NP-Completeness

Desain & Analisis Algoritma

Fakultas Ilmu Komputer

Universitas Indonesia

Compiled by **Alfan F. Wicaksono** from multiple sources

Credits

- Introduction to *Algorithms*, **Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein**
- **Slide kuliah DAA Pak L. Y. Stefanus (Pak Stef)**

Tractable vs Intractable

- Almost all the algorithms we have studied thus far have been polynomial-time algorithms: on inputs of size n, their worst-case running time is $O(n^k)$ for some constant k.
- Not all problems can be solved in polynomial time.
  - Some problems even cannot be solved by any computer! -> **Turing's Halting Problem**
  - There are also problems that can be solved, but **not in polynomial time**.
- Tractable or easy problems: problems that are solvable by polynomial-time algorithms.
- Intractable or hard problems: problems that require superpolynomial time to be solved.

An Unsolved Problem: Turing's Halting Problem

- Given a computer program and an input, will the program terminate or will it run forever?

- In 1936, **Alan Turing** proved that **no turing machine** can decide correctly (terminate and produce the correct answer) for all possible program/input pairs.
  - Undecidable problem!

NP-Complete Problems (NPC)

- We will study an interesting class of problems, called the NP-complete problems, whose status is unknown.

- No polynomial-time algorithm has yet been discovered for an NP-complete problem, nor has anyone yet been able to prove that no polynomial-time algorithm can exist for any one of them.

- Example: Integer Knapsack, Hamiltonian Cycle, Set Cover Problem, …

Why should I learn this topic?

- To become a good algorithm designer, you must understand the rudiments of the theory of NP-completeness.
- If you can establish a problem as NP-complete, you provide good evidence for its intractability.

- As an engineer, you would then do better to spend your time developing an approximation algorithm, or solving a tractable special case, rather than searching for a fast algorithm that solves the problem exactly.
- Moreover, many practical problems are in fact NP-complete.

Decision Problem vs Optimization Problem

- Many problems of interest are optimization problems, in which each feasible (i.e., "legal") solution has an associated value, and the goal is to find a feasible solution with the best value.

- Example:
  - **SHORTEST-PATH:** the input is an undirected graph **G** and vertices **u** and **v**, and the goal is to find a **path from u to v** that uses the fewest edges.

  - **KNAPSACK:** the input is knapsack capacity **W**, and a set of items with their weights and values, and the goal is find a subset of items that fits the knapsack and maximizes the total value.

Decision Problem vs Optimization Problem

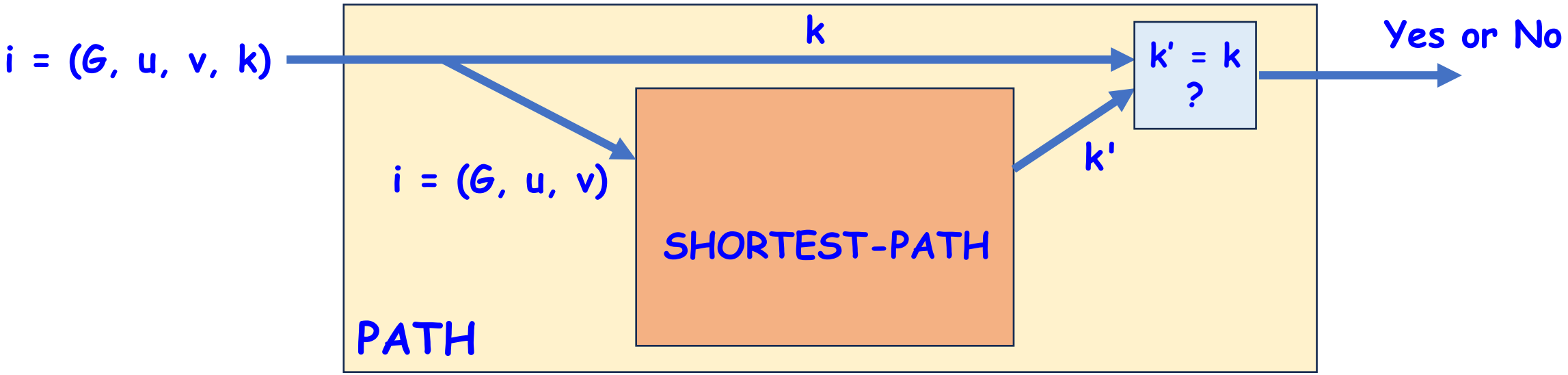
- However, NP-completeness applies directly not to optimization problems, but to **decision problems**, in which the answer is simply "yes" or "no" (or, more formally, 1 or 0)

- There is usually a way to **cast** a given optimization problem as a related decision problem by imposing a bound on the value to be optimized.

- For example, a decision problem related to SHORTEST-PATH is **PATH**: given an undirected graph G, vertices u and v, and an integer k, does a path exist from u to v consisting of at most k edges?

Decision Problem vs Optimization Problem

- The decision problem is in a sense "easier" or at least "no harder" than the optimization counterpart.
- If you can provide evidence that a decision problem is hard, you also provide evidence that its related optimization problem is hard.



Decision Problem vs Optimization Problem

Exercise:

- Cast an Integer Knapsack problem (**KNAPSACK**) as a decision problem (**KNAPSACK-DEC**)!

P vs NP

- The class **P** consists of decision problems that are solvable in $O(n^k)$ for some constant k.
- **P**: Polynomial

- The class **NP** consists of decision problems that are **"verifiable"** in polynomial time.
- If you were somehow given a **"certificate"** of a solution, then you could verify that the certificate is correct in time polynomial in the size of the input to the problem.
- **NP**: Nondeterministic Polynomial

P vs NP

- Consider the problem **HAMILTON**: A hamiltonian cycle of a directed graph **G = (V,E)** is a simple cycle that contains each vertex in **V**. Given an input **G**, determine whether a directed graph has a hamiltonian cycle!

- **HAMILTON** is **NP**
- Let a certificate be a sequence $\langle v_1, v_2, \ldots, v_{|V|} \rangle$ of $|V|$ vertices. We can check in polynomial time that the sequence contains each of the $|V|$ vertices exactly one, that $(v_i, v_{i+1}) \in E$ for $i = 1,2,3, \ldots, |V| - 1$, and that $(v_{|V|}, v_1) \in E$.

P ⊆ NP ?

- Any problem in P also belongs to NP, since if a problem belongs to **P** then it is solvable in polynomial time without even being supplied a certificate.

- The open question is whether **P** is a proper subset of **NP**.
  - P ≠ NP (improper subset) ?
  - P = NP (proper subset) ?

Can every problem whose solution can be quickly checked (NP) also be solved quickly (P)?

- Nowadays, many scientist believe that P ≠ NP. The reason is despite so much effort, no body can prove that P = NP.

NP-Complete

- **Informally**, a problem belongs to the class NPC if:
  - It belongs to NP; and
  - **It is as "hard" as any problem in NP**

- This class of problems is important; because if **any NPC problem** can be solved in polynomial time, then **every problem** in NP has a polynomial-time algorithm.
  - This is where the term "complete" comes from.

- Nobody has proven whether NPC problems can be solved in polynomial time. The Clay Mathematics Institute is offering a **US$1 million reward** to anyone who has a formal proof that P=NP or that P≠NP.

NP-Complete

- If you can demonstrate that a problem is NPC, you are making a statement about how hard it is, rather than about how easy it is.

- If you prove a problem NPC, you are saying that searching for efficient algorithm is likely to be a fruitless endeavor.

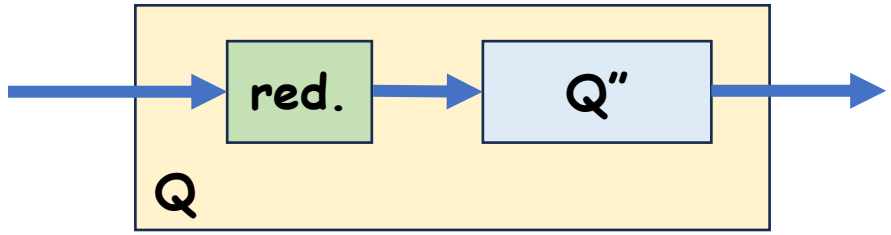
- How to show that your problem is NP-Complete?

Reductions

Reductions

- We call the input to a particular problem an instance of that problem.

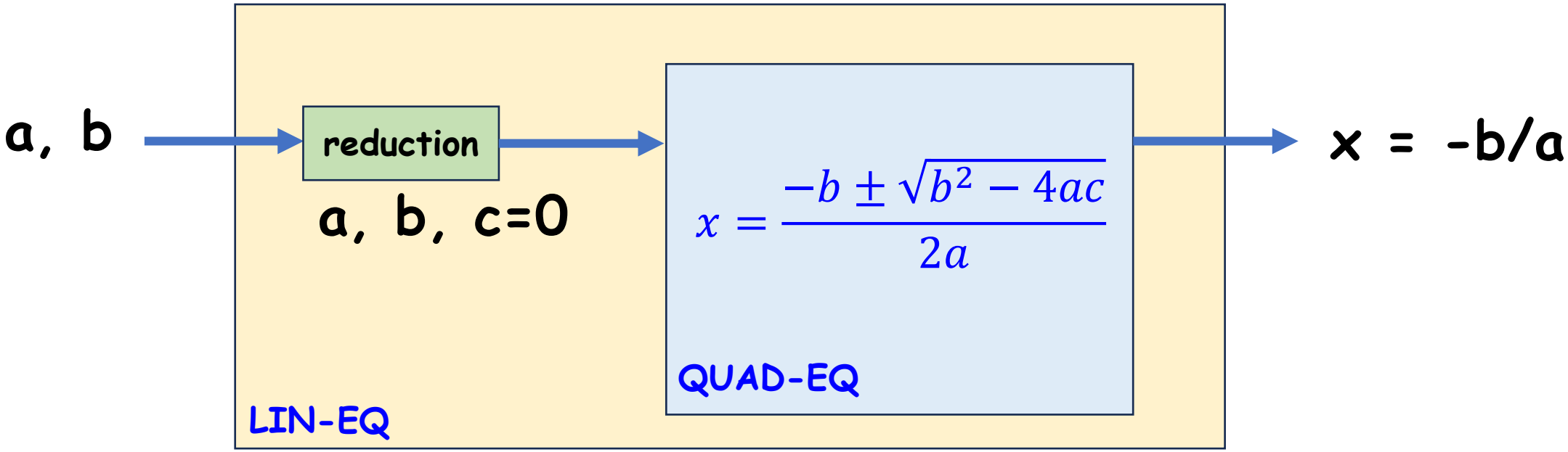
- One way that sometimes works for solving a problem is to recast it as a different problem -> reducing one problem to another.
- Think of a problem **Q** as being reducible to another problem **Q"** if any instance of **Q** can be recast as an instance of **Q"**, and the solution to the instance of **Q"** provides a solution to the instance of **Q**.



