CSE 417
Introduction to Algorithms
Winter 2005

NP-Completeness

(Chapter 6)

Some Algebra Problems (Algorithmic)

Given positive integers a, b, c

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

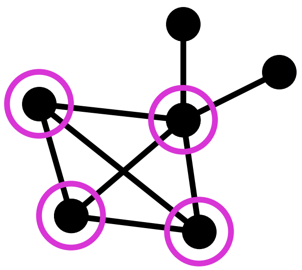
Question 2: does there exist a positive integer x such that $ax^2 + bx = c$?

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

The Clique Problem

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

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



Solving The Clique Problem

- A simple way:
  - Systematically list all possible sets of exactly k nodes
  - For each such set, check whether all pairs are neighbors

- A general approach for problems like this: Backtracking

Backtracking (abstractly)

- Want: a vector $(a_1, a_2, \dots, a_q)$ satisfying some property P, e.g. "$a_1..a_q$ is a q-clique".

  BT(A,j)

  if A, j satisfies P, report it

  else

  $j = j+1$

  let S_j be the set of "candidates" for slot j;

  for each a_j in S_j

  BT(A . a_j, j)

Top Level: Call BT(empty,0); report "no solution" if it found none.

Backtracking for k-Clique, I

a_i's are distinct vertices, in order.

- Want: a vector $(a_1, a_2, \ldots, a_q)$ satisfying some property P, e.g. "$a_1..a_q$ is a q-clique".

  BT(A,j)

      if A, j satisfies P, report it

      else

          j = j+1

          let S_j be the set of "candidates" for slot j

          for each a_j in S_j

              BT(A . a_j, j)

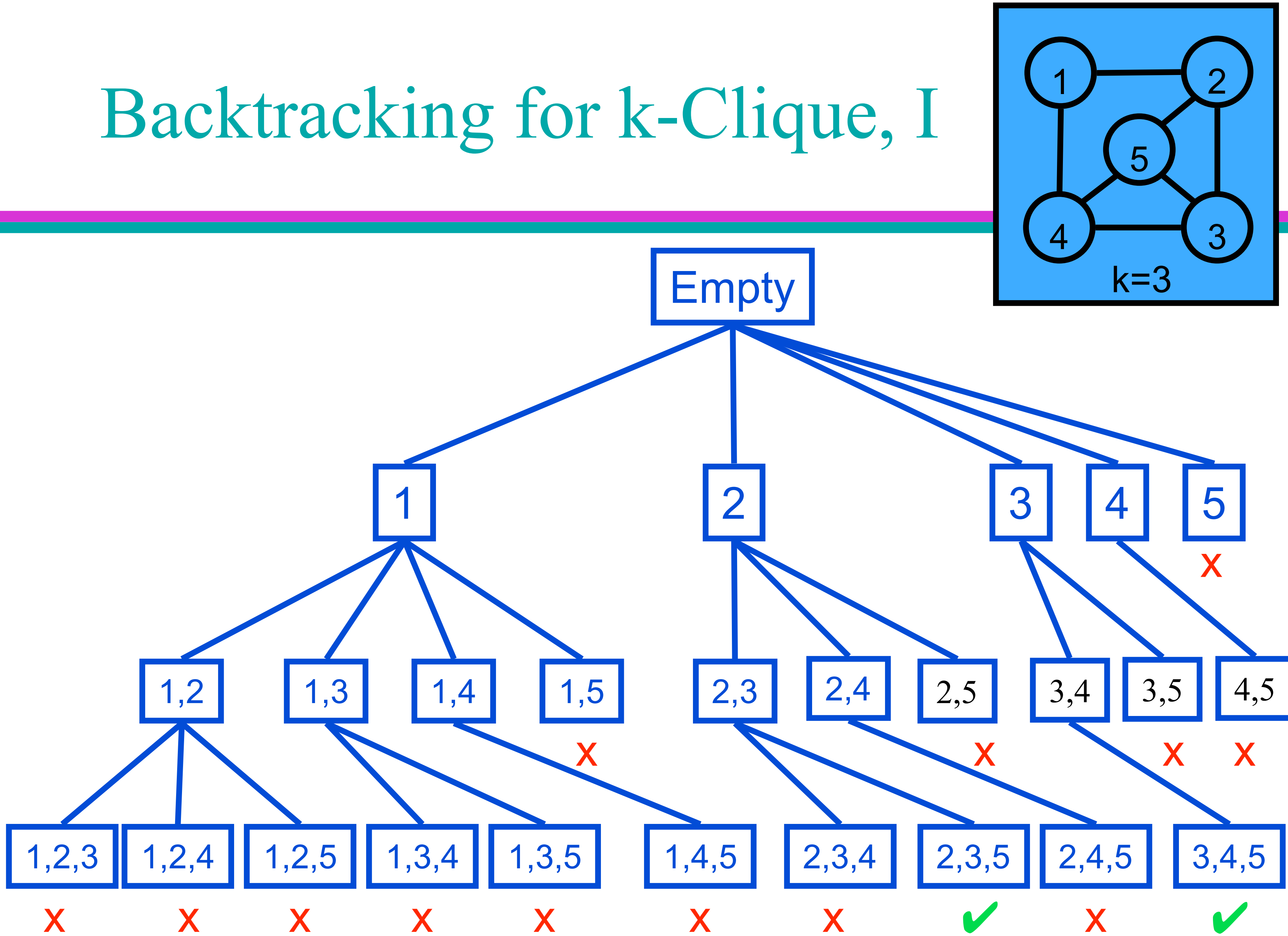
Test for j == k and presence of all edges

S_j is empty if $j \geq k$, else the set of all vertex numbers greater than last in A

Top Level: Call BT(empty,0); report "no solution" if it found none.

Time: $n*(n-1)*\ldots*(n-k+1)*k^2$

Backtracking for k-Clique, I



Backtracking for k-Clique, II

a_i's are distinct vertices, in order, that are adjacent to each other

- Want: a vector (a_1, a_2, … , a_q) satisfying some property P, e.g. "a_1..a_q is a q-clique".

BT(A,j)

if A, j satisfies P, report it

else

Just test for j == k

j = j+1

S_j is empty if j≥k, else set of all vertex numbers greater than last in A and adjacent to all in A

let S_j be the set of "candidates" for slot j;

