COMP4600
Complexity Theory
NP-Completeness

Lecture Notes* by

Paulette Lieby

ANU

paulette.lieby@anu.edu.au

*based on *Introduction to Algorithms*, 3nd edition, T.H. Cormen, C.E. Leiserson, R.L. Rivest and using slides from Beata Faller and Pascal Schweitzer

Complexity Theory

- Computational Complexity Theory studies how difficult algorithmic problems are:
  - Can we solve the problem efficiently?
  - What running time can we achieve?
  - When do we call an algorithm efficient?
  - Why have no fast algorithms been developed for some problems?

Complexity Theory

- Some problems are harder than others. This indicates that there might not be fast algorithms that solve them.
- Complexity theory provides the tools to analyse the 'hardness' of problems.

Outline

1. Polynomial time: the class P
2. Polynomial time verification: Verification algorithms, the class NP
3. NP-complete problems and Reductions:
    - classifying problems
    - the first NP-complete problem: Circuit-SAT
4. NP-completeness problems: examples and proofs

Polynomial time

- An algorithm has *polynomial running time* if it runs in time $O(n^k)$ for some fixed k and input size n.
- Recall

  that $O(a_1 n^d + a_2 n^{d-1} + \ldots + a_{d-1} n + a_d) \subseteq O(n^d)$.
- We consider polynomial time algorithms as tractable. (This is arguably not appropriate for running times of $\Theta(n^{100})$.)

Polynomial Time

- We know polynomial time algorithms for various problems:
  - Sorting: $O(n \log n)$
  - Shortest path: $O(m + n \log n)$
  - Matrix Multiplication: $O(n^{2.807}) \subseteq O(n^3)$
  - Max-Flow, Min-Cut: $O(n^3)$
  - Primality: $O((\log n)^7)$
  - Minimum Spanning Tree: $O(m + n \log n)$

Polynomial time

- Why consider polynomial time?
  - Invariant under many computational models (RAM, Turing machine, parallel computation with polynomially many processors)
  - Polynomials are closed under addition, multiplication, composition (transitivity: input of one program can be fed into another)
  - Polynomial time algorithms are essentially tractable

A Formal Approach

- What is a (computational) problem? define a *abstract problem* as a binary relation on a set I of problem *instances* and a set S of problem *solutions*.
- example:
  - Instance of SHORTEST-PATH problem: a graph, a source vertex s and a target vertex t.
  - Solution of SHORTEST-PATH problem: sequence of vertices forming a shortest path from s to t.

Decision Problems

- We will only consider decision problems (so we can use the theory of formal languages).
- A *decision problem* is a problem with solution set $S = \{\text{No}, Yes\}$ (equivalently $\{0, 1\}$).
- Decision problem associated with SHORTEST-PATH:
  Instance $i = \langle G, s, t, k \rangle$ asks: Is there a path in G of length at most k from s to t?

Decision Problems

- Every optimisation problem whose output is a number can be associated with a decision problem.
- Solving the decision problem is no harder that solving the optimisation problem.

Concrete Problems

- A *concrete problem* is a problem whose instance set is a set of binary strings $\{0,1\}^*$.

Polynomial Time

- An algorithm solves a problem *in time* $O(T(n))$, if for all inputs of length n the algorithm produces a solution in $O(T(n))$ time steps.
- A problem is *polynomial-time solvable* if there is an algorithm that solves it in polynomial time (i.e., $O(n^k)$ for some fixed k).
- The *complexity class* P is the set of concrete decision problems that are solvable in polynomial time.

Encodings

- Problem instances must be encoded for a computer to solve the problem.
- An *encoding* maps a set of abstract objects S to strings over an alphabet.
  - Encodings turn abstract problems into concrete problems.
- Examples of encodings of $\mathbb{N}$:
  - Binary encoding: $0 \mapsto 0$, $1 \mapsto 1$, $2 \mapsto 10$, $3 \mapsto 11$, $\ldots$
  - Unary encoding: $0 \mapsto \varepsilon$, $1 \mapsto 1$, $2 \mapsto 11$, $3 \mapsto 111$, $\ldots$

Polynomial Time and Encodings

- But solvability complexity may depend on the encoding chosen:
  - Unary vs. binary encoding: Input length k vs $\log k$. An algorithm that runs in $O(k)$ for unary runs in $O(2^{\log k})$ for binary.

Polynomial Time and Encodings

- In practice *reasonable* encodings are used:
  - "equivalent" encodings that make no difference to whether the problem can be solved in polynomial time
  - for example, a binary encoding (i.e. base 2) is such an encoding (or any encoding in some base b)
  - but a unary encoding is not

Polynomial Time and Encodings

- A function $f\colon \{0,1\}^* \to \{0,1\}^*$ is *polynomial-time computable* if there is a polynomial-time algorithm that computes $f(x)$ when given x as input.
- Two encodings are *polynomially related* if there are polynomial-time computable functions f_{12} and f_{21} that transform the encodings into each other.
- Example: Graphs given as adjacency list or adjacency matrix.

Polynomial Time and Encodings

For an encoding e of an abstract problem Q, denote by $e(Q)$ the concrete problem associated with Q.

Lemma 1. *If e_1 and e_2 are two polynomially related encodings of an abstract decision problem Q, then $e_1(Q) \in \mathsf{P}$ if and only if $e_2(Q) \in \mathsf{P}$.*

Proof. $\rightarrow$ Suppose $e_1(Q)$ can be solved in time $O(n^k)$ for input size n. Suppose that an encoding $e_1(i)$ can be computed from encoding $e_2(i)$ in time $O(|e_2(i)|^c)$.
Since $|e_1(i)| \in O(|e_2(i)|^c)$, $e_2(i)$ is solved in
$O(|e_1(i)|^k) \subseteq O\Big(\big(O(|e_2(i)|^c)\big)^k\Big) \subseteq O(|e_2(i)|^{ck})$. $\square$

A Formal-Language Framework

- Concrete decision problems are viewed as formal *languages*.
- Definitions:
  - An *alphabet* Σ is a finite set of symbols.
  - A language L over Σ is a subset of strings over Σ^* (i.e., $L \subseteq \Sigma^*$).
  - The *empty string* is ε.

A Formal-Language Framework

- Operations on Languages
  - union, intersection
  - complement: $\overline{L} = \Sigma^* \setminus L$
  - concatenation of L_1 and L_2 is

    $L_1L_2 = \{x_1x_2 \colon x_1 \in L_1 \text{ and } x_2 \in L_2\}$
  - closure (or Kleene star):

    $L^* = \{\varepsilon\} \cup L \cup LL \cup LLL \cup \ldots$

A Formal-Language Framework

- Since a decision problem Q is entirely characterised by those problem instances that produce a Yes answer, we can view Q as a *language* L:

$$L = \{x \in \Sigma^* \colon Q(x) = 1\}.$$

- Algorithms vs Languages:
  - An algorithm A *accepts a string* if $A(x) = 1$.
  - An algorithm A *rejects a string* if $A(x) = 0$.