Reductions

LIN-EQ: Find a solution to the linear equation **ax + b = 0**?

QUAD-EQ: Find a solution to the quadratic equation **$ax^2 + bx + c = 0$**?



If the **reduction** can be run in **polynomial time**, then **LIN-EQ** is no harder than **QUAD-EQ**, or **LIN-EQ** $\leq$ **QUAD-EQ**. **--> If we know that QUAD-EQ can be solved in polynomial time, then LIN-EQ can also be solved in polynomial time.**

Reductions

Exercise:

MAX: Given an array, find a maximum value!

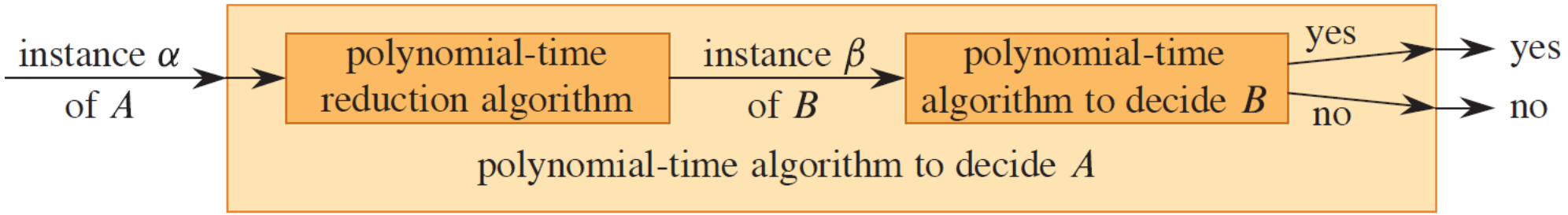
SORT: Given an array, output a sorted version of the array in ascending fashion!

We know that **SORT** can be solved in polynomial-time. Can you prove that **MAX** also has a polynomial solution?

How do you prove?

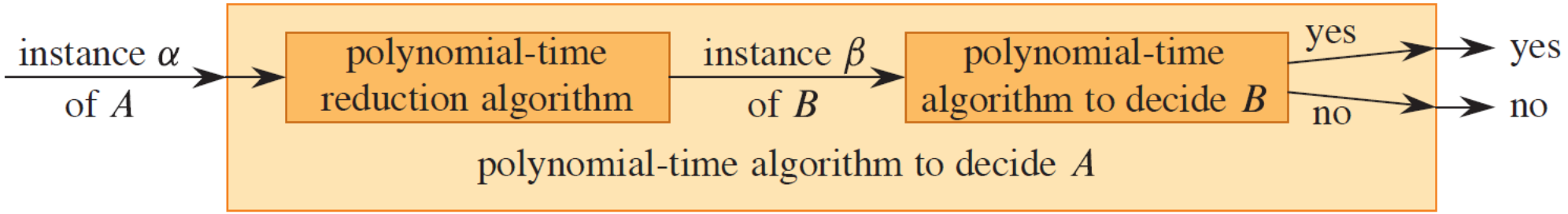
Polynomial-time Reductions

- Consider a decision problem A, which you would like to solve in polynomial time. Now suppose that you already know how to solve a different decision problem B in polynomial time.



Polynomial-time Reductions

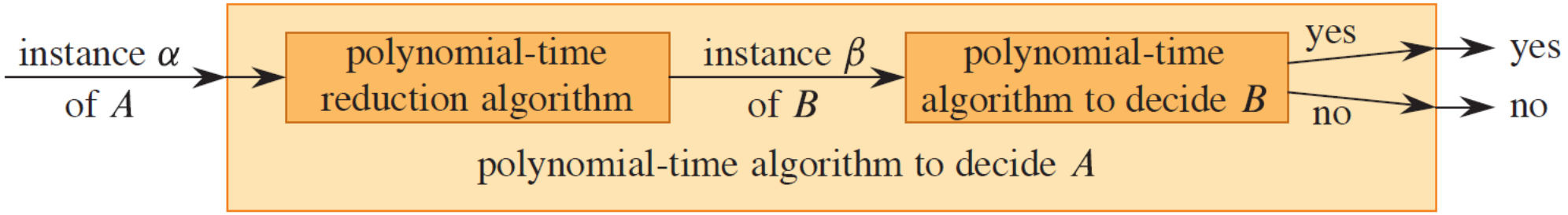
- Finally, suppose that you have a procedure that transforms any instance α of A into some instance β of B with the following characteristics:
  - The transformation takes polynomial time;
  - The answers are the same. That is, the answer for α is yes if and only if the answer for β is yes.
- We call such a procedure a polynomial-time reduction algorithm. it provides us a way to solve problem A in polynomial time.



Polynomial-time Reductions

- As long as each of these steps takes polynomial time, all three together do also, and so you have a way to decide on α in polynomial time. In other words, by "reducing" solving problem A to solving problem B , you use the "easiness" of B to prove the "easiness" of A.

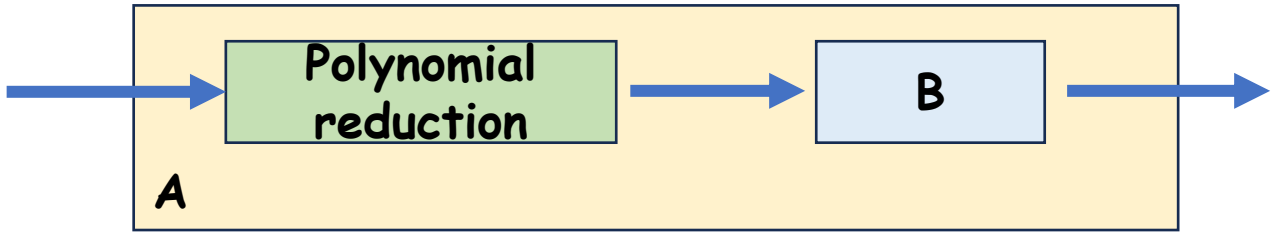
- B is easy --> A is easy



Polynomial-time Reductions

- We can use the contraposition to prove that no polynomial solution can exists for B, when we know that A is "hard".
  - A is hard --> B is hard.

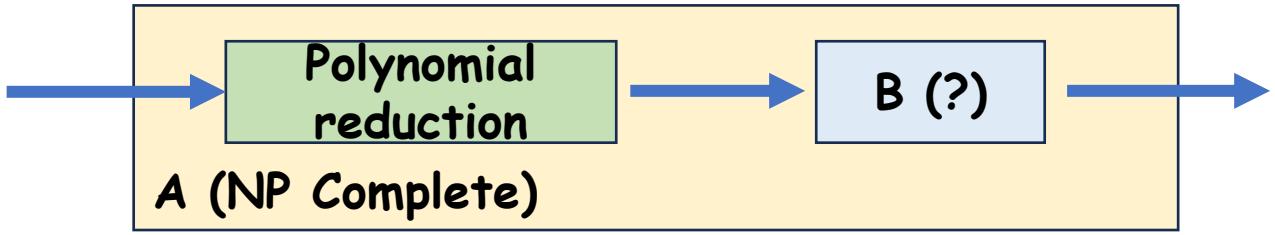
- **Proof:** Suppose B has a polynomial-time algorithm. Then, according to the below picture, we would have a way to solve problem A in polynomial time, which **contradicts** our assumption that there is no polynomial time solution for A.



Then, how to prove that your problem is NP-Complete?

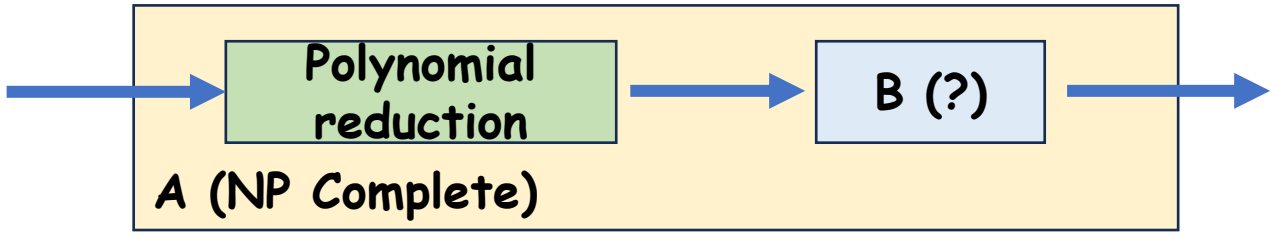
- Yes, the previous slide serves as a basis for us to prove that a particular problem is NPC.
- Suppose that ***A is known to be NPC***;
- If ***A is reducible to B in polynomial time***, then we have shown that **B is also NPC**.

- **But, what is the first problem proved to be NPC?**



A First NP-Complete Problem

- Because the technique of reduction relies on having a problem already known to be NPC in order to prove a different problem NP-complete, there must be some first NPC problem.
- We'll use the **circuit-satisfiability problem** (**CIRCUIT-SAT**), in which the input is a boolean combinational circuit composed of AND, OR, and NOT gates, and the question is whether there exists some set of boolean inputs to this circuit that causes its output to be 1.