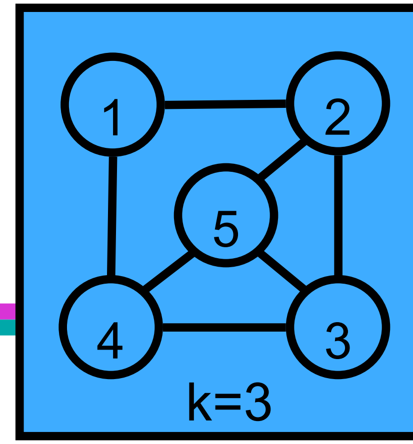
for each a_j in S_j

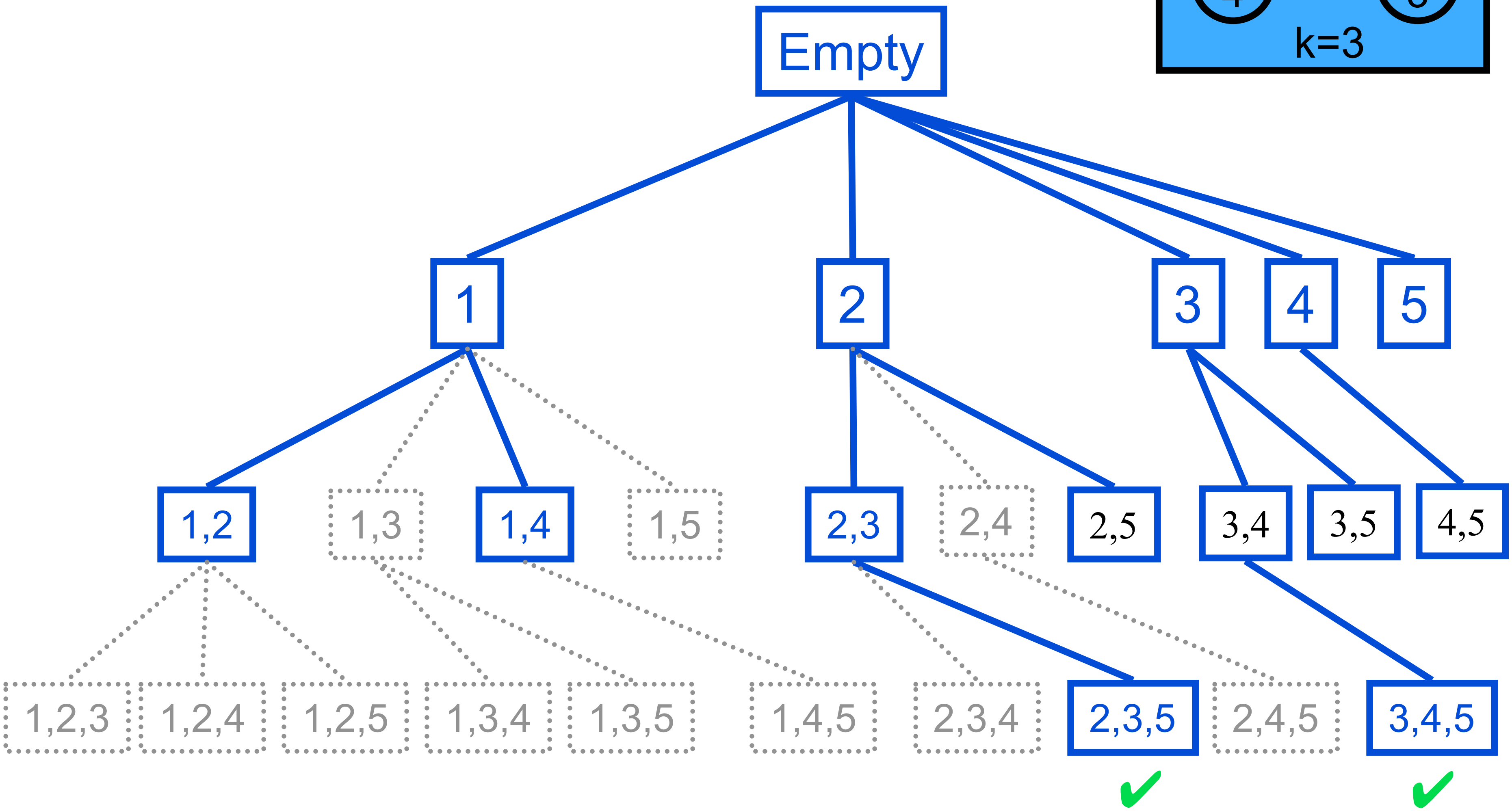
BT(A . a_j, j)

Top Level: Call BT(empty,0); report "no solution" if it found none.

Time: depends strongly on graph, but basically as bad in worst case.

Backtracking for k-Clique, II





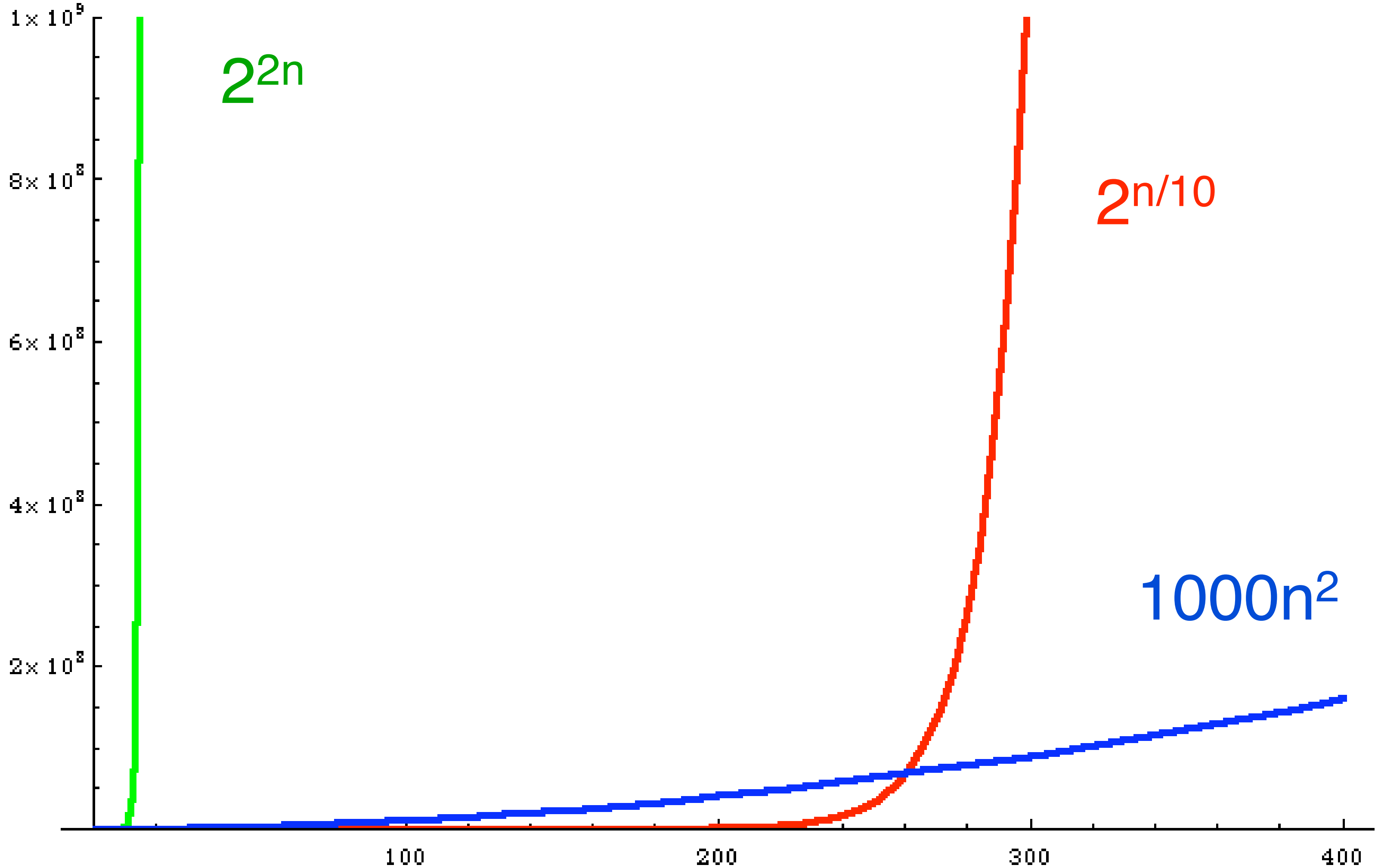
A Brief History of Ideas

- From Classical Greece, if not earlier, "logical thought" held to be a somewhat mystical ability
- Mid 1800's: Boolean Algebra and foundations of mathematical logic created possible "mechanical" underpinnings
- 1900: David Hilbert's famous speech outlines program: mechanize all of mathematics?
  http://mathworld.wolfram.com/HilbertsProblems.html
- 1930's: Gödel, Church, Turing, et al. prove it's impossible

More History

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

Polynomial vs Exponential Growth



Another view of Poly vs Exp

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

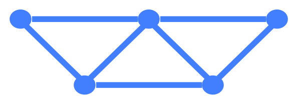
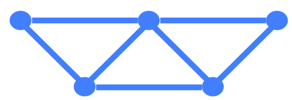
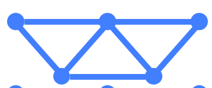
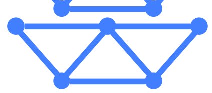
Complexity	Increase	E.g. $T=10^{12}$	
$O(n)$	$n_0 \rightarrow 2n_0$	10^{12}	2×10^{12}
$O(n^2)$	$n_0 \rightarrow \sqrt{2}\, n_0$	10^6	1.4×10^6
$O(n^3)$	$n_0 \rightarrow \sqrt[3]{2}\, n_0$	10^4	1.25×10^4
$2^{n/10}$	$n_0 \rightarrow n_0+10$	400	410
2^n	$n_0 \rightarrow n_0+1$	40	41

Polynomial versus exponential

- We'll say any algorithm whose run-time is
  - polynomial is good
  - bigger than polynomial is *bad*

- Note – of course there are exceptions:
  - n^{100} is bigger than $(1.001)^n$ for most practical values of n but usually such run-times don't show up
  - There are algorithms that have run-times like $O(2^{n/22})$ 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

Decision problems

- Computational complexity usually analyzed using decision problems
  - answer is just 1 or 0 (yes or no).
- Why?
  - much simpler to deal with
  - *deciding* whether G has a k-clique, is certainly no harder than *finding* a k-clique in G, so a *lower* bound on deciding is also a lower bound on finding
  - Less important, but if you have a good decider, you can often use it to get a good finder. (Ex.: does G still have a k-clique after I remove this vertex?)

