

Topic 7: NP-Completeness



Outline

- Overview
- Polynomial time
- Polynomial-time verification
- NP-completeness and reducibility
- NP-completeness and proofs
- NP-complete problems



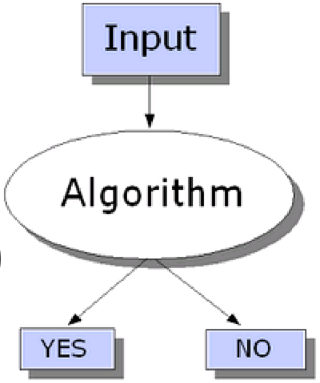
Overview

Polynomial Time vs. Superpolynomial Time

- ***Polynomial-time problem (tractable problems)***:
  - On inputs of size *n*, the problems are solvable in $O(n^k)$ time for some constant *k*.
  - ***Polynomial-time algorithms***: on inputs of size n, their worst-case running time is $O(n^k)$ for some constant *k*.
- ***Superpolynomial-time problem (intractable problems)***:
  - On inputs of size *n*, the problems are not solvable by polynomial-time algorithms.
  - E.g., Halting problem:
    - The **halting problem** is a decision problem of deciding, given a program and an input, whether the program will eventually halt when run with that input, or will run forever.

Optimization Problems vs. Decision Problem

- ***Optimization problems***
  - Each feasible solution has an associated value, and we wish to find a feasible solution with the best value.
  - E.g., **SHORTEST-PATH**
    - Given an undirected graph *G* and vertices *u* and *v*, and we wish to find a path from *u* to *v* that uses the fewest edges.
    - This is the single-pair shortest-path problem in an unweighted, undirected graph.



- **Decision problems (yes-or-no (1-or-0) problems)**
  - A decision problem is a question with a yes-or-no answer, depending on the values of some input parameters.
    - For example, the problem "given two numbers *x* and *y*, does *x* evenly divide *y*?" is a decision problem. The answer can be either 'yes' or 'no', and depends upon the values of *x* and *y*.
  - E.g., **PATH** (decision version of **SHORTEST-PATH**)
    - Given a directed graph *G*, vertices *u* and *v*, and an integer *k*, does a path exist from *u* to *v* consisting of ***at most k edges***?



P, NP, NP-Complete, and NP-Hard

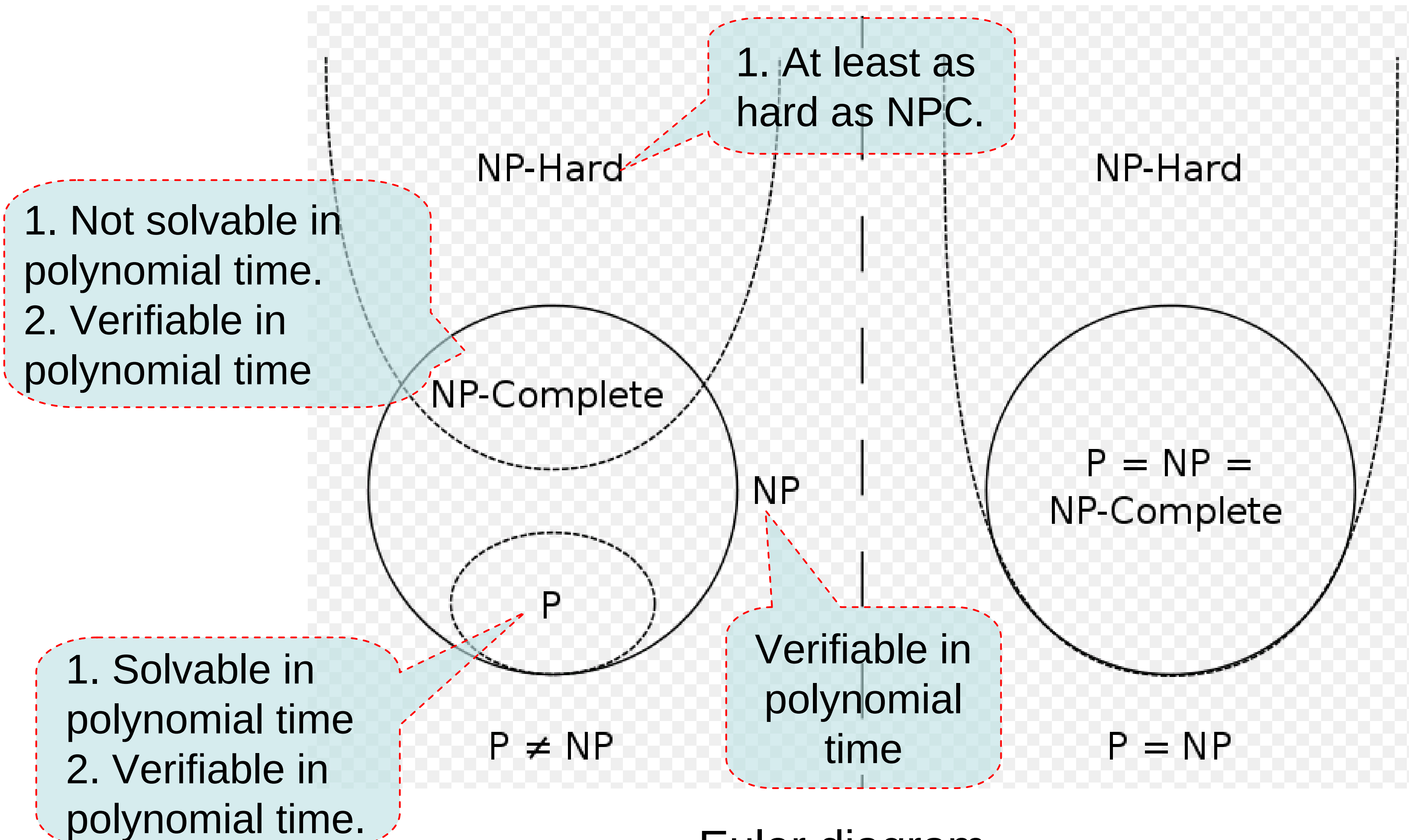
- Class **P** (polynomial-time problems)
  - The problems that are solvable in polynomial time.
- Class **NP** (Nondeterministic polynomial-time problems)
  - The problems that are verifiable in polynomial time.
  - ***NP is the set of problems*** for which the instances where the answer is "yes" have efficiently verifiable proofs of the fact that the answer is indeed "yes".
  - Given a certificate of a solution, then we could verify that the certificate is correct in time polynomial in the size of the input to the problem.
  - E.g.,
    - Given a directed graph G=(V,E), a certificate would be a sequence $<v_1$, v_2, v_3, … $v_{|V|}\}$ of |V| vertices. We can check in polynomial time to see whether the sequence forms a Hamiltonian cycle.
    - For 3-CNF satisfiability, a certification would be an assignment of values to variables.



P, NP, NP-Complete, and NP-Hard (Cont.)

- Class **NPC** (NP-Complete problems)
  - The problems that are not solvable in polynomial time (or intractable), but are verifiable in polynomial time.
  - The hardest problems in class NP.
- Class **NP-Hard**
  - The problems that are at least as hard as the hardest problems in NP (i.e., NPC).
  - A problem *H* is NP-hard <u>if and only if</u> there is an <u>NP-complete</u> problem L that is <u>polynomial time reducible</u> to *H*.
    - Problem *H* is at least as hard as L, because H can be used to solve *L.*
    - Since *L* is NP-complete, and hence the hardest in class NP, also problem *H* is at least as hard as NP, but **<u>H does not have to be in NP and hence does not have to be a decision problem</u>** (even if it is a decision problem, it need not be in NP) .

P, NP, NP-Complete, and NP-Hard (Cont.)



Euler diagram



NP-Complete Problems

- NP-Complete problems are also call NPC problems.
- No polynomial-time algorithm has yet been discovered for an NP-complete problem, except $P = NP$.
  - If any single NP-complete problem can be solved in polynomial time, then every NP-complete problem has a polynomial time algorithm.
  - To become a good algorithm designer, you must understand the rudiments (基本原理) of the theory of NP-completeness.
- Whether P = NP or P $\neq$ NP is an open question.
  - P $\neq$ NP question has been one of the deepest, most perplexing(令人費解的) open research problems in theoretical computer science since it was first posed in 1971.
- Although NP-complete problems are confined to the realm of decision problems, we can take advantage of a convenient relationship between *optimization problems* and *decision problems*.
  - The decision problem is in a sense “easier”, or at least “no harder” than its corresponding optimization problem.
  - ***If we can provide evidence that a decision problem is hard, we also provide evidence that its related optimization problem is hard as well.***



Polynomial-Time vs. NPC Problems

- ***Shortest vs. longest simple paths:***
  - With negative edge weights, we can find shortest paths form a singe source in a directed graph G=(V,E) in O(VE) time. (***Polynomial-time problem***)
  - Finding a longest simple path between two vertices is difficult. Merely determining whether a graph contains a simple path with at least a given number of edges is ***NP-complete***.
- ***Euler tour vs. hamiltonian cycle***:
  - An Euler tour of a connected, directed graph G=(V,E) is a cycle that traverses each edge of G excatly once, although it is allowed to visit each vertex more than once. (Solvable in O(E) time: ***Polynomial-time problem***)
  - A hamiltonian cycle of a directed graph G=(V,E) is a simple cycle that contains each vertex in V: Determining whether a directed graph has a hamiltonian cycle is ***NP-complete***.
    - Note: **Travelling Salesman Problem** (**TSP**) is an NP-hard problem. Given a list of cities and their pairwise distances, the task is to find a shortest possible tour that visits each city exactly once. The Hamiltonian cycle problem is a special case of the traveling salesman problem, obtained by setting the distance between two cities to a finite constant if they are adjacent and infinity otherwise.



Polynomial-Time vs. NPC Problems (Cont.)

