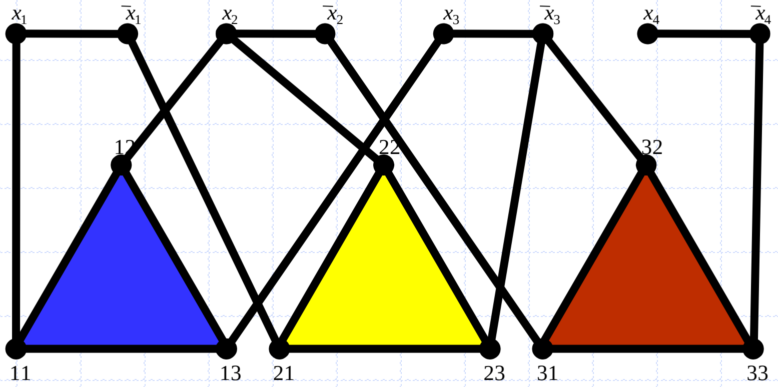
Chapter 13: NP-Completeness

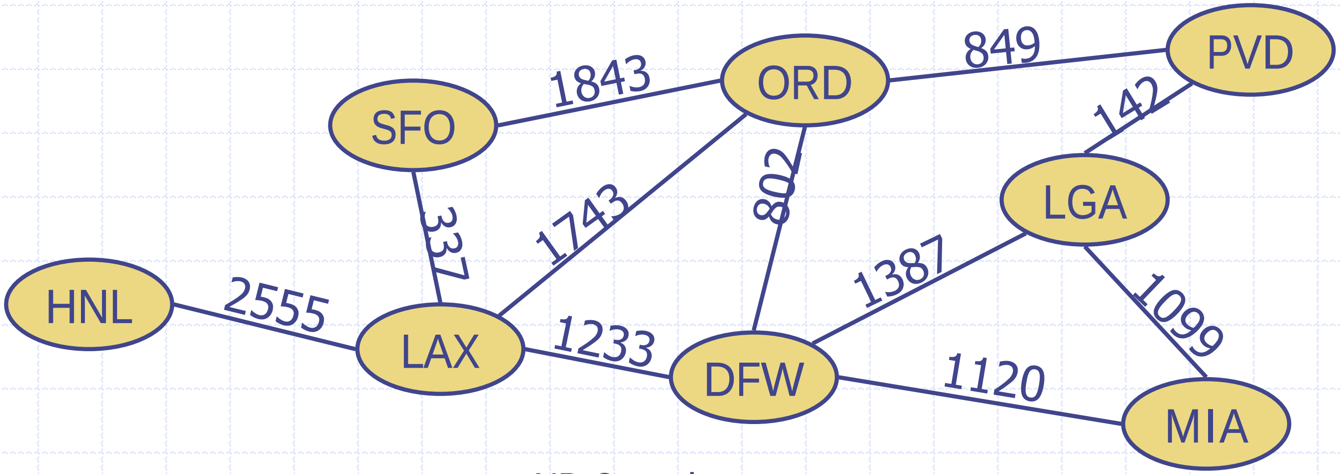


Outline and Reading

- P and NP (§13.1)
- NP-completeness (§13.2)
- Some NP-complete problems (§13.3)
- Approximation Algorithms for NP-Complete Problems (§13.4)
- Optional: Backtracking and Branch-and-Bound (§13.4)

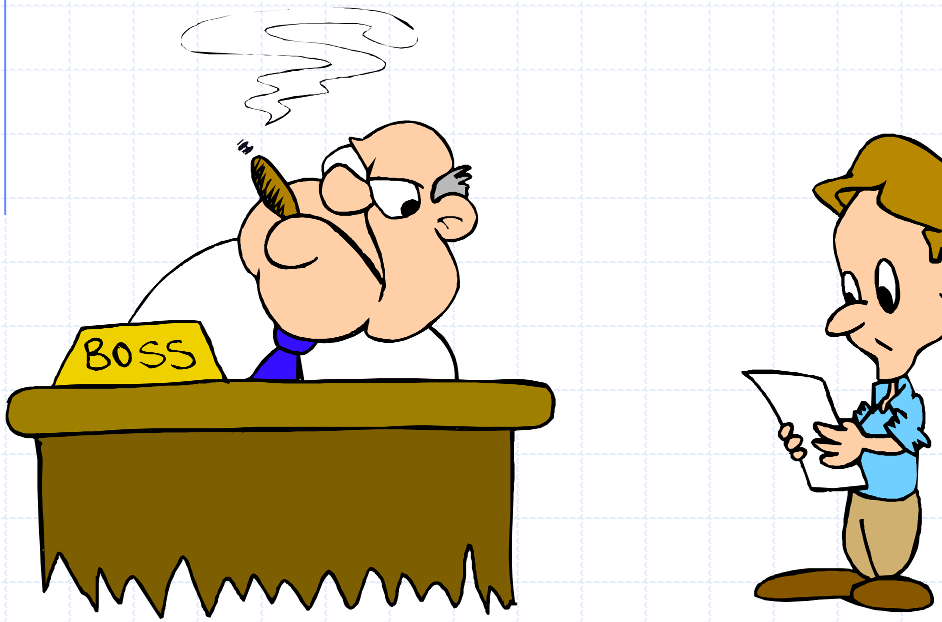
Running Time Revisited

- Input size, ***n***
  - To be exact, let *n* denote the number of **bits** in a nonunary encoding of the input
- All the polynomial-time algorithms studied so far in this course run in polynomial time using this definition of input size.
  - Exception: any pseudo-polynomial time algorithm