The class P

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

i.e. $T(n) = O(n^k)$ for some fixed k.

- These problems are sometimes called **tractable** problems.

Examples: sorting, shortest path, MST, connectivity, biconnectivity, various dynamic programming *– all of 417 up to now except Knapsack/Change-Making*

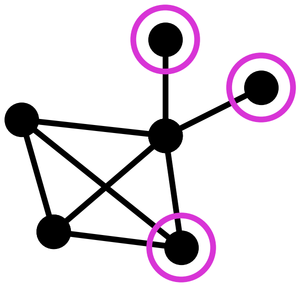
Beyond P?

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

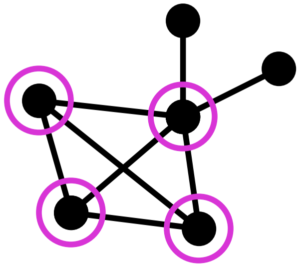
- e.g. CLIQUE:
  - Given a weighted graph G and an integer k, does there exist a k-clique in G?
- e.g. quadratic Diophantine equations:
  - Given a, b, c $\in$ N, $\exists$ x, y $\in$ N s.t. $ax^2 + by = c$?

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 $\leq k$.

Satisfiability

- Boolean variables $x_1,\ldots,x_n$
  - taking values in $\{0,1\}$. 0=false, 1=true
- Literals
  - x_i or $\neg x_i$ for $i=1,\ldots,n$
- Clause
  - a logical OR of one or more literals
  - e.g. $(x_1 \vee \neg x_3 \vee x_7 \vee x_{12})$
- CNF formula
  - a logical AND of a bunch of clauses

Satisfiability

- CNF formula example
  - $(x1 \vee \neg x3 \vee x7 \vee x12) \wedge (x2 \vee \neg x4 \vee x7 \vee x5)$
- 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
  - $x1 \wedge (\neg x1 \vee x2) \wedge (\neg x2 \vee x3) \wedge \neg x3$
- Satisfiability: Given a CNF formula F, is it satisfiable?

More History – As of 1970

- Many of the above problems had been studied for decades
- All had real, practical applications
- *None* had poly time algorithms; exponential was best known

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

Some Problem Pairs

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

Superficially different; similar computationally

Similar pairs; seemingly different computationally

Common property of these problems

- There is a special piece of information, a short hint or proof, that allows you to efficiently (in polynomial-time) verify that the YES answer is correct. This hint might be very hard to find

- e.g.
  - TSP: the tour itself,
  - Independent-Set, Clique: the set U
  - Satisfiability: an assignment that makes F true.
  - Quadratic Diophantine eqns: the numbers x & y.

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

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 $\in$ 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

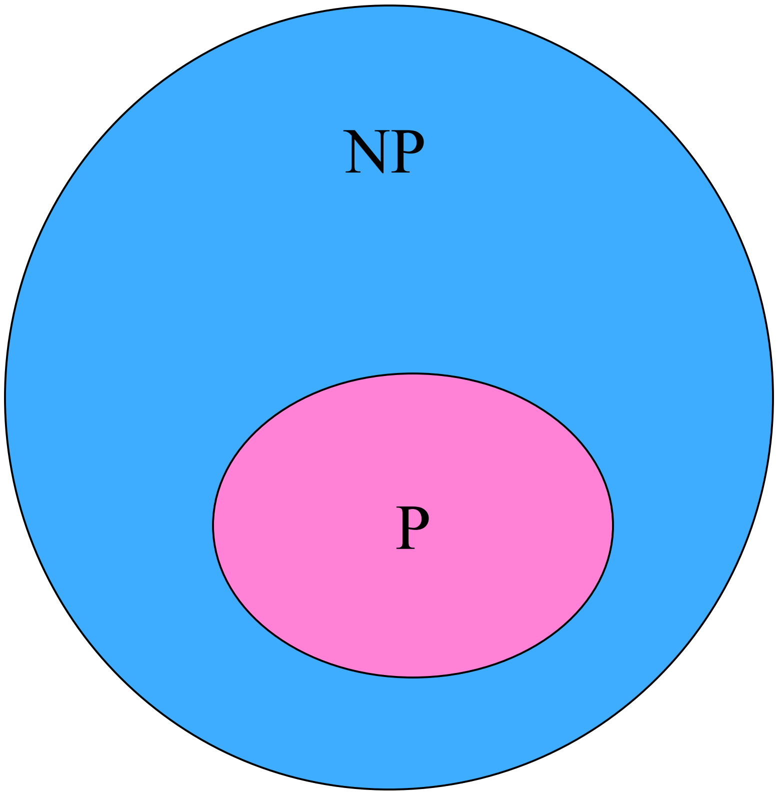
Keys to showing that a problem is in NP

- 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?

Complexity Classes

NP = Polynomial-time
verifiable

P = Polynomial-time
solvable



Solving NP problems without hints

- The only obvious algorithm for most of these problems is brute force:
  - try all possible hints and 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, even much less-obvious algs are slow, too

Problems in P can also be verified in polynomial-time

Shortest 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*?

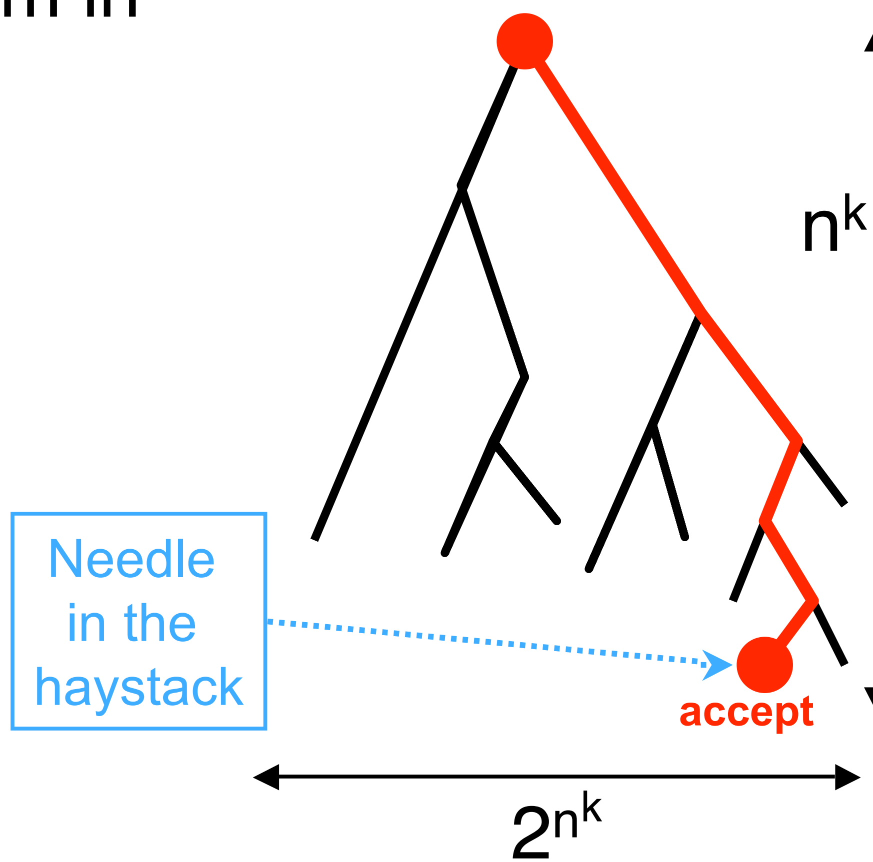
Small Spanning Tree: Given a weighted undirected graph *G*, is there a spanning tree of weight ≤ *k*?

