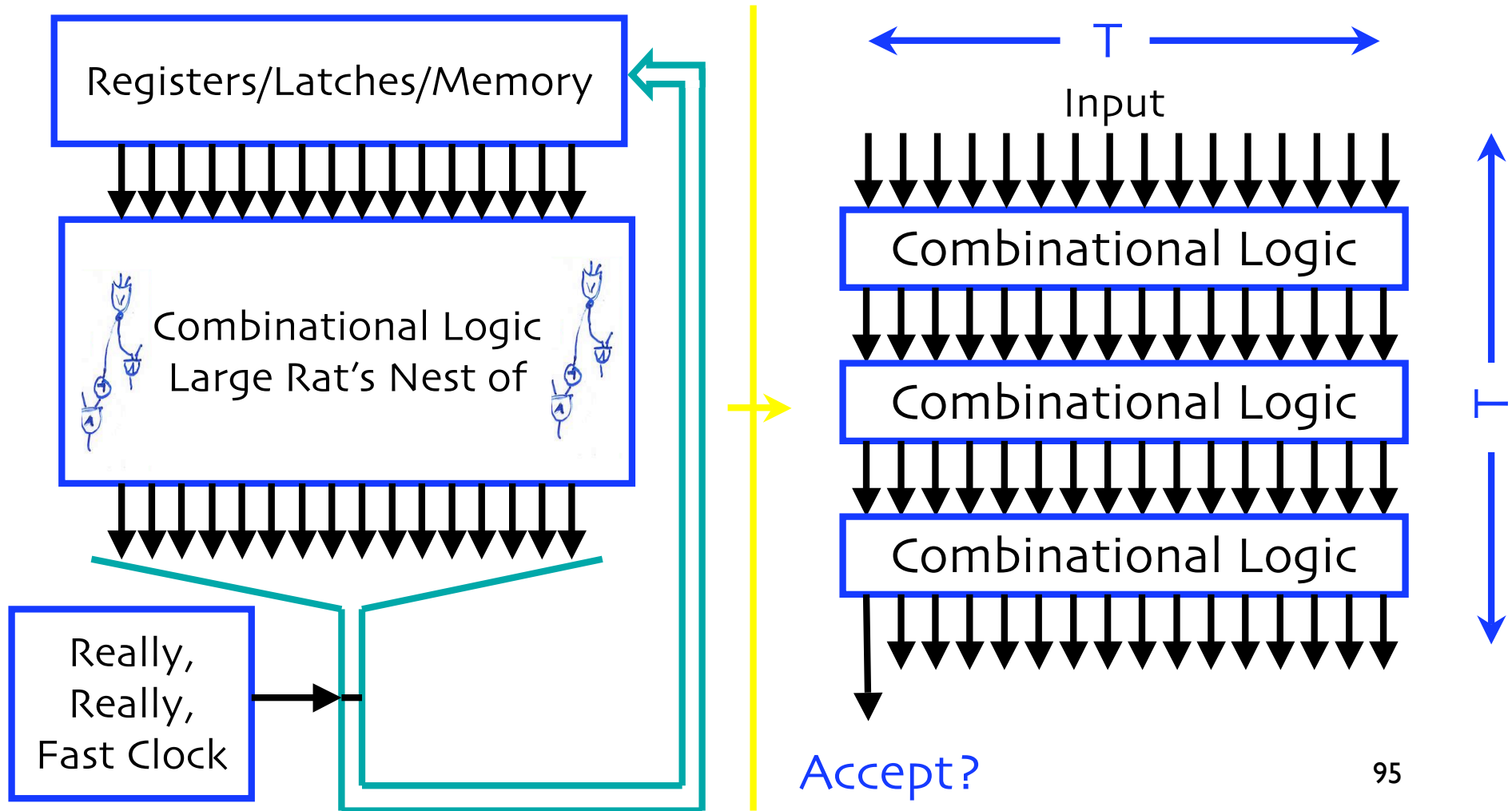
NP Is Reducible To The Circuit Satisfiability Problem

ND bit

Registers/Latches/Memory

Combinational Logic, Large Rat's Nest of

Really, Really Fast Clock

T

"Input"