Dealing with Hard Problems

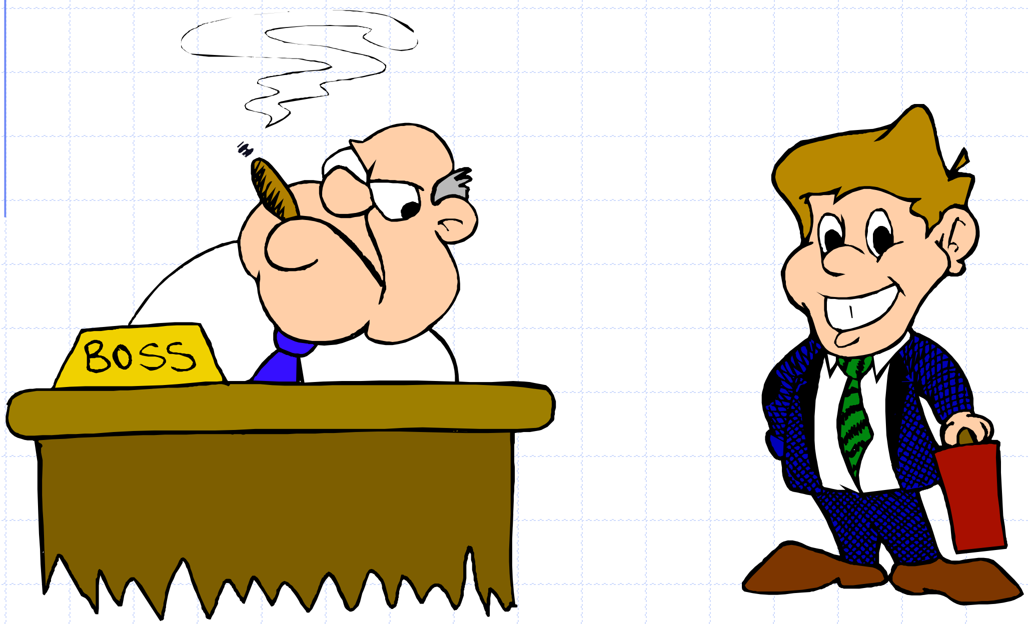
- What to do when we find a problem that looks hard...



I couldn't find a polynomial-time algorithm;
I guess I'm too dumb.

Dealing with Hard Problems

- Sometimes we can prove a strong lower bound... (but not usually)