Accepting/Deciding

- Given a language L,
  - L is *accepted by an algorithm* A if $L = \{x \in \Sigma^*\colon A(x) = 1\}$.
  - L is *decided by an algorithm* A, if every string in L is accepted by A and every string not in L is rejected by A.
- Algorithms do not always halt. Thus "A decides a language" is *NOT* equivalent to "A accepts a language".

Accepting/Deciding

- An algorithm A *accepts L in polynomial time* if A accepts L and for all $x \in L$ algorithm A accepts x in $O(|x|^k)$ (for some constant k).
- An algorithm A *decides L in polynomial time* if A decides L and for all $x \in \{0, 1\}^*$ algorithm A decides whether $x \in L$ in $O(|x|^k)$ (for some constant k).

Accepting/Deciding

- INPUT: a linked list.
- TASK: decide whether starting at the root one reaches a NIL pointer.
- An algorithm: Start at the root and repeatedly jump to the position of the next pointer. If NIL is reached accept.
- The algorithm accepts all inputs for which NIL can be reached starting from the root.
- It accepts in polynomial time but it does not decide in polynomial time.

The class P Revisited

- Recall: The class P is the set of concrete decision problems that are solvable in polynomial time.
- An equivalent definition:

  P $= \{L \subseteq \{0,1\}^*:$

  there is an algorithm that decides L in polynomial time$\}$

P-decided = P-accepted

- But:

  The languages that can be decided in polynomial time are exactly the languages that can be accepted in polynomial time.

Theorem 1.

$\mathsf{P} = \{\boldsymbol{L} \subseteq \{0,1\}^* :$

there is an algorithm that accepts $\boldsymbol{L}$ in polynomial time$\}$.

P-decided = P-accepted: Proof

Proof. If a language is decided in polynomial time, it is accepted in polynomial time.

Need to show: If a language L is accepted in polynomial time by some algorithm A, it is also decided by some algorithm A' in polynomial time.

A accepts in time $O(n^k)$, thus there is a positive constant c such that A accepts in time at most cn^k.

P-decided = P-accepted: Proof cont.

Proof cont. We design A' as follows: Simulate algorithm A for cn^k steps keeping track of the number of steps that have been performed. If A has accepted, then accept. Otherwise reject.

Correctness: Algorithm A' decides L.

Running time: Simulating A takes only a constant factor longer than running A itself. Thus A' has polynomial running time. $\square$

Polynomial-time Verification

- Some problems are hard to solve, but they are easy to verify when a certificate is available.

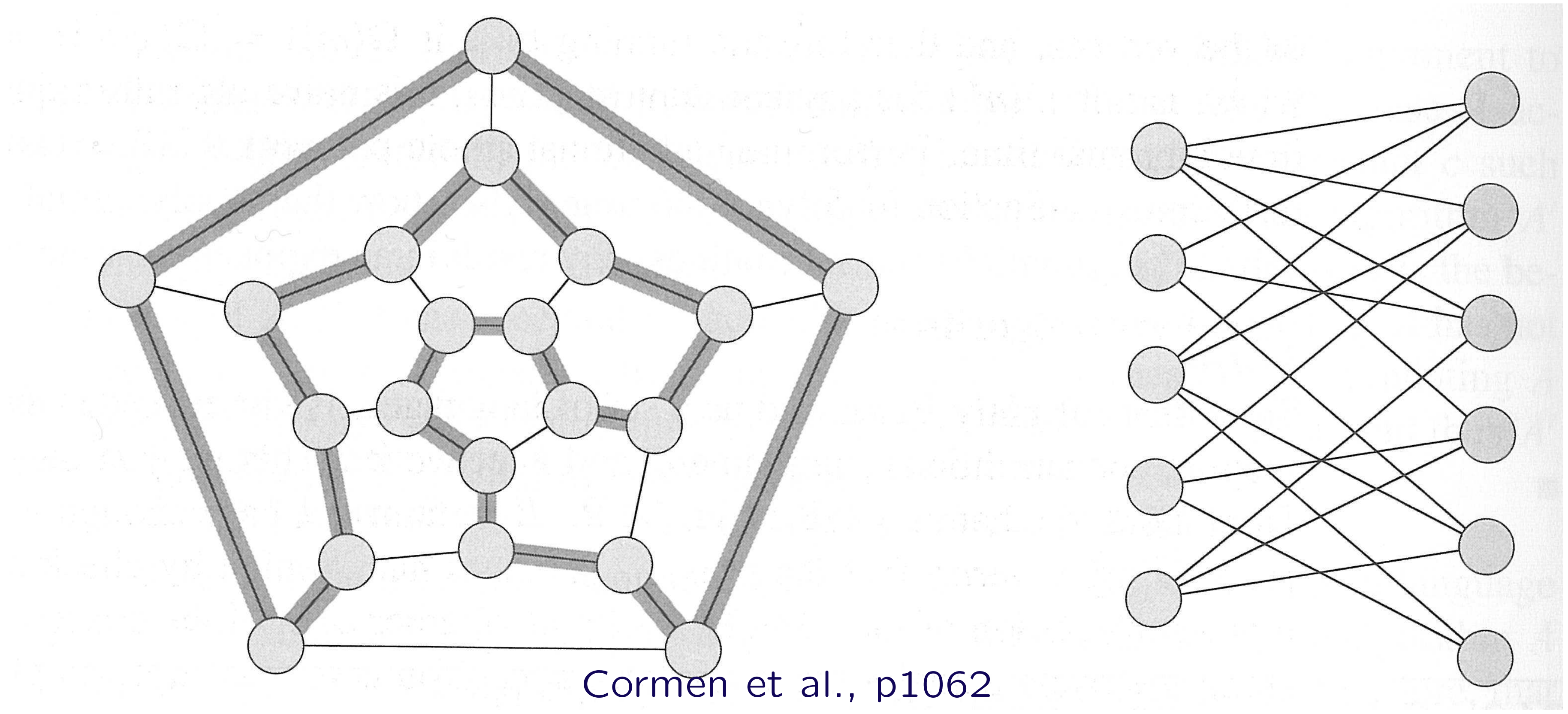
Polynomial-time Verification: Examples

- Factorisation: Given a set of integers, it is easy to verify they are the prime factors of some number n.
- Traveling salesman problem: Given a walk in an edge-weighted graph, it is easy to verify that it goes through all the nodes and has length no greater than l.
- Maximum clique: Given a set of vertices of a graph, it is easy to verify that they form a clique.
- Chromatic number: Given a colouring of the vertices of a graph, it is easy to verify that it is a proper k-colouring.