ND bits

Combinational Logic

Combinational Logic

Combinational Logic

T

Accept?
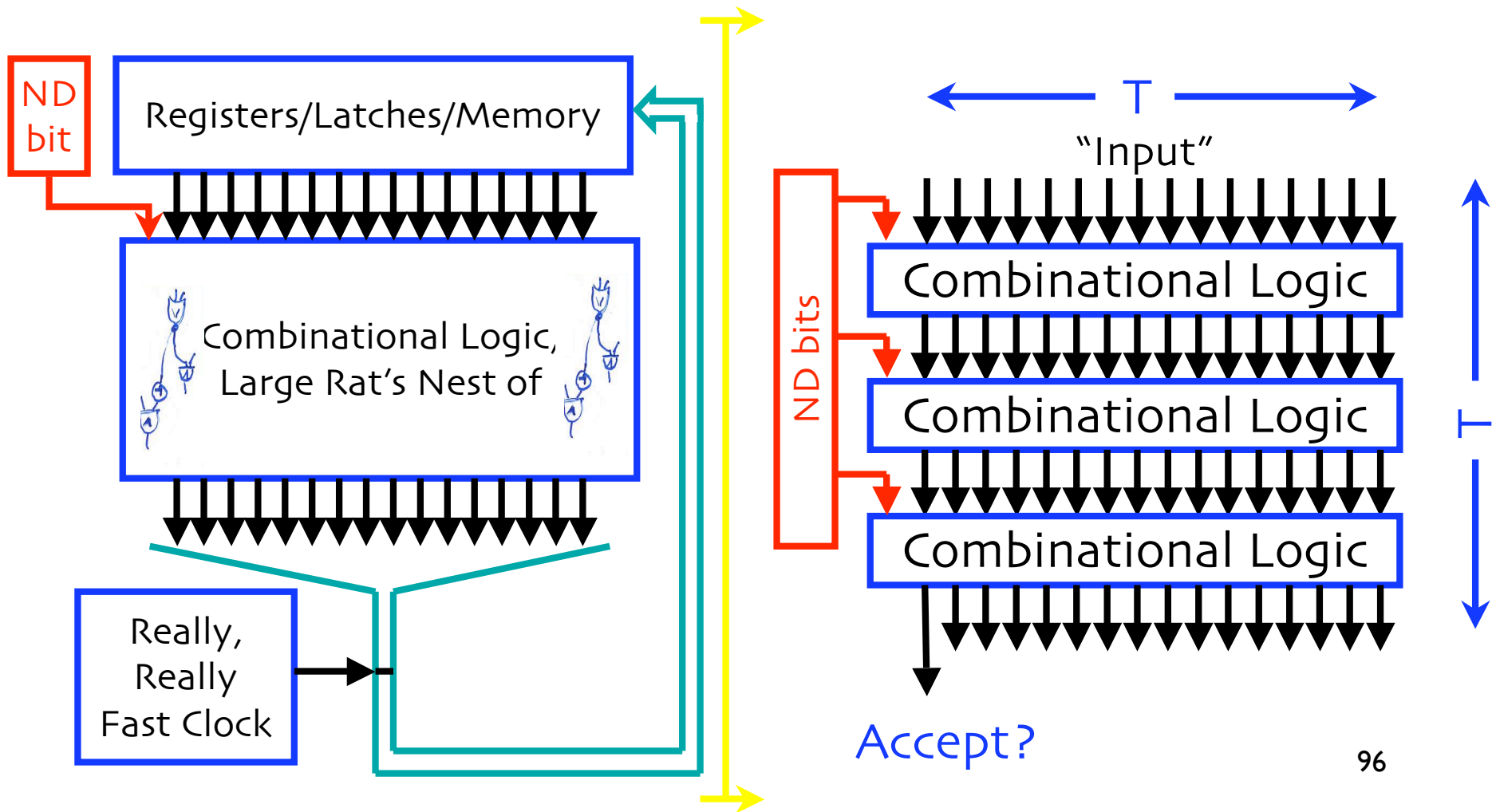
To Prove SAT is NP-complete

Show it's in NP: Exercise
(Hint: what's an easy-to-check certificate of satisfiability?)

Pick a known NP-complete problem
& reduce it to SAT
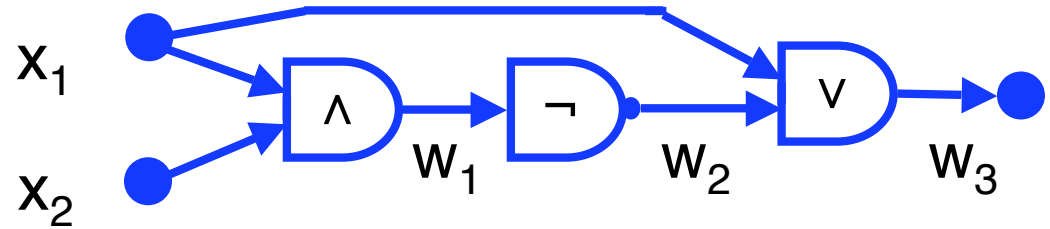
Gee, How about Circuit-SAT?
Good idea; it's the only NP-complete problem we have so far