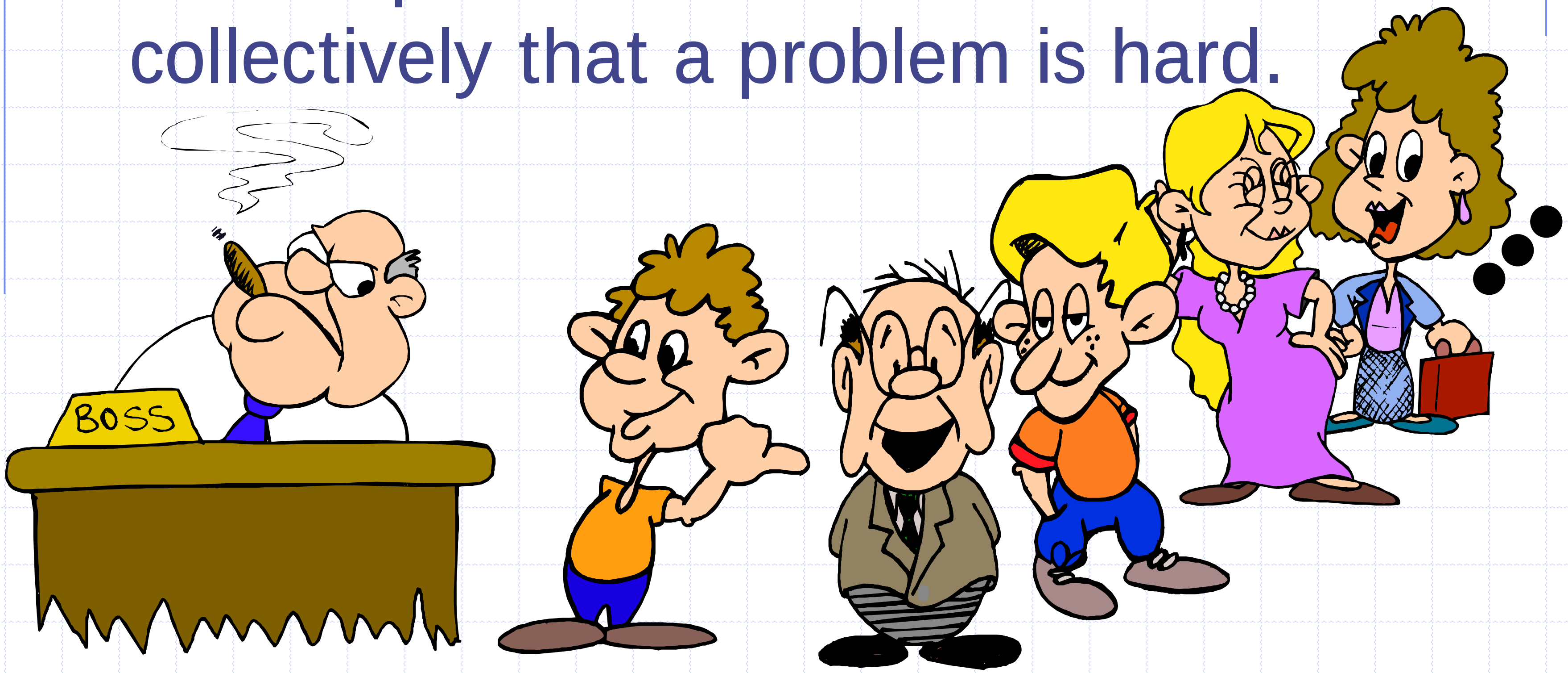


I couldn't find a polynomial-time algorithm,
because no such algorithm exists!

(cartoon inspired by [Garey-Johnson, 79])

Dealing with Hard Problems

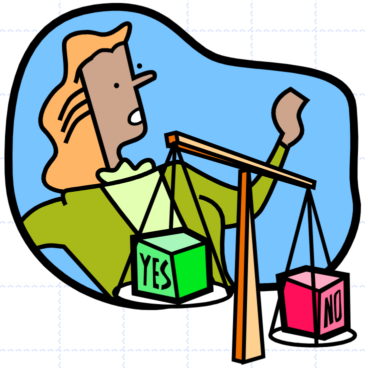
- NP-completeness let's us show collectively that a problem is hard.



I couldn't find a polynomial-time algorithm, but neither could all these other smart people.

(cartoon inspired by [Garey-Johnson, 79])

Polynomial-Time Decision Problems



- To simplify the notion of "hardness," we will focus on the following:
  - Polynomial-time as the cut-off for efficiency
  - Decision problems: output is 1 or 0 ("yes" or "no")
    - Examples:
    - Does a given graph G have an Euler tour?
    - Does a text T contain a pattern P?
    - Does an instance of 0/1 Knapsack have a solution with benefit at least K?
    - Does a graph G have an MST with weight at most K?

Problems and Languages

- A **language** L is a set of strings defined over some alphabet Σ
- Every decision algorithm A defines a language L
  - L is the set consisting of every string x such that A outputs "yes" on input x.
  - We say "A **accepts** x" in this case
    - Example:
    - If A determines whether or not a given graph G has an Euler tour, then the language L for A is all graphs with Euler tours.

The Complexity Class P

- A **complexity class** is a collection of languages
- P is the complexity class consisting of all languages that are accepted by **polynomial-time** algorithms
- For each language L in P there is a polynomial-time decision algorithm A for L.
  - If n=|x|, for x in L, then A runs in p(n) time on input x.
  - The function p(n) is some polynomial

The Complexity Class NP

- We say that an algorithm is non-deterministic if it uses the following operation:
  - Choose(b): chooses a bit b non-deterministically (0 or 1)
  - Can be used to choose an entire string y (with |y| choices)
- We say that a non-deterministic algorithm A **accepts** a string x if there exists some sequence of choose operations that causes A to output "yes" on input x.
- NP is the complexity class consisting of all languages accepted by **polynomial-time non-deterministic** algorithms.

NP example

- Problem: Decide if a graph has an MST of weight K

- Algorithm:
  1. Non-deterministically choose a set T of n-1 edges
  2. Test that T forms a spanning tree
  3. Test that T has weight at most K

- Analysis: Testing takes O(n+m) time, so this algorithm runs in polynomial time.

The Complexity Class NP Alternate Definition

- We say that an algorithm B **verfies** the acceptance of a language L if and only if, for any x in L, there exists a certificate y such that B outputs “yes” on input (x,y).
- NP is the complexity class consisting of all languages verified by **polynomial-time** algorithms.

- We know: P is a subset of NP.
- Major open question: P=NP?
- Most researchers believe that P and NP are different.

NP example (2)

- Problem: Decide if a graph has an MST of weight K

- Verification Algorithm:
  1. Use as a certificate, y, a set T of n-1 edges
  2. Test that T forms a spanning tree
  3. Test that T has weight at most K

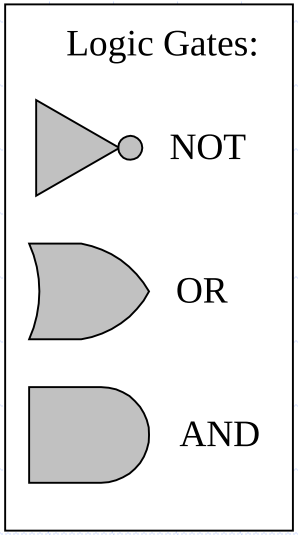
- Analysis: Verification takes O(n+m) time, so this algorithm runs in polynomial time.