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

P vs NP vs Exponential Time

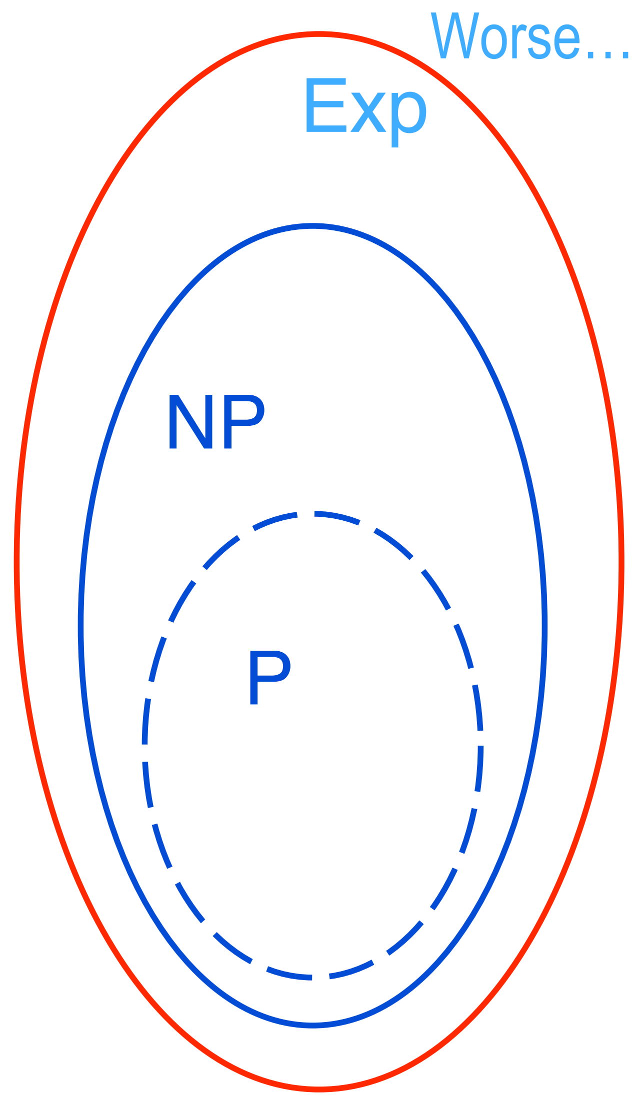
- Theorem: Every problem in NP can be solved deterministically in exponential time

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



P and NP

- Every problem in P is in NP
  - one doesn't even need a hint for problems in P so just ignore any hint you are given
- Every problem in NP is in exponential time
- I.e., P ⊆ NP ⊆ Exp
- We know P ≠ Exp, so either P ≠NP, or NP ≠ Exp (most likely both)



P vs NP

- Theory
  - P = NP ?
  - Open Problem!
  - I bet against it
- Practice
  - Many interesting, useful, natural, well-studied problems known to be NP-complete
  - With rare exceptions, no one routinely succeeds in finding exact solutions to large, arbitrary instances

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

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

NON Theorem: EEXP in NP

- "Proof" 1: Hint = step-by-step trace of the computation of p on x; verify step-by-step

More Connections

- Some Examples in NP
  - Satisfiability
  - Independent-Set
  - Clique
  - Vertex Cover
- All hard to solve; hints seem to help on all
- Very surprising fact:

  - Fast solution to *any* gives fast solution to *all!*

The class NP-complete

We are pretty sure that no problem in NP – P can be solved in polynomial time.

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

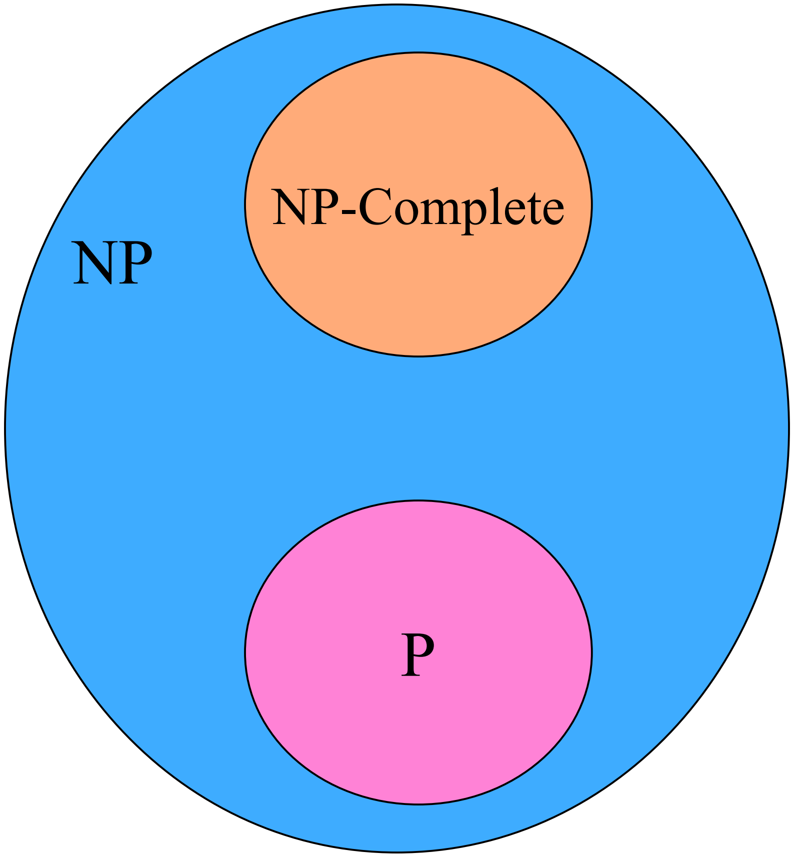
Interesting fact: If any one NP-complete problem could be solved in polynomial time, then ***all*** NP problems could be solved in polynomial time.