Polynomial-time Verification: HAMILTON-CYCLE

- The problem HAMILTON-CYCLE.
  - INPUT: A graph.
  - TASK: Find a simple cycle in the input graph that includes every vertex.
  - If such a cycle exists, the graph is called *Hamiltonian*.

Polynomial-time Verification: HAMILTON-CYCLE



Cormen et al., p1062

Polynomial-time Verification: HAMILTON-CYCLE

- HAMILTON-CYCLE can be verified in linear time.
- How fast can we decide HAMILTON-CYCLE?
- The brute force algorithm takes $\Omega(n!)$.
- Better algorithms are known:
  - $O(2^n)$ [Held, Karp] (1962)
  - $O(1.657^n)$ [Björklund] (2010)
- No polynomial time algorithm is known.

Verification Algorithms

- A *verification algorithm* is an algorithm A that takes two arguments:
  - a normal input string x,
  - a string y called a *certificate*.
- A *verifies* x if there exists a certificate y such that $A(x, y) = 1$.
- The *language verified by A* is $L = \{x \in \{0, 1\}^* : \exists y \in \{0, 1\}^* \text{ such that } A(x, y) = 1\}$.
- Example: A Hamiltonian cycle serves as a certificate that a graph is Hamiltonian.

Polynomial-time Verification: The Complexity Class NP

- The *complexity class* NP is the class of languages that can be verified in polynomial-time.
- Thus: $L \in$ NP if there exists a verification algorithm running in polynomial time and a constant c such that $\boldsymbol{L = \{x \in \{0,1\}^* \colon \exists y}$ with $\boldsymbol{|y| \in O(|x|^c)}$ and $\boldsymbol{A(x,y) = 1\}}$.

The Class NP

- So, HAMILTON-CYCLE $\in$ NP.
- **Observation**: P $\subseteq$ NP.

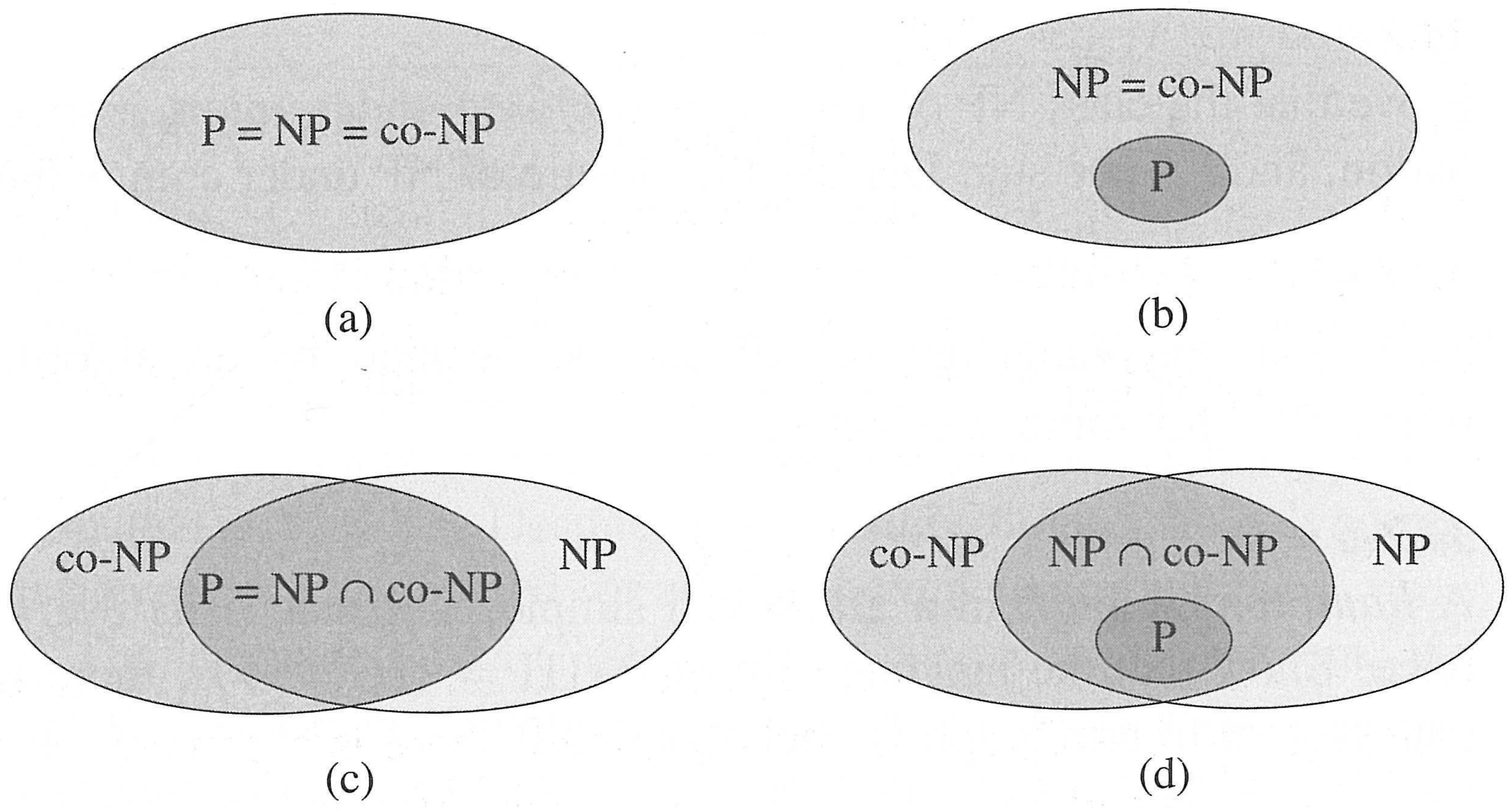
  Proof. For $L \in$ P there is a polynomial time algorithm A that decides L. Using no certificate (i.e., $y = \varepsilon$) this algorithm verifies L in polynomial time. $\square$
- It is unknown whether P $=$ NP.

The Class co-NP

- The *complexity class* co-NP is the class of languages L such that $\overline{L} \in \mathrm{NP}$.

- Open problems concerning NP:
  - $\mathrm{P} = \mathrm{NP}$?
  - $L \in \mathrm{NP} \Rightarrow \overline{L} \in \mathrm{NP}$ (equivalent co-NP $=$ NP)?
  - $\mathrm{P} = \mathrm{NP} \cap \text{co-NP}$?