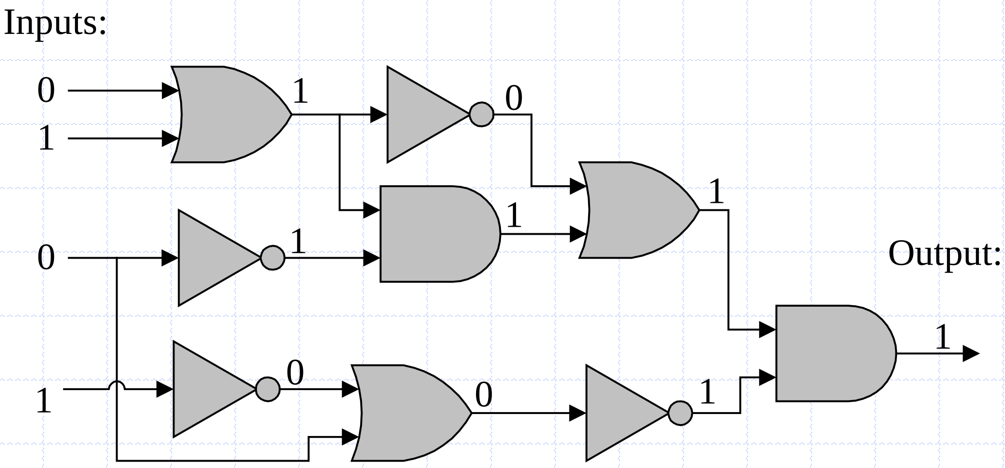
Equivalence of the Two Definitions

- Suppose A is a non-deterministic algorithm
  - Let y be a certificate consisting of all the outcomes of the choose steps that A uses
  - We can create a verification algorithm that uses y instead of A's choose steps
  - If A accepts on x, then there is a certificate y that allows us to verify this (namely, the choose steps A made)
  - If A runs in polynomial-time, so does this verification algorithm
- Suppose B is a verification algorithm
  - Non-deterministically choose a certificate y
  - Run B on y
  - If B runs in polynomial-time, so does this non-deterministic algorithm

An Interesting Problem

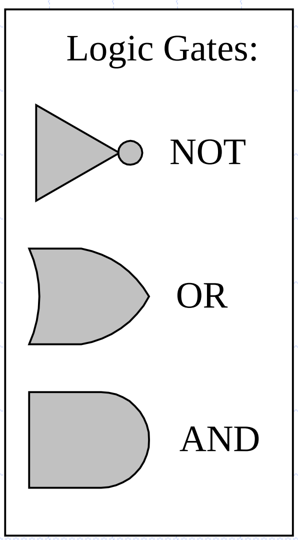
- A Boolean circuit is a circuit of AND, OR, and NOT gates; the CIRCUIT-SAT problem is to determine if there is an assignment of 0's and 1's to a circuit's inputs so that the circuit outputs 1.

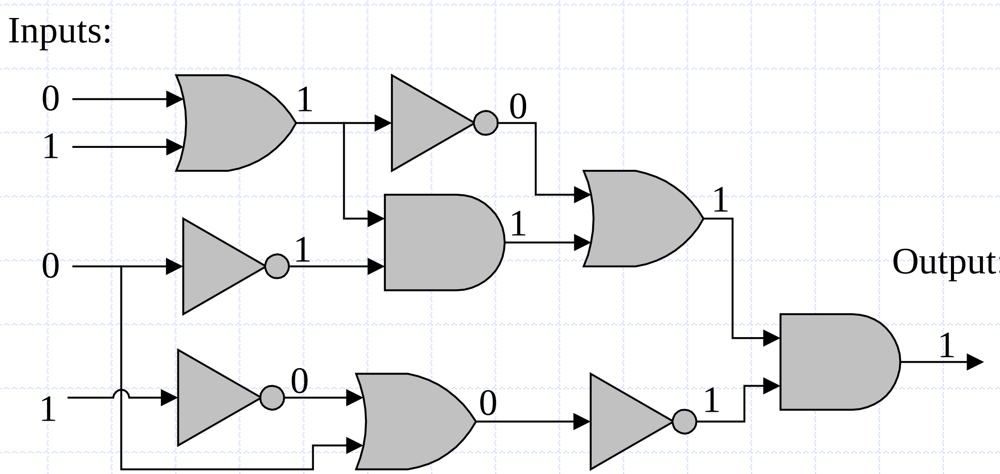




CIRCUIT-SAT is in NP

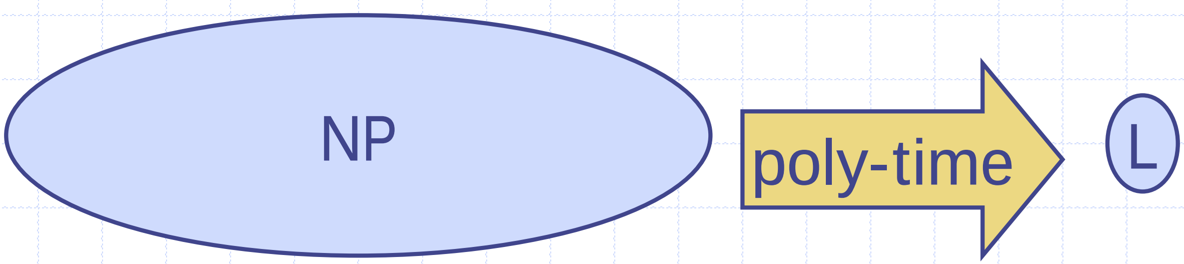
- Non-deterministically choose a set of inputs and the outcome of every gate, then test each gate's I/O.