What we need:
a fast, mechanical way to "simulate" a circuit by a formula

Circuit-SAT $\leq_p$ 3-SAT

x_1

x_2

$(w_1 \Leftrightarrow (x_1 \wedge x_2)) \wedge (w_2 \Leftrightarrow (\neg w_1)) \wedge (w_3 \Leftrightarrow (w_2 \vee x_1)) \wedge w_3$

Replace with 3-CNF Equivalent:

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	$\leftarrow x_1 \wedge \neg x_2 \wedge \neg w_1$
1	1	1	1	0	

$f(\ \text{◦}\underset{\wedge}{\triangleright}\underset{\neg}{\triangleright}\underset{\vee}{\triangleright}\text{◦}\ ) = (x_1 \vee x_2 \vee \neg w_1) \wedge (x_1 \vee \neg x_2 \vee \neg w_1) \wedge (\neg x_1 \vee x_2 \vee \neg w_1) \wedge (\neg x_1 \vee x_2 \vee w_1) \dots$

$\neg$ clause → Truth Table → DNF $\xrightarrow{\text{DeMorgan}}$ CNF

98

Correctness of "Circuit-SAT ≤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: f does not know whether circuit or formula are satisfiable or not; does not try to find satisfying assignment.

Correctness:

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

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).

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, does not show it's hard.)

Sloooow Reductions
"Find a satisfying assignment, then output…"

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

Coping with NP-Completeness

Is your real problem a special subcase?

> E.g. 3-SAT is NP-complete, but 2-SAT is not;
>
> Ditto 3- vs 2-coloring
>
> E.g. maybe you only need planar graphs, or degree 3 graphs, or …

Guaranteed approximation good enough?

> E.g. Euclidean TSP within 1.5 * Opt in poly time

Clever exhaustive search may be fast enough in practice, e.g. Backtrack, Branch & Bound, pruning

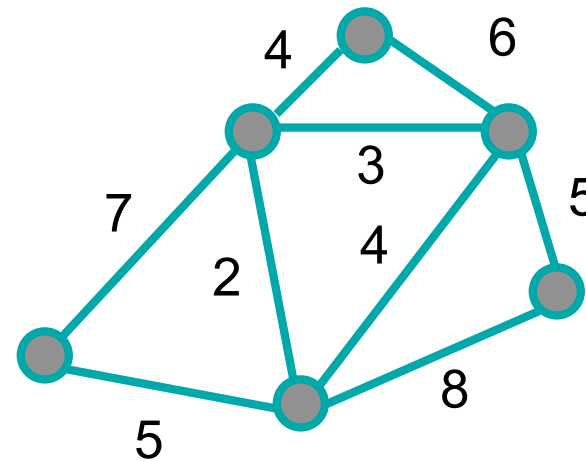Heuristics – usually a good approximation and/or usually fast

NP-complete problem: TSP

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

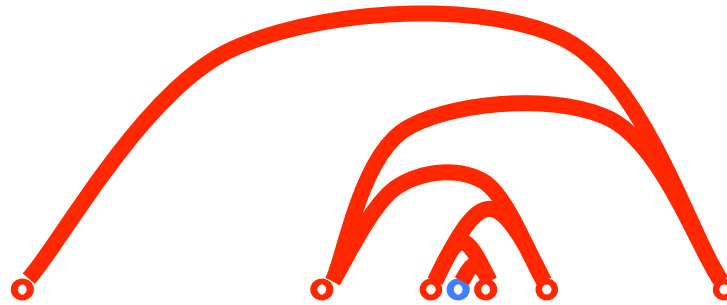Output: YES iff there is a simple cycle in G passing through all vertices (once), with total cost ≤ b.

Example:

b = 34

TSP - Nearest Neighbor Heuristic

Recall NN Heuristic

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

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

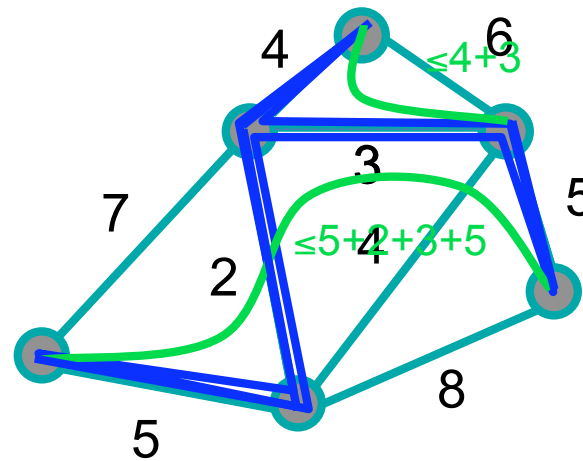(above example is not that bad)