The Class co-NP

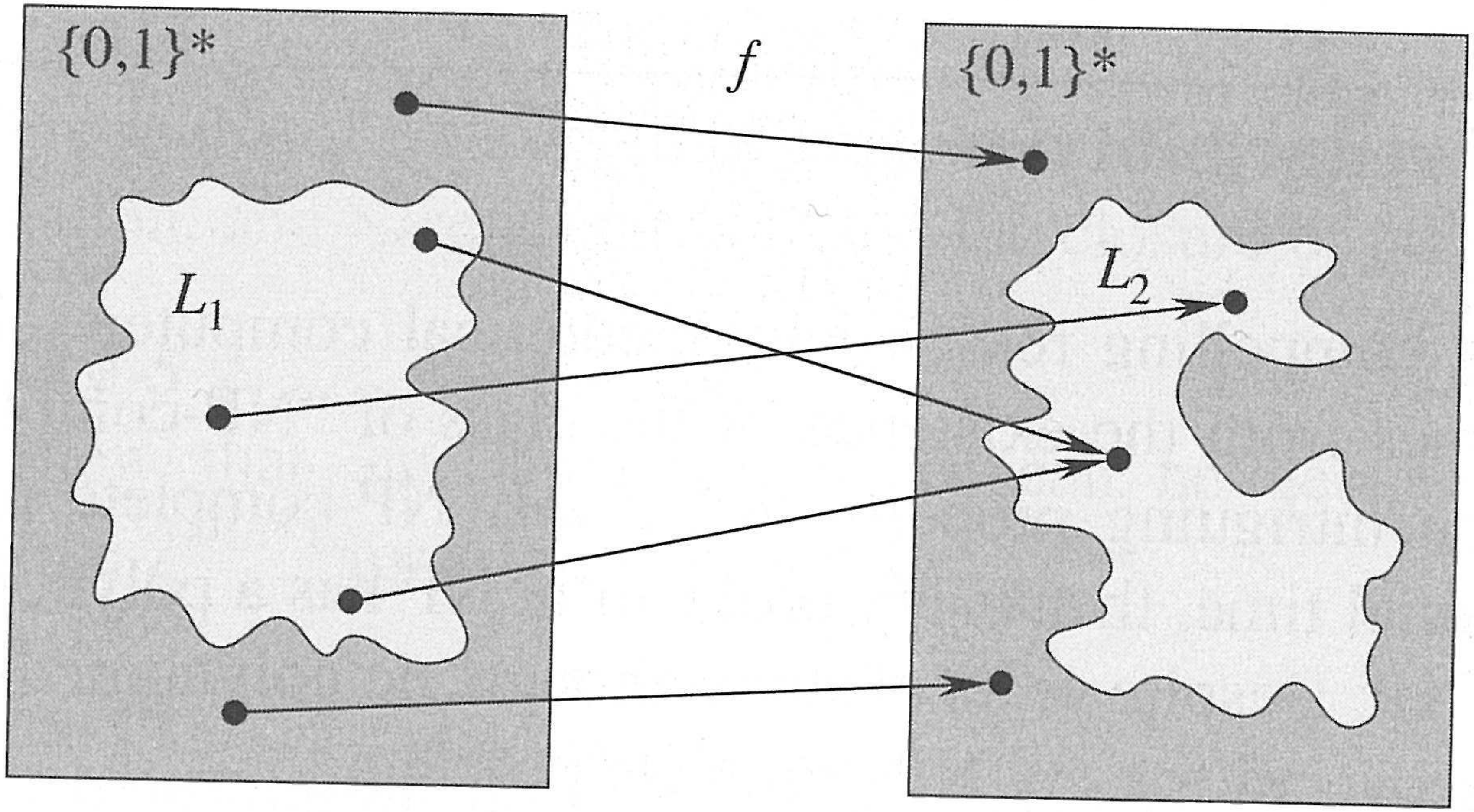


Cormen et al., p1065

NP-Completeness and Reducibility

- Informally NP-complete problems are the hardest problems in NP.
- If some NP-complete problem is in P then P = NP (we'll show this later).
- We need the concept of *reduction*:
  - Language L_1 *polynomial-time reduces* to L_2 if there exists a polynomial-time computable function $f\colon \{0,1\}^* \to \{0,1\}^*$ such that for all $x \in \{0,1\}^*$, $x \in L_1$ if and only if $f(x) \in L_2$.
  - Write $L_1 \leq_P L_2$.

Reducibility



Cormen et al., p1068

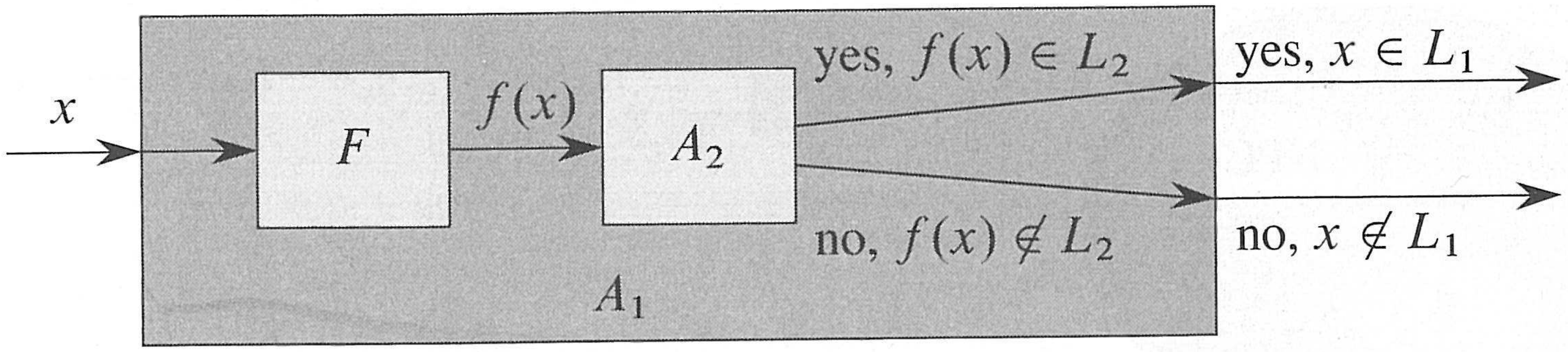
Reducibility

Lemma 2. *Let $L_1, L_2 \subseteq \{0,1\}^*$ be two languages.*

If $L_1 \leq_P L_2$, then $L_2 \in \mathsf{P} \Rightarrow L_1 \in \mathsf{P}$.

Proof.
A_2: polynomial-time algorithm that decides L_2.
f: polynomial-time computable reduction reducing L_1 to L_2.
We need: A_1, a polynomial-time algorithm that decides L_1.
Given $x \in \{0,1\}^*$, compute $f(x)$. Run A_2 on input $f(x)$.
If $f(x) \in L_2$, then output "Yes", otherwise output "No".
Running time: $f(x)$ can be computed in time polynomial in $|x|$.
Also, A_2 runs in time polynomial in $|f(x)|$, where $|f(x)|$ is polynomial in $|x|$. Thus, the total running time is polynomial in $|x|$. $\square$

Reducibility



Cormen et al., p1069

NP-Completeness

- A language L is NP-*complete* if
  - $L \in \mathrm{NP}$ and
  - $L' \leq_P L$ for every $L' \in \mathrm{NP}$.
- If only the second property holds then L is called NP-*hard*.
- NPC is the complexity class of all NP-complete languages.

