Winter School on Optimization Techniques
December 15-20, 2016
Organized by ACMU, ISI and IEEE CEDA

NP Completeness and Approximation Algorithms

Susmita Sur-Kolay
Advanced Computing and Microelectronic Unit
Indian Statistical Institute

Tractability

- Some problems are *intractable*:
  as they grow large, we are unable to solve them in reasonable time
- What constitutes reasonable time?
  - » Standard working definition: *polynomial time*
  - » On an input of size n the worst-case running time is $O(n^k)$ for some constant k
  - » $O(n^2)$, $O(n^3)$, $O(1)$, $O(n \lg n)$, $O(2^n)$, $O(n^n)$, $O(n!)$
  - » Polynomial time: $O(n^2)$, $O(n^3)$, $O(1)$, $O(n \lg n)$
  - » Not in polynomial time: $O(2^n)$, $O(n^n)$, $O(n!)$

Polynomial-Time Algorithms

- Are some problems solvable in polynomial time?
  - » Of course: many algorithms we've studied provide polynomial-time solutions to some problems
- Are all problems solvable in polynomial time?
  - » No: Turing's "Halting Problem" is not solvable by any computer, no matter how much time is given
- Most problems that do not yield polynomial-time algorithms are either optimization or decision problems.

Optimization/Decision Problems

- Optimization Problems
  - » An optimization problem is one which asks, “What is the optimal solution to problem X?”
  - » Examples:
    - v Minimum Spanning Tree
    - v 0-1 Knapsack
    - v Fractional Knapsack
- Decision Problems
  - » An decision problem is one with yes/no answer
  - » Examples:
    - v Does a graph G have a MST of weight ≤ W?

Optimization/Decision Problems

- An optimization problem tries to find an optimal solution
- A decision problem tries to answer a yes/no question
- Many problems will have decision and optimization versions
  - Eg: Traveling salesman problem
    - optimization: find hamiltonian cycle of minimum weight
    - decision: is there a hamiltonian cycle of weight ≤ k
- Some problems are decidable, but *intractable*: as they grow large, we are unable to solve them in reasonable time
  - *Is there a polynomial-time algorithm that solves the problem?*

The Class *P*

P: the class of decision problems that have polynomial-time deterministic algorithms.

- » That is, they are solvable in $O(p(n))$, where $p(n)$ is a polynomial on n
- » A deterministic algorithm is (essentially) one that always computes the correct answer

Why polynomial?

- » if not, very inefficient
- » nice closure properties
  - v *the sum and composition of two polynomials are always polynomials too*

Sample Problems in P

- Gaussian Elimination
- Sorting
- Minimum Spanning Tree
- Shortest Path
- Maximum Matching
- Maximum Flow
- String Matching
- Others?

Problems that Cross the Line

- What if a problem has:
  - An exponential upper bound
  - A polynomial lower bound

- We have only found **exponential** algorithms, so it appears to be **intractable.**

- But... we can't **prove** that an exponential solution is needed, we can't **prove** that a polynomial algorithm cannot be developed, so we **can't say the problem is intractable...**

The class *NP*

NP: the class of decision problems that are solvable in polynomial time on a *nondeterministic* machine (or with a nondeterministic algorithm)

- (A *determinstic* computer is what we know)
- A *nondeterministic* computer is one that can "guess" the right answer or solution
  - » Think of a nondeterministic computer as a parallel machine that can freely spawn ***an infinite number*** of processes
- Thus *NP* can also be thought of as the class of problems
  - » whose solutions can be verified in polynomial time
- Note that *NP* stands for "Nondeterministic Polynomial-time"

Sample Problems in NP

- Gaussian Elimination
- Sorting
- Minimum Spanning Tree
- Shortest Path
- Maximum Matching
- Maximum Flow
- String Matching
- Others?
  - Hamiltonian Cycle (Traveling Salesman)
  - Satisfiability (SAT)
  - Graph Coloring
  - Integer Linear Programming

Determinism vs. Nondeterminism

- **Deterministic** algorithms (like those that a computer executes) make decisions based on information.
- **Nondeterministic** algorithms produce an answer by a series of "correct guesses"

NP and P

- **What is NP?**
- NP is the set of all decision problems (question with yes-or-no answer) for which the 'yes'-answers can be **verified** in polynomial time (O(n^k) where n is the problem size, and k is a constant) by a deterministic Turing machine. Polynomial time is sometimes used as the definition of *fast* or *quickly*.
- **What is P?**
- P is the set of all decision problems which can be **solved** in polynomial time by a deterministic Turing machine. Since it can solve in polynomial time, it can also be verified in polynomial time. Therefore P is a subset of NP.

Does Non-Determinism matter?

Finite Automata?

No!

DFA ≈ NFA

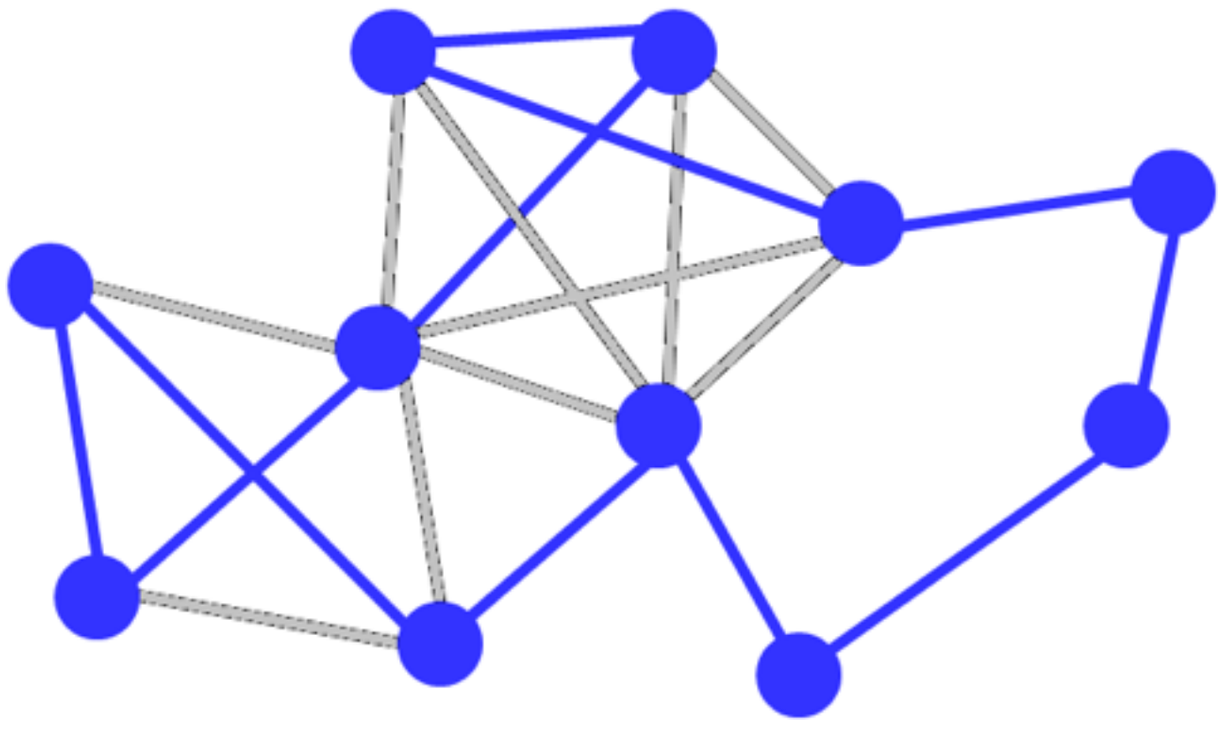
Push Down Automata?

Yes!

DFA not ≈ NFA

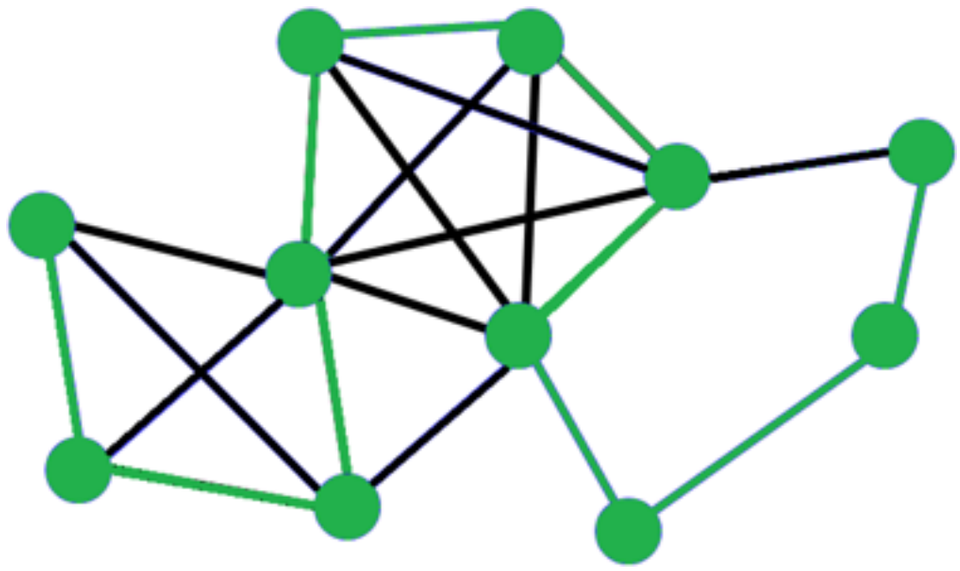
Traveling Salesperson Problem

- You have to visit *n* cities
- You want to make the shortest trip
- How could you do this?
- What if you had a machine that could guess?