- ***2-CNF satisfiability vs. 3-CNF satisfiability***:
  - A boolean formula contains variables whose values are 0 or 1.
    - Boolean connectives such as $\wedge$ (AND), $\vee$ (OR), $\neg$ (NOT), and parentheses.
    - A boolean formula is ***satisfiable*** if there exists some assignment of the values 0 and 1 to its variables that causes it to evaluate to 1.
    - K-conjunctive normal form (k-CNF):
      - If it is the AND of clauses of Ors of exactly k variables or their negations.
      - For example: $(x1 \vee \neg x2) \wedge (\neg x1 \vee x3) \wedge (\neg x2 \vee \neg x3)$ is in 2-CNF. It has the satisfying assignment x1=1, x2=0, x3=1.
    - We can determine in polynomial time to see whether a 2-CNF formula is satisfiable. (***Polynomial-time problem***)
    - Determining whether a 3-CNF formula is satisfiable is ***NP-complete.***

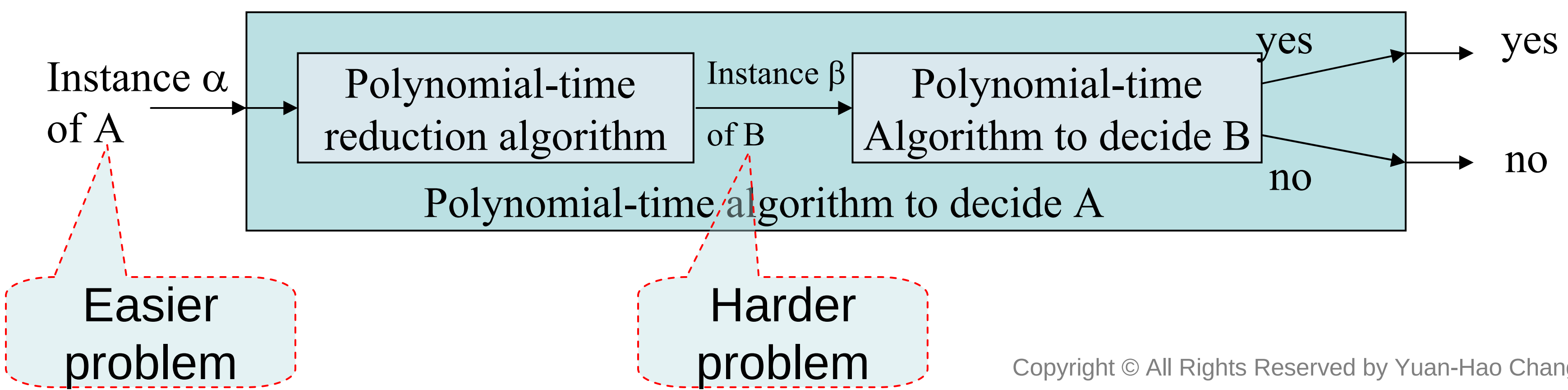


Reductions

- Suppose that there is a different decision problem, say B, that we already know how to solve in polynomial time.
- Suppose that we have a procedure that transforms any instance α of A into some instance β of B with the following characteristics:
  - 1. The transformation takes polynomial time.
  - 2. The answer are the same.
    That is, the answer for α is “yes” if and only if the answer for β is also “yes.”

Reductions (Cont.)

- We can call such a procedure a polynomial-time reduction algorithm and, it provides us a way to solve problem A in polynomial time:
  1. Given an instance α of problem A, use a polynomial-time reduction algorithm to transform it to an instance β of problem B.
  2. Run the polynomial-time decision algorithm for B on the instance β.
  3. Use the answer for β as the answer for α.





Reductions (Cont.)

- NP-completeness is about showing how hard a problem is rather than how easy it is.
- We use polynomial-reductions in the opposite way to show that no polynomial-time algorithm can exist for a particular problem B.:
  - Suppose we have a decision problem A for which we already know that no polynomial-time algorithm can exist.
  - Suppose further that we have a polynomial-time reduction transforming instances of A to instances of B.
- ***A first NP-complete problem***
  - Because the technique of reduction relies on having a problem already known to be NP-complete in order to prove a different problem NP-complete, we need a "first" NPC problem.
    - E.g., ***Circuit-satisfiability problem***.



Polynomial Time



Polynomial Time

Note: A Turing machine takes a tape with a string of symbols on it as an input, and can respond to a given symbol by changing its internal state, writing a new symbol on the tape, shifting the tape right or left to the next symbol, or halting. The inner state of the Turing machine is described by a **finite state machine**. It has been shown that if the answer to a computational problem can be computed in a finite amount of time, then there exists an ***abstract Turing machine*** that can compute it.

- Polynomial time solvable problem are regarded as tractable.
  - Even if the current best algorithm for a problem has a running time of $\Theta(n^{100})$, it is likely that an algorithm with a much better running time will soon be discovered.
  - For many reasonable models of computation, a problem can be solved in one model can be solved in polynomial in another.
    - E.g., the problems solvable in polynomial time by the serial random-access machine are solvable in polynomial time on abstract Turing machines.
  - Polynomial-time solvable problems has a nice closure property.
    - Polynomials are closed under addition, multiplication, and composition.
    - E.g., If an algorithm takes a constant number of calls to polynomial-time subroutines and performs an additional amount of work that also takes polynomial time, then the running time of the composite algorithm is polynomial.



Abstract Problems

- An abstract problem *Q* is a binary relation on a set *I* of problem *instances* and a set *S* of problem *solutions*.
  - E.g., An instance for SHORTEST-PATH is a triple consisting of a graph and two vertices.
    - A solution is a sequence of vertices in the graph, with perhaps the empty sequence denoting that no path exists.
    - The problem SHORTEST-PATH itself is the relation that associates each instance of a graph and two vertices with a shortest path in the graph that connects the two vertices.
- The theory of NP-completeness restricts attention to *decision problems:* those having a yes/no solution.
  - We can view an abstract decision problem as a function that maps the instance set *I* to the solution set {0, 1}.
  - E.g., A decision problem related to SHORTEST-PATH is the problem PATH:
    - If I = <G, u, v, k> is an instance of the decision problem PATH, then *PATH(i)=1* if a shortest path from *u* to *v* has at most *k* edges. Otherwise, *PATH(i)=0*.
- *Optimization problems* can be re-casted as a decision problem that is no harder.



Encodings

- In order for a computer program to solve an abstract problem, we must represent problem instances in a way that the problem understands.
- An ***encoding*** of a set ***S*** of abstract objects is a mapping ***e*** from *S* to *the set of binary strings*.
  - E.g.,
    - {0,1,2,3,…}={0,1,10,11,…}
    - Using this encoding, e(17) = 10001.
  - E.g.,
    - ASCII code: the encoding of A is 10000001.
- We can encode a compound object as a binary string by combining the representations of its constituent parts.
    - E.g., Polygons, graphs, functions, ordered pairs, and programs.



Concrete Problems

- A computer algorithm that solves some abstract decision problem actually takes an ***encoding*** *of a problem instance as* ***input***.
- We call a problem whose instance sets is *the set of binary strings* a ***concrete problem***.
  - We say that an algorithm ***solves*** a concrete problem in time $O(T(n))$ if, when it is provided a problem instance i if length $n=|i|$, the algorithm can produce the solution in at most time $O(T(n))$.
- A concrete problem is ***polynomial-time solvable*** if there exists an algorithm to solve it in time $O(n^k)$ for some constant k.
- The ***complexity class P*** is the set of *concrete decision problems* that are solvable in polynomial time.



Abstract Problems vs. Concrete Problems.

- Using *encoding* as the bridge between abstract problems and concrete problems.
- We can use encodings to map abstract problems to concrete problems.
  - Given an abstract decision problem *Q:*
    - Input: $i \in$ instance set I
    - Output: $Q(i) \in \{0, 1\}$
  - An encoding to encode i to $e(i)$, where $e: I \rightarrow \{0, 1\}^*$ (binary string)
  - The related concrete decision problem $e(Q)$:
    - Input: $e(i) \in \{0, 1\}^*$
    - Output: $Q(i) \in \{0, 1\}$
- The concrete problem produces the same solutions as the abstract problem on binary-string instances that represent the encodings of abstract-problem instances.
  - For convenience, we assume that any non-meaningful abstract-problem instance maps arbitrarily to 0.



Unary vs. Binary Encodings

- The efficiency of solving a problem should not depend on how the problem is encoded.
- However, it depends heavily on the encoding.
  - ***For example***:
    - An integer *k* is to be provided as the sole input to an algorithm, and the running time of the algorithm is Θ(k).
    - ***n*** is the input length (i.e., the input size).

Problem	input *k*	complexity O(*k*)
unary	$k \rightarrow 11\ldots1$	$\Theta(k) = \Theta(n)$
binary	$n = \lfloor \lg k \rfloor + 1$	$\Theta(k) = \Theta(2^n)$

Polynomial time

Exponential time (Superpolynomial time)



Unary vs. Binary Encodings (Cont.)

- For NP-complete problems:
  - If the input is encoded in unary, then there exists a polynomial-time algorithm for it. We say that the problem is NP-complete *in the ordinary sense*. The algorithm is said to be *pseudo-polynomial*.
  - If a problem remains even NP-complete when the input is encoded in unary, then we say that the problem is NP-complete *in the strong sense*.
- In practice, if we rule out "*expensive*" encodings such as unary ones, the actual encoding of a problem makes little difference to whether the problem can be solved in polynomial time.
  - E.g., Representing integers in base 3 instead of binary has no effect on whether a problem is solvable in polynomial time, because we can convert an integer represented in base 3 to base 2 in polynomial time.