NP-Completeness

Theorem 2. *If some* NP*-complete problem is in* P *then* P $=$ NP*.*

Proof. Suppose $L \in$ P and $L \in$ NPC.

Let $L' \in$ NP be any language.

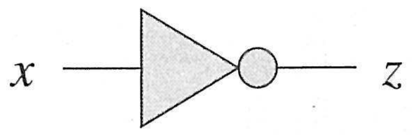
Since $L \in$ NPC we know $L' \leq_p L$.

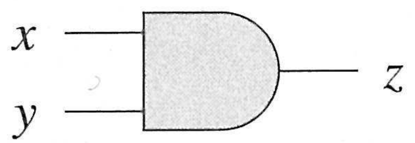
Thus, by Lemma 2, we conclude $L' \in$ P. $\square$

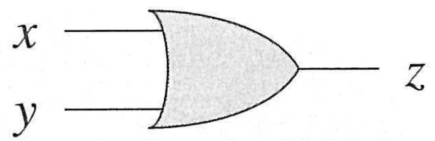
A first NP-Complete Problem: Circuit Satisfiability

- Do NP-complete problems exist?
- A first NP-complete problem: circuit satisfiability:
  - Boolean combinational circuits are built from basic building blocks.
  - We consider building blocks with 1 or 2 inputs (wires) and 1 output. These are called logic gates.
  - The gates are:
    - NOT gate
    - AND gate
    - OR gate

A first NP-Complete Problem: Circuit Satisfiability







x	$\neg x$
0	1
1	0

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

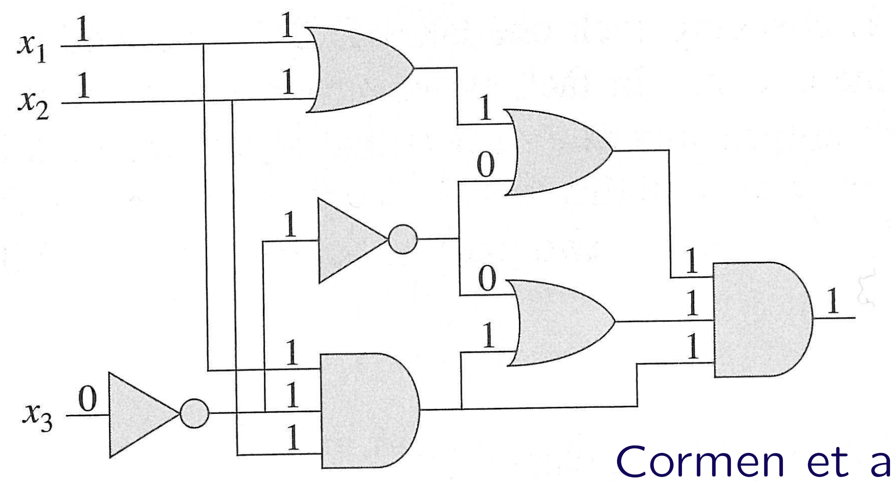
x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

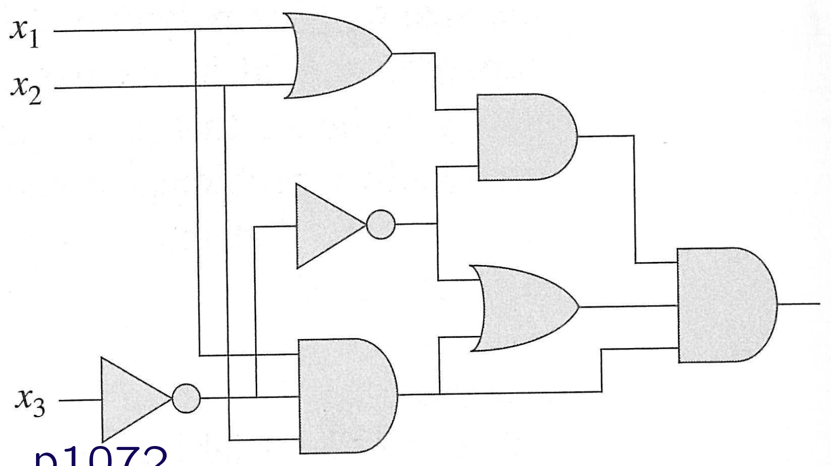
Cormen et al., p1071

Circuit Satisfiability (CIRCUIT-SAT)

- A boolean circuit consists of several gates that are connected by wires.
- We consider only circuits that have one output (decision problem).
- A *truth assignment* is a set of values for all inputs of a circuit.
- A circuit is *satisfiable* if it has a truth assignment that produces an output value 1.
- Circuits can be encoded in a similar fashion as graphs.
- The circuit *size* is the number of gates plus the number of wires used.

CIRCUIT-SAT





Cormen et al., p1072

CIRCUIT-SAT

Lemma 3. *CIRCUIT-SAT is in* NP*.*

Proof. We need: Algorithm A that verifies circuit satisfiability in polynomial time.

Input: Encoding of a circuit.

Certificate: Truth assignment for all wires.

Algorithm A: For each logic gate in the circuit compute the output of the gate and check whether the output wire has the correct assignment. If the output is correct for every gate, and the output wire of the whole circuit is 1 then output 1. Otherwise, output 0.

CIRCUIT-SAT

Proof cont. *Correctness*: If a circuit is satisfiable, then there is an assignment of the wires that will make A output 1. If a circuit is not satisfiable, then every assignment for the wires is inconsistent with the gates and A outputs 0.

Running time: The running time is polynomial in the number of gates, thus polynomial in the input size. $\square$

CIRCUIT-SAT is in NP

- CIRCUIT-SAT can be verified in $O(n)$ (where $n =$ size of the circuit).
- CIRCUIT-SAT can be decided in $O(2^k \cdot n)$ (when $k =$ the number of inputs) using brute force.

CIRCUIT-SAT is NP-Hard

- Showing that CIRCUIT-SAT is NPC: i.e. that every language in NP is P-reducible to CIRCUIT-SAT.
- A program/algorithm running on a computer requires
  - storage assigned to its set of instructions
  - an input
  - storage assigned for the output
  - a program counter keeping track of the instructions to be performed
  - working storage
  - storage for general bookkeeping

Program/Algorithm Configuration

- This defines a *configuration*.
- The execution of an instruction maps a configuration to another configuration.
- Mapping a configuration to another can be implemented as a boolean circuit M.