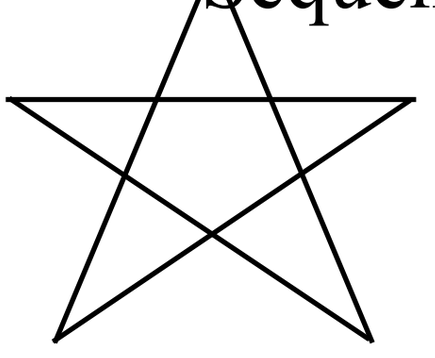


Hamiltonian Cycle and Hamiltonian Path

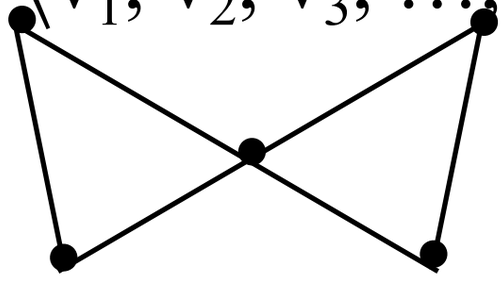
- Given: a directed graph G = (V, E), determine a simple cycle that contains each vertex in V
  - Each vertex can only be visited once
- Certificate:
  - Sequence: $\langle v_1, v_2, v_3, \ldots, v_{|V|} \rangle$



Hamiltonian Path



hamiltonian



not hamiltonian

The *Satisfiability* (SAT) Problem

- *Satisfiability* (SAT):
  - » Given a Boolean expression on n variables, can we assign values such that the expression is TRUE?
  - – Ex: $((x_1 \rightarrow x_2) \vee \leftarrow((\leftarrow x_1 \leftrightarrow x_3) \vee x_4)) \wedge \leftarrow x_2$
  - » Seems simple enough, but no known deterministic polynomial time algorithm exists
  - » Easy to verify in polynomial time!
  - » SAT was the first problem shown to be NP-complete! – Cook-Levin's Theorem (1971)

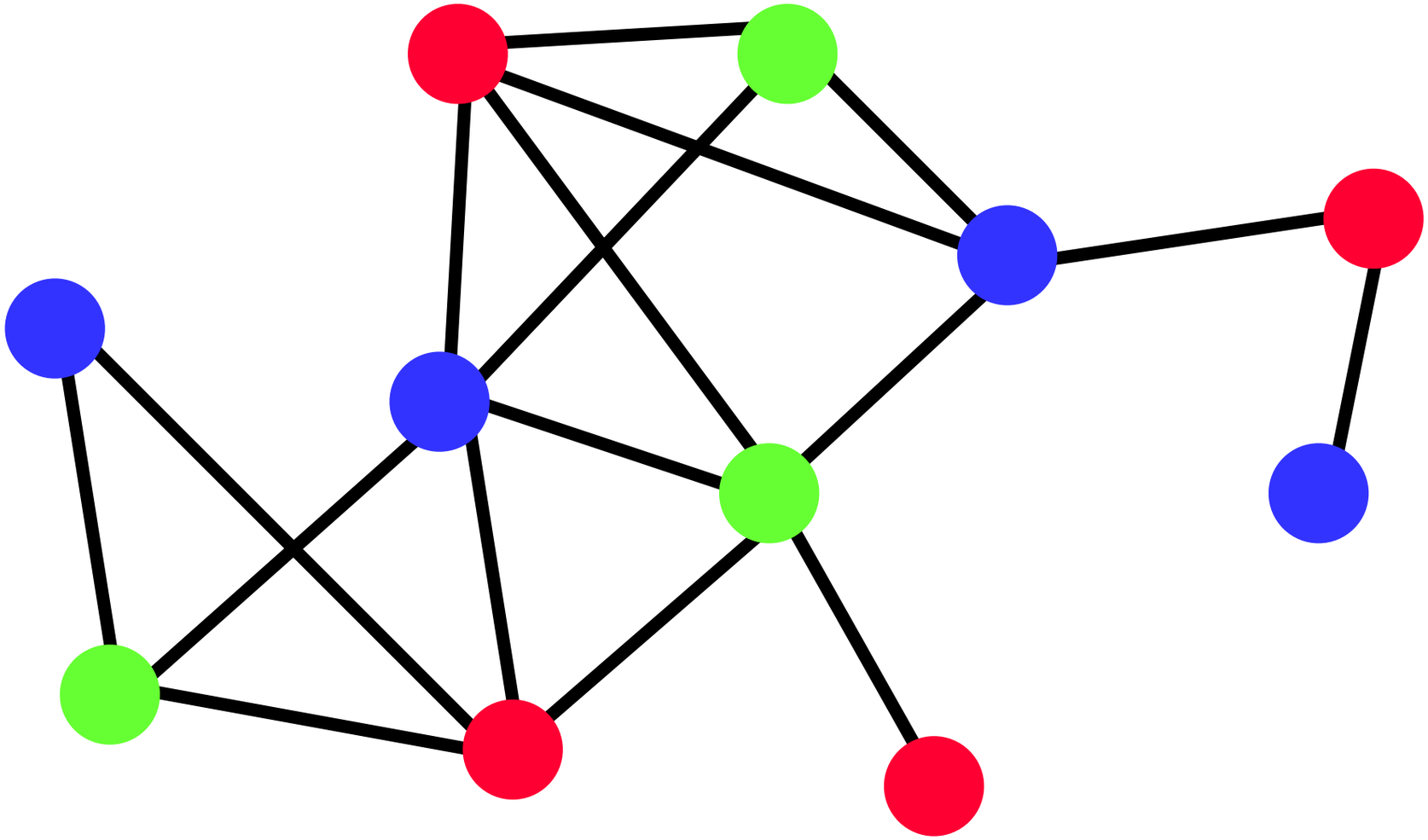
Class Scheduling Problem

- With N teachers with certain hour restrictions M classes to be scheduled, can we:
  - Schedule all the classes
  - Make sure that no two teachers teach the same class at the same time
  - No teacher is scheduled to teach two classes at once

Certificates

- **Returning true**: in order to show that the schedule can be made, we only have to show one schedule that works
  - This is called a **certificate.**
- **Returning false**: in order to show that the schedule cannot be made, we must test all schedules.

Vertex Coloring

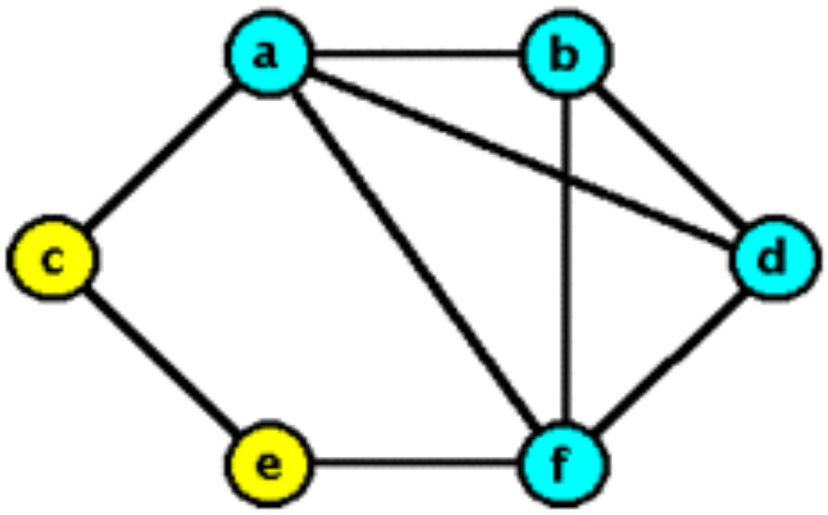


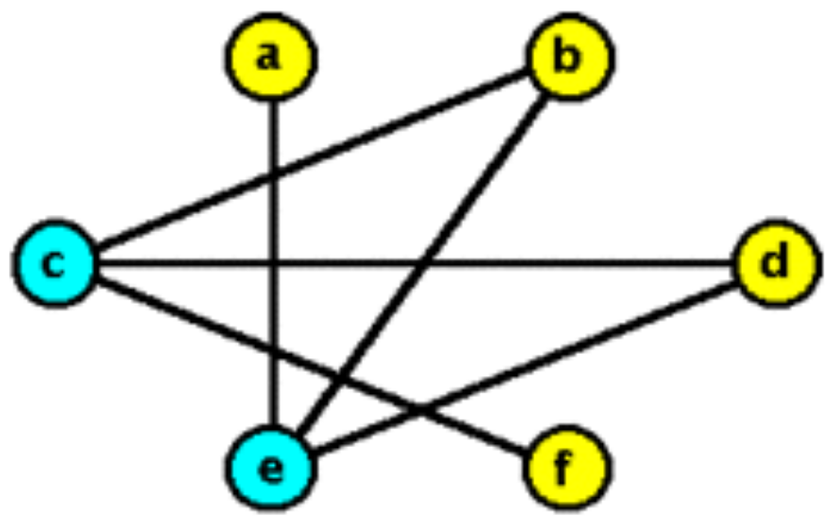
Subset Sum

» SUBSET-SUM={<S,*t*>: S is a set of integers and there exists a S'⊆S such that $t=\sum_{s\in S'}s$.}