Polynomial-Time Computable

- A function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ is polynomial-time computable if there exists a polynomial time algorithm A that, given any $x \in \{0, 1\}^*$, produces as output $f(x)$.
  - For any set I of problem instances, we say that two encodings e_1 and e_2 are ***polynomial related*** if there exist two polynomial-time computable functions f_{12} and f_{21} such that for any $i \in I$, we have $f_{12}(e_1(i))=e_2(i)$ and $f_{21}(e_2(i))=e_1(i)$.
  - That is, a polynomial-time algorithm can compute the encodings $e_2(i)$ from the encoding $e_1(i)$, and vice versa.



Polynomial-Time Computable (Cont.)

- ***Lemma 34.1***
  - Let Q be an abstract decision problem on an instance set I, let e_1 and e_2 be polynomially related encodings on I. Then, $e_1(Q) \in P$ if and only if $e_2(Q) \in P$.
- ***Proof***
  - Suppose that $e_1(Q)$ can be solved in $O(n^k)$ for some constant k.
  - Suppose that for any problem instance i, the encoding $e_1(i)$ can be computed from the encoding $e_2(i)$ in time $O(n^c)$ for some constant c, where $n = |e_2(i)|$.
  - To solve problem $e_2(Q)$, on input $e_2(i)$,
    - $|e_1(i)| = O(n^c)$ since the output of a serial computer cannot be longer than its running time.
    - Solving the problem on $e_1(i)$ takes time $O(|e_1(i)|^k) = O(n^{ck})$ that is polynomial since both c and k are constants.



Polynomial-Time Computable (Cont.)

- The encoding of a finite set is polynomially related to its encoding as a list of its elements, enclosed in *braces* and *separated by commas*. (ASCII is one such encoding scheme)
  - With such a “standard” encoding in hand, we can derive reasonable encodings of other mathematical objects (such as *tuples*, *graphs*, and *formulas*)
  - ***<G>*** denotes the standard encoding a graph G.
- We shall assume that the encoding of an integer is polynomially related to its ***binary representation***.
- We shall assume that all problem instances are ***binary strings*** encoded using the standard encoding.
- *We shall typically neglect the distinction between abstract and concrete problems.* (because the actual encoding of a problem makes little difference to whether the problem can be solved (in polynomial time)).



Formal-Language Framework

- An ***alphabet*** Σ is a finite set of symbols.
- A ***language L*** over Σ is any set of strings made up of symbols from Σ.
  - E.g., if $\Sigma = \{0, 1\}$, the set $L = \{10, 11, 101, 111, 1011, 1101, 10001, \ldots\}$ Is the language of binary representation of prime numbers.
- The ***empty string*** is ε.
- The ***empty language*** is ϕ.
- The language of all strings over Σ is Σ^*.
  - E.g., $\Sigma = \{0, 1\}$, then $\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$



Formal-Language Framework - Set-Theoretic Operations

- **Union**: $L_1 \cup L_2$
- **Intersection**: $L_1 \cap L_2$
- **Complement** (of L): $\overline{L} = \Sigma^* - L$
- **Concatenation**: $L = \{x_1x_2 : x_1 \in L_1 \text{ and } x_2 \in L_2\}$
- **Closure** (or **Kleene star**): $L^* = \{\varepsilon\} \cup L^1 \cup L^2 \cup L^3 \cup \ldots$
  - **L**^{k} is the language obtained by concatenating *L* to itself *k* times.



Formal-Language Framework - Language Theory

- The set of instances for any decision problem ***Q*** is simply the set Σ^*, where $\Sigma = \{0, 1\}$.
- Since ***Q*** is entirely characterized by those problem instances that produce a ***1(yes) answer***, we can view *Q* as a language *L* over $\Sigma = \{0, 1\}$, where $\boldsymbol{L = \{x \in \Sigma^* : Q(x) = 1\}}$.
- ***For example***:
  - The decision problem **PATH**:
    - PATH = { $\langle G, u, v, k \rangle$: G = (V, E) is an undirected graph,
$u, v \in V$,
$k \geq 0$ is an integer, and
there exists a path from u to v in G
consisting of at most *k* edges. }



Formal-Language Framework Accept vs. Decide

- The formal-language framework allows us to express concisely the relation between ***decision problems*** and ***algorithms that solve them***.
  - Algorithm *A* ***accepts*** a string $x \in \{0, 1\}^*$ if, given input *x*, the algorithm's output ***A(x)=1***.
  - The language ***accepted*** by an algorithm *A* is the set of strings ***$L=\{x \in \{0, 1\}^* : A(x)=1\}$***.
    - That is, *the set of strings that the algorithm accepts*.
  - Algorithm *A* ***rejects*** a string x if A(x)=0.
    - Even if language *L* is accepted by an algorithm *A*, the algorithm will not necessarily reject a string $x \notin L$.
    - A language *L* is ***decided*** by an algorithm *A* if every *binary string* in *L* is accepted by *A* and every binary string not in *L* is rejected by *A*.
- A language *L* is ***accepted in polynomial time by an algorithm A*** if
  - It is accepted by *A* and
  - If there exists a constant *k* such that for any length-n string $x \in L$, algorithm *A* accepts x in time $O(n^k)$.
- A language *L* is ***decided in polynomial time by an algorithm A*** if there exists a constant *k* such that for any length-n string $x \in \{0, 1\}^*$, the algorithm correctly decides whether $x \in L$ in time $O(n^k)$.
- Note:
  - To ***accept*** a language, an algorithm must produce an answer when provided a string in L.
  - To ***decide*** a language, an algorithm must correctly accept or reject every string in $\{0, 1\}^*$.



Formal-Language Framework Accept vs. Decide (Cont.)

- The language PATH
  - If G encodes an undirected graph and the path found from u to v has at most k edges, then the algorithm outputs 1 and halts.
    - This algorithm does not decide PATH, since it does not explicitly output 0 for instances in which a shortest path has more than k edges.
    - If this algorithm could output 0 and halts when there is not a path from *u* to *v* with at most *k* edges, then the algorithm decides PATH.
- Turing's halting problem
  - This is the problem of deciding, given a program and an input, whether the program will eventually halt when run with that input, or will run forever.
  - *There exists an accepting algorithm, but no decision algorithm exists.*



Complexity Class P

- A ***complexity class*** is a set of languages of an algorithm that determines whether a given string *x* belongs to language *L*.
- The membership in a set of languages is determined by a ***complexity measure***, such as running time.
- Definition of the complexity class **P**:
  - **P** = { $L \subseteq \{0, 1\}^*$: there exists an algorithm *A* that **decides** *L* in polynomial time }.
  - In fact, P is also the class of languages that can be accepted in polynomial time.



Complexity Class P (Cont.)

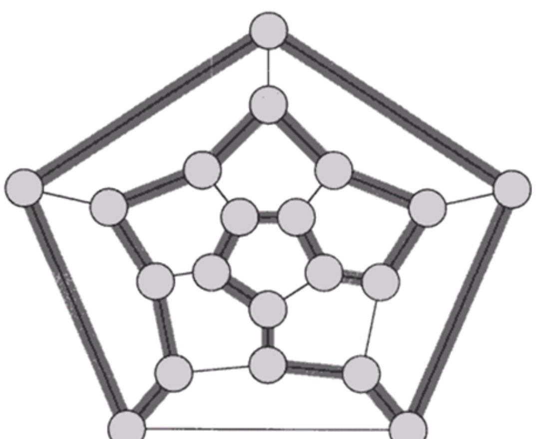
- ***Theorem 34.2***
  - P = { L : L is accepted by a polynomial-time algorithm }.
- ***Proof***
  - We need only show that if *L* is ***accepted*** by a polynomial-time algorithm, it is ***decided*** by a polynomial-time algorithm.
    - Let *L* be the language accepted by some polynomial-time algorithm *A*. We construct A’ that decides L.
    - Because *A* accepts *L* in time $O(n^k)$ steps for some constant *k*, there also exists a constant *c* such that *A* accepts *L* in at most cn^k steps.
    - For any input string x, the algorithm A’ simulates cn^k steps of A. After simulating cn^k steps,
      - A’ accepts x if A has accepted x.
      - A’ rejects x if A has not accepted x.

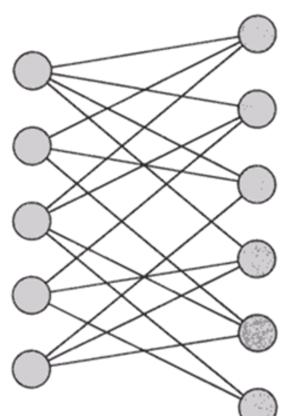


Polynomial-Time Verification

Hamiltonian Cycles

- A ***hamiltonian cycle*** of an undirected graph G=(V, E) is a simple cycle that contains each vertex in V.
- A graph that contains a hamiltonian cycle is said to be ***hamiltonian***; otherwise, it is ***nonhamiltonian***.
- W. R. Hamilton described a mathematical game on the dodecahedron (十二面體), in which one player sticks five pins in any five consecutive vertices and the other play must complete the path to form a cycle containing all the vertices. → ***The dodecahedron is hamiltonian.***
- A ***bipartite graph*** with an ***odd number*** of vertices is ***nonhamiltonian***.





A **bipartite graph** (or **bigraph**) is a graph whose vertices can be divided into two disjoint sets *U* and *V* such that every edge connects a vertex in *U* to one in *V*; that is, *U* and *V* are independent sets.



Hamiltonian Cycles (Cont.)

- Hamiltonian-cycle problem:
  - HAM-CYCLE = { <G> : G is a hamiltonian graph }
- How might an algorithm decide the language HAM-CYCLE?
  - Given a problem instance <G>, one possible decision algorithm lists all permutations of the vertices of G and then checks each permutation to see if it is a hamiltonian path.
    - Input size $n = |<G>|$ is the length of the encoding of G.
    - The number of vertices $m = \Omega(n^{1/2})$ or $m \geq n^{1/2}$.
    - Thus, the running time is $\Omega(m!) = \Omega((n^{1/2})!) = \Omega(2^{n^{1/2}})$ that is not $O(n^k)$ for any constant *k*.
  - In fact, the hamiltonian-cycle problem is NP-complete.