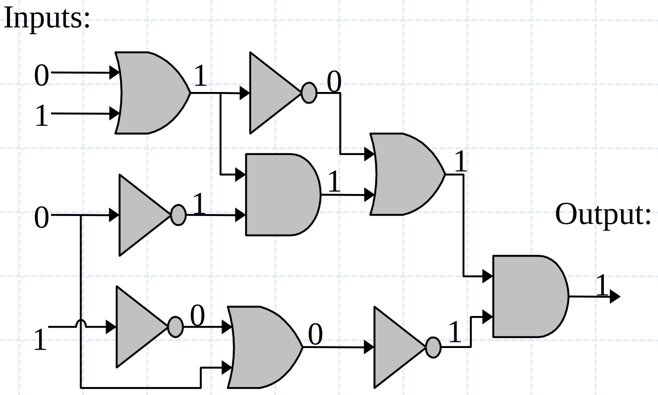
NP-Completeness

- A problem (language) L is **NP-hard** if every problem in NP can be reduced to L in polynomial time.
- That is, for each language M in NP, we can take an input x for M, **transform** it in polynomial time to an input x' for L such that x is in M if and only if x' is in L.
- L is **NP-complete** if it's in NP and is NP-hard.

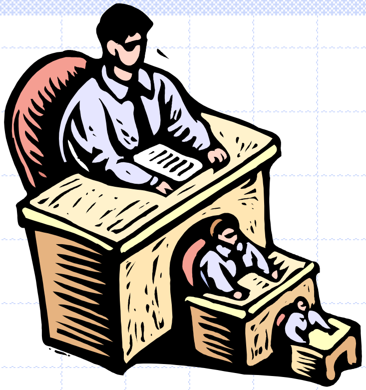


Problem Reduction

- A language M is polynomial-time **reducible** to a language L if an instance x for M can be transformed in polynomial time to an instance x' for L such that x is in M if and only if x' is in L.
  - Denote this by $M \rightarrow L$.
- A problem (language) L is **NP-hard** if every problem in NP is polynomial-time reducible to L.
- A problem (language) is **NP-complete** if it is in NP and it is NP-hard.
- CIRCUIT-SAT is NP-complete:
  - CIRCUIT-SAT is in NP
  - For every M in NP, $M \rightarrow$ CIRCUIT-SAT.

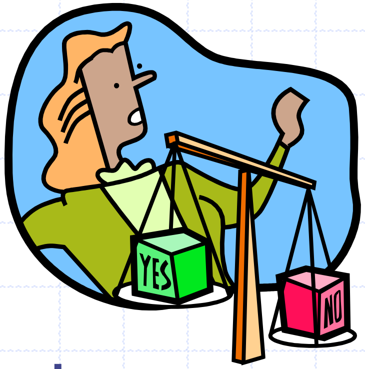


Transitivity of Reducibility



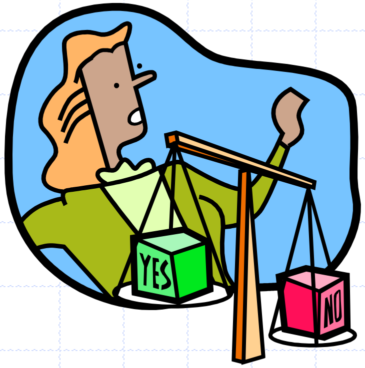
- If A $\rightarrow$ B and B $\rightarrow$ C, then A $\rightarrow$ C.
  - An input x for A can be converted to x' for B, such that x is in A if and only if x' is in B. Likewise, for B to C.
  - Convert x' into x'' for C such that x' is in B iff x'' is in C.
  - Hence, if x is in A, x' is in B, and x'' is in C.
  - Likewise, if x'' is in C, x' is in B, and x is in A.
  - Thus, A $\rightarrow$ C, since polynomials are closed under composition.
- Types of reductions:
  - **Local replacement:** Show A $\rightarrow$ B by dividing an input to A into components and show how each component can be converted to a component for B.
  - **Component design:** Show A $\rightarrow$ B by building special components for an input of B that enforce properties needed for A, such as "choice" or "evaluate."