Complexity Classes

NP = Poly-time **verifiable**

P = Poly-time **solvable**

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



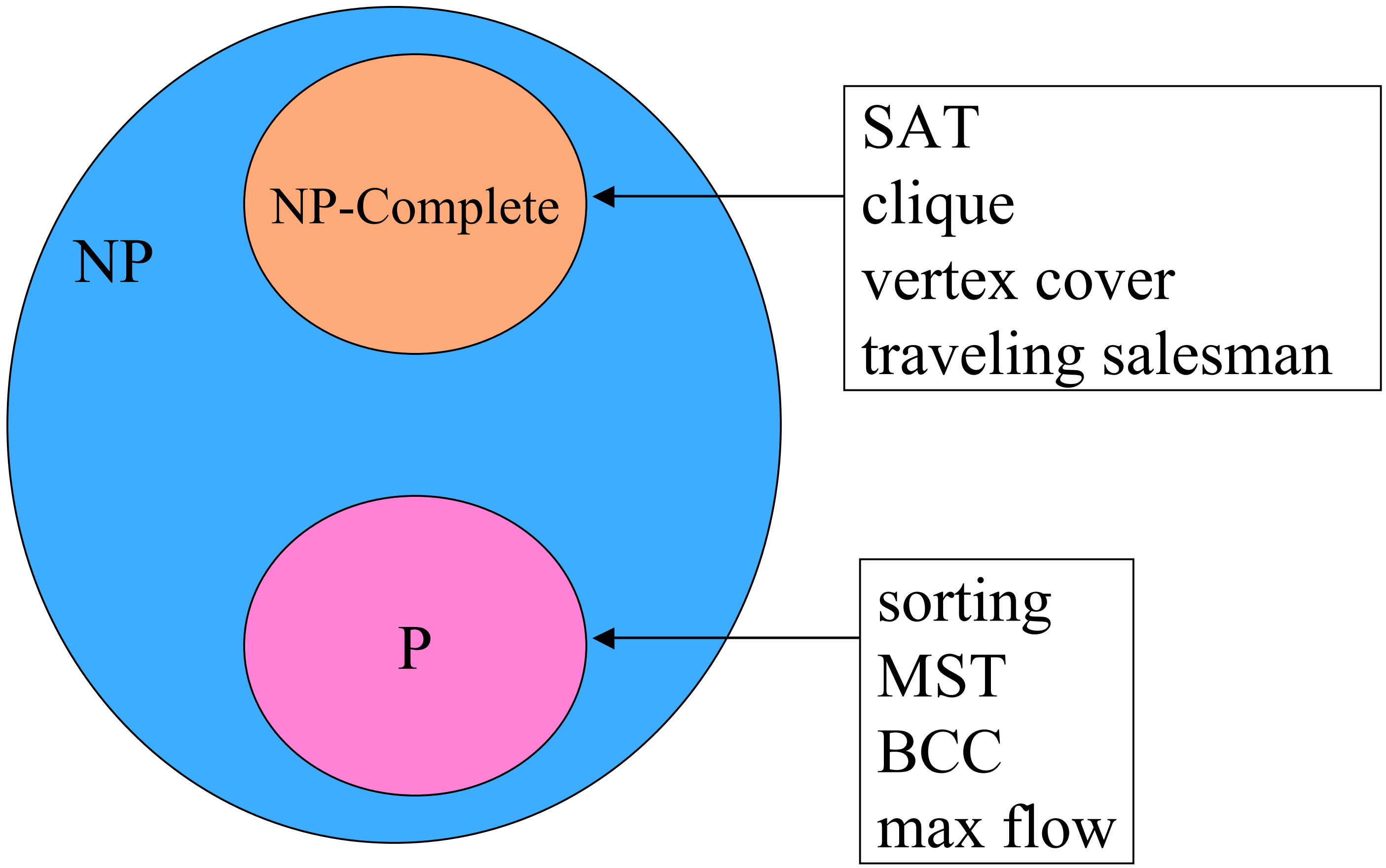
The class NP-complete (cont.)

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.

Complexity Classes of Problems



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.)

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.

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"*

The growth of the number of NP-complete problems

- Steve Cook (1971) showed that SAT was NP-complete.
- Richard Karp (1972) found 24 more NP-complete problems.
- Today there are thousands of known NP-complete problems.
  - Garey and Johnson (1979) is an excellent source of NP-complete problems.

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:

$$S=(x+y+\neg z)\bullet(\neg x+y+z)\bullet(\neg x+\neg y+\neg z)$$

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

How do you prove problem *A* is NP-complete?

1) **Prove *A* is in NP:** show that given a solution, it can be verified in polynomial time.
2) **Prove that *A* is NP-hard:**

   a) Select a **known** NP-complete problem *B*.

   b) Describe a polynomial time computable algorithm that computes a function *f*, **mapping *every* instance of *B* to an instance of *A***. (that is: $B \leq_p A$)

   c) Prove that if b is a ***yes***-instance of *B* then f(b) is a ***yes***-instance of *A*. Conversely, if f(b) is a ***yes***-instance of *A,* then b must be ***yes***-instance of *B*.

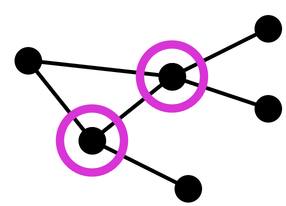
   d) Prove that the algorithm computing ***f* runs in polynomial time**.

NP-complete problem: **Vertex Cover**

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

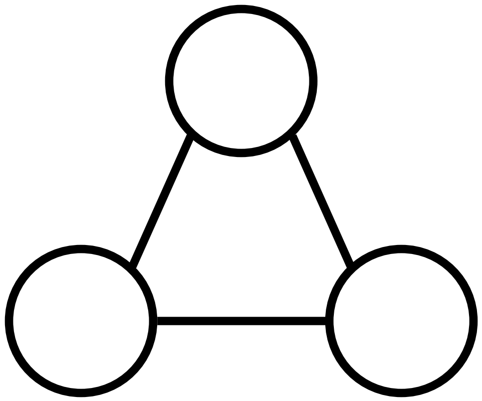
Output: True iff there is a subset C of V of size $\leq k$ such that every edge in E is incident to at least one vertex in C.

Example: Vertex cover of size ≤ 2.