CIRCUIT-SAT is NP-Hard

Lemma 4. *CIRCUIT-SAT is* NP*-hard.*

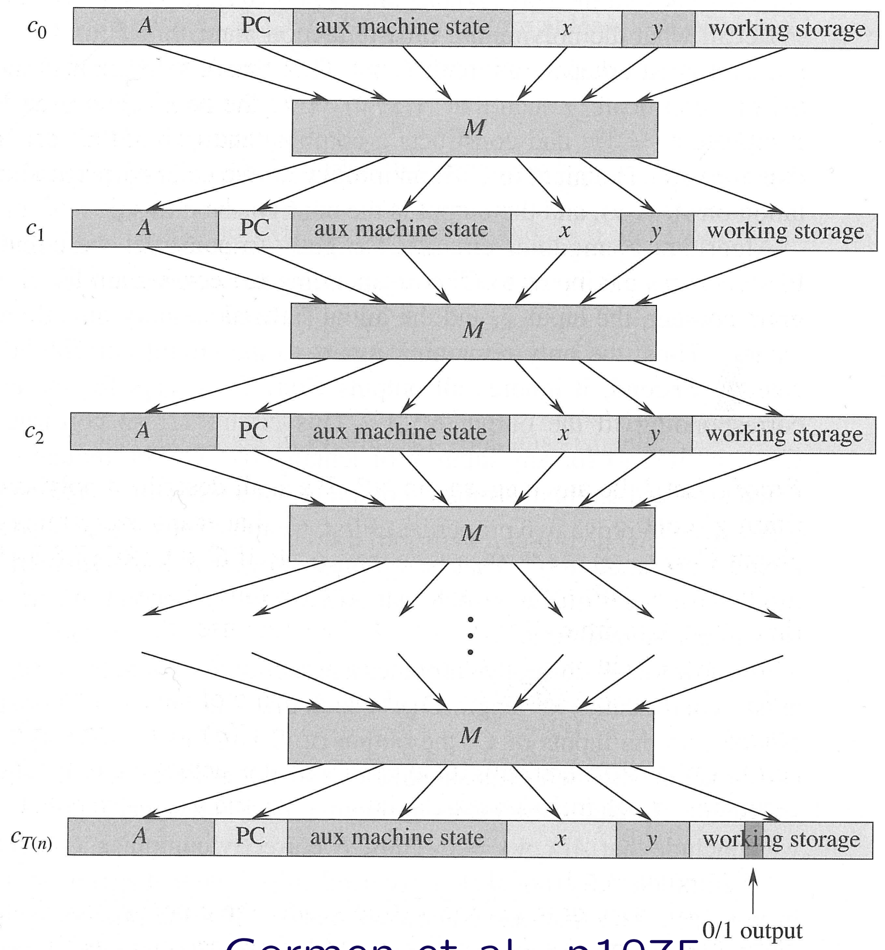
Proof sketch. Let $L \in \mathrm{NP}$ be a language. We construct a P-time reduction from L to CIRCUIT-SAT (so $L \leq_P$ CIRCUIT-SAT).

Let A be the algorithm that verifies L in time $O(n^k)$, for some k and construct the circuit C that maps all the configurations of A corresponding to its execution (i.e. $O(n^k)$ copies of M).

Transform C into C' that takes the certificate y as input and outputs 0 or 1.

Correctness is easy to verify, as well as P-time. $\square$

CIRCUIT-SAT is NP-Hard



Cormen et al., p1075

CIRCUIT-SAT is NP-Complete

Theorem 3. *CIRCUIT-SAT is* NP*-complete.*

Proof. Lemma 3 and 4 together give the theorem. $\square$

NP-Completeness Proofs

- Knowing one NP complete problem allows us to derive more NP-complete problems.

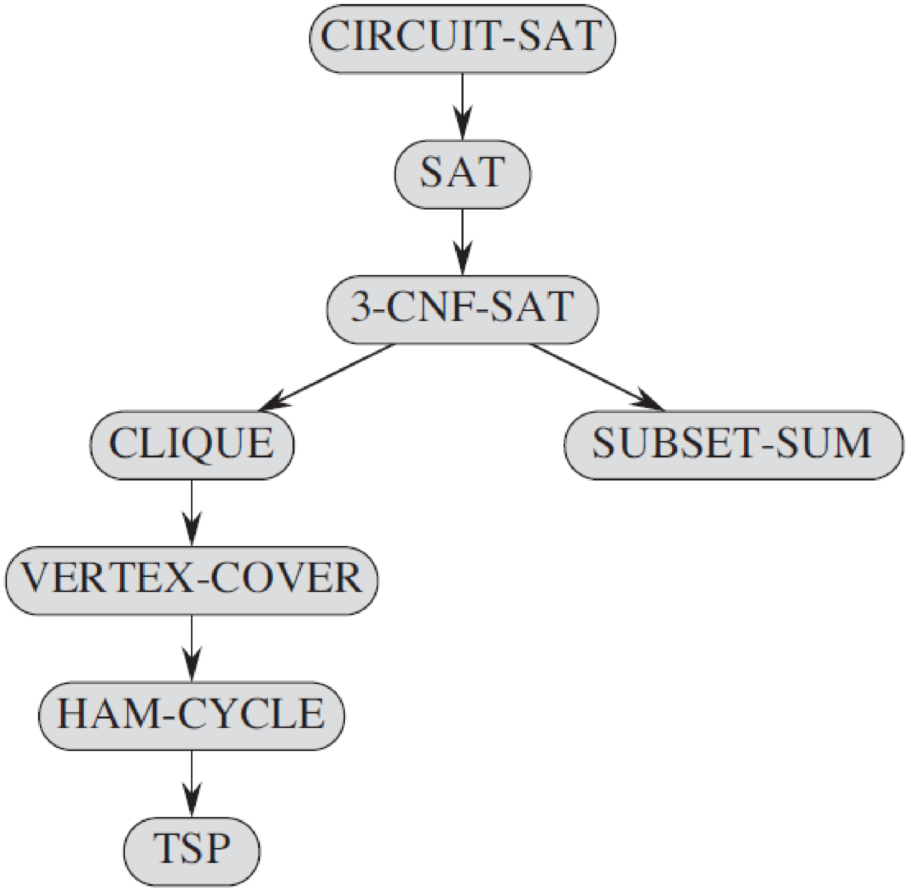
Theorem 4. *If L is a language and $L' \leq_P L$ for a language $L' \in \mathrm{NPC}$ then L is NP-hard. If additionally $L \in \mathrm{NP}$, then $L \in \mathrm{NPC}$.*

Proof. L' is NP-complete. Thus $\forall L'' \in \mathrm{NP}$, $L'' \leq_P L' \leq_P L$, and $L'' \leq_P L$. □

NP-Completeness Proofs

- To prove that a language L is NP-complete:
  1. Show $L \in$ NP.
  2. Choose an language $L' \in$ NPC.
  3. Design a function f that maps instances of L' to instances of L.
  4. Prove that f is a reduction: for all $x \in \{0, 1\}^*$, $x \in L'$ if and only if $f(x) \in L$.
  5. Prove that f can be computed in polynomial time.
- This method is called reduction [Karp (1972)].