2x Approximation to 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
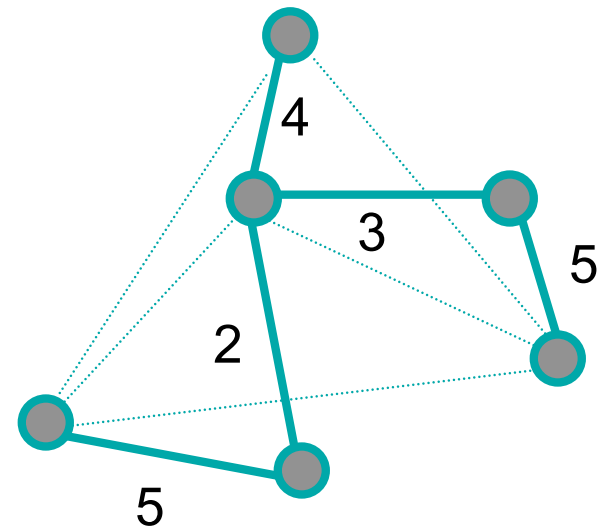
1.5x Approximation to EuclideanTSP

Find MST

Find min cost matching among odd-degree tree vertices

Cost of matching ≤ TSP/2

Find Euler Tour

Shortcut

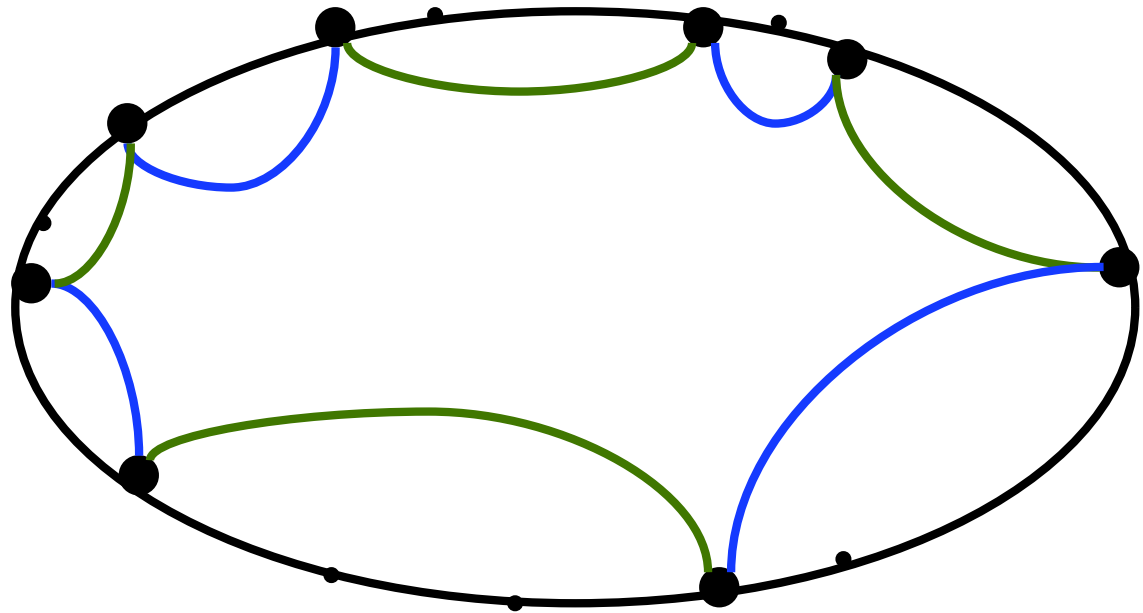Shortcut ≤ ET ≤ MST + TSP/2 < 1.5* TSP

107

Matching ≤ TSP/2

Oval=TSP

Big dots=
odd tree
nodes

Blue, Green
= 2 matchings

Blue + Green ≤ TSP
(by triangle inequality)

So min matching ≤ TSP/2

A problem NOT in NP; A bogus "proof" to the contrary

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

NON Theorem: EEXP in NP

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

"Proof" II: Hint = a bit; accept iff it's 1

Summary

Big-O   – good

P   – good

Exp   – bad

Exp, but hints help? NP

NP-hard, NP-complete – bad (I bet)

To show NP-complete – reductions

NP-complete = hopeless? – no, but you need to lower your expectations: heuristics & approximations.

Worse…

Exp

NP-C

NP

P