What Wikipedia Says?

on

oblems

problems

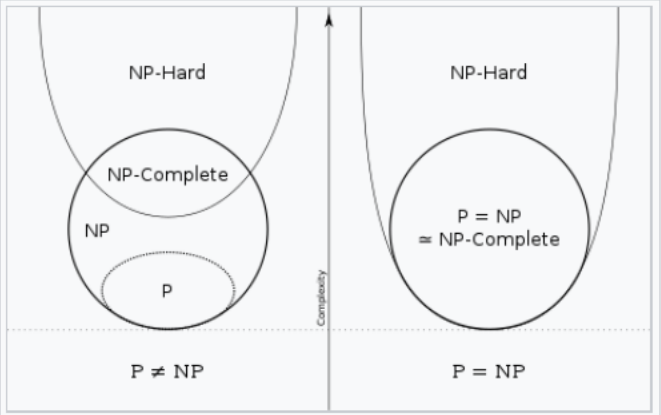
plete problems

under different
on

nceptions

The concept of NP-completeness was introduced in 1971 (see Cook–Levin theorem), though the term *NP-complete* was introduced later. At the 1971 STOC conference, there was a fierce debate between the computer scientists about whether NP-complete problems could be solved in polynomial time on a deterministic Turing machine. John Hopcroft brought everyone at the conference to a consensus that the question of whether NP-complete problems are solvable in polynomial time should be put off to be solved at some later date, since nobody had any formal proofs for their claims one way or the other. This is known as "the question of whether P=NP".

Nobody has yet been able to determine conclusively whether NP-complete problems are in fact solvable in polynomial time, making this one of the great unsolved problems of mathematics. The Clay Mathematics Institute is offering a US$1 million reward to anyone who has a formal proof that P=NP or that P≠NP.[5]



Euler diagram for P, NP, NP-complete, and NP-hard sets of problems. The left side is valid under the assumption that P≠NP, while the right side is valid under the assumption that P=NP (except that the empty language and its complement are never NP-complete, and in general, not every problem in P or NP is NP-complete).

The existence of NP-complete problems is not obvious. The Cook–Levin theorem states that the Boolean satisfiability problem is NP-complete, thus establishing that such problems do exist. In 1972, Richard Karp proved that several other problems were also NP-complete (see Karp's 21 NP-complete problems); thus, there is a class of NP-complete problems (besides the Boolean satisfiability problem). Since the original results, thousands of other problems have been shown to be NP-complete by reductions from other problems previously shown to be NP-complete; many of these problems are collected in Garey & Johnson (1979).

Formal Way to Understand What a "Problem" is

Yes, this is where you find the relation between DAA and TBA.

What a "problem" is?

We define an abstract problem **Q** to be a binary relation on a set **I** of problem instances and a set **S** of problem solutions.

Example: **PATH** problem

- $\boldsymbol{i \in I}$, with $\boldsymbol{i = \langle G, u, v, k \rangle}$, meaning that the input is a graph G, a start node u, an end node v, and a value k.
- $\boldsymbol{S = \{0, 1\}}$
- $\mathbf{PATH(i) = 1}$ if G contains a path from u to v with a most k edges;
- $\mathbf{PATH(i) = 0}$ otherwise.

Encodings

In order for a computer program to solve an abstract problem, its problem instances must appear in a way that the program understands.

An **encoding** of a set **S** of abstract objects is a mapping **e** from **S** to the set of **binary strings**.

Example, natural numbers {0, 1, 2, …} are encoded as the strings {0, 1, 10, 11, 100, …}.

Polygons, graphs, functions, ordered pairs, programs - all can be encoded as binary strings.

Encodings

We call a problem whose instance set is the set of binary strings a **concrete problem**.

We say that an algorithm solves a concrete problem in $O(T(n))$ time if, when it is provided a problem instance i of size $n = |i|$ (the length of binary string), the algorithm can produce the solution in $O(T(n))$.

Formally, A concrete problem is polynomial-time solvable, it there exists an algorithm to solve it in $O(n^k)$ time.

Encodings

Encodings map abstract problems to concrete problems. Given an abstract decision problem **Q** mapping an instance set **I** to **{0,1}**, an encoding **e : I → {0,1}*** can induce a related concrete decision problem, which we denote by **e(Q)**.

If the solution to an abstract-problem instance **i ∈ I** is **Q(i) ∈ {0,1}**, then solution to the concrete problem instance **e(i) ∈ {0,1}*** is also **Q(i) ∈ {0,1}**.

Formal-Language Framework

- In the formal-language framework, 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 $L = \{x \in \Sigma^* : Q(x) = 1\}$.
- The formal-language framework allows us to express the relation between decision problems and algorithms that solve them concisely.
- We say that an algorithm A accepts a string $x \in \{0, 1\}^*$ if, given input x, the algorithm's output $A(x)$ is 1.
- The language L accepted by an algorithm A is the set of strings that the algorithm accepts, that is, $L = \{x \in \{0, 1\}^* : A(x) = 1\}$.
- An 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 $x \notin L$ given as input to it. For example, the algorithm may loop forever.
- 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.
- Thus, to accept a language L, an algorithm need only worry about strings in L, but to decide a language, it must correctly accept or reject every string in $\{0, 1\}^*$.
- There are problems, such as Turing's Halting Problem, for which there exists an accepting algorithm, but no decision algorithm exists.
- Using this formal-language framework, we can define the complexity class P as follows:

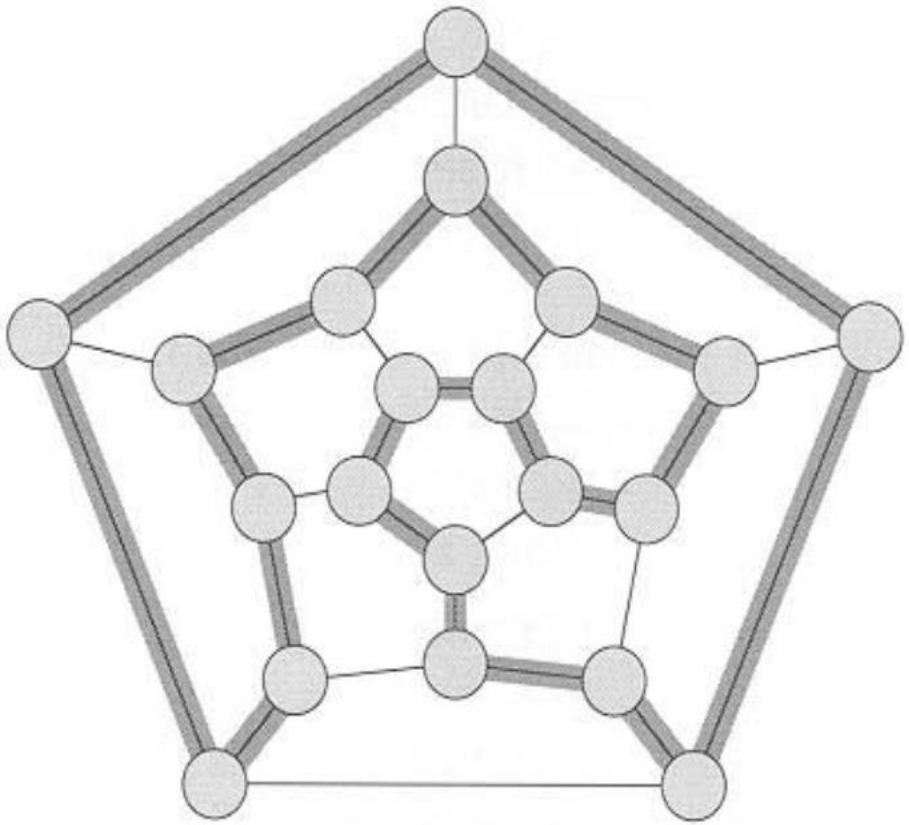
  $P = \{L \subseteq \{0, 1\}^* :$ there exists an algorithm A that decides L in polynomial time$\}$.

Hamiltonian Cycles

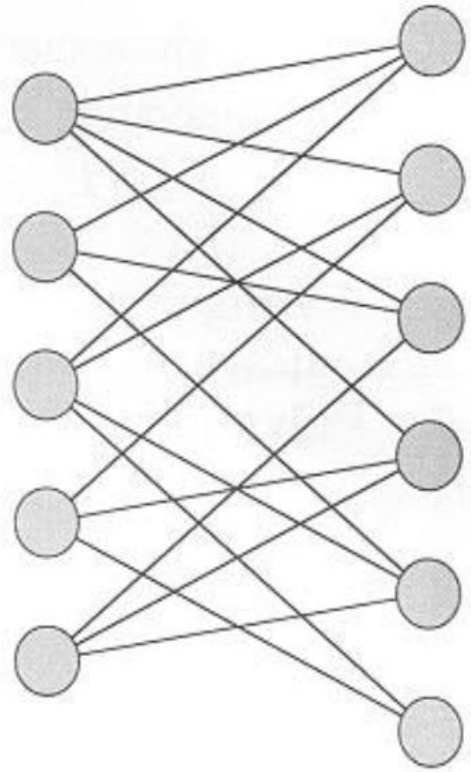
- A hamiltonian cycle of an undirected graph $G = (V, E)$ is a simple cycle that contains each vertex in V.
- A graph that contains a hamiltonian cycle is said to be hamiltonian; otherwise, it is nonhamiltonian.
- We can define the hamiltonian-cycle problem, "Does a graph G have a hamiltonian cycle?" as a formal language:

  HAM-CYCLE = {$\langle G \rangle$: G is a hamiltonian graph}.

  $\langle G \rangle$ denotes the standard binary encoding of G.
- In the following slide, the dodecahedron is hamiltonian, while the bipartite graph (with an odd number of vertices) is nonhamiltonian.