NP-Completeness Proofs



Cormen et al., p1087

NP-completeness Proofs: Formula Satisfiability

- Show that satisfiability of boolean formulas (SAT) is NP-complete.
- A SAT instance consists of
  - n boolean variables $x_1, x_2, \ldots x_n$,
  - m boolean operators taken from: $\wedge$ (AND), $\vee$ (OR), $\neg$ (NOT), $\rightarrow$ (implication), $\leftrightarrow$ (iff)
  - parentheses (w.l.o.g. we assume the number of parentheses is $O(m)$)
- A formula is *satisfiable* if there exists a truth assignment for the variables such that the formula evaluates to 1.

NP-completeness Proofs: SAT

- Example instances of SAT:
  - $(x_1 \wedge x_2) \leftrightarrow x_3$:

    satisfiable with $x_1 = 0, x_2 = 0, x_3 = 0$
  - $((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$:

    satisfiable with $x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1$
  - $(x_1 \rightarrow x_2) \wedge (x_1 \wedge \neg x_2)$:

    not satisfiable
- The naive algorithm takes time $\Omega(2^n)$.

NP-completeness Proofs: SAT

Theorem 5 (Cook (1976)). *SAT ∈ NPC.*

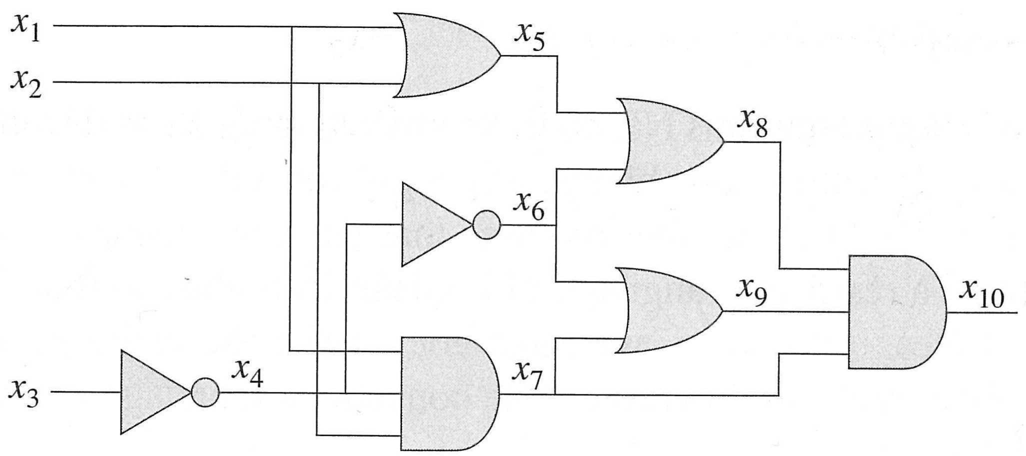
Proof. 1. SAT∈ NP: SAT can be verified in polynomial time. Certificate: Satisfying assignment. The verification algorithm: Replace each variable of formula with its truth value and evaluate each expression. If the final value is 1 then accept.

2. SAT is NP-hard. Reduce from CIRCUIT-SAT: Design a function that transforms a circuit C into a formula ϕ.

CIRCUIT-SAT $\leq_P$ SAT

Proof cont. Each input of C is a variable of ϕ as well as each wire. Each gate is written as a clause ϕ_i $y_1 \leftrightarrow (x_1 \odot x_2 \ldots \odot x_k)$ where y_1 is the output of the clause, $x_1, \ldots, x_k$ its inputs, and $\odot$ the connector implemented by the gate. Combining all the ϕ_i into a conjunctive formula gives ϕ.
ϕ is constructed in P-time (linear in number of gates), and ϕ is satisfiable iff C is satisfiable. $\square$

CIRCUIT-SAT $\leq_P$ SAT



Cormen et al., p1081

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

NP-Completeness Proofs: 3-CNF-SAT

- *3-CNF-SAT* (3-conjunctive normal form satisfiability) is the language that consists of boolean formulas ϕ such that
  - ϕ is a conjunction (AND) of clauses ϕ_i
  - each clause ϕ_i is a disjunction (OR) of exactly three literals
  - (a literal is the occurrence of a variable x or its negation $\neg x$)
- Example:

  $(x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee \neg x_2 \vee \neg x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$

NP-Completeness Proofs: 3-CNF-SAT

Theorem 6. *3-CNF-SAT is* NP*-complete.*

Proof sketch. 1. 3-CNF-SAT $\in$ NP: easy since SAT $\in$ NP.

2. SAT $\leq_P$ 3-CNF-SAT: i) parse a formula ϕ as a binary tree and write it in conjunctive form ϕ' as in the proof of SAT (with the tree vertices replacing the gates).

ii) write the truth table for ϕ' and extract the disjunctive form for $\neg\phi'$.

iii) applying DeMorgan's laws gives a conjunctive form equivalent to ϕ.

iv) extend all clauses to three literals. $\square$

NP-Complete Problems

- NP-complete problems occur in many areas of computer science and mathematics.
- Several thousand are known, the list still growing.
- Knowing a variety of NP-complete problems is helpful, since each NP-complete problem can be used to show that other problems are NP-complete.

NP-Complete Problems: The Clique Problem

- Given a graph $G = (V, E)$ and an integer k, the *clique problem* (CLIQUE) asks whether there exists a clique of size k in G.
- naively: try all possible subsets of size k of V; this takes $\Omega(k^2\binom{|V|}{k})$ time, where $n = |V|$
- This is exponential in $|V|$ when $k \sim |V|/2$.

Theorem 7. *CLIQUE $\in$ NPC.*

CLIQUE $\in$ NPC

Proof. 1. CLIQUE $\in$ NP (verification takes $O(k^2)$).