Vertex Cover (VC)

- *Given a graph and an integer k, is there a collection of k vertices such that each edge is connected to one of the vertices in the collection?*





Review: P And NP Summary

- **P** = set of problems that can be solved in polynomial time
  - » Examples: Sorting, Fractional Knapsack, …
- **NP** = set of problems for which a solution can be verified in polynomial time
  - » Examples: Sorting, Fractional Knapsack,…, Hamiltonian Cycle, CNF SAT, 3-CNF SAT
- Clearly **P** $\subseteq$ **NP**
- Open question: Does **P = NP**?
  - » Most suspect not
  - » An August 2010 claim of proof that P ≠ NP, by Vinay Deolalikar, researcher at HP Labs, Palo Alto, has flaws

NP-Complete

"NP-Complete" comes from:

- **Nondeterministic Polynomial**
- **Complete - "Solve one, Solve them all"**

There are more NP-Complete problems than provably intractable problems.

Oracles

- If we could make the **'right decision'** at all decision points, then we can determine whether a solution is possible very quickly!
  - If the found solution is valid, then **True**
  - If the found solution is invalid, then **False**
- If we could find the certificates quickly, NP-complete problems would become tractable – **O(N)**
- This (magic) process that can always make the right guess is called an **Oracle**.

NP-Complete Problems

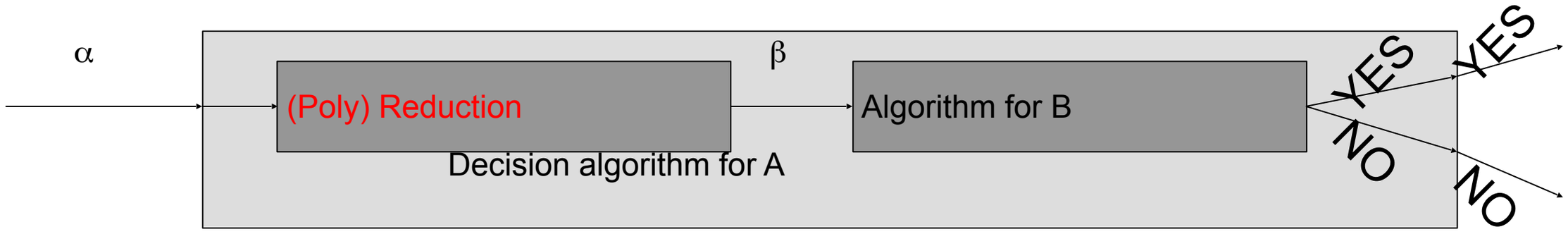
- The upper bound suggests the problem is intractable
- The lower bound suggests the problem is tractable

- The lower bound is linear: O(N)
- They are all reducible to each other
  - If we find a reasonable algorithm (or prove intractability) for one, then we can do it for all of them!

Reduction

- A problem R can be *reduced* to another problem Q if any instance of R can be rephrased to an instance of Q, the solution to which provides a solution to the instance of R
  - » This rephrasing is called a *transformation*
- Intuitively: If R reduces in polynomial time to Q, R is "no harder to solve" than Q
- Example: lcm(m, n) = m * n / gcd(m, n),
  lcm(m,n) problem is reduced to gcd(m, n) problem

Implication of (poly) reduction



1. If decision algorithm for B is poly, so does A.
   A is no harder than B (or B is no easier than A)

2. If A is hard (e.g., NPC), so does B.

3. How to prove a problem B to be NPC ??

   (at first, prove B is in NP, which is generally easy.)
   3.1 find a already proved NPC problem A
   3.2 establish an (poly) reduction from A to B

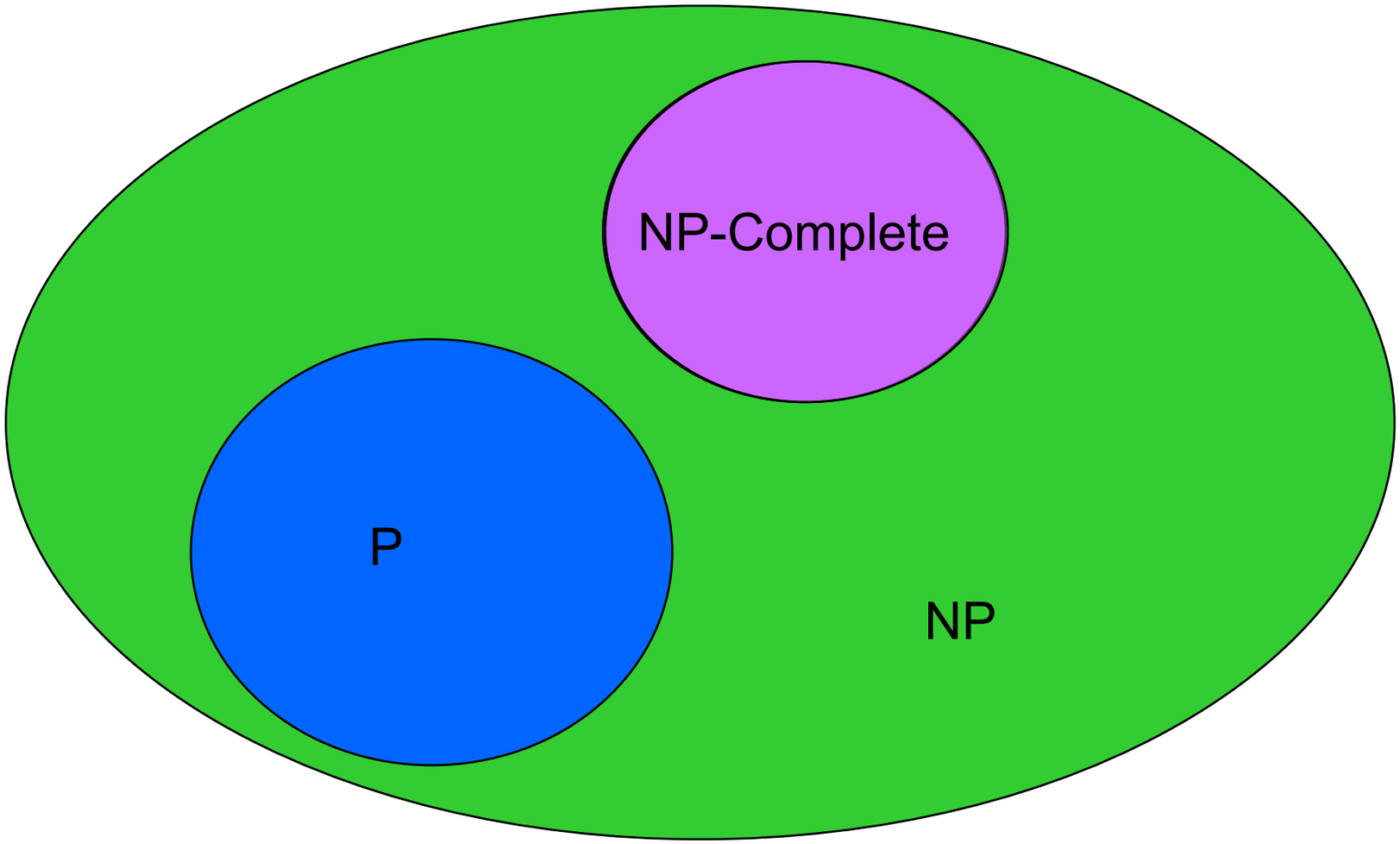
Question: What is and how to prove the first NPC problem?

NP-Complete

- **What is NP-Complete?**
- A problem A that is in NP is also in NP-Complete if and only if every other problem in NP can be quickly (ie. in polynomial time) transformed into A. In other words:
- x is in NP, and
- Every problem in NP is reducible to A
- So what makes NP-Complete so interesting is that if any one of the NP-Complete problems was to be solved quickly then all NP problems can be solved quickly

NP-Completeness

- How would you define NP-Complete?
- They are the “hardest” problems in NP



NP-complete problems

- A decision problem *D* is *NP*-complete iff
  1. *D* ε *NP*
  2. every problem in *NP* is polynomial-time reducible to *D*

- Cook's theorem (1971): CNF-sat is *NP*-complete

NP-Hard

- **What is NP-Hard?**
- NP-Hard are problems that are at least as hard as the hardest problems in NP. Note that NP-Complete problems are also NP-hard. However not all NP-hard problems are NP (or even a decision problem), despite having 'NP' as a prefix. That is the NP in NP-hard does not mean 'non-deterministic polynomial time'. Yes this is confusing but its usage is entrenched and unlikely to change.