(a)



(b)

Verification Algorithms

- Suppose that a friend 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.
- It would certainly be easy enough to verify the proof: simply verify that the provided cycle is hamiltonian by checking whether it is a permutation of the vertices of V and whether each of the consecutive edges along the cycle actually exists in the graph.
- This verification algorithm can be implemented to run in $O(n^2)$ time, where n is the length of the encoding of G.
- Thus, a proof that a hamiltonian cycle exists in a graph can be verified in polynomial time.

- In general, a verification algorithm is defined as being a two-argument algorithm A, where one argument is an ordinary input string x and the other is a binary string y called a 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

  $L = \{x \in \{0, 1\}^* : \text{there exists } y \in \{0, 1\}^* \text{ such that } A(x, y) = 1\}$.

- Intuitively, an algorithm A verifies a language L if for any string $x \in L$, there is a certificate y that A can use to prove that $x \in L$. Moreover, for any string $x \notin L$, there must be no certificate proving that $x \in L$.

The complexity class NP

- The complexity class NP is the class of languages that can be verified by a polynomial-time algorithm.
- More precisely, 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

$$L = \{x \in \{0, 1\}^* : \text{there exists a certificate } y \text{ with } |y| = O(|x|^c) \text{ such that } A(x,y) = 1\}$$

- We say that algorithm A verifies language L in polynomial time.

The complexity class NP

- HAM-CYCLE $\in$ NP.
- If L $\in$ P, then L $\in$ NP.

  Why? Because if there is a polynomial-time algorithm to 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.

  Most computer scientists believe that P $\neq$ NP.
- Intuitively, the class P consists of problems that can be solved quickly. The class NP consists of problems for which a solution can be verified quickly.

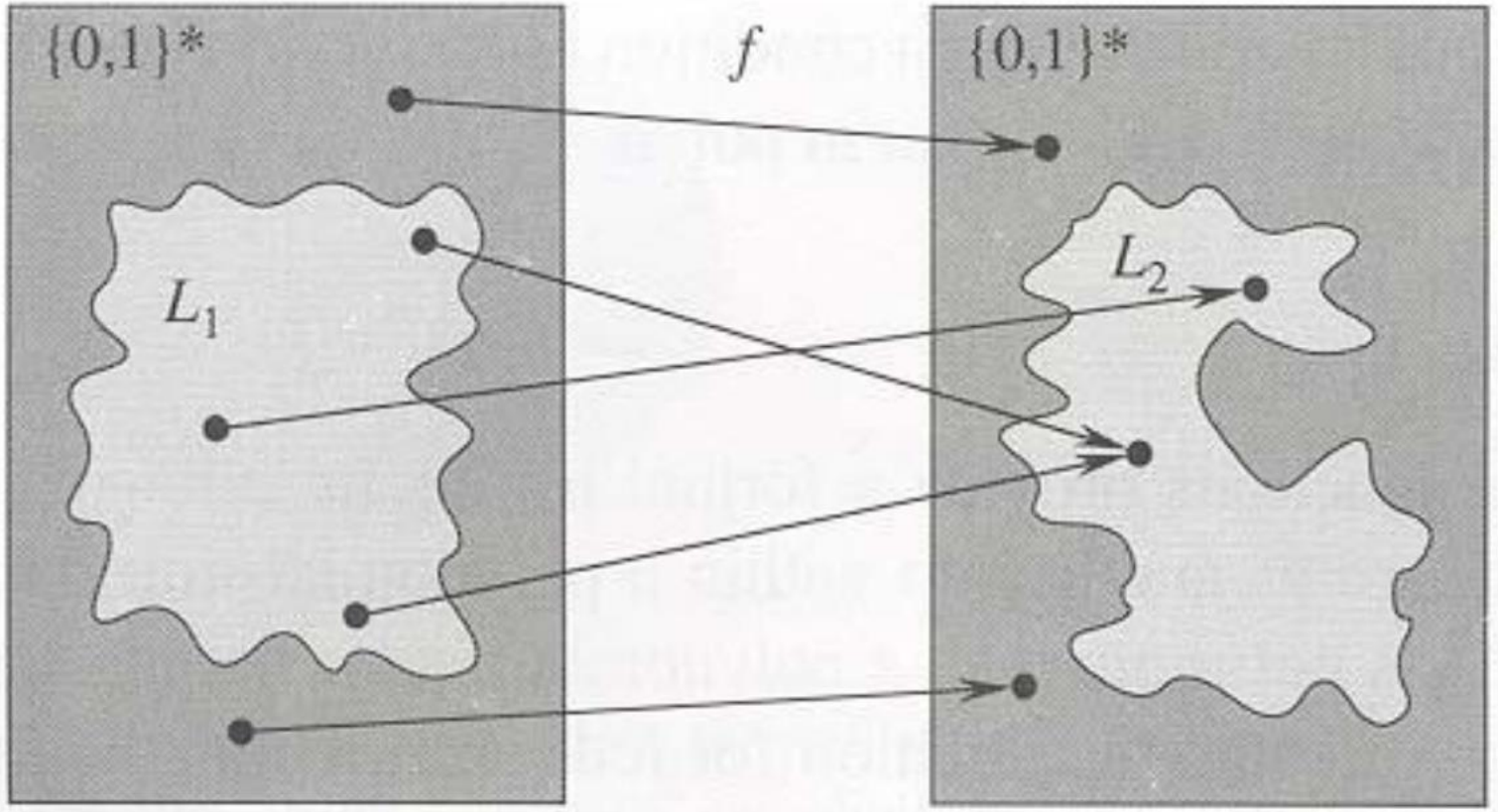
NP Completeness & Reducibility

- A language L_1 is polynomial-time reducible to a language L_2, denoted as $L_1 \leq_P L_2$, if there exists a polynomial-time computable function $f: \{0, 1\}^* \rightarrow \{0, 1\}^*$ such that for all $x \in \{0, 1\}^*$,

  $x \in L_1$ if and only if $f(x) \in L_2$.

The function f is called the reduction function, and a polynomial-time algorithm F that computes f is called a reduction algorithm.

NP Completeness & Reducibility



NP Completeness & Reducibility

- The reduction function f maps any instance x of the decision problem represented by the language L_1 to an instance $f(x)$ of the decision problem represented by L_2.
- Giving an answer to whether $f(x) \in L_2$ directly provides the answer to whether $x \in L_1$.

- Polynomial-time reductions provide a formal means for showing that one problem is at least as hard as another, to within a polynomial-time factor.

  That is, if $L_1 \leq_P L_2$, then L_1 is not more than a polynomial factor harder than L_2.

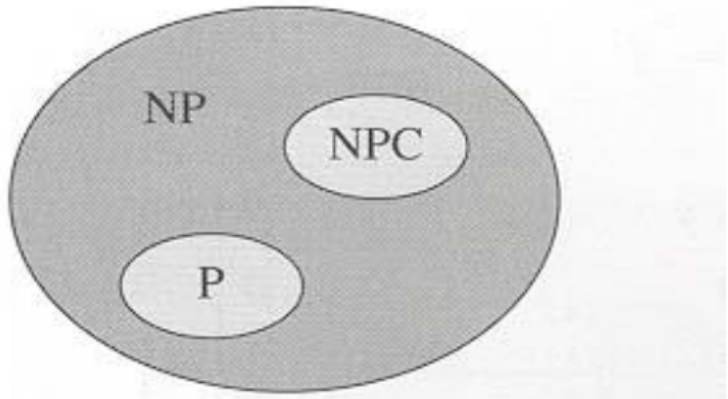
Definition:

A language $L \subseteq \{0, 1\}^*$ is NP-complete if

1. $L \in NP$, and
2. $L' \leq_p L$ for every $L' \in NP$.

This part is honestly difficult!

- If a language L satisfies property 2, but not necessarily property 1, we say that L is NP-hard.
- If any NP-complete problem is polynomial-time solvable, then P = NP. Equivalently, if any problem in NP is not polynomial-time solvable, then no NP-complete problem is polynomial-time solvable.
- Most computer scientists believe that $P \neq NP$.



Good News!

https://www.britannica.com/biography/Richard-Karp
https://www.britannica.com/biography/Stephen-Arthur-Cook

Thanks to **Prof. Cook** and **Prof. Karp** who have shown our first NPC problem: **Circuit Satisfiability Problem** (**CIRCUIT-SAT**).

Both of you deserved **Turing Award** for this work!

WIKIPEDIA
The Free Encyclopedia

Search Wikipedia | Search

Cook–Levin theorem

Article | Talk | Read | Edit | View history | Tools

From Wikipedia, the free encyclopedia

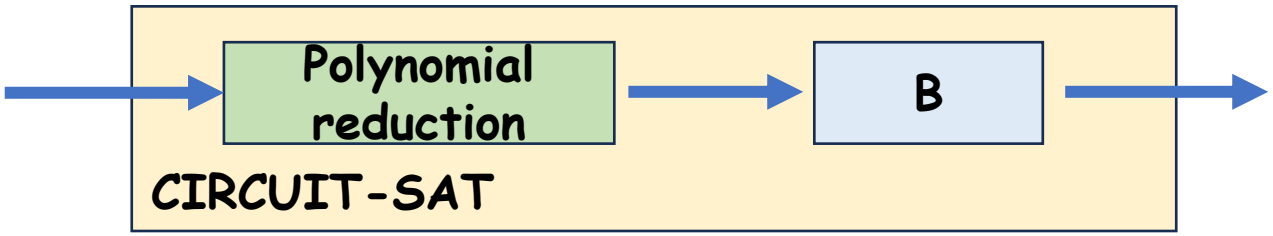
In computational complexity theory, the **Cook–Levin theorem**, also known as **Cook's theorem**, states that the Boolean satisfiability problem is NP-complete. That is, it is in NP, and any problem in NP can be reduced in polynomial time by a deterministic Turing machine to the Boolean satisfiability problem.