SAT

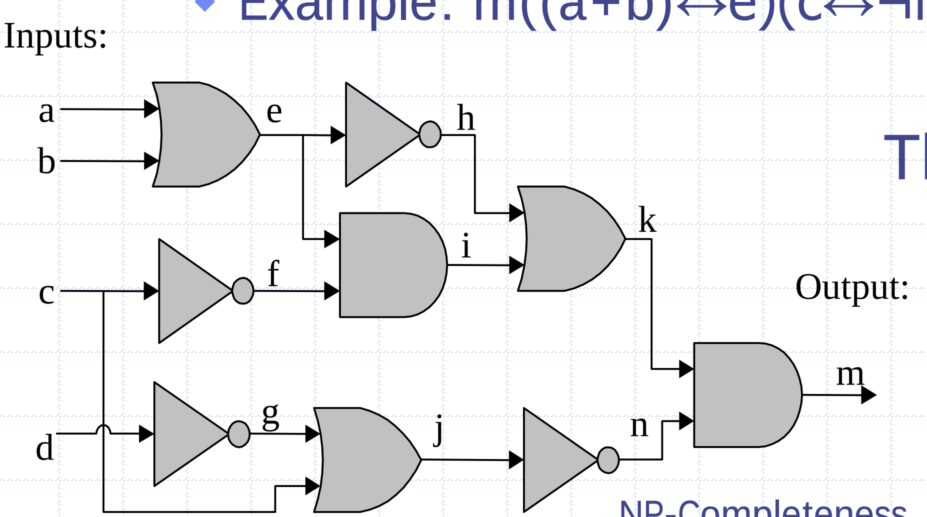


- A Boolean formula is a formula where the variables and operations are Boolean (0/1):
  - (a+b+¬d+e)(¬a+¬c)(¬b+c+d+e)(a+¬c+¬e)
  - OR: +, AND: (times), NOT: ¬
- SAT: Given a Boolean formula S, is S satisfiable, that is, can we assign 0's and 1's to the variables so that S is 1 ("true")?
  - Easy to see that CNF-SAT is in NP:
    - Non-deterministically choose an assignment of 0's and 1's to the variables and then evaluate each clause. If they are all 1 ("true"), then the formula is satisfiable.

SAT is NP-complete



- Reduce CIRCUIT-SAT to SAT.
  - Given a Boolean circuit, make a variable for every input and gate.
  - Create a sub-formula for each gate, characterizing its effect. Form the formula as the output variable AND-ed with all these sub-formulas:
    - Example: $m((a+b)\leftrightarrow e)(c\leftrightarrow \neg f)(d\leftrightarrow \neg g)(e\leftrightarrow \neg h)(ef\leftrightarrow i)\ldots$

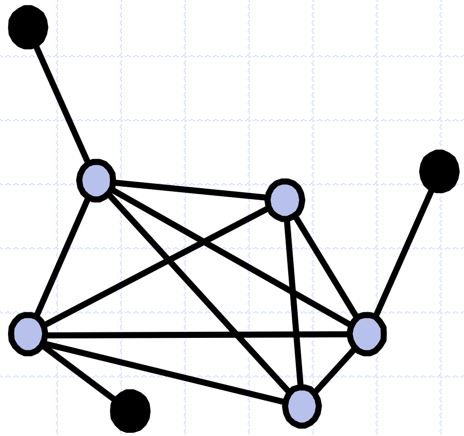


The formula is satisfiable if and only if the Boolean circuit is satisfiable.

Clique

- A **clique** of a graph G=(V,E) is a subgraph C that is fully-connected (every pair in C has an edge).
- CLIQUE: Given a graph G and an integer K, is there a clique in G of size at least K?

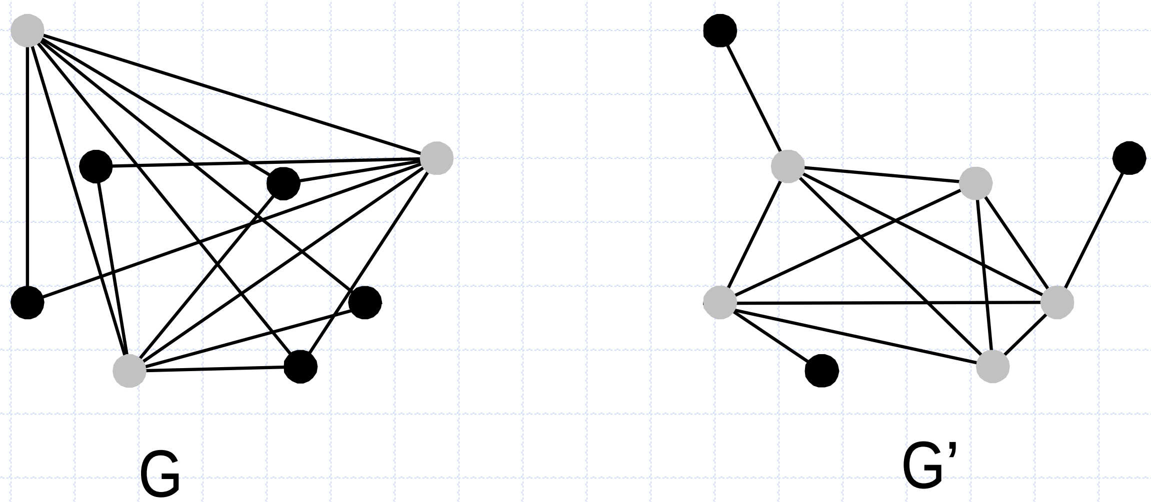
This graph has
a clique of size 5



- CLIQUE is in NP: non-deterministically choose a subset C of size K and check that every pair in C has an edge in G.

CLIQUE is NP-Complete

- Reduction from VERTEX-COVER.
- A graph G has a vertex cover of size K if and only if it’s complement has a clique of size n-K.