NP-Hard and NP-Complete

- If R is *polynomial-time reducible* to Q, we denote this R $\leq_p$ Q
- Definition of NP-Hard and NP-Complete:
  - » If all problems R ε **NP** are *polynomial-time* reducible to Q, then Q is *NP-Hard*
  - » We say Q is *NP-Complete* if Q is NP-Hard and Q ε **NP**
- If R $\leq_p$ Q and R is NP-Hard, Q is also NP-Hard (why?)

NP-naming convention

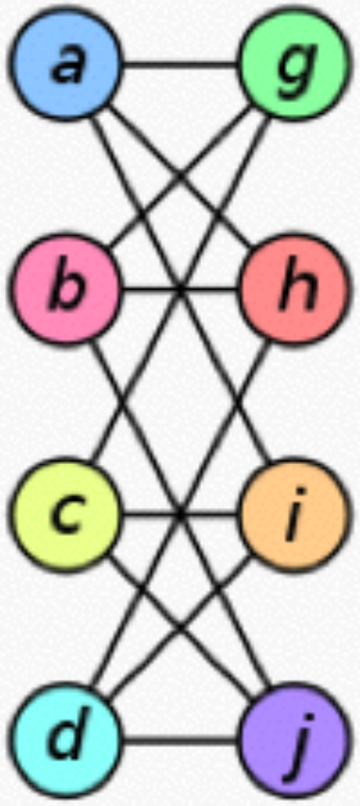
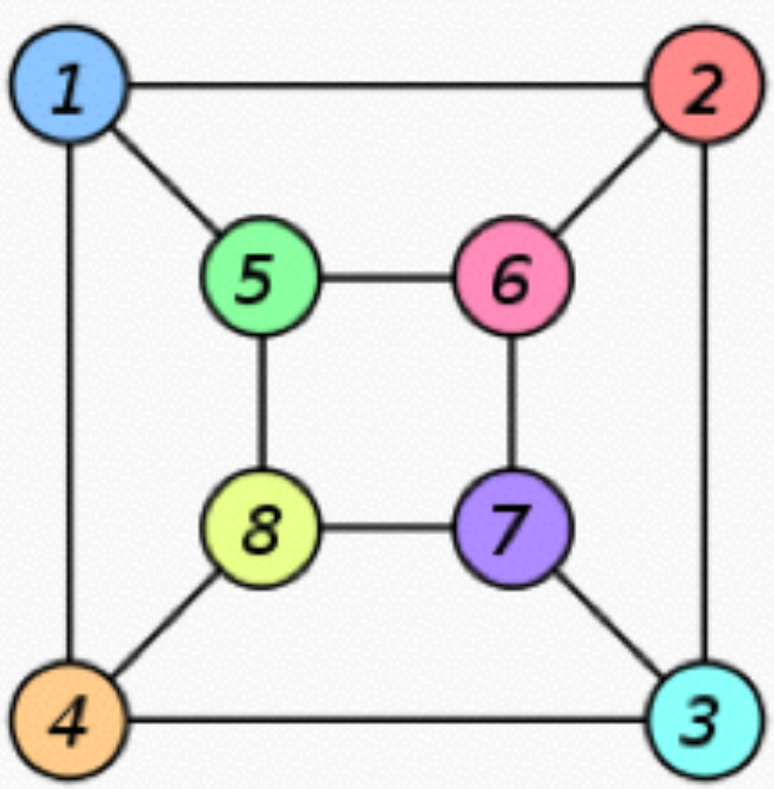
- NP-complete - means problems that are 'complete' in NP, i.e. the most difficult to solve in NP
- NP-hard - stands for 'at least' as hard as NP (but not necessarily in NP);
- NP-easy - stands for 'at most' as hard as NP (but not necessarily in NP);
- NP-equivalent - means equally difficult as NP, (but not necessarily in NP);

Pair Programming Problem

- With N students and K projects, where N is even, can we:
  - Assign pairs of students to each project
  - Every student works on every project
  - No student has the same partner more than once
- Is this an NP-complete problem?

Graph isomorphism

- Graph isomorphism is in NP; but is it NP-complete?

Graph G	Graph H	An isomorphism between G and H
		$f(a) = 1$ $f(b) = 6$ $f(c) = 8$ $f(d) = 3$ $f(g) = 5$ $f(h) = 2$ $f(i) = 4$ $f(j) = 7$

Proving NP-Completeness

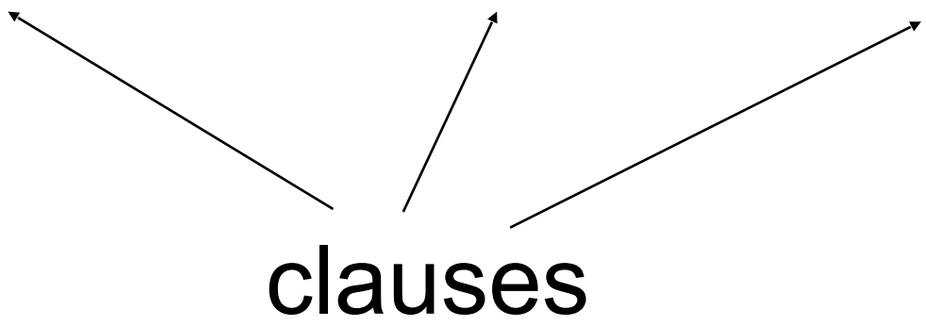
- Show that the problem is in NP. (i.e. Show that a certificate can be verified in polynomial time.)
- Assume it is not NP complete
- Show how to convert an existing NPC problem into the problem that we are trying to show is NP Complete (in polynomial time).

- If we can do it we've done the proof!
- Why?
- If we can turn an existing NP-complete problem into our problem in polynomial time... $\rightarrow|\leftarrow$

CNF Satisfiability

- CNF is a special case of SAT
- Φ is in "Conjuctive Normal Form" (CNF)
  - "AND" of expressions (i.e., clauses)
  - Each clause contains only "OR"s of the variables and their complements

E.g.: $\Phi = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_2)$



CNF satisfiability

- This problem is in *NP*. Nondeterministic algorithm:
  - » Guess truth assignment
  - » Check assignment to see if it satisfies CNF formula

- Example:

  $(A \vee \neg B \vee \neg C) \wedge (\neg A \vee B) \wedge (\neg B \vee D \vee F) \wedge (F \vee \neg D)$

- Truth assignments:

	A	B	C	D	E	F
v	0	1	1	0	1	0
v	1	0	0	0	0	1
v	1	1	0	0	0	1
v	... (how many more?)					

Checking phase: $\Theta(n)$

3-CNF Satisfiability

A subcase of CNF problem:

– Contains three clauses

- *E.g.:*

$$\Phi = (x_1 \vee \leftarrow x_1 \vee \leftarrow x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\leftarrow x_1 \vee \leftarrow x_3 \vee \leftarrow x_4)$$

- **3-CNF** is NP-Complete

- Interestingly enough, **2-CNF** is in P!