The theorem is named after Stephen Cook and Leonid Levin. The proof is due to Richard Karp, based on an earlier proof (using a different notion of reducibility) by Cook.[1]

An important consequence of this theorem is that if there exists a deterministic polynomial-time algorithm for solving Boolean satisfiability, then every NP problem can be solved by a deterministic polynomial-time algorithm. The question of whether such an algorithm for Boolean satisfiability exists is thus equivalent to the P versus NP problem, which is still widely considered the most important unsolved problem in theoretical computer science as of 2024.

Good News!

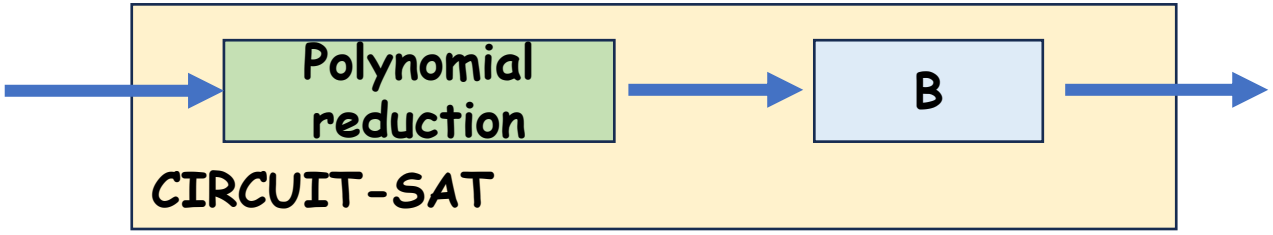
If we know CIRCUIT-SAT is NPC, then we can use it to prove "many" other problems to be NPC using polynomial-reduction.



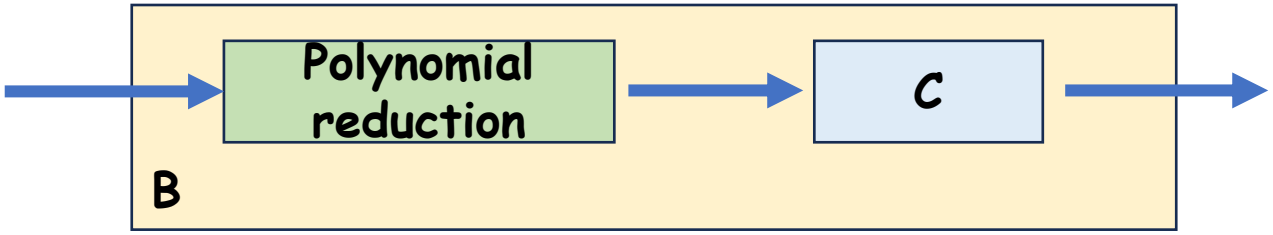
If CIRCUIT-SAT is reducible to B, then B is NPC

Good News!

If we know CIRCUIT-SAT is NPC, then we can use it to prove "many" other problems to be NPC using polynomial-reduction.



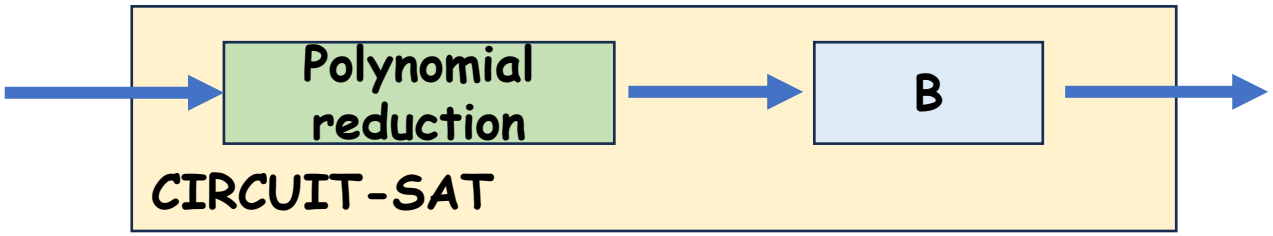
If CIRCUIT-SAT is reducible to B, then B is NPC



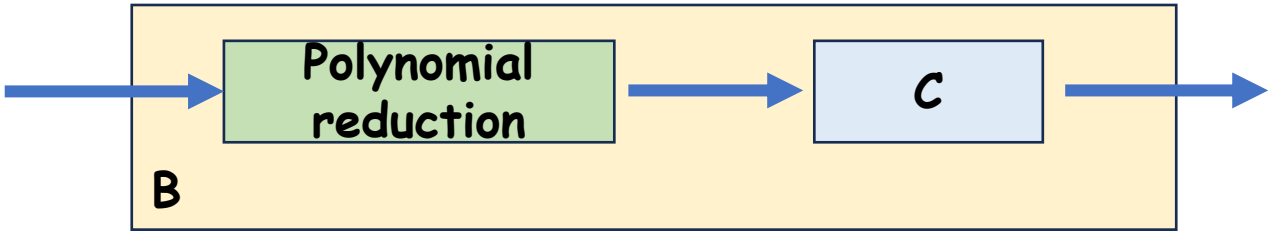
If B is reducible to C, then C is NPC

Good News!

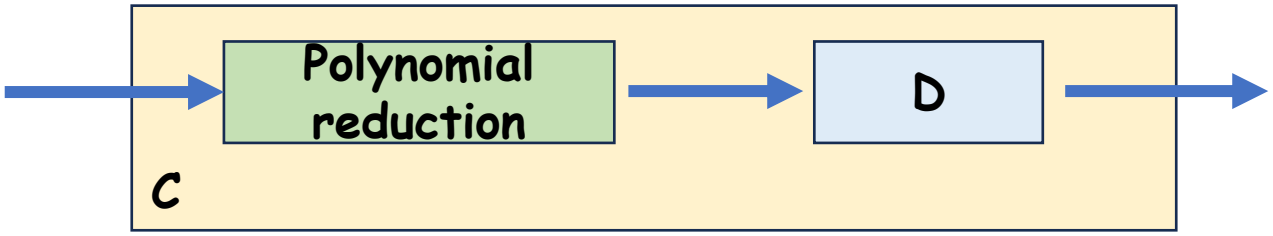
If we know CIRCUIT-SAT is NPC, then we can use it to prove "many" other problems to be NPC using polynomial-reduction.