$m \leq n \leq m^2$



Verification Algorithms

- Suppose that someone tells you that a given graph *G* is hamiltonian, and then offers to prove it by giving you the vertices in order along the hamiltonian cycle.
  - You should certainly implement an $O(n^2)$-time verification algorithm to check whether the given vertices form a hamiltonian cycle.
- A verification algorithm is a ***two-argument algorithm A***.
  - One argument is an ordinary *input string x*.
  - The other is a *binary string y* called ***certificate***.
- A two-argument algorithm *A* ***verifies*** an input string *x* if there exists a certificate *y* such that $A(x, y) = 1$.
  - The ***language verified*** by a verification algorithm *A* is $L = \{ x \in \{0, 1\}^* : \text{there exists } y \in \{0, 1\}^* \text{ such that } A(x, y) = 1 \}$.



Verification Algorithms (Cont.)

- Intuitively, an algorithm A verifies a language L,
  - If for any string $x \in L$, there exists a certificate y that A can use to prove that $x \in L$.
  - If any string $x \notin L$, there must be no certificate proving that $x \in L$.
- ***For example***
  - In the hamiltonian-cycle problem, the *certificate* is the list of vertices in some hamiltonian cycle.
    - If a graph is hamiltonian, the hamiltonian cycle itself offers enough information to verify this fact.
    - If a graph is not hamiltonian, there is no list of vertices that fools the verification algorithm into believing that the graph is hamiltonian.



Complexity Class NP

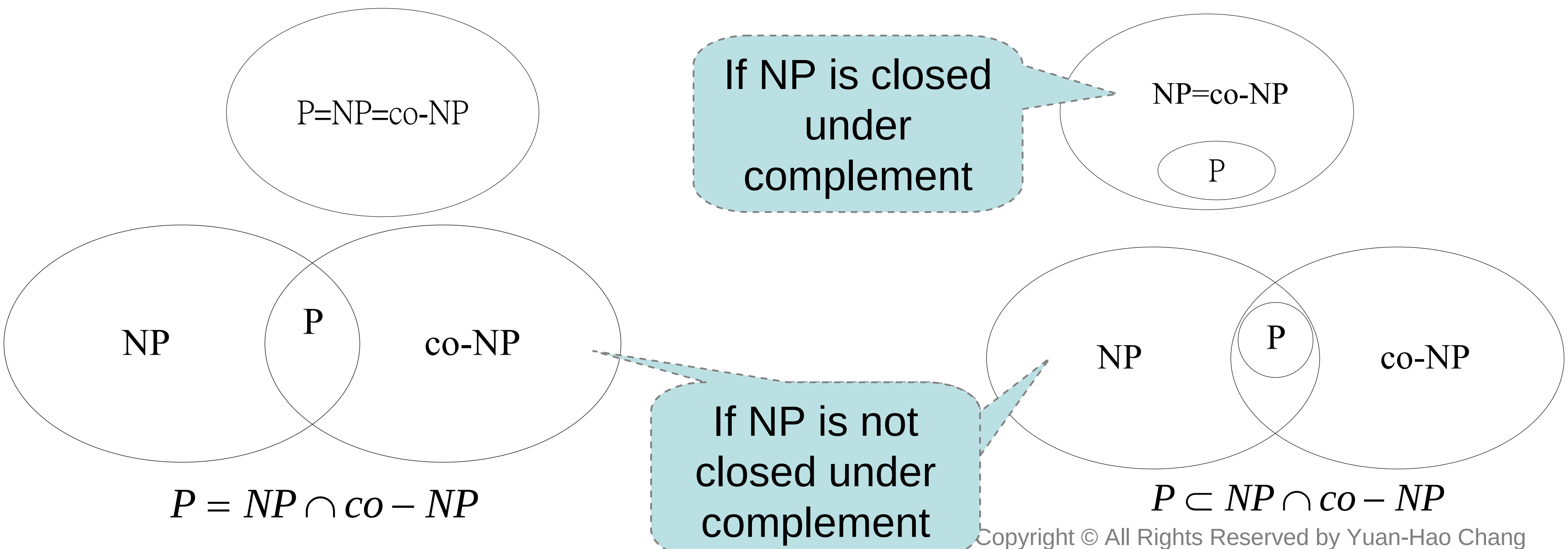
- The ***complexity class NP*** is the class of languages that can be verified by a polynomial-time algorithm.
  - That is, a language L belongs to NP if and only if there exist a *two-input polynomial-time algorithm A* and a constant *c* such that $L = \{ x \in \{0, 1\}^* :$ there exists a certificate y with $|y| = O(|x|^c)$ such that $A(x, y) = 1 \}$.
  - We say that *A* ***verifies*** language *L* in ***polynomial time***.

|x| is the input size

- ***For example***:
  - HAM-CYCLE $\in$ NP.
  - If L $\in$ P, then L $\in$ NP.
    - Because if there is a polynomial-time algorithm ***decide*** *L*, the algorithm can be easily converted to a two-argument verification algorithm that simply ignores any certificate and accepts exactly those input strings it determines to be in *L*. Thus, P $\subseteq$ NP.
  - It is unknown whether P = NP, but most researchers believe that P $\neq$ NP.

Co-NP

- Does $L \in NP$ imply $\overline{L} \in NP$?
  - No one knows whether the class NP is closed under complement.
- The ***complexity class co-NP*** is the set of languages L such that $\overline{L} \in NP$.
- Since P is closed under complement, so that $P \subseteq NP \cap co\text{-}NP$.





NP-Completeness and Reducibility

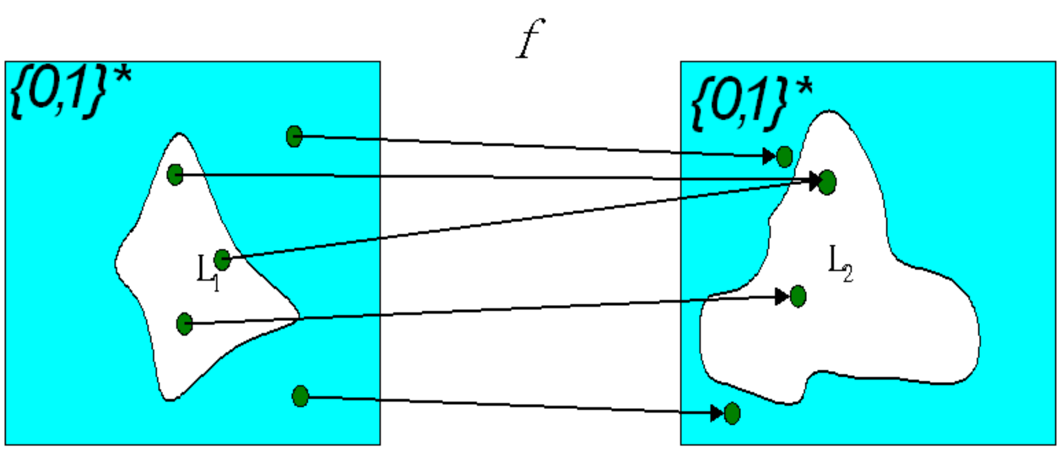


Reducibility

- If ***any*** NP-complete problem can be solved in polynomial time, then ***every*** problem in NP has a polynomial-time solution.
- A problem ***Q*** can be reduced to another problem ***Q'*** if any instance of ***Q*** can be "***easily rephrased***" as an instance of ***Q'***, then solution to which provides a solution to the instance of ***Q***.
  - E.g., Given an instance $ax + b = 0$, we can transform it to $0x^2 + ax + b = 0$.
  - Thus, if a problem Q reduces to another problem Q', then Q is "***no harder to solve***" than Q'.

Reducibility (Cont.)

- A language L_1 is ***polynomial-time reducible*** to a language L_2, written $L_1 \leq_P L_2$, if there exists a polynomial-time computable function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ such that for all $x \in \{0, 1\}^*$, $x \in L_1$ iff $f(x) \in L_2$.
  - **f** : the ***reduction function***.
  - **F** : a ***reduction algorithm*** that computes f in polynomial time.

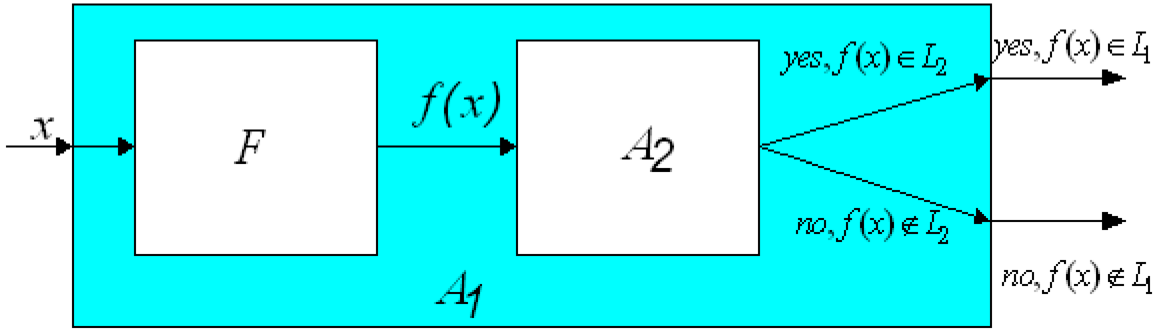


Providing an answer to whether $f(x) \in L_2$ directly provides the answer to whether $x \in L_1$.

If $x \in L_1$ then $f(x) \in L_2$.
If $x \notin L_1$ then $f(x) \notin L_2$.