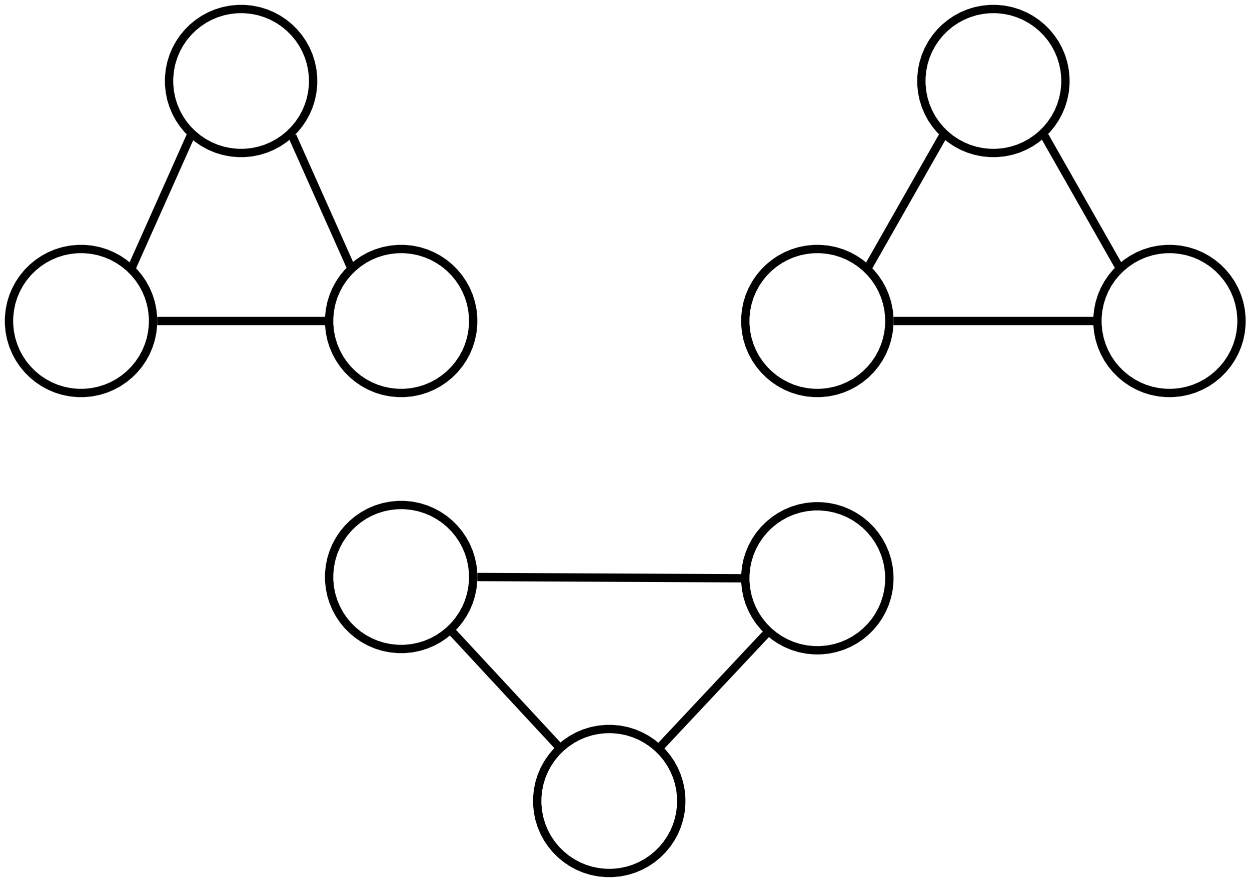


In NP? Exercise

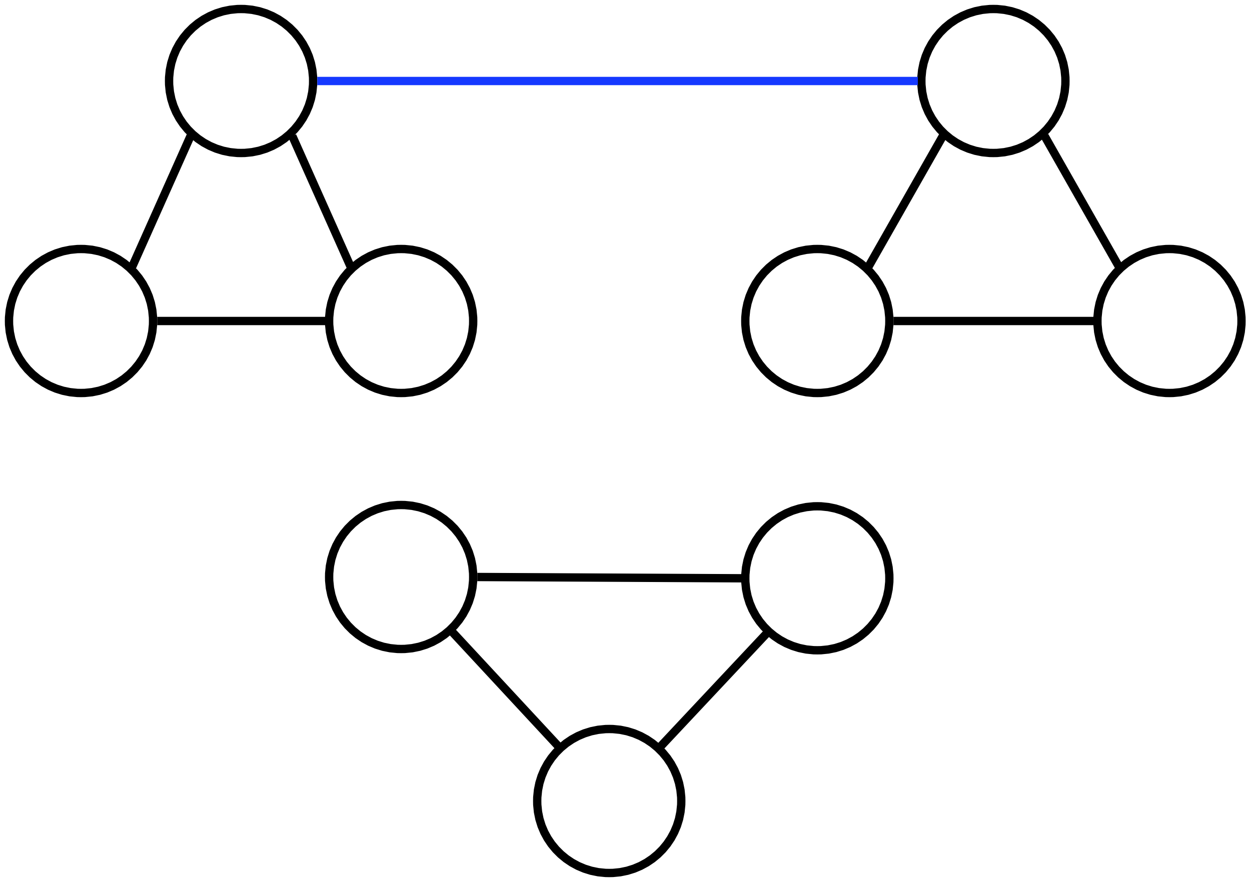
3SAT $\leq_p$ VertexCover



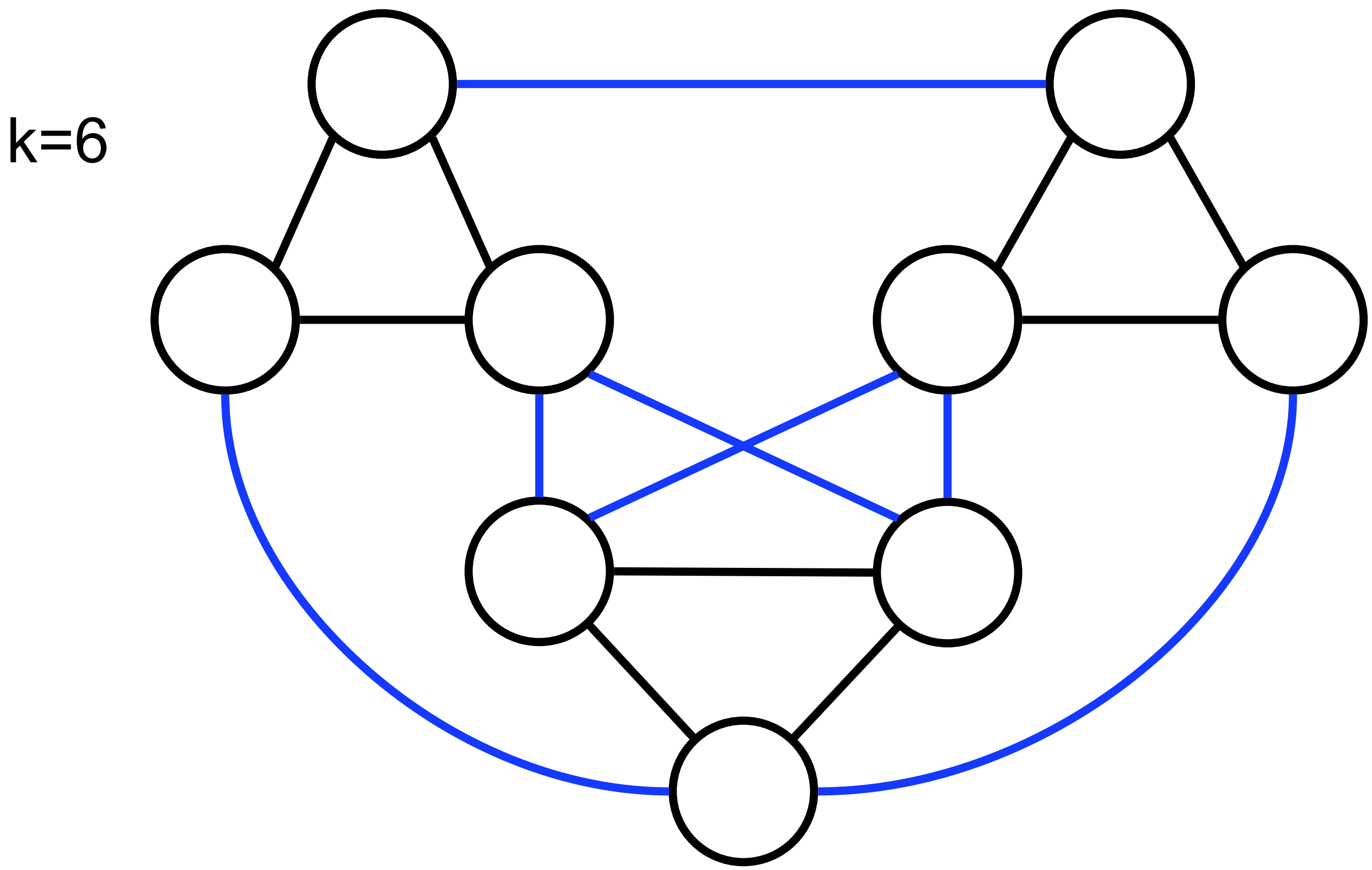
3SAT $\leq_p$ VertexCover



3SAT $\leq_p$ VertexCover

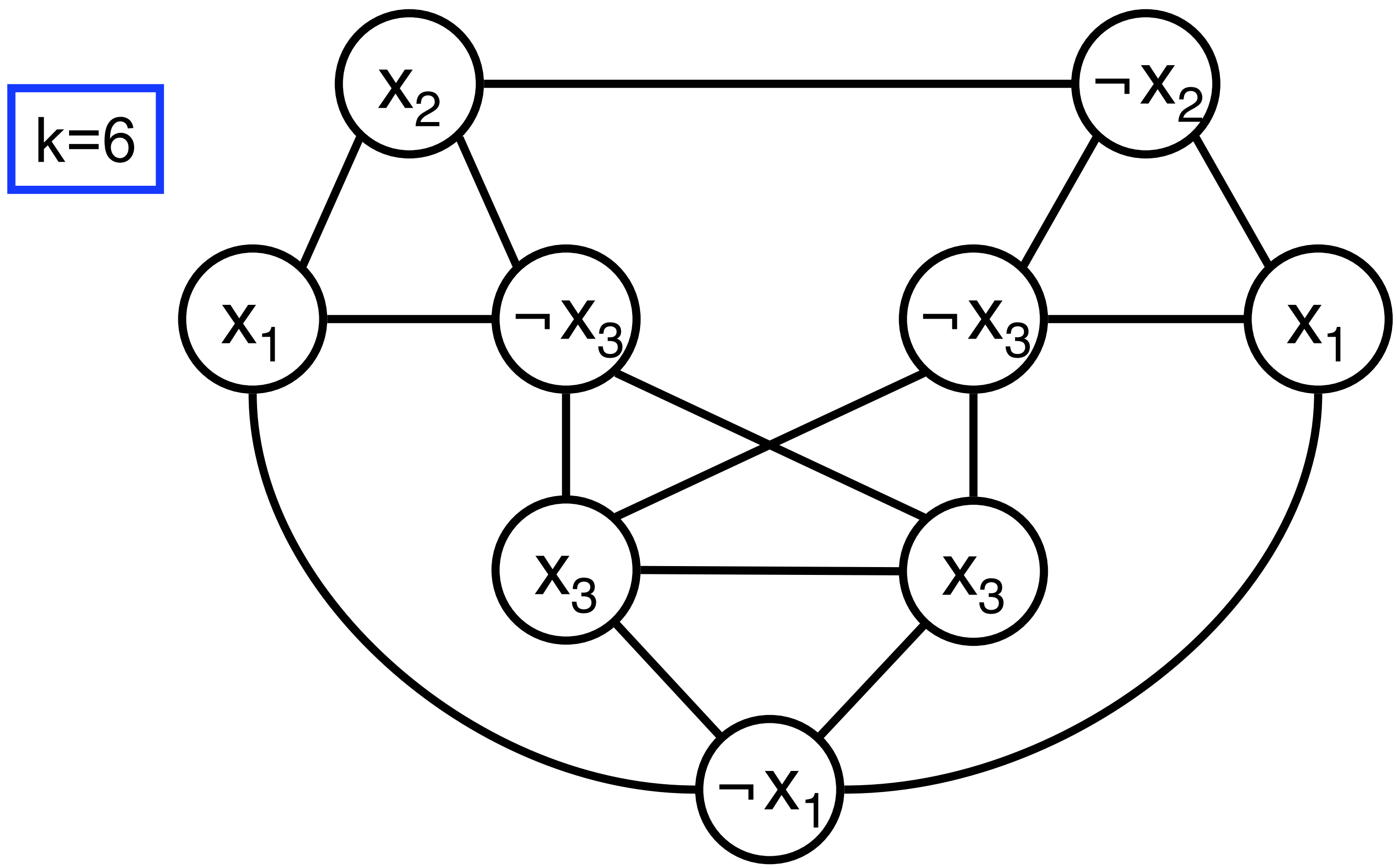


3SAT $\leq_p$ VertexCover



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 (

3-SAT Instance:

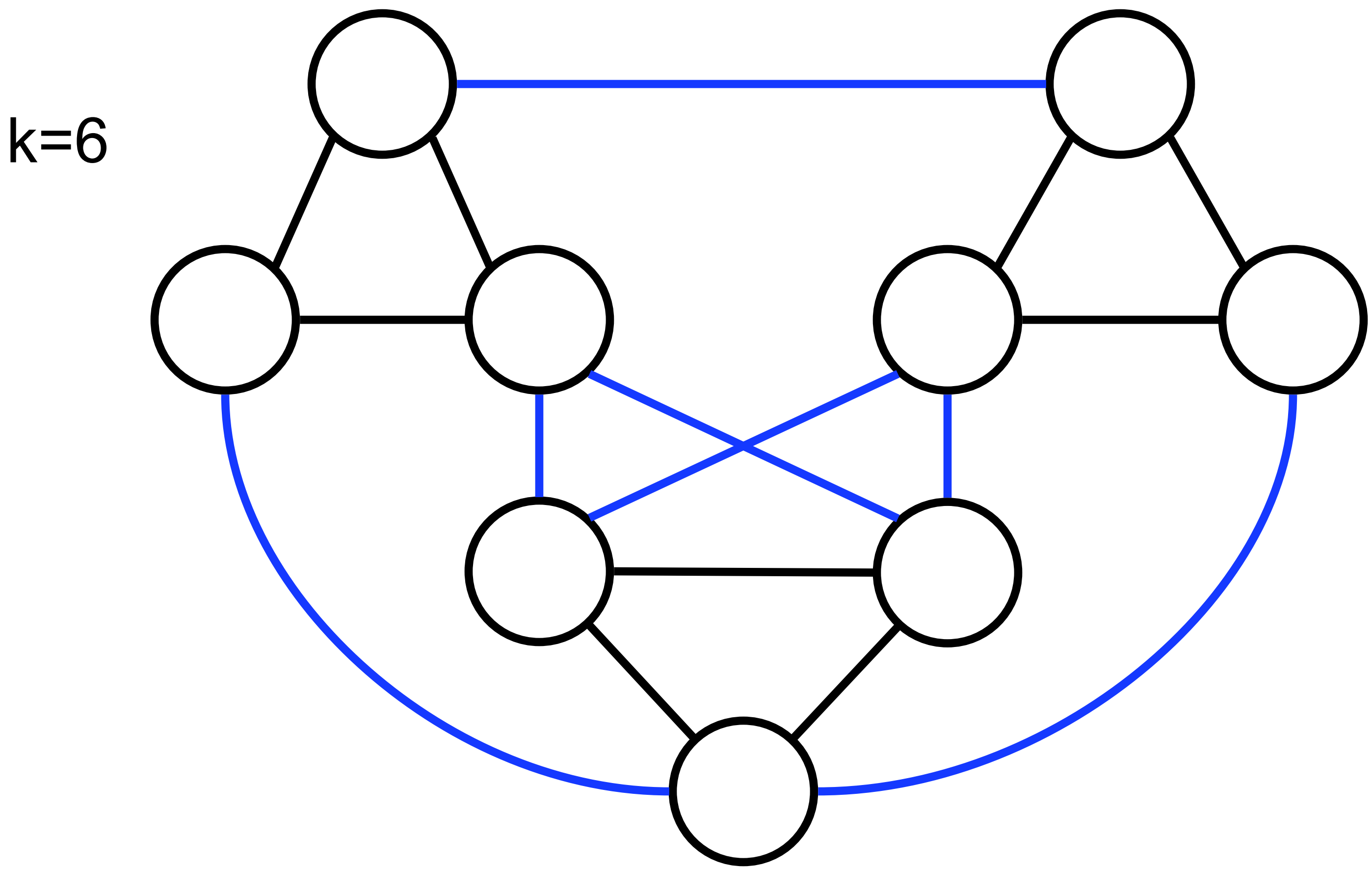
- Variables: $x_1, x_2, \ldots$
- Literals: $y_{i,j}$, $1 \leq i \leq q$, $1 \leq j \leq 3$
- Clauses: $c_i = y_{i1} \vee y_{i2} \vee y_{i3}$, $1 \leq l \leq q$
- Formula: $c = c_1 \wedge c_2 \wedge \ldots \wedge c_q$

) =

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



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

Summary of reduction function f:

Given formula, make graph G with one group per clause, one node per literal. Connect each to all nodes in *same* group, *plus* complementary literals (x, ¬x). Output graph G plus integer k = 2 * number of clauses.