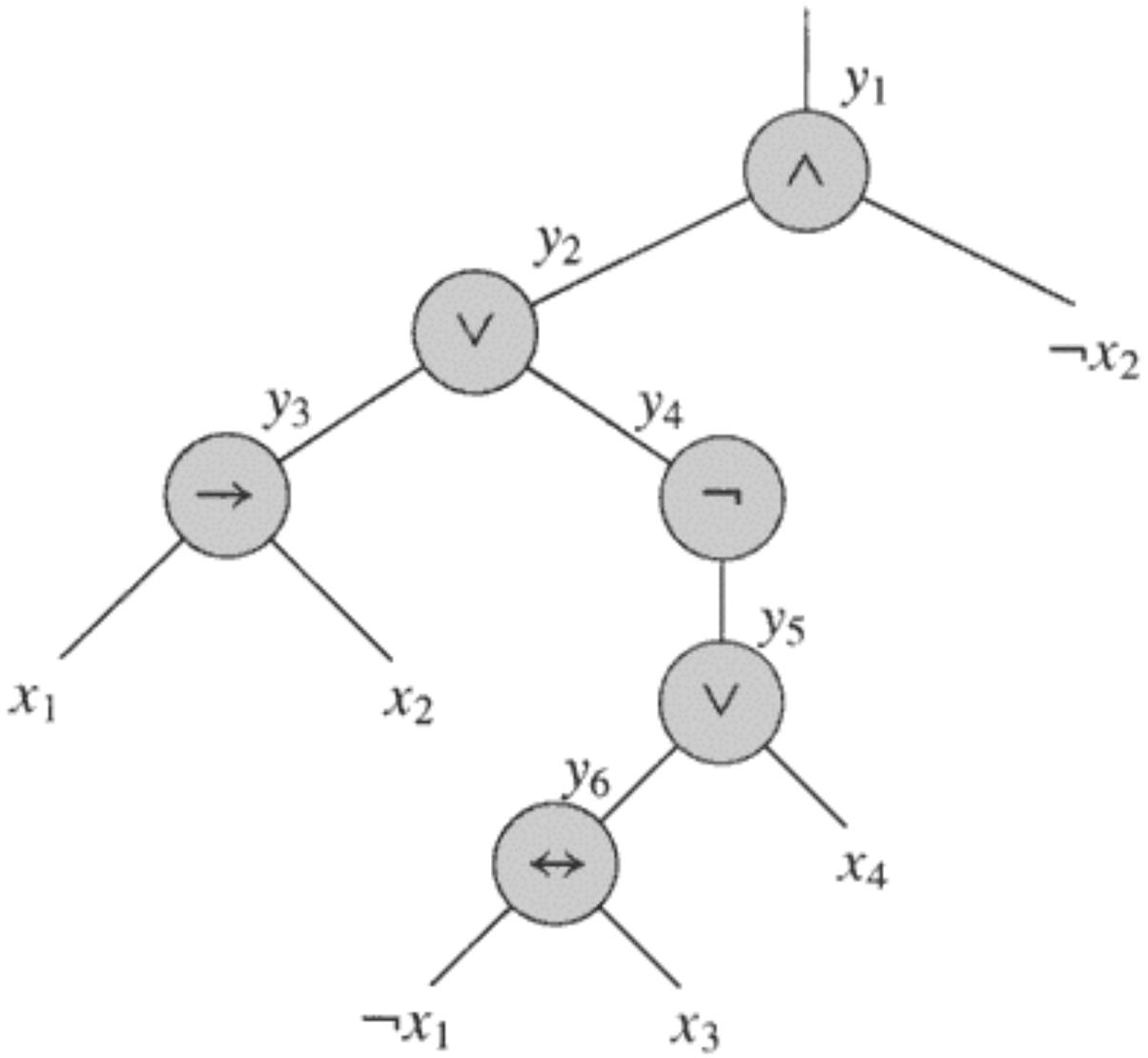
3-CNF-SAT is NP-complete

» Proof: 3-CNF-SAT ∈NP. Easy.

– 3-CNF-SAT is NP-hard. (show SAT $\leq_p$3-CNF-SAT)

- Suppose ϕ is any boolean formula, Construct a binary 'parse' tree, with literals as leaves and connectives as internal nodes.
- Introduce a variable y_i for the output of each internal nodes.
- Rewrite the formula to ϕ' as the AND of the root variable and a conjunction of clauses describing the operation of each node.
- The result is that in ϕ', each clause has at most three literals.
- Change each clause into conjunctive normal form as follows:
  – Construct a truth table, (small, at most 8 by 4)
  – Write the disjunctive normal form for all true-table items evaluating to 0
  – Using DeMorgan law to change to CNF.
- The resulting ϕ" is in CNF but each clause has 3 or less literals.
- Change 1 or 2-literal clause into 3-literal clause as follows:
  – If a clause has one literal l, change it to $(l \vee p \vee q) \wedge (l \vee p \vee \leftarrow q) \wedge (l \vee \leftarrow p \vee q) \wedge (l \vee \leftarrow p \vee \leftarrow q)$.
  – If a clause has two literals $(l_1 \vee l_2)$, change it to $(l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \leftarrow p)$.

Binary parse tree for $\phi=((x_1 \rightarrow x_2) \vee \leftarrow((\leftarrow x_1 \leftrightarrow x_3) \vee x_4)) \wedge \leftarrow x_2$



$$\begin{aligned}\phi' = {} & y_1 \wedge (y_1 \leftrightarrow (y_2 \wedge \leftarrow x_2)) \\ & \wedge (y_2 \leftrightarrow (y_3 \vee y_4)) \\ & \wedge (y_4 \leftrightarrow \leftarrow y_5) \\ & \wedge (y_3 \leftrightarrow (x_1 \rightarrow x_2)) \\ & \wedge (y_5 \leftrightarrow (y_6 \vee x_4)) \\ & \wedge (y_6 \leftrightarrow (\leftarrow x_1 \leftrightarrow x_3))\end{aligned}$$

Figure 34.11 The tree corresponding to the formula $\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$.

y_1	y_2	x_2	$(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$
1	1	1	0
1	1	0	1
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	1
0	0	0	1

Disjunctive Normal Form:

$\phi_i' = (y_1 \wedge y_2 \wedge x_2) \vee (y_1 \wedge \leftarrow y_2 \wedge x_2)$
$\vee (y_1 \wedge \leftarrow y_2 \wedge \leftarrow x_2) \vee (\leftarrow y_1 \wedge y_2 \wedge \leftarrow x_2)$

Conjunctive Normal Form:

$\phi_i'' = (\leftarrow y_1 \vee \leftarrow y_2 \vee \leftarrow x_2) \wedge (\leftarrow y_1 \vee y_2 \vee \leftarrow x_2)$
$\wedge (\leftarrow y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \leftarrow y_2 \vee x_2)$

Figure 34.12 The truth table for the clause $(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$.

3-CNF is NP-complete

≈ ϕ and reduced 3-CNF are equivalent:

- From ϕ to ϕ' , keep equivalence.
- From ϕ' to ϕ'' , keep equivalence.
- From ϕ'' to final 3-CNF, keep equivalence.

» Reduction is in poly time,

- From ϕ to ϕ' , introduce at most 1 variable and 1 clause per connective in ϕ.
- From ϕ' to ϕ'' , introduce at most 8 clauses for each clause in ϕ'.
- From ϕ'' to final 3-CNF, introduce at most 4 clauses for each clause in ϕ''.

Clique

Clique Problem:

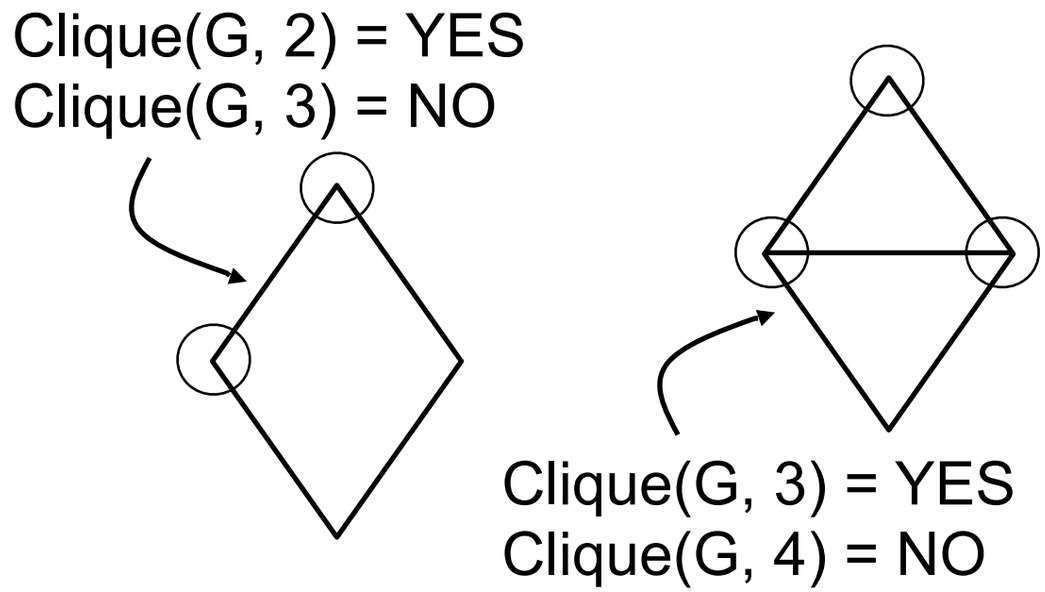
- Undirected graph G = (V, E)
- Clique: a subset of vertices in V all connected to each other by edges in E (i.e., forming a complete graph)
- Size of a clique: number of vertices it contains

Optimization problem:

- Find a clique of maximum size

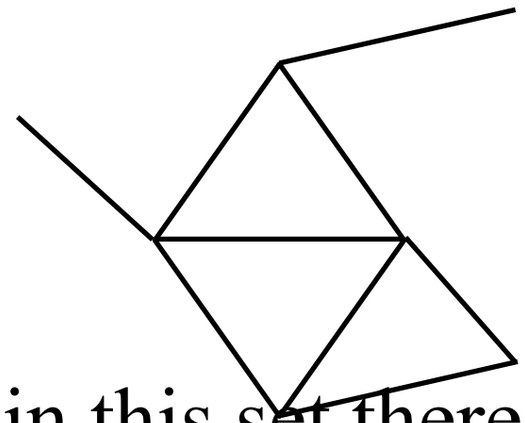
Decision problem:

- Does G have a clique of size k?