Polynomial-Time Reducible

- ***Lemma 34.3***
  - If L1, L2 $\subseteq \{0, 1\}^*$ are languages such that $L_1 \leq_P L_2$, then $L_2 \in P$ implies $L_1 \in P$.
- ***Proof***
  - Let A_2 be a polynomial-time algorithm that decides L_2.
  - Let F be a polynomial-time reduction algorithm that computes the reduction function f.
  - Thus, we can construct a polynomial-time algorithm A_1 that decides L_1.





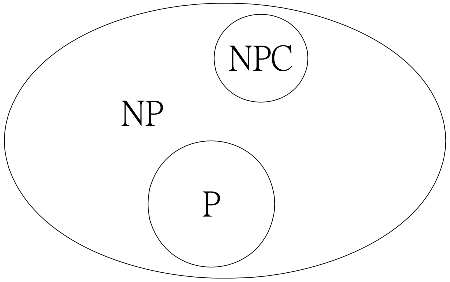
NP-Completeness

- ***Polynomial-time reduction*** provides a formal means for showing that *one problem is at least as hard as another*.
- A language $L \subseteq \{0, 1\}^*$ is ***NP-complete (NPC)*** if
  - 1. $L \in NP$, (L 屬於NP) and
  - 2. $L' \leq_P L$ for every $L' \in NP$. (所有NP的問題L'都可reduce到L)
- If a language *L* satisfies *property 2*, but not necessarily *property 1*, then ***L*** is ***NP-hard***.



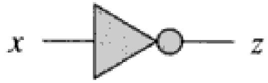
NP-Completeness (Cont.)

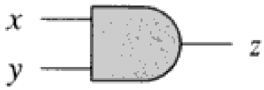
- ***Theorem 34.4***
  - If any NP-complete problem is polynomial-time solvable, then P = NP. Equivalently, if any (one) problem in NP is not polynomial-time solvable, then no NP-complete problem is polynomial-time solvable.
- ***Proof***
  - Suppose that $L \in P$ and $L \in NPC$.
  - For any $L' \in NP$, we have $L' \leq_P L$ because $L \in NPC$.
  - By Lemma 34.3, $L' \in P$, which proves the first statement of this theorem. (Because $L' \leq_P L$ and $L \in P$)

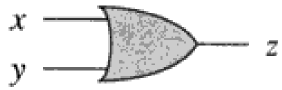


Circuit Satisfiability

- ***The first NPC problem***
- ***Term definition***
  - ***Boolean combinational circuits*** are built from boolean combinational elements that are interconnected by wires.
  - A ***boolean combinational element*** is any circuit element that has a constant number of boolean inputs and outputs and that performs a well-defined function.
  - ***Boolean values*** are drawn from the set {0, 1}.
  - The three basic logic gates are







Truth table

NOT

x	$\neg x$
0	1
1	0

AND

x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

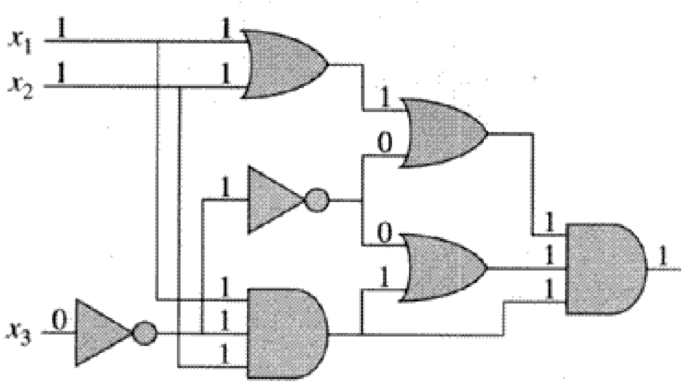
OR

x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

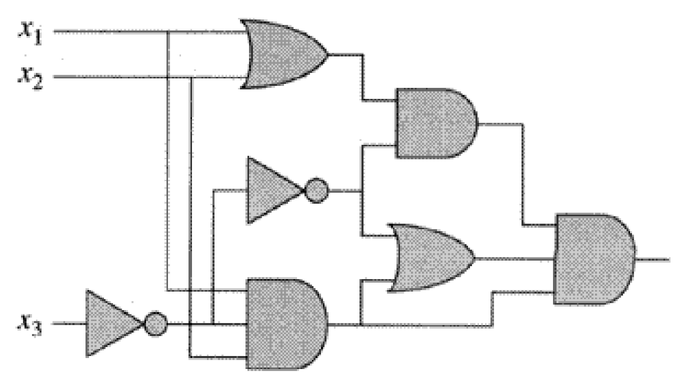


Circuit Satisfiability (Cont.)

- A ***boolean combinational circuit*** consists of one or more ***boolean combinational elements*** interconnected by ***wires***.
- A ***wire*** can connect the output of one element to the input of another.
- The number of element inputs fed by a wire is called the ***fan-out*** of the ***wire***.
- A ***truth assignment*** is a set of ***boolean input value***.
- A circuit is ***satisfiable*** if it has a ***satisfying assignment*** that makes the circuit output ***1***.



$\langle x_1=1, x_2=1, x_3=0 \rangle$
Satisfiable circuit



Unsatisfiable circuit



Computer Hardware

- A ***computer program*** is stored in the computer memory as *a sequence of instruction*.
- A typical instruction encodes
  - An operation to be performed,
  - Addresses of operands in memory, and
  - An address where the result is to be stored.
- A special memory location (called the ***program counter***) keeps track of which instruction is to be executed next.
  - The program counter automatically increments upon fetching each instruction, causing the computer to execute instructions sequentially.
  - The execution of some instructions can cause a value to be written to the program counter, so as to allow loop or conditional braches.
- Any particular state of computer memory is called a ***configuration***.
  - We can view the execution of an instruction as ***mapping one configuration to another***.
  - The hardware that accomplishes this mapping can be implemented as a *boolean combinational circuit* (denoted as ***M***).



Circuit Satisfiability Problem

- ***Problem definition***:
  - Given a boolean combinational circuit composed of AND, OR, and NOT gates, is it satisfiable?
- The ***size*** of a boolean combinational circuit is ***the number of boolean combinational elements*** plus ***the number of wires*** in the circuit.
  - We can encode any given ***circuit C*** into a ***binary string <C>*** whose length is polynomial in *the size of the circuit*.
  - When *the size of C is polynomial in the **k** inputs*, checking each one takes $\Omega(2^k)$.
- As formal language, we can define:

CIRCUIT-SAT = { <C> : C is a satisfiable boolean combinational circuit. }

In the computer-aided hardware optimization, if a subcircuit always produces 0, that subcircuit is unnecessary.



Circuit Satisfiability Problem (Cont.)

- To prove the circuit satisfiability problem ***C*** an NPC problem, we should prove the following two things:
  - 1. C$\in$NP
  - 2. C is at least as hard as any language in NP (or ***C is NP-hard***)

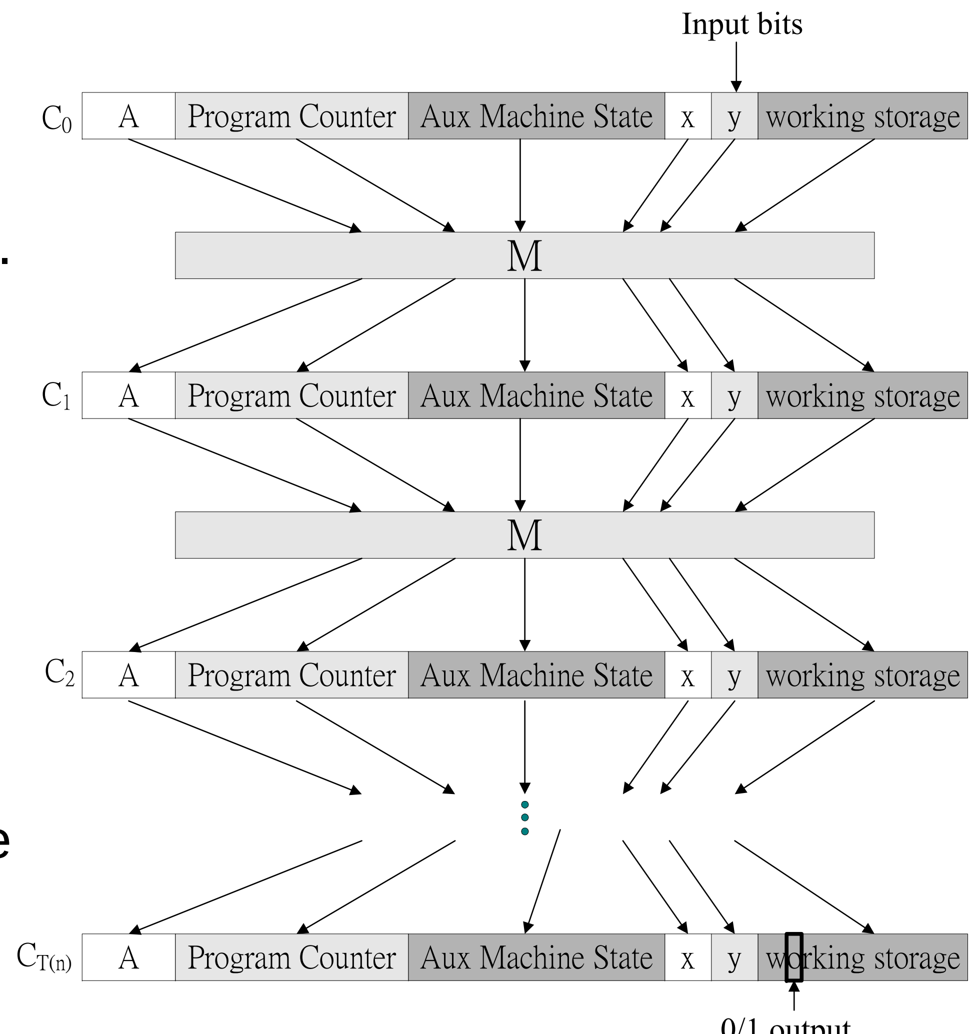
- ***Lemma 34.5***
  - The circuit-saisfiability problem belongs to the class NP.
- ***Proof***
  - We shall provide a two-input, *polynomial-time* algorithm ***A***
    - One input is a boolean combinational circuit ***C***.
    - The other input is a ***certificate*** corresponding to an assignment of boolean values to the wires in *C*.
  - For each logic gate in the circuit, the algorithm ***A*** checks that the value provided by the certificate on the output wire is correctly computed.
  - Then, if the output of the entire circuit is 1, the algorithm ***A*** outputs 1. Otherwise, ***A*** outputs 0.