If CIRCUIT-SAT is reducible to B, then B is NPC

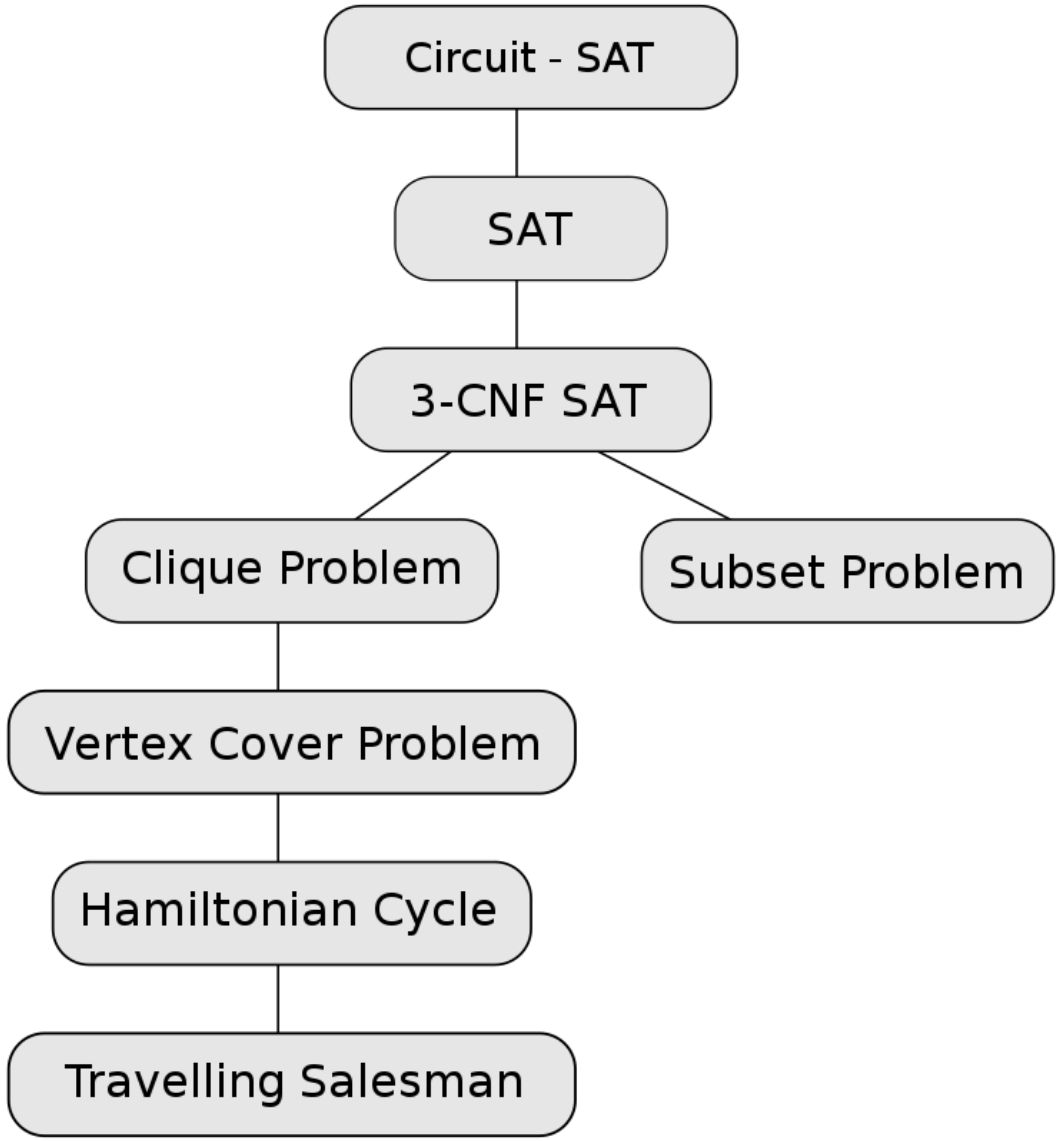


If B is reducible to C, then C is NPC



If C is reducible to D, then D is NPC

And so on …

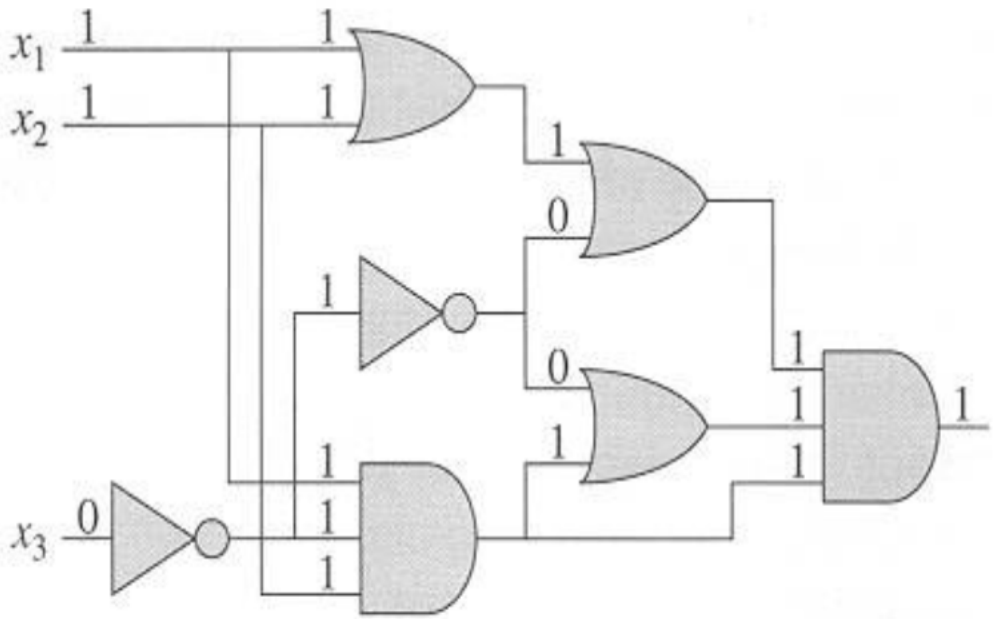


https://en.wikipedia.org/wiki/NP-completeness#/media/File:Relative_NPC_chart.svg

Intro. To CIRCUIT-SAT

Our first problem proven to be NP-Complete!

- Up to this point, we have not actually proved any problem to be NPC.
- Once we prove that at least one problem is NPC, we can use polynomial-time reducibility as a tool to prove the NP-completeness of other problems.
- Soon, we will demonstrate the existence of an NPC problem: the circuit-satisfiability problem.
- First let's review some related concepts of combinational circuits.
- A boolean combinational circuit consists of one or more logic gates (NOT, AND, OR) interconnected by wires. Boolean combinational circuits contain no cycles.
- A truth assignment for a boolean combinational circuit is a set of boolean input values. A one-output boolean combinational circuit is satisfiable if it has a satisfying assignment, that is, a truth assignment that causes the output of the circuit to be 1.



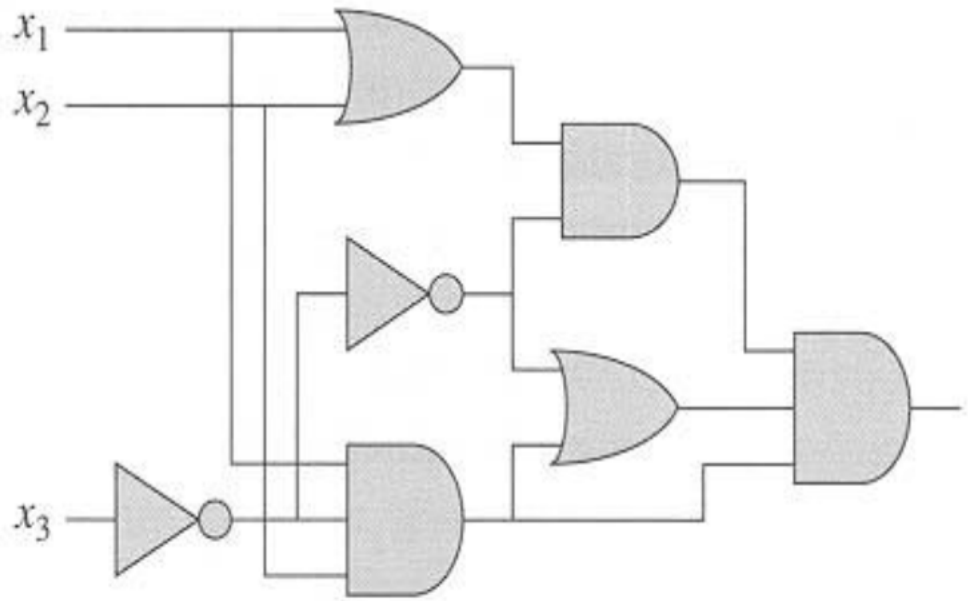


fig348

(a) satisfiable
(b) unsatisfiable

- The **circuit-satisfiability problem** is

  "Given a boolean combinational circuit composed of NOT, AND, and OR gates, is it satisfiable?"

- This problem arises in the area of computer-aided hardware optimization. If a subcircuit always produces 0, that subcircuit can be replaced by a simpler subcircuit that omits all logic gates and provides the constant 0 value as its output.

- The size of a boolean combinational circuit is the number of logic gates plus the number of wires in the circuit.

- As a formal language, the circuit-satisfiability problem can be defined as

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

  where <*C*> denotes the binary string encoding the circuit *C*.

- Given a circuit C, we might attempt to determine whether it is satisfiable by simply checking all possible assignments to the inputs. Unfortunately, if there are k inputs, there are 2^k possible assignments.
- We will prove that CIRCUIT-SAT is NP-complete.
- The proof is divided into two parts, based on the two parts of the definition of NP-completeness.

Lemma 1:

The circuit-satisfiability problem belongs to the class NP.

Lemma 2:

The circuit-satisfiability problem is NP-hard.

We provide the proof of this theorem in the last slides

Theorem:

The circuit-satisfiability problem is NP-complete.

How to use CIRCUIT-SAT to prove that other problems are NPC?

- A language can be proved to be NP-complete without directly reducing every language in NP to the given language.
- The basis of the method is the following lemma.

Lemma:

If L is a language such that $L' \leq_p L$ for some $L' \in NPC$, then L is NP-hard. Moreover, if $L \in NP$, then $L \in NPC$.

Proof:

Since L' is NP-complete, for all $L'' \in NP$, we have $L'' \leq_p L'$. By transitivity, since $L' \leq_p L$, we have $L'' \leq_p L$, which shows that L is NP-hard. If $L \in NP$, we also have $L \in NPC$.

♦

- Note that by reducing a known NP-complete language L' to L, we implicitly reduce every language in NP to L.

- The previous lemma gives us a method for proving that a language L is NP-complete:

1. Prove $L \in NP$.
2. Select a known NP-complete language L'.
3. Describe an algorithm that computes a function f mapping every instance $x \in \{0, 1\}^*$ of L' to an instance $f(x)$ of L.
4. Prove that the function f satisfies $x \in L'$ if and only if $f(x) \in L$ for all $x \in \{0, 1\}^*$.
5. Prove that the algorithm computing f runs in polynomial time.

- Steps 2-5 show that L is NP-hard.
- Knowing that CIRCUIT-SAT is NP-complete now allows us to prove much more easily that other problems are NP-complete.

General Strategy for Proving New Problems NP-Complete

- Given a new problem X, here is the basic strategy for proving that it is NP-complete.
  1. Prove that $X \in NP$.
  2. Choose a problem Y that is known to be NP-complete.
  3. Prove that $Y \leq_P X$.

Example: We introduce a new problem, **formula satisfiability problem (SAT)**.

And we show that SAT is NPC using our "almighty" CIRCUIT-SAT

Formula Satisfiability Problem (SAT)

- The (formula) satisfiability problem can be formulated as the language SAT as follows.
- 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), $\neg$ (NOT), $\rightarrow$ (implication), $\leftrightarrow$ (if and only if); and
3. parentheses.

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

  SAT = $\{\langle\phi\rangle : \phi$ is a satisfiable boolean formula$\}$.

- A boolean formula ϕ can be encoded in a length that is polynomial in $n + m$.

Formula Satisfiability Problem (SAT)

- For example, the formula

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

  has the satisfying assignment

  $\langle x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1 \rangle$.

  Thus, this formula ϕ belongs to SAT.

Theorem:

The problem of satisfiability of boolean formulas is NP-complete.

Formula Satisfiability Problem (SAT)

How to prove that SAT is NPC?

1. To prove that SAT belongs to NP, we show that a certificate consisting of a satisfying assignment for an input formula ϕ can be verified in polynomial time.

   The verifying algorithm simply replaces each variable in the formula with its corresponding value and then evaluates the expression. This task can be done in polynomial time. If the expression evaluates to 1, the formula is satisfiable.

Formula Satisfiability Problem (SAT)

How to prove that SAT is NPC?

2. To prove that SAT is NP-hard, we show that CIRCUIT-SAT $\leq_P$ SAT.

   Induction can be used to express any boolean combinational circuit as a boolean formula. We simply look at the gate that produces the circuit output and inductively express each of the gate's inputs as formulas. The formula for the circuit is then obtained by writing an expression that applies the gate's function to its inputs' formulas. For each wire x_i in the circuit C, the formula ϕ has a variable x_i. The proper operation of a gate can be expressed as a formula involving the variables of its incident wires. For example,

the operation of the output AND gate is

$x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9)$.