2. 3-CNF-SAT $\leq_P$ CLIQUE. Construct a graph $G = (V, E)$ from a 3-CNF formula ϕ with k clauses:

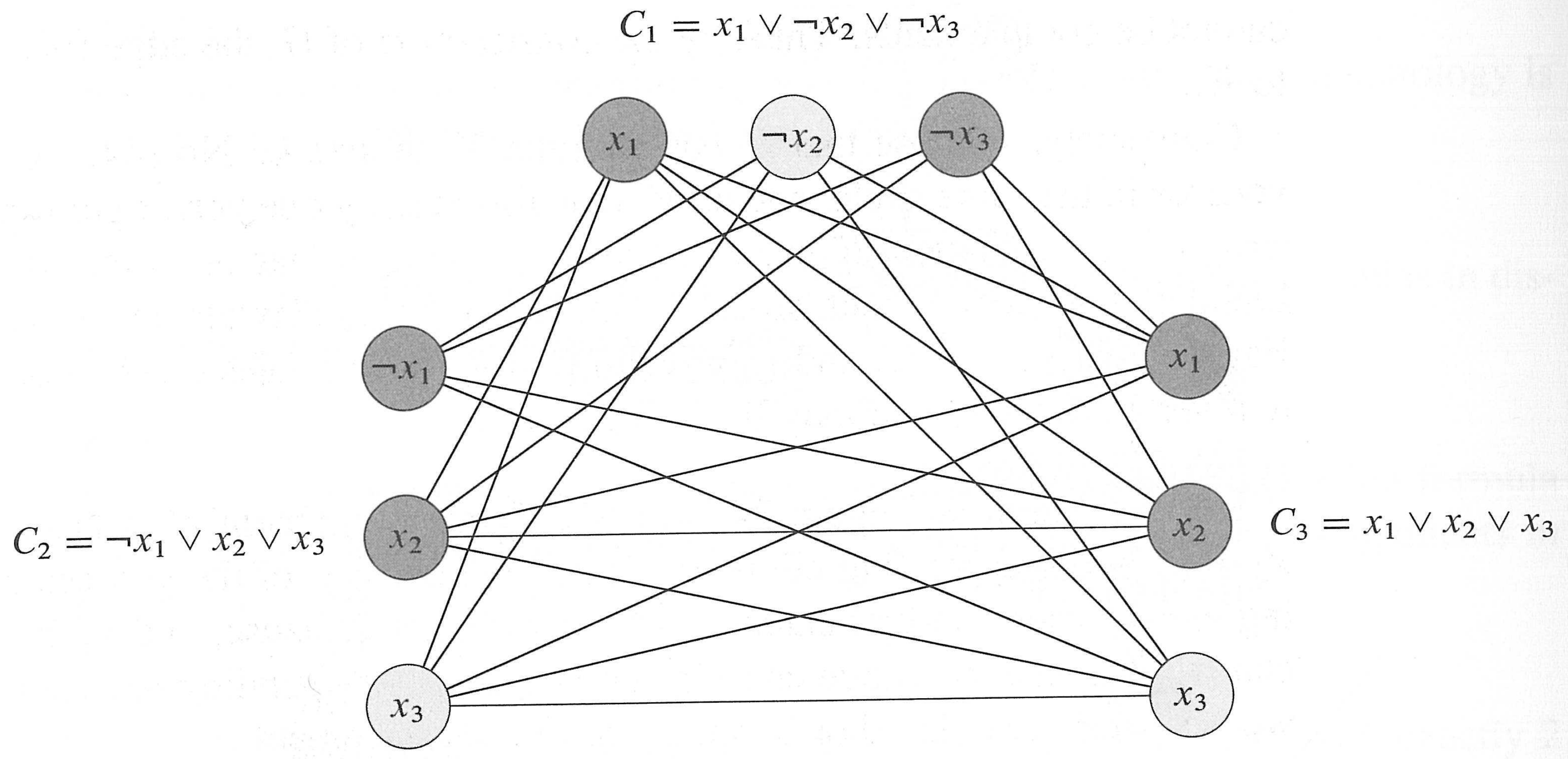
i) for each clause $\phi_i = l_1^i \vee l_2^i \vee l_3^i$ add the vertices l_1^i, l_2^i, l_3^i to V.

ii) add an edge (u, v) to E if u and v belong to different clauses and they are not negation of each other. (Note: G is constructed in P-time and is a multipartite graph with k parts.)

iii) show that ϕ is satisfiable iff there is a clique of size k in G.

CLIQUE $\in$ NPC

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



$C_1 = x_1 \vee \neg x_2 \vee \neg x_3$
x_1
$\neg x_2$
$\neg x_3$
$\neg x_1$
x_1
$C_2 = \neg x_1 \vee x_2 \vee x_3$
x_2
x_2
$C_3 = x_1 \vee x_2 \vee x_3$
x_3
x_3

Cormen et al., p1088

3-CNF-SAT $\leq_P$ CLIQUE

Proof cont. a) Assume that ϕ is satisfiable. Pick a true literal from each clause (there is at least one) and form the set $V' \subset V$. Then V' is a clique.

b) Suppose V' is a clique of size k of G: then vertices of V' all originate from different clauses of ϕ. Assign 1 to each vertex/literal in V' and ϕ is true: there is no inconsistency. $\square$

NP-Complete Problems: The Vertex Cover Problem

- A *vertex cover* of a graph $G = (V, E)$ is a subset $V' \subseteq V$ such that for all edges e in E, at least one endpoint of e is in V'.
- A *minimum vertex cover* of a graph is a smallest vertex cover.
- The *vertex-cover problem* (VERTEX-COVER) asks, given a graph G and an integer k, whether G has a vertex cover with k vertices.

Theorem 8. *VERTEX-COVER $\in$ NPC.*

VERTEX-COVER $\in$ NPC

Proof. 1. VERTEX-COVER $\in$ NP: given $V' \subseteq V$, can check in $O(|E|)$ if V' is a vertex cover.
2. CLIQUE $\leq_P$ VERTEX-COVER. Given a graph G, let $\overline{G}$ be its complement (computed in $O(|E|)$.
i) Let V' be a clique of size k in G. Then $V \setminus V'$ is vertex-cover of size $n-k$ of $\overline{G}$ (all edges in $\overline{G}$ have at least one endpoint in $V \setminus V'$).
ii) If V' is vertex-cover of size $n-k$ in $\overline{G}$, then $V \setminus V'$ is an independent set of $\overline{G}$ and thus a clique of G (of size k). $\square$

NP-Complete Problems: Hamiltonian Cycle

- Hamiltonian cycle (HAM-CYCLE) problem: does there exist a simple cycle through all vertices in an input graph.

Theorem 9. *HAM-CYCLE $\in$ NPC.*

Proof. 1. HAM-CYCLE $\in$ NP.

2. VERTEX-COVER $\leq_P$ HAM-CYCLE (see book).

$\square$

NP-Complete Problems: Traveling-Salesman

- The *traveling-salesman problem* (TSP) asks for the shortest tour that visits all vertices of weighted graph G.
- Or (decision problem): given $k \in N$, does G have a tour of length k that visits all vertices of G.

Theorem 10. *TSP* $\in$ *NPC.*

TSP $\in$ NPC

Proof. 1. TSP $\in$ NP.

2. HAM-CYCLE $\leq_P$ TSP. From a weighted graph $G = (V, E)$ form the graph $G' = K_n$ ($n = |V|$) where an edge e is assigned weight 1 if $e \in E$ and weight 2 otherwise.

Then G' has a tour of length n if and only if G has a Hamiltonian cycle. $\square$