Note: f does *not* know whether formula is satisfiable or not; does *not* know if G has k-cover; does *not* try to find satisfying assignment or cover.

Correctness:

1. Show f poly time computable: A key point is that graph size is polynomial in formula size; mapping basically straightforward.
2. 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 "3-SAT $\leq_p$ VertexCover"

- *Suppose* we had a fast algorithm for VertexCover, then we could get a fast algorithm for 3SAT:
  – Given 3-CNF formula *w*, build VertexCover 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.

“3-SAT $\leq_p$ VertexCover” Retrospective

- Previous slide: two *suppositions*
- Somewhat clumsy to have to state things that way.
- Alternative: abstract out the key elements, give it a name (“polynomial time reduction”), then properties like the above always hold.

Polynomial-Time Reductions

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

We say that ***A*** is **polynomially reducible** to ***B*** 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***.

$$\mathbf{x} \in \boldsymbol{A} \;\Leftrightarrow\; \boldsymbol{f(x)} \in \boldsymbol{B}$$

Polynomial-Time Reductions (cont.)

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

Why the notation?

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

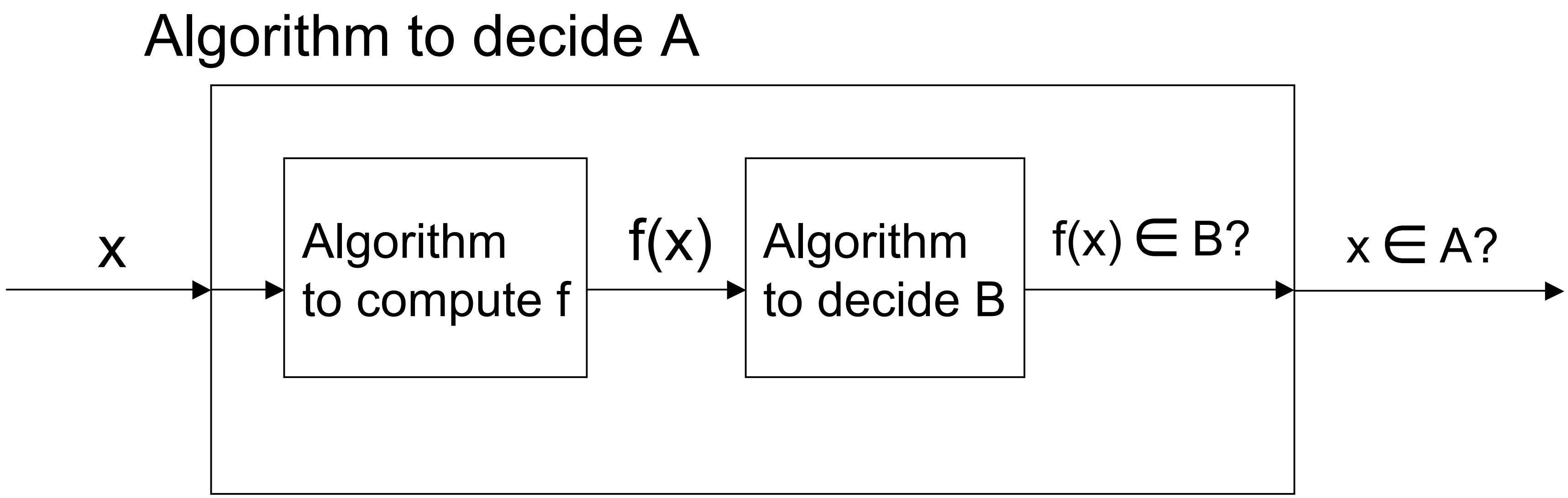
polynomial

(1) $\boldsymbol{A \leq_p B}$ *and* $\boldsymbol{B \in P} \Rightarrow \boldsymbol{A \in P}$

(2) $\boldsymbol{A \leq_p B}$ *and* $\boldsymbol{A \notin P} \Rightarrow \boldsymbol{B \notin P}$

(3) $\boldsymbol{A \leq_p B}$ *and* $\boldsymbol{B \leq_p C} \Rightarrow \boldsymbol{A \leq_p C}$ *(transitivity)*

Using an Algorithm for *B* to Decide *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?

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!) This 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 <u>some</u> problem ***L'*** ∈ ***NP-complete***, in order to show that **L** is NP-hard. Why is this equivalent?

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

∀ ***L''*** ∈ ***NP****, we have* ***L''*** $\leq_p$ ***L'***

2) If we show ***L'*** $\leq_p$ ***L****,* then by transitivity we know that: ∀ ***L''*** ∈ ***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

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

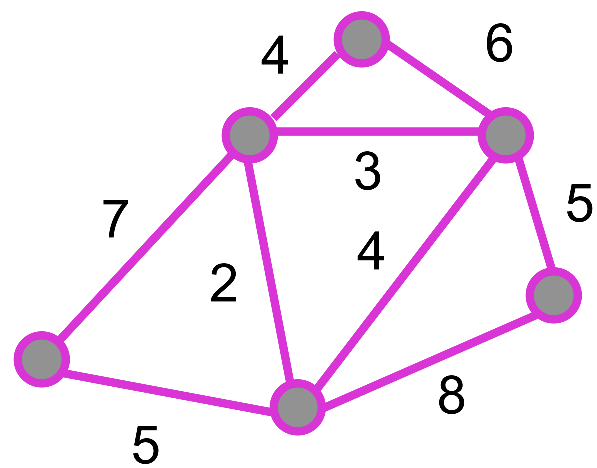
NP-complete problem: **TSP**

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.

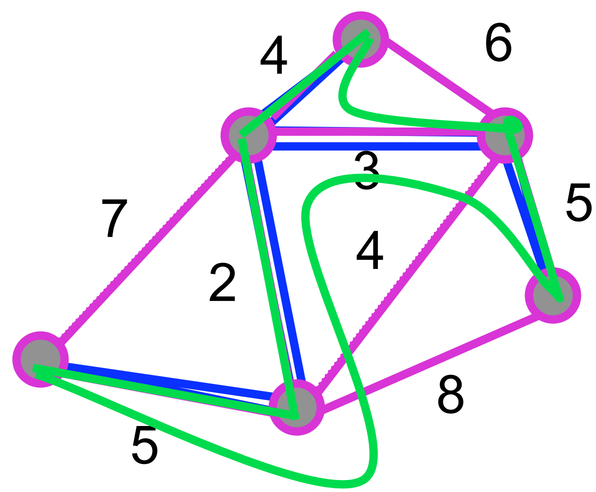
Example:

b = 34



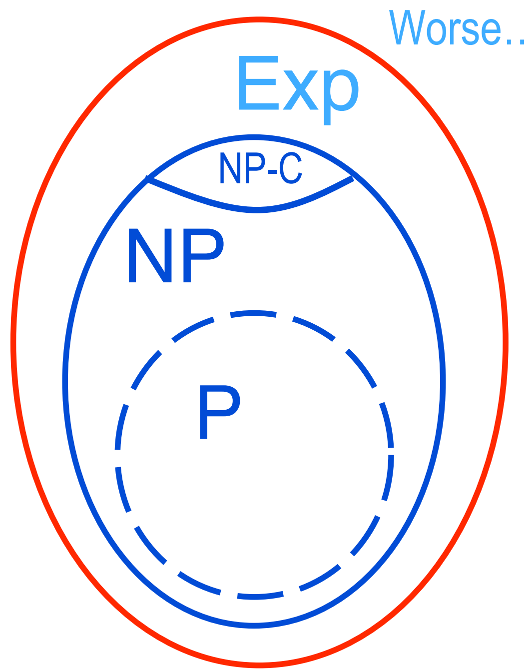
2x Approximation to EuclideanTSP

- 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

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.