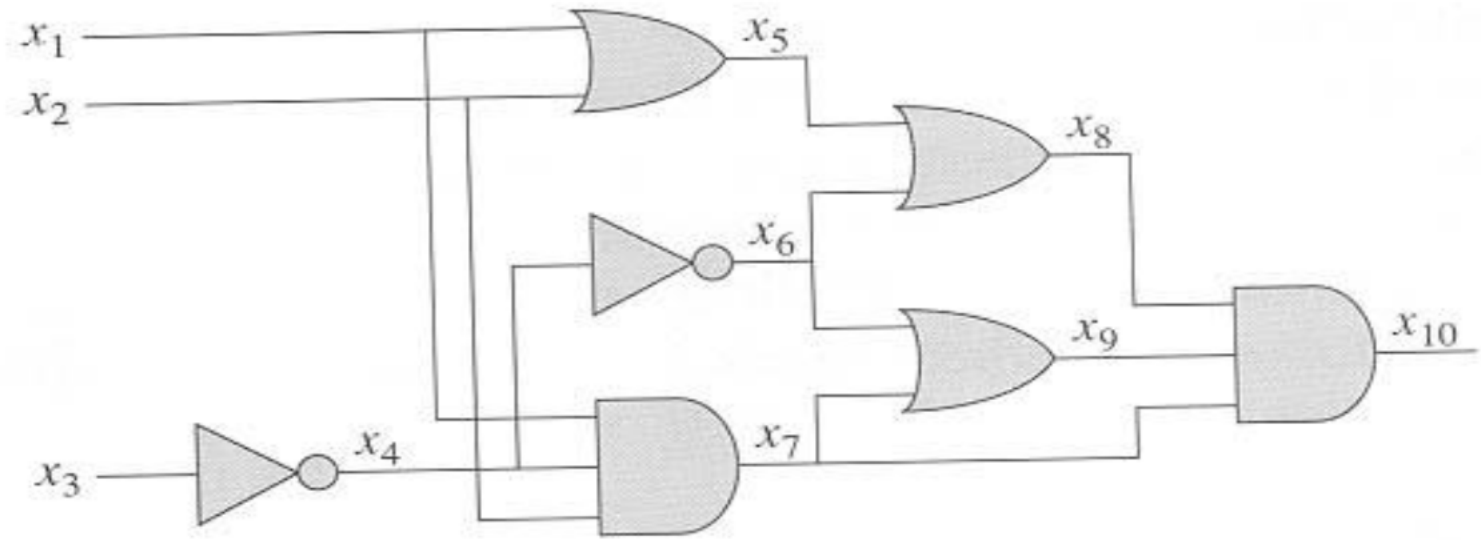


fig3410

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. For example, the formula for the circuit in the figure is

$$\phi = x_{10} \wedge (x_4 \leftrightarrow \neg x_3)$$
$$\wedge (x_5 \leftrightarrow (x_1 \vee x_2))$$
$$\wedge (x_6 \leftrightarrow \neg\, x_4\,)$$
$$\wedge (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4))$$
$$\wedge (x_8 \leftrightarrow (x_5 \vee x_6))$$
$$\wedge (x_9 \leftrightarrow (x_6 \vee x_7))$$
$$\wedge (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9)).$$

The circuit C is satisfiable exactly when the formula is satisfiable.

Why? If C has a satisfying assignment, each wire of the circuit has a well-defined value and the output of the circuit is 1.

Given a circuit C, such a formula can be produced in polynomial time.

Therefore, the assignment of wire values to variables in ϕ makes each clause of ϕ evaluate to 1, and thus the conjunction of all evaluates to 1.

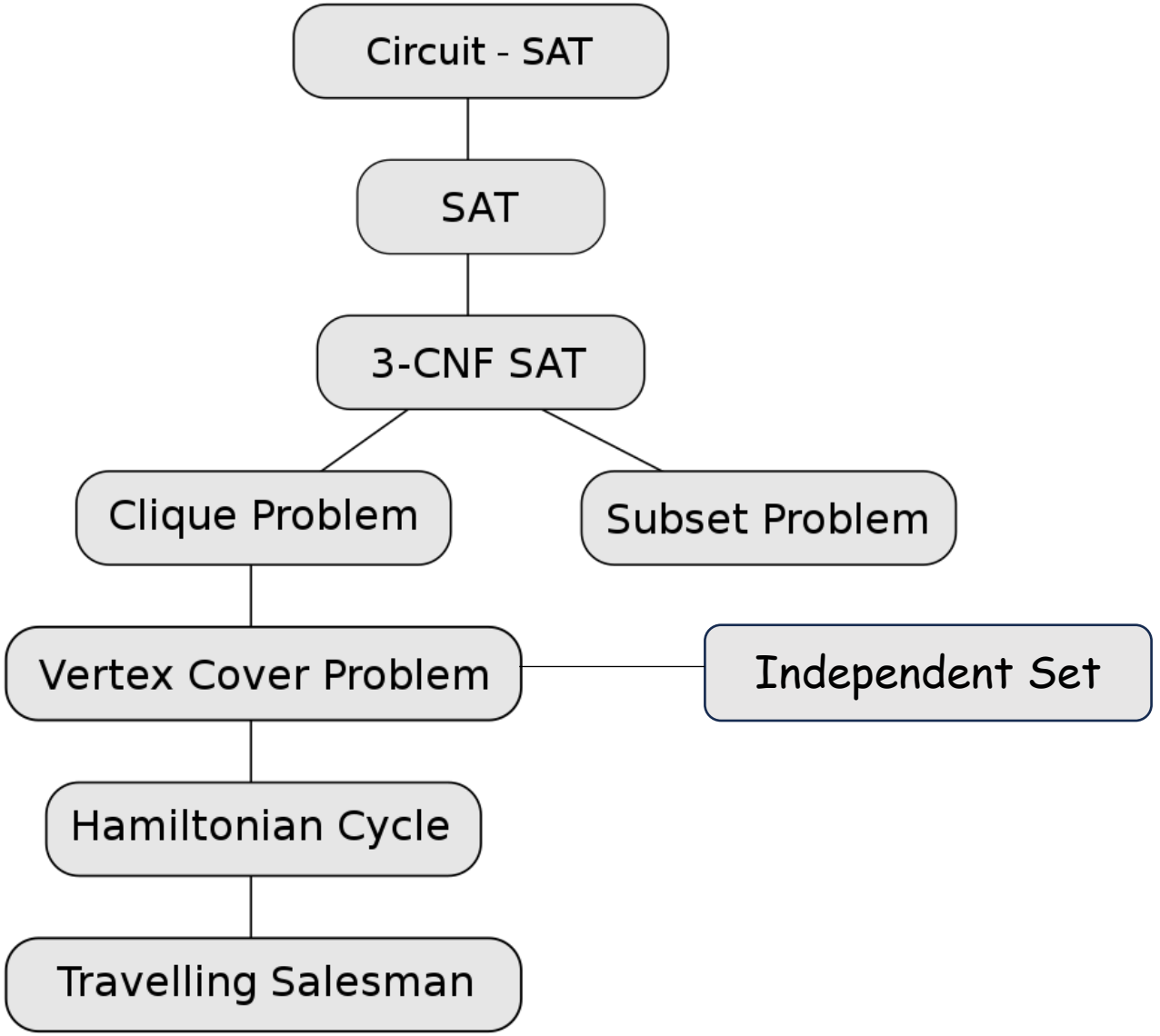
Conversely, if there is an assignment that causes ϕ to evaluate to 1, the circuit C is satisfiable by an analogous argument. Thus, we have shown that CIRCUIT-SAT $\leq_P$ SAT, which completes the proof.

♦

Various NP-complete Problems

- the 3-CNF satisfiability problem
- the clique problem
- the vertex-cover problem
- The independent-set problem
- the hamiltonian-cycle problem
- the traveling-salesman problem
- the subset-sum problem

and many more …

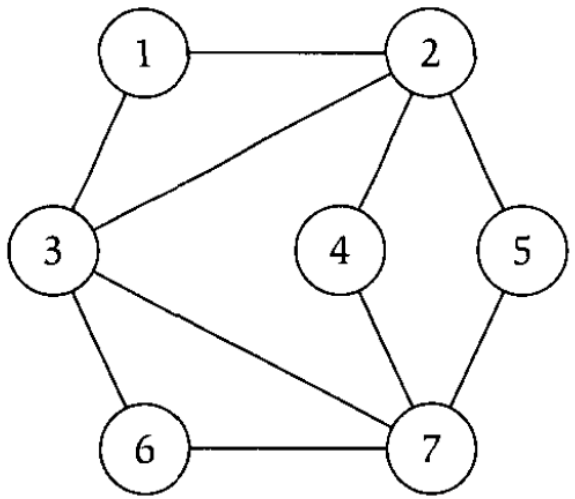


https://en.wikipedia.org/wiki/NP-completeness#/media/File:Relative_NPC_chart.svg

Another Example: Independent Set & Vertex Cover Problems

Independent Set

- For a graph $G = (V, E)$, we say a set of nodes $S \subseteq V$ is independent if no two nodes in S are joined by an edge.
- It is easy to find small independent sets in a graph (for example, a single node forms an independent set); the hard part is to find a largest independent set.
- For example, in the following figure, the set of nodes {3, 4, 5} is an independent set of size 3, while the set of nodes {1, 4, 5, 6} is a largest independent set.