Clique Verifier

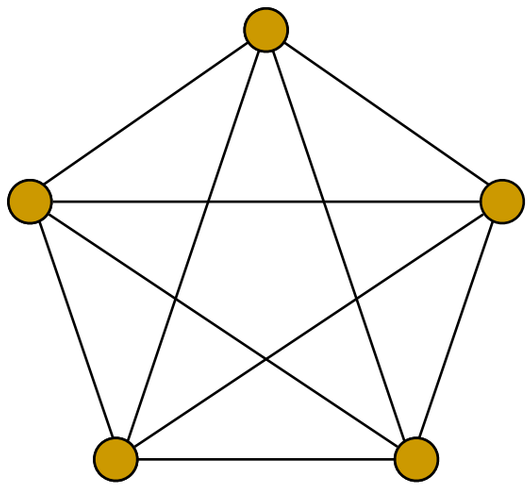
- Given: an undirected graph G = (V, E)
- Problem: Does G have a clique of size k?
- Certificate:
  - A set of k nodes



- Verifier:
  - Verify that for all pairs of vertices in this set there exists an edge in E

Example: Clique

- CLIQUE = { <G,k> | G is a graph with a clique of size k }
- A clique is a subset of vertices that are all connected
- Why is CLIQUE in NP?



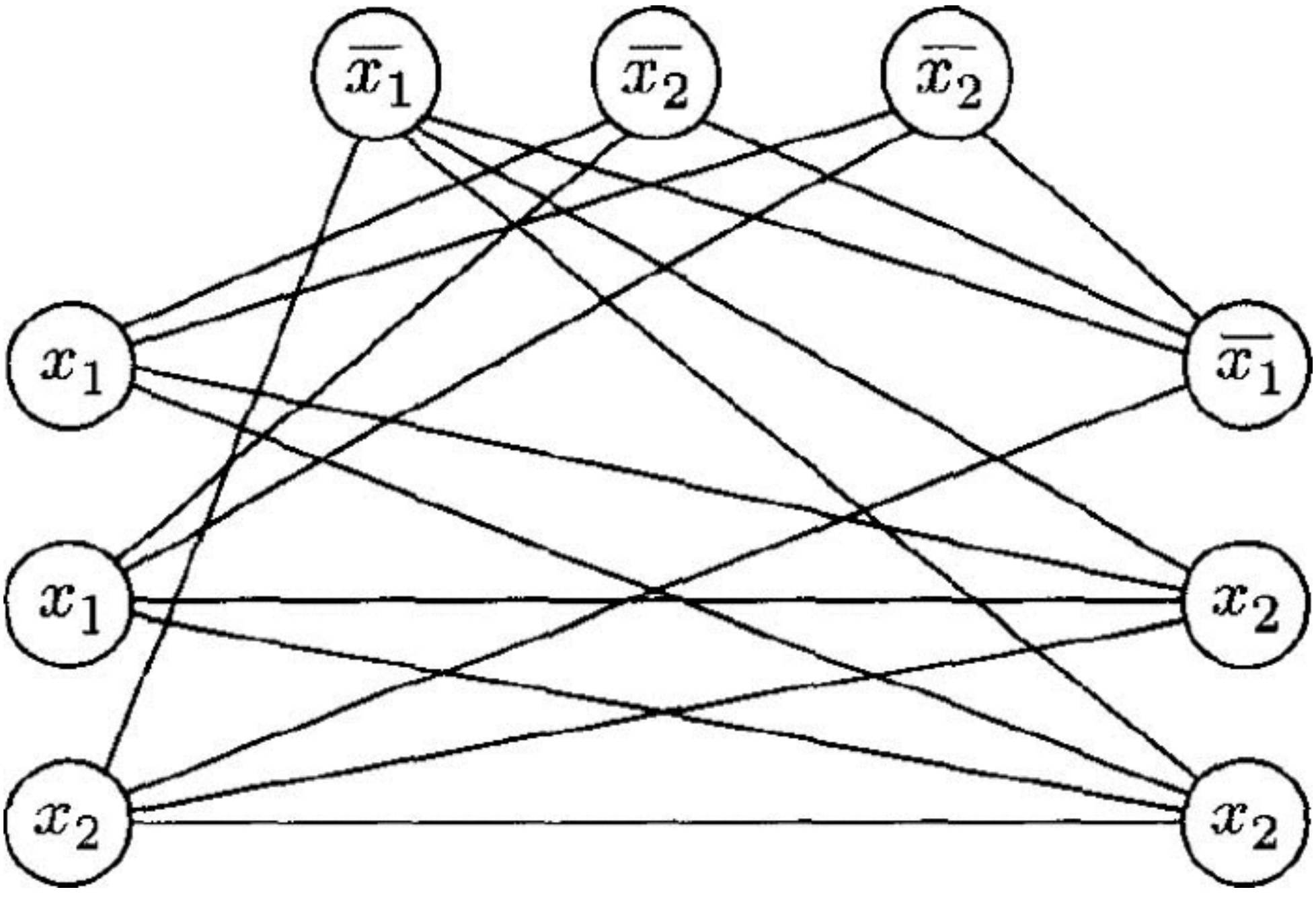
3-CNF $\leq_p$ Clique

- Idea:
  - Construct a graph G such that Φ is satisfiable only if G has a clique of size k

Reduce 3-SAT to Clique

- Pick an instance of 3-SAT, Φ, with k clauses
- Make a vertex for each literal
- Connect each vertex to the literals in other clauses that are not the negation
- Any k-clique in this graph corresponds to a satisfying assignment

$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$

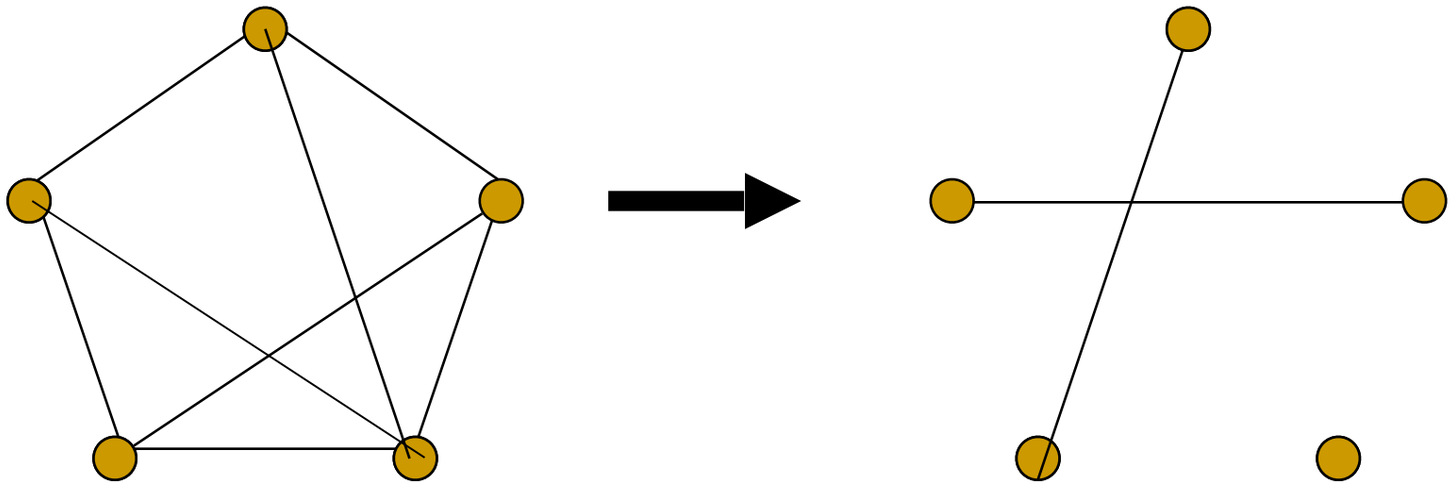


Example: Independent Sett

- INDEPENDENT SET = { <G,k> | where G has an independent set of size k }
- An independent set is a set of vertices that have no edges
- How can we reduce this to clique?

Independent Set to CLIQUE

- This is the *dual problem*!



Subset Sum is NPC

» SUNSET-SUM={<S,*t*>: S is a set of integers and there exists a S'⊆S such that $t=\sum_{s\in S'}s$.}

» SUBSET-SUM is NP-complete. SUBSET-SUM belongs to NP.

– Given a certificate S', check whether *t* is sum of S' can be finished in poly time.

» SUBSET-SUM is NP-hard (show 3-CNF-SAT$\leq_p$SUBSET-SUM).