Circuit Satisfiability Problem (Cont.)

- ***Lemma 34.6***
  - The circuit satisfiability problem is NP-hard
- ***Proof***
  - Let ***L*** be any language in NP.
  - We shall describe a polynomial-time algorithm ***F*** computing a reduction function ***f*** that maps every binary string ***x*** to a circuit ***C = f(x)*** such that *$x \in L$ iff $C \in$ CIRCUIT-SAT*.
  - The algorithm ***F*** uses the ***two-input polynomial-time algorithm A*** ('cause $L \in NP$) to compute the reduction function ***f***.
    - Let ***T(n)*** denote the worst-case running time of A on *length-n input strings*.
    - Let $k \geq 1$ be a constant such that ***$T(n) = O(n^k)$*** and the length of the *certificate* is ***$O(n^k)$***

Circuit Satisfiability Problem (Cont.)

- ***Proof (Cont.)***
  - Represent the computation of ***A*** as a sequence of configurations.
  - We can break down each configuration into parts consisting of the program for ***A***.
  - The combinational circuit M that implements the computer maps each configuration c_i to the next configuration c_{i+1}, from $\boldsymbol{c_0}$.
  - If A runs for at most $T(n)$ steps, the output appears as one of the bits in $\boldsymbol{c_{T(n)}}$.





Circuit Satisfiability Problem (Cont.)

- ***Proof (Cont.)***
  - The ***reduction algorithm F*** constructs a single combinational circuit that computes all configurations produced by a given initial configuration.
    - Paste the circuit M for ***T(n)*** copies.
    - The output if the i_{th} circuit (c_i) feeds directly into the input of the (i+1)st circuit.
    - Thus, the configurations reside as values on the wires connecting copies of ***M***.
  - Mission of the ***reduction algorithm F***
    - Given an input x, it must compute a circuit C=f(x) that is satisfiable iff there exists a certificate y such that A(x, y) = 1.
    - When ***F*** obtains an input *x*, it computes n = |x| and constructs a combinational circuit ***C'*** consisting of T(n) copies of M.
    - The input to C' is an initial configuration corresponding to a computation on A(x, y), and the output is the configuration $c_{T(n)}$.



Circuit Satisfiability Problem (Cont.)

- Mission of the ***reduction algorithm F*** ***(Cont.)***
  - Algorithm F modifies C' to construct **C = f(x)**.
    - First, it wires the inputs and certificate *y* to C'.
    - Second, it ignore all outputs from C', except the output bit of $c_{T(n)}$.
  - This circuit C compute **C(y) = A(x, y)** for any *input y of length $O(n^k)$*.
  - We need to provide two properties:
    - **First, *F* correctly computes a reduction function *f*.**
      i.e., C is satisfiable iff there exists a certificate y such that A(x, y)=1.
      - » Suppose there exists a certificate y such that A(x, y)=1. Because C(y) = A(x, y), C(y) = 1 → C is satisfiable (***reverse direction***).
      - » Suppose C is satisfiable such that C(y)=1. Because A(x, y) = C(y), A(x, y) = 1 (***forward direction***)



Circuit Satisfiability Problem (Cont.)

- **Second, F runs in polynomial time.**
- The number of bits to represent a configuration is polynomial in $n = |x|$.
  - » The program for A itself has constant size.
  - » The length of the input x is n.
  - » The length of the certificate y is $O(n^k)$. (by algorithm A's definition)
  - » The amount of working storage required by A is polynomial in *n* because the algorithm runs for at most $O(n^k)$.
  - » The combinational circuit M has size polynomial in the length of a configuration ($O(n^k)$). → ***M*** usually implements *the logic of the memory system*.
  - » The circuit C consists of at most $t = O(n^k)$ copies of M. Hence it has size polynomial in *n*.
- → Therefore, F can construct C in polynomial time.

• Therefore, CIRCUIT-SAT is at least as hard as any language in NP, and since it is in NP, it is NP-Complete.



NP-Completeness Proofs



NP-Complete Basis

- ***Lemma 34.8***
  - If L is a language such that L’ $\leq_P$ L for some L’$\in$NPC, then L is NP-hard. If, in addition, L$\in$NP, then L$\in$NPC.
- ***Proof***
  - Since L’ is NP-complete, for all L’’$\in$NP, we have L’’$\leq_P$ L’.
  - By supposition, L’ $\leq_P$L, and thus by transitivity, we have L’’ $\leq_P$ L, which shows that L is NP-hard.
  - If L$\in$NP, we also have L$\in$NPC.

NP-Complete Proof Method

- By reducing a known NP-complete language L’ to L, we implicitly reduce every language in NP to L. Thus, the proving steps:
  - 1. Prove $L \in NP$.
  - 2. Prove L is NP-hard.
    - 1. Select a known NP-complete language L’.
    - 2. Describe an algorithm that computes a function *f* mapping every instance $x \in \{0, 1\}^*$.
    - 3. Prove that the function *f* satisfies $x \in L'$ iff $f(x) \in L$ for all $x \in \{0, 1\}^*$.
    - 4. Prove that the algorithm computing *f* runs in polynomial time.
- Proving CIRCUIT-SAT$\in$NPC has given us a “foot in the door.”



Formula Satisfiability Problem

- Formula satisfiability problem (***SAT***):
  - An instance of SAT is a boolean formula ϕ composed of
    - 1. *n* boolean variables: $x_1, x_2, \ldots, x_n$;
    - 2. *m* boolean connectives: any boolean function with one or two inputs and one output, such as $\wedge$ (AND), $\vee$(OR), and $\neg$(NOT), $\rightarrow$(implication), $\leftrightarrow$(if and only if); and
    - 3. parentheses. (Without loss of generality, we assume that there are no redundant parentheses.)
  - We can easily encode a boolean formula ϕ in a length that is polynomial in *n+m*.
- Satisfiable formula
  - A ***truth assignment*** for a boolean formula ϕ is a set of values for the variables of ϕ.
  - A satisfying assignment is a truth assignment that causes it to evaluate to 1.
  - A formula with a satisfying assignment is a ***satisfiable*** formula.



Formula Satisfiability Problem (Cont.)

- The satisfiability problem asks whether a given boolean formula is satisfiable:

  SAT = {$<\phi>$: ϕ is a satisfiable boolean formula}

- ***For example***:

$$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$

has the satisfying assignment $\langle x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1\rangle$, since

$$\begin{aligned} \phi &= ((0 \rightarrow 0) \vee \neg((\neg 0 \leftrightarrow 1) \vee 1)) \wedge \neg 0 \\ &= (1 \vee \neg(1 \vee 1)) \wedge 1 \\ &= (1 \vee 0) \wedge 1 \\ &= 1 , \end{aligned}$$

and thus this formula ϕ belongs to SAT.

A formula with n variables has 2^n possible assignments. If the length of $<\phi>$ is polynomial in *n*, then checking every assignment requires $\Omega(2^n)$ time.

Formula Satisfiability Problem (Cont.)

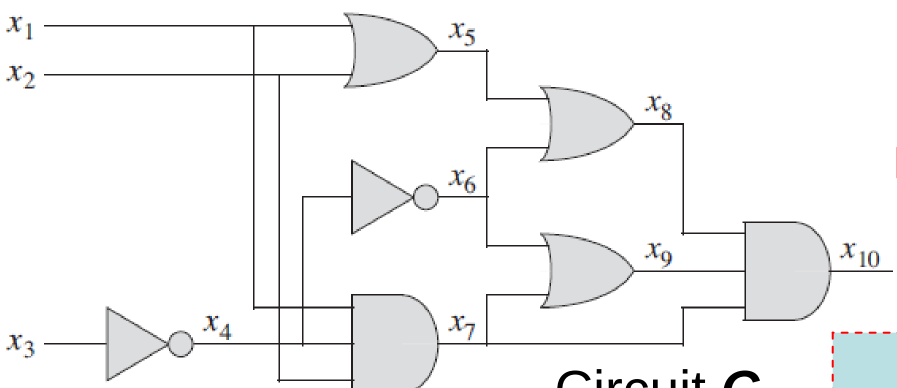
- ***Theorem 34.9***
  - Satisfiability of boolean formulas is NP-complete.
- ***Proof***
  - **1. Start by proving that SAT$\in$NP. (Verify in polynomial time)**
    - Show that a certificate consisting of a satisfying assignment for an input formula ϕ can be verified in polynomial time.
      - The verifying algorithm replaces each variable in the formula with its corresponding value and then evaluates the expression.
      - This task is easy to do in polynomial time.
  - **2. Then prove SAT is NP-hard. (CIRCUIT-SAT $\leq_P$ SAT)**
    - For each wire x_i in the circuit C, the formula ϕ have a variable x_i.
    - Then, express each gate as a small formula involving the variables of its incident wires.



Formula Satisfiability Problem (Cont.)

– **2. Then prove SAT is NP-hard. (Cont.)**

- The formula ϕ produced by the reduction algorithm is the AND of the circuit-output variable with the conjunction of clauses describing the operation of each gate.