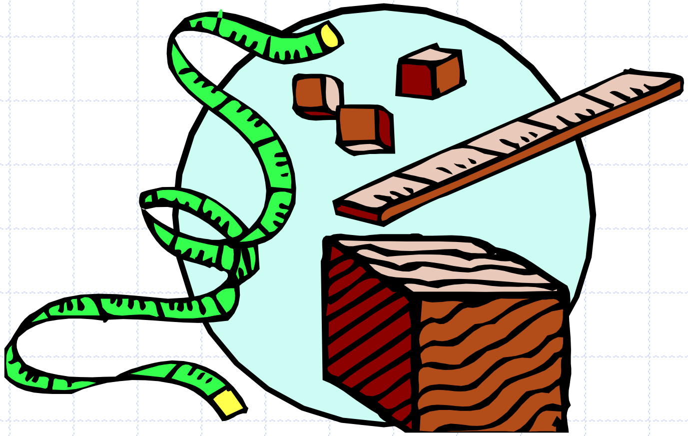
Some Other NP-Complete Problems

- SET-COVER: Given a collection of m sets, are there K of these sets whose union is the same as the whole collection of m sets?
  - NP-complete by reduction from VERTEX-COVER
- SUBSET-SUM: Given a set of integers and a distinguished integer K, is there a subset of the integers that sums to K?
  - NP-complete by reduction from VERTEX-COVER

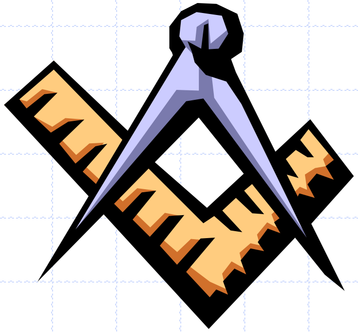
Some Other NP-Complete Problems

- 0/1 Knapsack: Given a collection of items with weights and benefits, is there a subset of weight at most W and benefit at least K?
  - NP-complete by reduction from SUBSET-SUM
- Hamiltonian-Cycle: Given an graph G, is there a cycle in G that visits each vertex exactly once?
  - NP-complete by reduction from VERTEX-COVER
- Traveling Saleperson Tour: Given a complete weighted graph G, is there a cycle that visits each vertex and has total cost at most K?
  - NP-complete by reduction from Hamiltonian-Cycle.

Approximation Algorithms

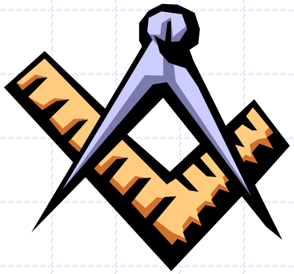


Approximation Ratios



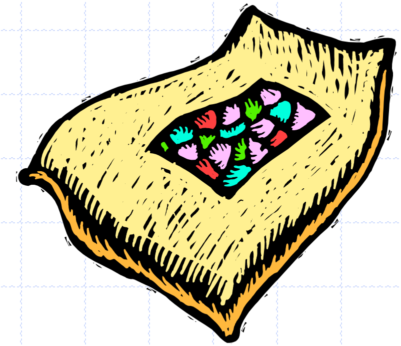
- Optimization Problems
  - We have some problem instance x that has many feasible "solutions".
  - We are trying to minimize (or maximize) some cost function c(S) for a "solution" S to x. For example,
    - Finding a minimum spanning tree of a graph
    - Finding a smallest vertex cover of a graph
    - Finding a smallest traveling salesperson tour in a graph
- An approximation produces a solution T
  - T is a **k-approximation** to the optimal solution OPT if $c(T)/c(OPT) \leq k$ (assuming a min. prob.; a maximization approximation would be the reverse)

Polynomial-Time Approximation Schemes

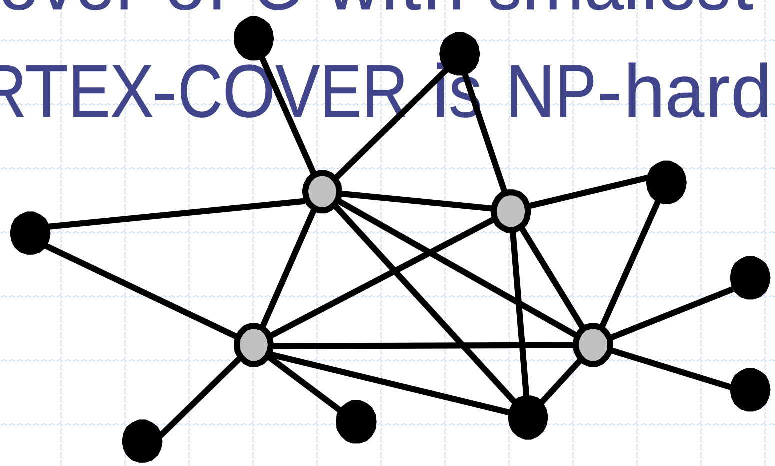


- A problem L has a **polynomial-time approximation scheme (PTAS)** if it has a polynomial-time $(1+\varepsilon)$-approximation algorithm, for any fixed $\varepsilon > 0$ (this value can appear in the running time).
- 0/1 Knapsack has a PTAS, with a running time that is $O(n^3/\varepsilon)$. Please see §13.4.1 in Goodrich-Tamassia for details.