NP-completeness proof structure

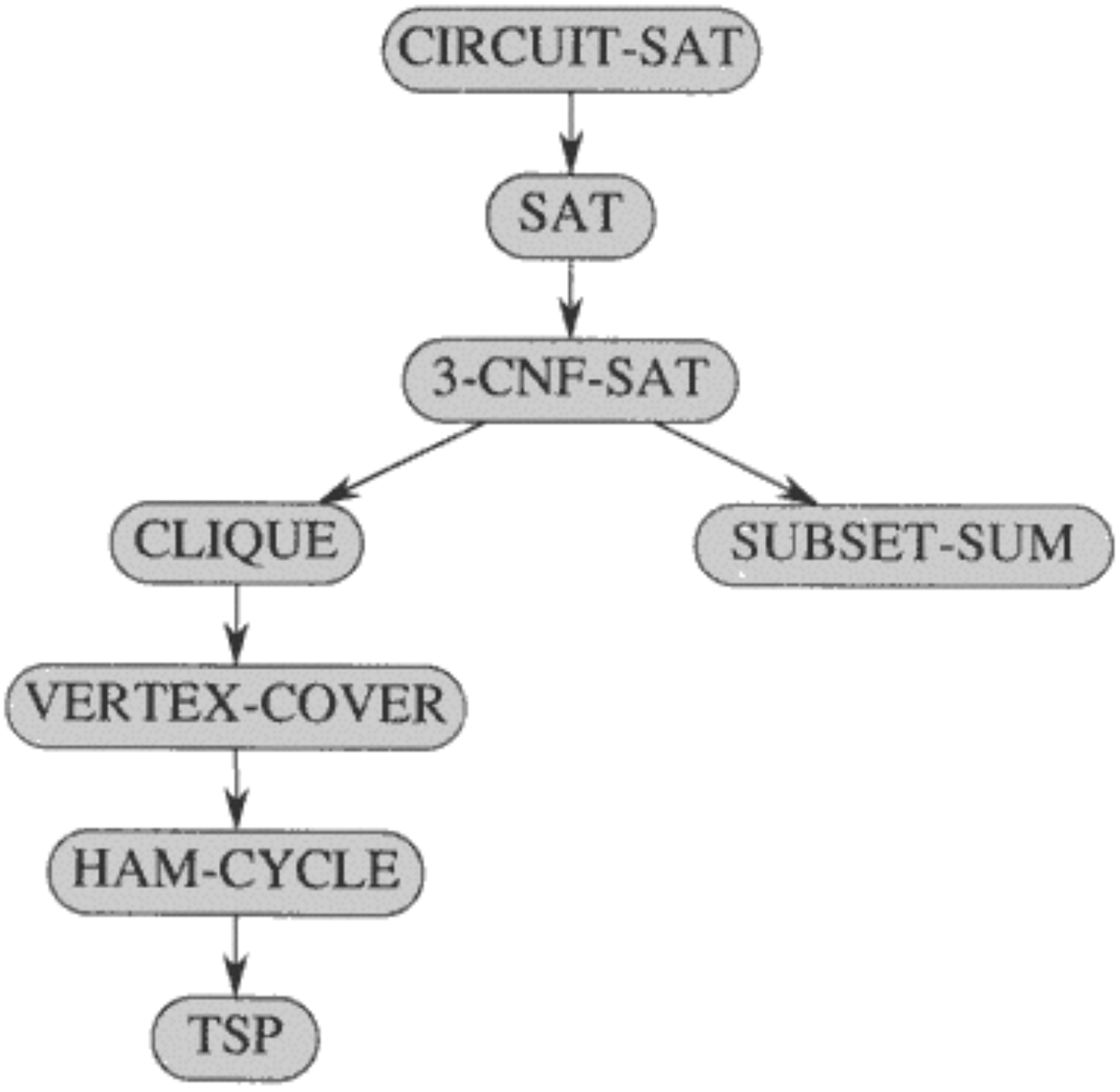


Figure 34.13 The structure of NP-completeness proofs in Sections 34.4 and 34.5. All proofs ultimately follow by reduction from the NP-completeness of CIRCUIT-SAT.

Relation among P, NP and co-NP={L: L ∈NP where L= ∑*-L}

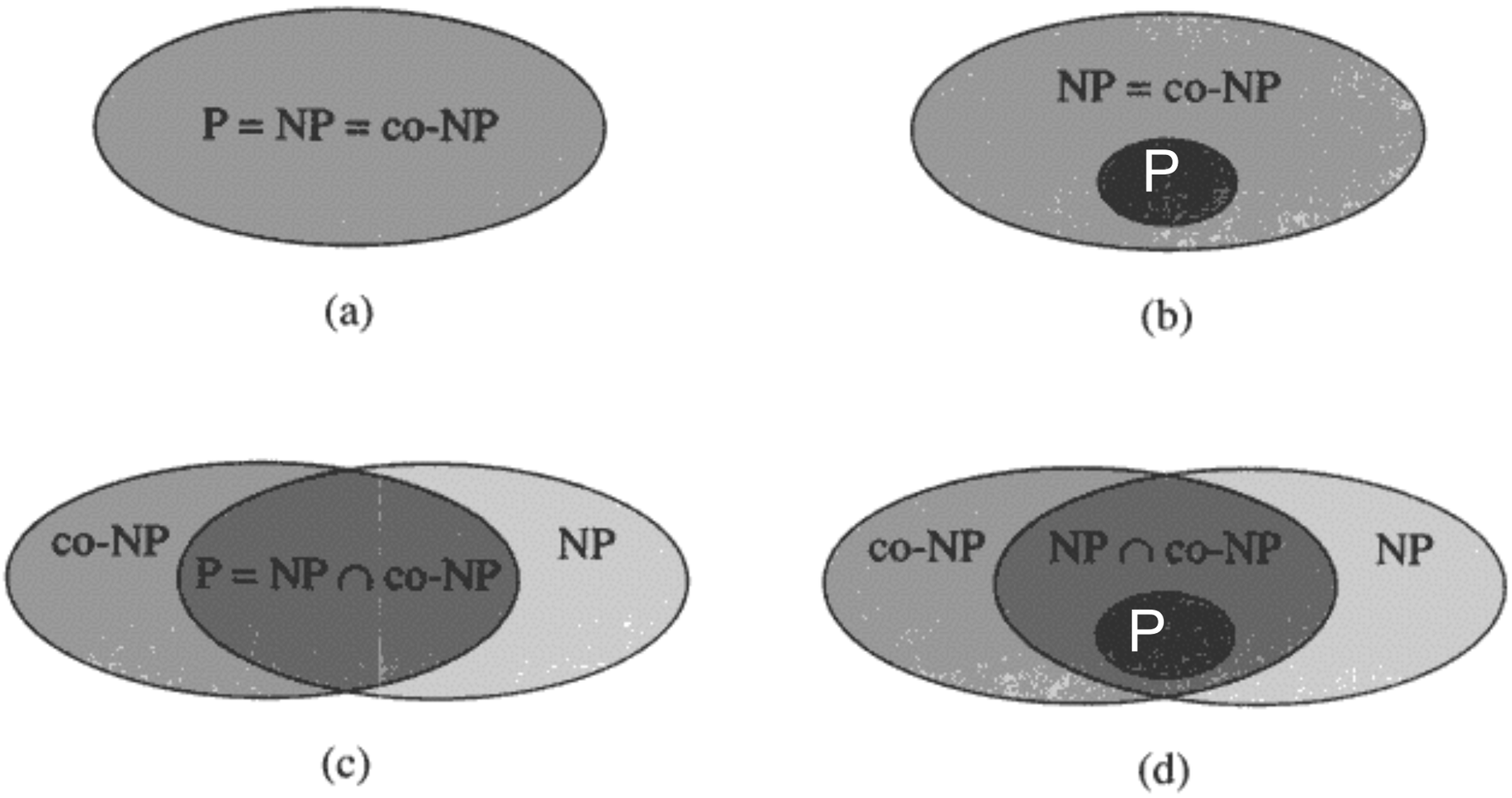


Figure 34.3 Four possibilities for relationships among complexity classes. In each diagram, one region enclosing another indicates a proper-subset relation. **(a)** P = NP = co-NP. Most researchers regard this possibility as the most unlikely. **(b)** If NP is closed under complement, then NP = co-NP, but it need not be the case that P = NP. **(c)** P = NP ∩ co-NP, but NP is not closed under complement. **(d)** NP ≠ co-NP and P ≠ NP ∩ co-NP. Most researchers regard this possibility as the most likely.