Circuit ***C***

$$
\begin{aligned}
\phi \;=\; x_{10} &\wedge (x_4 \leftrightarrow \neg x_3) \\
&\wedge (x_5 \leftrightarrow (x_1 \vee x_2)) \\
&\wedge (x_6 \leftrightarrow \neg x_4) \\
&\wedge (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4)) \\
&\wedge (x_8 \leftrightarrow (x_5 \vee x_6)) \\
&\wedge \boxed{(x_9 \leftrightarrow (x_6 \vee x_7))} \\
&\wedge (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9))
\end{aligned}
$$

A clause

- Given a circuit C, it is straightforward to produce such a formula ϕ in polynomial time.
- When we assign wire values to variables in ϕ, each clause of ϕ evaluates to 1, and thus the conjunction of all evaluates to 1.
- Conversely, if some assignment causes ϕ to evaluate to 1, the circuit C is satisfiable by an analogous argument.

Conjunctive Normal Form (CNF)

- The reduction algorithm must handle *any input formula*, and 3-CNF-SAT is one convenient language to simply the NPC proofs.
- 3-CNF-SAT
  - A ***literal*** in a boolean formula is an occurrence of a variable or its negation.
  - A boolean formula is in ***conjunctive normal form***, or ***CNF***, if it is expressed as an **AND** of ***clauses***.
  - Each clause is the **OR** of one or more ***literals.***
  - A boolean formula is in ***3-CNF***, if each clause has exactly three distinct literals.
  - E.g., $\underbrace{(x_1 \vee \neg x_1 \vee \neg x_2)}_{\text{Clause}} \wedge (\underbrace{x_3}_{\text{Literal}} \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$

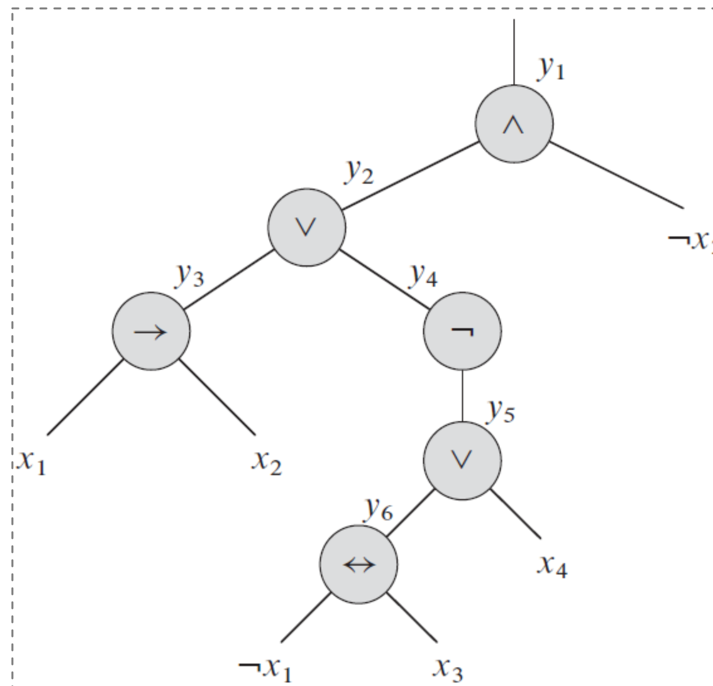


3-CNF Satisfiability (3-CNF-SAT)

- ***Theorem 34.10***
  - Satisfiability of boolean formulas in 3-CNF is NP-complete.

$$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$

- ***Proof***
  - **1. 3-CNF-SAT∈NP. (Verify in polynomial time)**
    - The verifying algorithm (in polynomial time) replaces each variable in the formula with its corresponding value and then evaluates the expression.
  - **2. SAT $\leq_P$ 3-CNF-SAT (Separated into 3 steps)**
    - **Step 1:** Construct a binary "parse" tree
      - Construct a binary "parse" tree for the input formula ϕ with ***literals as leaves*** and ***connectives as internal nodes***.
      - The input formula is fully parenthesized.



Parse tree

3-CNF Satisfiability (3-CNF-SAT) (Cont.)

- **Step 1:** Construct a binary "parse" tree **(Cont.)**
  - Introduce a variable y_i for the output of each internal node.
  - Thus, obtain a formula ϕ', each clause of which has *at most 3* literals, but fail to meet that each clause has exactly 3literals.

$$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$

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

3-CNF Satisfiability (3-CNF-SAT) (Cont.)

- **Step 2:** Convert each clause of ϕ' into CNF.
  - Construct a truth table for each clause to evaluate all possible assignments to its variables.
  - Build a ***disjunctive normal form (DNF)*** – ***an OR of ANDs***, and then use ***DeMorgan's laws*** for propositional logic.

$$\neg(a \wedge b) = \neg a \vee \neg b$$
$$\neg(a \vee b) = \neg a \wedge \neg b$$

DeMorgan's law:
Break the line and change the sign

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

Truth table

$$\neg\phi_1' = (y_1 \wedge y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge \neg x_2) \vee (\neg y_1 \wedge y_2 \wedge \neg x_2)$$

$$\phi_1'' = \phi_1' = (\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee x_2)$$

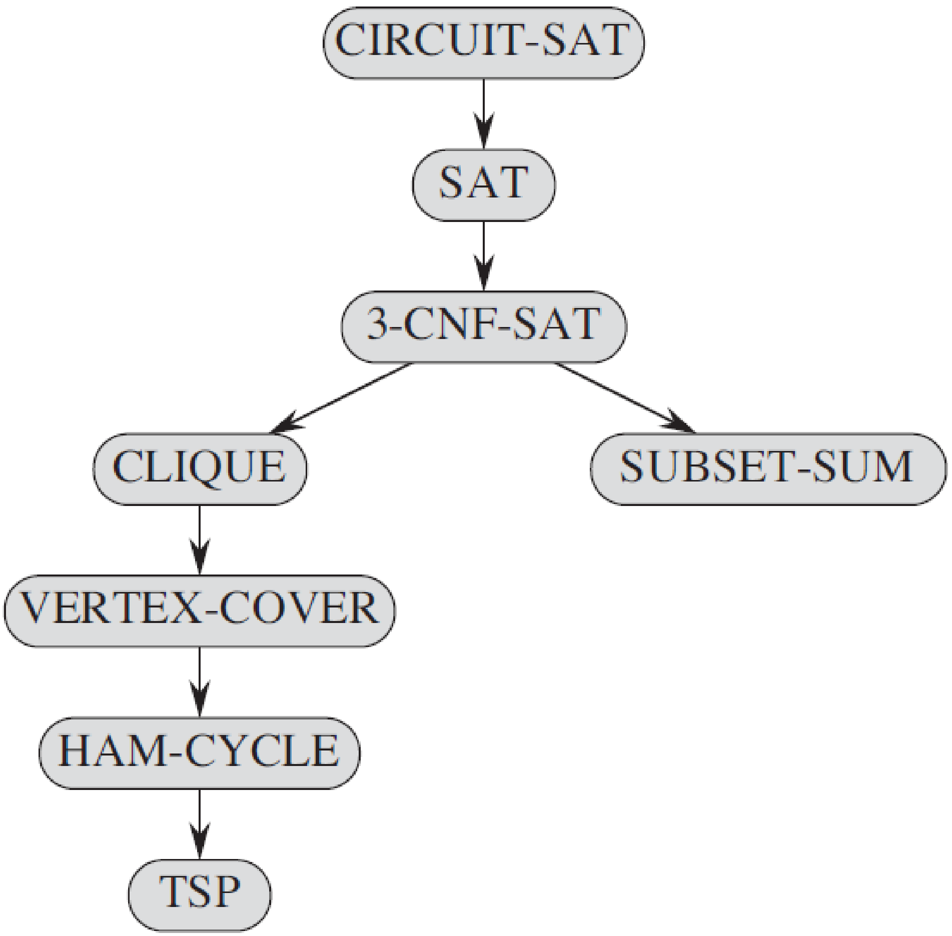
3-CNF Satisfiability (3-CNF-SAT) (Cont.)

- **Step 3:** Transform each clause of ϕ' into exactly 3 literals. We construct the final 3-CNF formula ϕ'''
  - If C_i has 3 distinct literals, then simply include C_i as a clause of ϕ'''.
  - If C_i has 2 distinct literals, that is, if $C_i = (l_1 \vee l_2)$, where l_1 and l_2 are literals, then include $(l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$ as clauses of ϕ'''.
  - If C_i has just 1 distinct literal l, then include $(l \vee p \vee q) \wedge (l \vee p \vee \neg q) \wedge (l \vee \neg p \vee q) \wedge (l \vee \neg p \vee \neg q)$ as clauses of ϕ'''.
- *The reduction can be computed in polynomial time.*
  - Constructing ϕ' from ϕ introduces *at most 1 variable and 1 clause* per connective in ϕ. (one y_i variable and its corresponding clause)
  - Constructing ϕ'' from ϕ' introduces *at most 8 clauses* into ϕ'' for each clause from ϕ' . (according to the *truth table*)
  - Constructing ϕ''' from ϕ'' introduces *at most 4 clauses* into ϕ''' for each clause from ϕ''. (according to the number of literals in the clause)



NP-Complete Problems

The Structure of NPC Proofs





The Clique Problem

- A ***clique*** is a complete subgraph of G.
  - In other words, a ***clique*** in an undirected graph G = (V, E) is a subset V’ $\subseteq$ V of vertices, each pair of which is connected by an edge in E.
- The ***size*** of a clique is the number of vertices it contains.
- The ***clique problem*** is the optimization problem of finding a clique of maximum size in a graph.
- As a *decision problem*, we ask simply whether a ***clique*** of a given size ***k*** exists in the graph:

$$\text{CLIQUE} = \{\langle G, k \rangle : G \text{ is a graph containing a clique of size } k\}$$

The running time of this algorithm is $\Omega(k^2 \binom{|V|}{k})$.
In general, k could be near $|V|/2$,
in which case the algorithm runs in superpolynomial time.



The Clique Problem (Cont.)