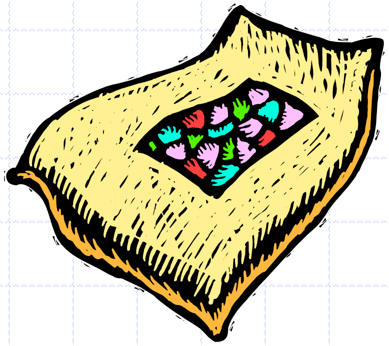
Vertex Cover



- A **vertex cover** of graph G=(V,E) is a subset W of V, such that, for every (a,b) in E, a is in W or b is in W.
- OPT-VERTEX-COVER: Given an graph G, find a vertex cover of G with smallest size.
- OPT-VERTEX-COVER is NP-hard.



A 2-Approximation for Vertex Cover

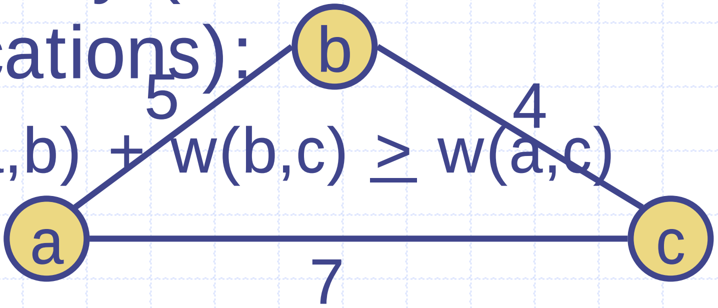


- Every chosen edge e has both ends in C
- But e must be covered by an optimal cover; hence, one end of e must be in OPT
- Thus, there is at most twice as many vertices in C as in OPT.
- That is, C is a 2-approx. of OPT
- Running time: O(m)

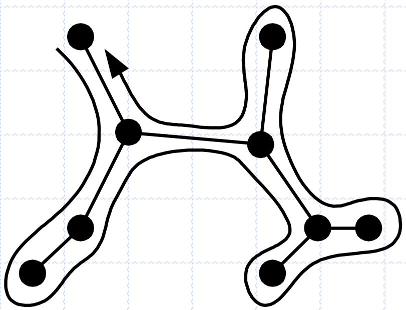
Algorithm *VertexCoverApprox*(*G*)
    Input graph *G*
    Output a vertex cover *C* for *G*
    C ← empty set
    H ← *G*
    while *H* has edges
        e ← ***H.removeEdge***(***H.anEdge***())
        v ← ***H.origin(e)***
        w ← ***H.destination(e)***
        C.add(*v*)
        C.add(*w*)
        for each *f* incident to *v* or *w*
            H.removeEdge(f)
    return *C*

Special Case of the Traveling Salesperson Problem

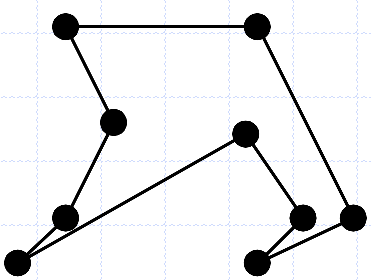
- OPT-TSP: Given a complete, weighted graph, find a cycle of minimum cost that visits each vertex.
  - OPT-TSP is NP-hard
  - Special case: edge weights satisfy the triangle inequality (which is common in many applications):
    - $w(a,b) + w(b,c) \geq w(a,c)$



A 2-Approximation for TSP Special Case



Euler tour *P* of MST *M*



Output tour *T*

Algorithm *TSPApprox*(*G*)

  Input weighted complete graph ***G***, satisfying the triangle inequality

  Output a TSP tour ***T*** for ***G***

  M $\leftarrow$ a minimum spanning tree for ***G***

  P $\leftarrow$ an Euler tour traversal of ***M***, starting at some vertex ***s***

  T $\leftarrow$ empty list

  for each vertex ***v*** in ***P*** (in traversal order)

    if this is ***v***'s first appearance in ***P*** **then**

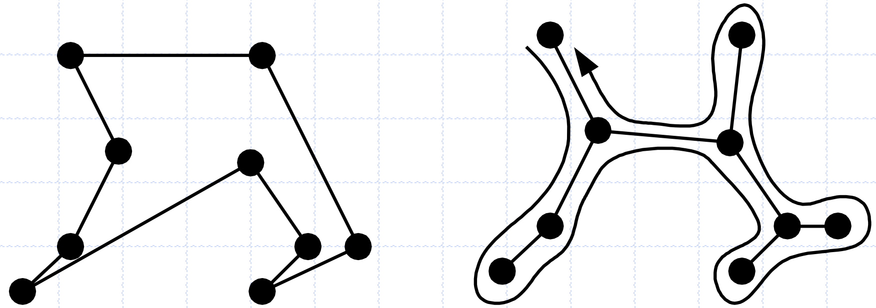
      T.insertLast(***v***)

  T.insertLast(***s***)

  return ***T***

A 2-Approximation for TSP Special Case - Proof

- The optimal tour is a spanning tour; hence $|M| \leq |OPT|$.
- The Euler tour P visits each edge of M twice; hence $|P| = 2|M|$
- Each time we shortcut a vertex in the Euler Tour we will not increase the total length, by the triangle inequality ($w(a,b) + w(b,c) \geq w(a,c)$); hence, $|T| \leq |P|$.
- Therefore, $|T| \leq |P| = 2|M| \leq 2|OPT|$



Output tour T
(at most the cost of P)

Euler tour P of MST M
(twice the cost of M)

Optimal tour OPT
(at least the cost of MST M)