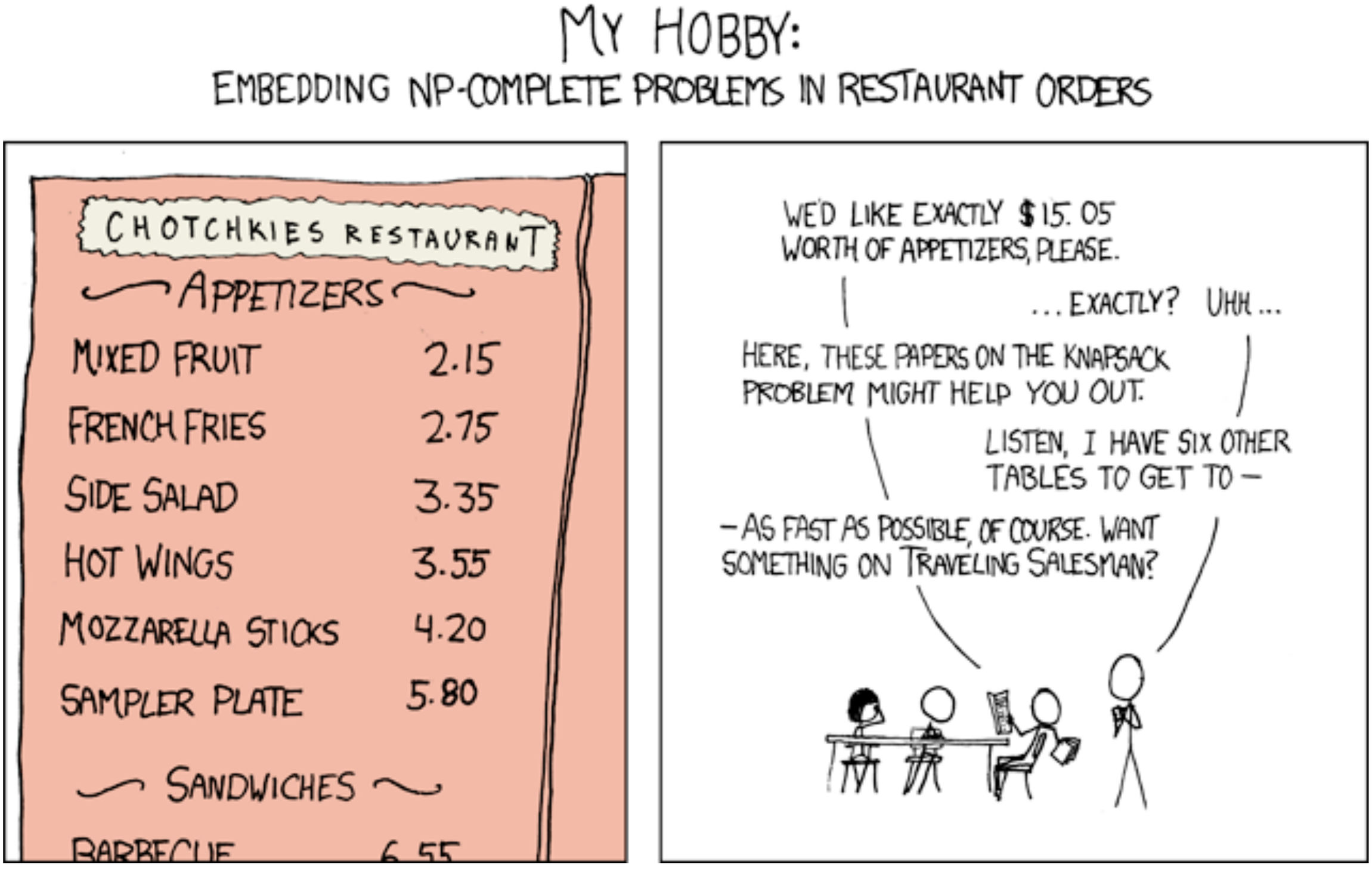
Become Famous!

To get famous in a hurry, for any NP-Complete problem:

- **Raise the lower bound (via a stronger proof)**
- **Lower the upper bound (via a better algorithm)**

They'll be naming buildings after you before you are dead!

http://xkcd.com/287/



Practical Implication

- Given a problem that is known to be NP-Complete
  - Try to solve it by designing a polynomial-time algorithm?
    - Prove P=NP
  - Alleviate the intractability of such problems
    - To make some large instances of the problem solvable
    - To find good approximations

Appendix

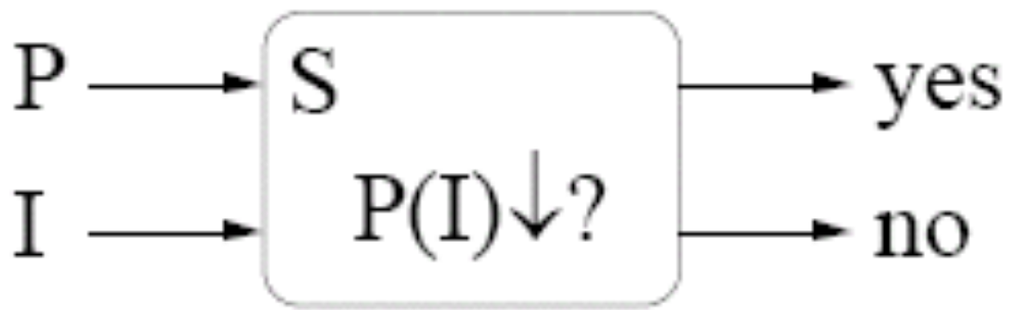
- The following slides are from a document by Dr. G. Robins, University of Virginia

The Halting Problem

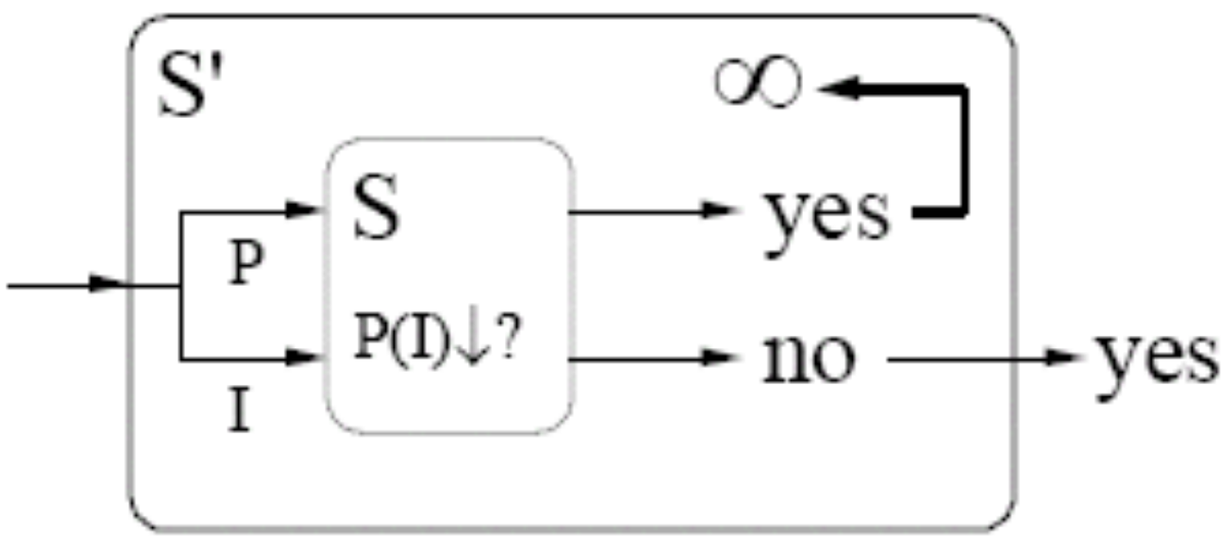
H: Given a program P and input I,
does P halt on I? i.e., does P(I)↓ ?

Thm: H is uncomputable

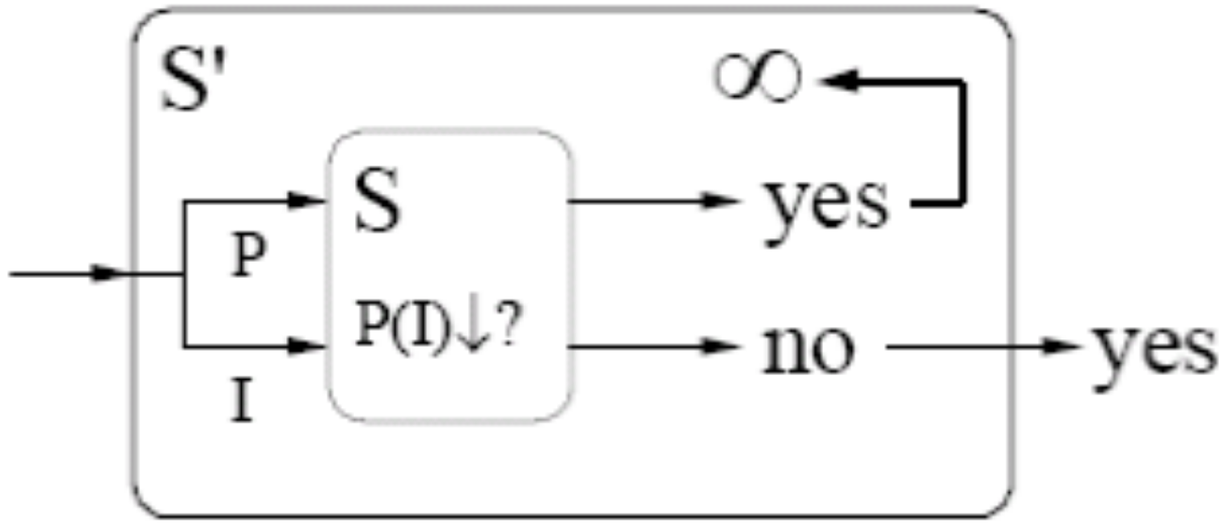
Pf: Assume subroutine S solves H.



Construct:



Analyze:



$S'(S')\downarrow \Rightarrow S'(S')\uparrow$

$S'(S')\uparrow \Rightarrow S'(S')\downarrow$

so, $S'(S')\uparrow \Leftrightarrow S'(S')\downarrow$

a contradiction!

$\Rightarrow$ S does not correctly compute H

But S was an arbitrary subroutine, so

$\Rightarrow$ H is not computable!