- ***Theorem 34.11***
  - The clique problem is NP-complete.
- ***Proof***
  - **1. CLIQUE$\in$NP. (Verify in polynomial time)**
    - For a given graph G = (V, E), we use the set *V'* $\subseteq$ V of vertices in the clique as a *certificate* for G. We can check whether V' is a clique in polynomial time by checking whether, for each pair u, v $\in$V', the edge *(u, v)* belongs to E.
  - **2. 3-CNF-SAT$\leq_P$ CLIQUE (Prove CLIQUE is NP-hard)**
    - Let $\phi = C_1 \wedge C_2 \wedge \ldots \wedge C_k$ be a boolean formula in 3-CNF with *k* clauses.
    - For *r* = 1, 2, …, k, each clause C_r has exactly three distinct literals l_1^r, l_2^r, l_3^r.
    - We shall construct a graph *G* such that ϕ ***is satisfiable*** iff ***G has a clique of size k***.



The Clique Problem (Cont.)

– **2. 3-CNF-SAT$\leq_P$ CLIQUE (Cont.)**

- For each clause $C_r = (l_1^r \vee l_2^r \vee l_3^r)$ in ϕ, we place a triple of vertices v_1^r , v_2^r, and v_3^r into V.
- We put an edge between two vertices v_i^r and v_j^s if both of the following holds (***Reduction rules***):
  - 1. v_i^r and v_j^s are in different triples, that is $r \neq s$, and (同一triple的vertex無edge)
  - 2. their corresponding literals are *consistent* (l_i^r is not the negation of l_j^s) (互為negation的literals無edge)
- We can build this graph from the formula ϕ in polynomial time.
  - ***Forward proof:***
    Suppose that ϕ has a satisfying assignment.
    - » Then each clause C_r contains at least one literal l_i^r that is assigned to 1, and each such literal corresponds to a vertex v_j^s.
    - » Picking one such “true” literal from each clause yields a set V’ of k vertices.
      → ***V’ is a clique***. (according to the reduction rules)

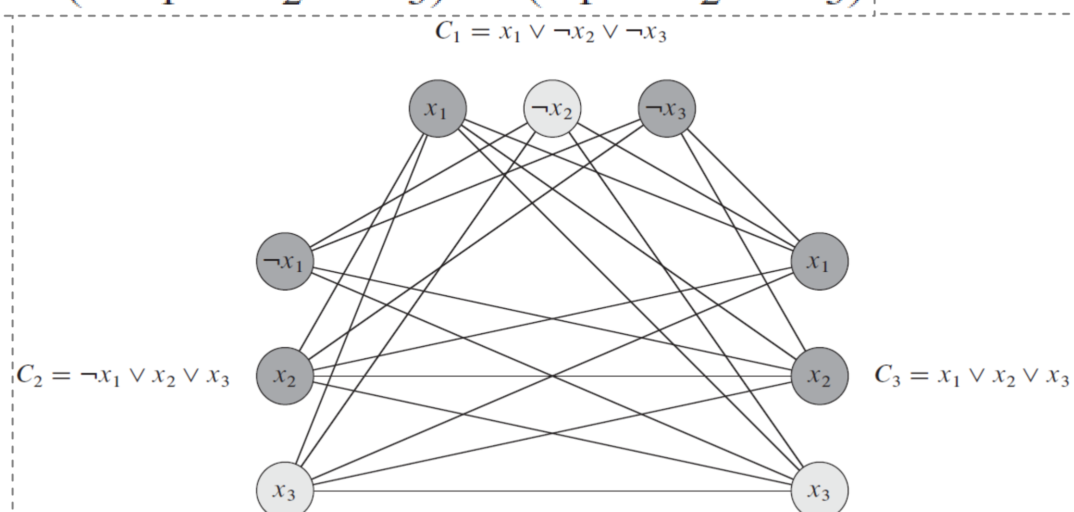
The Clique Problem (Cont.)

– 2. 3-CNF-SAT $\leq_P$ CLIQUE (Cont.)

- ***Reverse proof:***
  Suppose that G has a clique V' of size k.
  - » No edges in G connect vertices in the same triple → V' contains exactly one vertex per triple.
  - » Assign 1 to each literal l_i^r such that $v_i^r \in V'$ → ***Each clause is satisfied and so ϕ is satisfied.***

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

Satisfy assignment:
$x_2=0$, $x_3=1$





The Vertex-Cover Problem

- A ***vertex cover*** of an undirected graph G = (V, E) is a subset V’ $\subseteq$ V such that if (u, v)$\in$E, then u $\in$ V’ or v $\in$ V’ (or both).
  - That is, each vertex “covers” its incident edges, and a vertex cover for G is a set of vertices that covers all the edges in E.
  - The ***size*** of a vertex cover is the number of vertices in it.
- The ***vertex-cover problem*** is to find a vertex cover of *minimum size* in a given graph.
- Restating this optimization problem as a *decision problem*, we wish to determine whether a graph has a vertex cover of a given size *k*. As a language, we define

$$\text{VERTEX-COVER} = \{\langle G, k\rangle : \text{graph } G \text{ has a vertex cover of size } k\}$$



The Vertex-Cover Problem (Cont.)

- ***Theorem 34.12***
  - The vertex-cover problem is NP-complete.
- ***Proof***
  - **1. VERTEX-COVER$\in$NP. (Verify in polynomial time)**
    - Suppose we are given a graph G = (V, E) and an integer *k*.
    - The certificate we choose is the vertex cover V'$\subseteq$V itself.
    - The verification algorithm affirms that |V'|=k, and then it checks, for each edge (u, v)$\in$E, that u $\in$ V' or v $\in$ V'. We can easily verify the certificate in polynomial time.
  - **2. CLIQUE $\leq_P$ VERTEX-COVER (Prove CLIQUE is NP-hard)**

Given an undirected graph $G = (V, E)$, we define the ***complement*** of G as $\overline{G} = (V, \overline{E})$, where $\overline{E} = \{(u, v) : u, v \in V, u \neq v, \text{ and } (u, v) \notin E\}$. In other words, $\overline{G}$ is the graph containing exactly those edges that are not in G.



The Vertex-Cover Problem (Cont.)

– **2. CLIQUE $\leq_P$ VERTEX-COVER (Cont.)**
  - The reduction algorithm takes as input an instance <G, k> of the clique problem. It computes the complement $\overline{G}$, which we can easily do in polynomial time.
  - To complete the proof, we show that this ***transformation*** is indeed a ***reduction***:

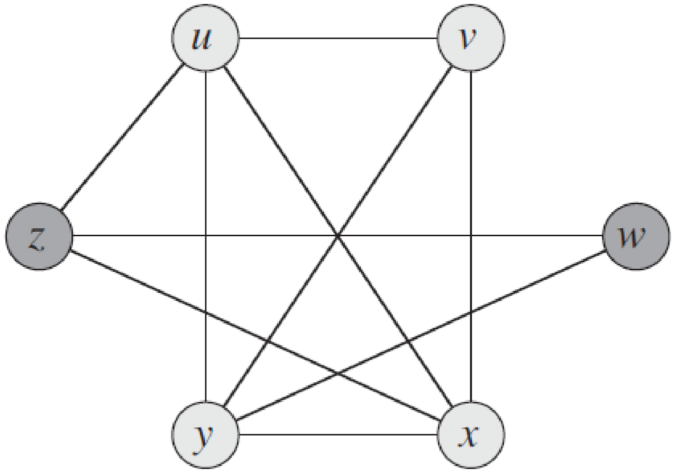
    The graph G has a clique of size k if and only if the graph $\overline{G}$ has a vertex cover of size |V|-k.

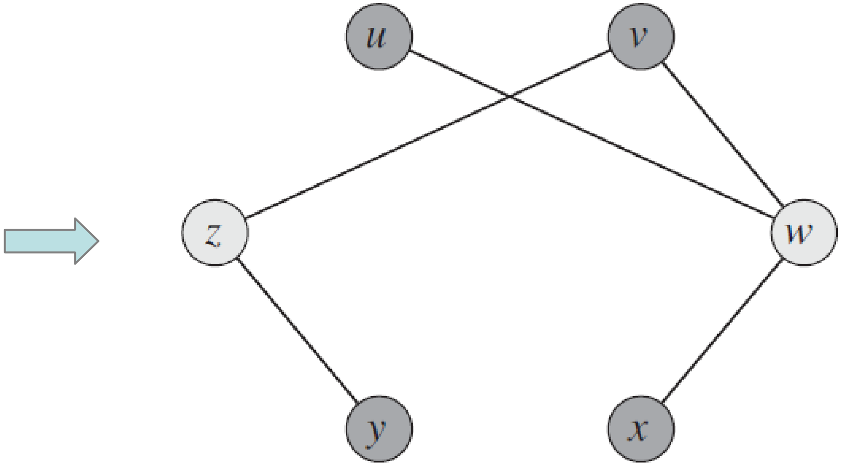
  - ***Forward proof:***
    Suppose that G has a clique V'⊆V with |V'|=k. we claim that V-V' is a vertex cover in $\overline{G}$.
    - Let (u, v) be any edge in $\overline{E}$. Then (u, v)∉E, which implies that at least one of u or v does not belong to V', since every pair of vertices in V' is connected by an edge of E.
      → Every edge (u, v) is covered by a vertex in V-V'.
  -



The Vertex-Cover Problem (Cont.)

- ***Reverse proof:***
  Suppose that $\overline{G}$ has a vertex cover $V' \subseteq V$, where $|V'|=|V|-k$.
  - Then for all $u,v \in V$, if $(u, v) \in \overline{E}$, then $u \in V'$ or $v \in V'$ or both.
    → for all $u, v \in V$, if $u \notin V'$ and $v \notin V'$, then $(u, v) \in E$.
    → In other words, $V-V'$ is a clique, and its size $= |V|-|V'| = k$.





CLIQUE: V'={u, v, x, y}

VERTEX-COVER: V-V' = {w, z}