Independent Set Problem

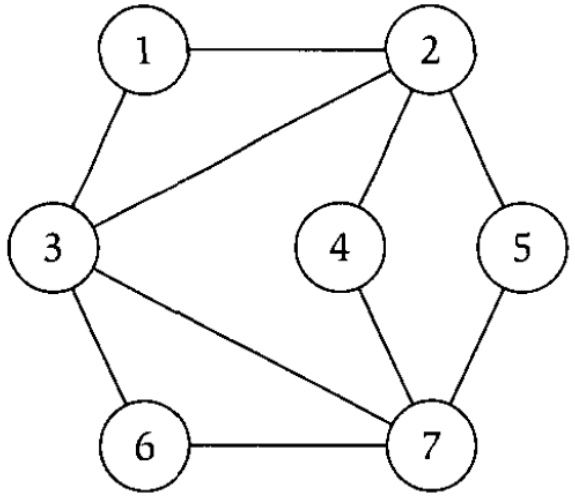
- As a decision problem, the independent set problem is formulated as follows: Given a graph G and a number k, does G contain an independent set of size at least k?
- As an optimization problem, the independent set problem is formulated as follows: Given a graph G, find an independent set of maximum size.
- From the point of view of polynomial-time solvability, the optimization version is equivalent to the decision version .

Independent Set Problem

- Given a method to solve the optimization version, we automatically solve the decision version (for any k) as well.
- But there is also a converse to this: If we can solve the decision version for every k, then we can also find a maximum independent set.
- For given a graph G of n nodes, we simply solve the decision version for each k; the largest k for which the answer is "yes" is the size of the largest independent set in G.
- By using binary search, we need only solve the decision version for O(log n) different values of k.

Vertex Cover

- Given a graph $G = (V, E)$, we say that a set of nodes $S \subseteq V$ is a vertex cover if every edge $e \in E$ has at least one end in S.
- It is easy to find large vertex covers in a graph (for example, the full vertex set is one); the hard part is to find a smallest vertex cover. We formulate the Vertex Cover
- For example, in the graph below, the set of nodes {1, 2, 6, 7} is a vertex cover of size 4, while the set {2, 3, 7} is a vertex cover of smallest size.



Vertex Cover Problem

- As a decision problem, the vertex cover problem is formulated as follows: Given a graph G and a number k, does G contain a vertex cover of size at most k?
- As an optimization problem, the vertex cover problem is formulated as follows: Given a graph G, find a vertex cover of minimum size.

Example of Polynomial Reduction

- Nobody knows how to solve either Independent Set problem or Vertex Cover problem in polynomial time; but their relative difficulty (hardness) can be determined.
- It can be shown that

  Independent Set $\leq_p$ Vertex Cover

  and

  Vertex Cover $\leq_p$ Independent Set

  by using the following theorem.

- Theorem: Let $G = (V, E)$ be a graph. Then $S \subseteq V$ is an independent set if and only if its complement $V - S$ is a vertex cover.
- Proof: First, suppose that S is an independent set. Consider an arbitrary edge $e = (u,v)$. Since S is independent, it cannot be the case that both u and v are in S; so one of them must be in V - S. It follows that every edge has at least one end in V - S, and so V - S is a vertex cover. **Conversely**, suppose that V - S is a vertex cover. Consider any two nodes u and v in S. If they were joined by edge e, then neither end of e would lie in V - S, contradicting our assumption that V - S is a vertex cover. It follows that no two nodes in S are joined by an edge, and so S is an independent set.

Independent Set $\leq_p$ Vertex Cover

- Proof:

  If we have a black box to solve Vertex Cover, then we can decide whether G has an independent set of size at least k by asking the black box whether G has a vertex cover of size at most n - k.

Vertex Cover $\leq_p$ Independent Set

- Proof:

  If we have a black box to solve Independent Set, then we can decide whether G has a vertex cover of size at most k by asking the black box whether G has an independent set of size at least n - k.

Hard Computational Problems in Various Domains

Aerospace engineering: optimal mesh partitioning for finite elements.
Biology: protein folding.
Chemical engineering: heat exchanger network synthesis.
Civil engineering: equilibrium of urban traffic flow.
Economics: computation of arbitrage in financial markets with friction.
Electrical engineering: VLSI layout.
Environmental engineering: optimal placement of contaminant sensors.
Financial engineering: find minimum risk portfolio of given return.
Game theory: find Nash equilibrium that maximizes social welfare.
Genomics: phylogeny reconstruction.
Mechanical engineering: structure of turbulence in sheared flows.
Medicine: reconstructing 3-D shape from biplane angiocardiogram.
Operations research: optimal resource allocation.
Physics: partition function of 3-D Ising model in statistical mechanics.
Politics: Shapley-Shubik voting power.
Pop culture: Minesweeper consistency.
Statistics: optimal experimental design.

A Proof: CIRCUIT-SAT is NPC

Optional … ☺

Proof of Lemma 1:

We construct a two-input, polynomial-time algorithm A that can verify CIRCUIT-SAT. One of the inputs to A is a standard binary encoding of a boolean combinational circuit C. The other input is a certificate corresponding to an assignment of boolean values to the wires in C.

The algorithm A works as follows. For each logic gate in the circuit, it checks that the value provided by the certificate on the output wire is correctly computed as a function of the values on the input wires. Then, if the output of the entire circuit is 1, the algorithm outputs 1, since the values assigned to the inputs of C provide a satisfying assignment. Otherwise, A outputs 0.

Proof of Lemma 1 (cont.)

Whenever a satisfiable circuit C is input to algorithm A, there is a certificate whose length is polynomial in the size of C and that causes A to output 1. Whenever an unsatisfiable circuit is input, no certificate can make A into concluding that the circuit is satisfiable. Algorithm A runs in polynomial time. Thus, CIRCUIT-SAT can be verified in polynomial time, and CIRCUIT-SAT $\in$ NP.

♦

Preparation for Proving Lemma 2

- The second part of proving that CIRCUIT-SAT is NP-complete is to show that the language is NP-hard. That is, we must show that every language in NP is polynomial-time reducible to CIRCUIT-SAT.
- The proof is based on the workings of computer hardware. Here is a review.
- A computer program is stored in the computer memory as a sequence of instructions. 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, thereby causing the computer to execute instructions sequentially.

Preparation for Proving Lemma 2

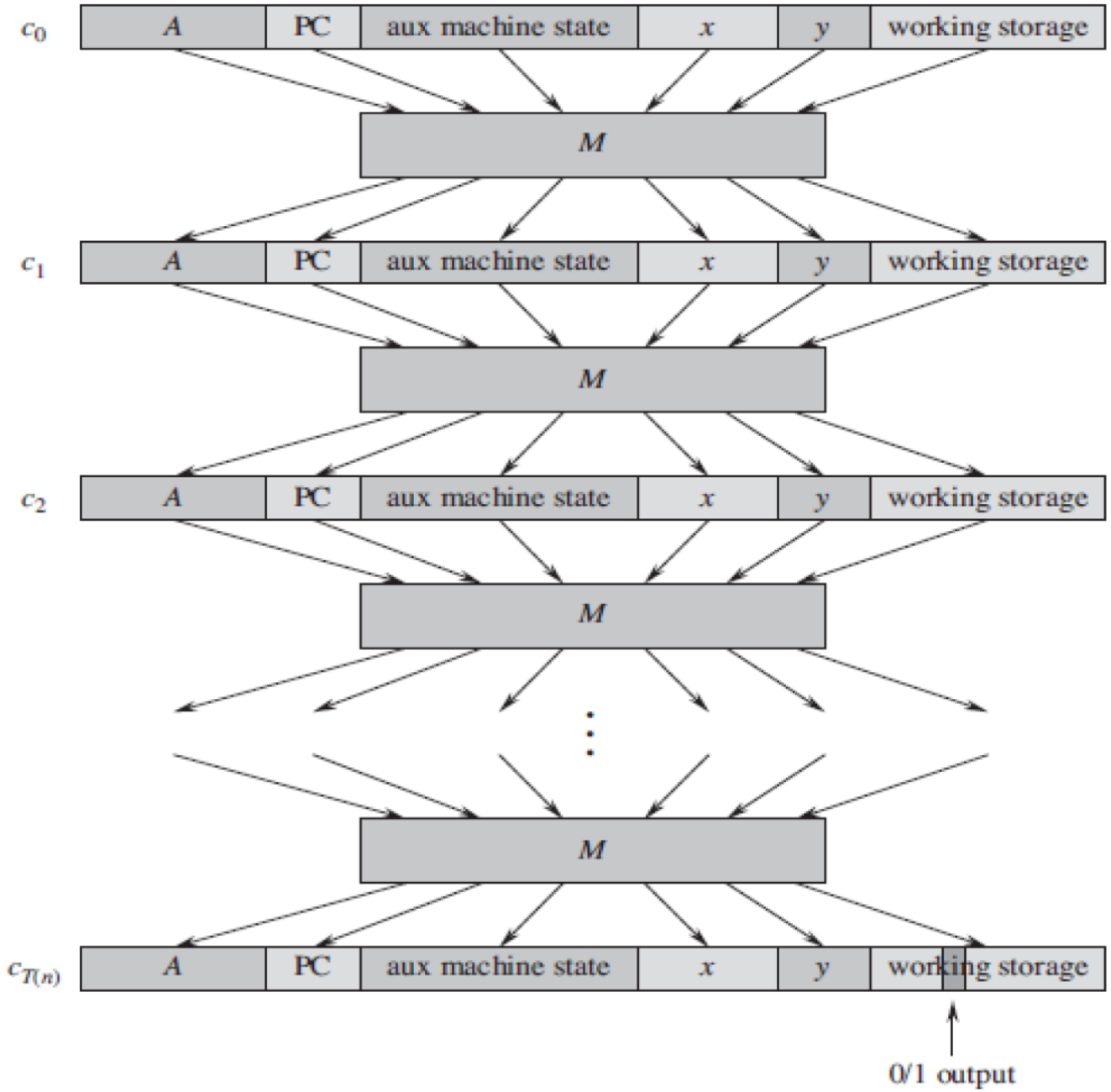
- The execution of an instruction can cause a value to be written to the program counter, which alters the normal sequential execution and allows the computer to loop and perform conditional branches.
- At any point during the execution of a program, the computer's memory holds the entire state of the computation.
- We call any particular state of computer memory a configuration.
- We can view the execution of an instruction as mapping one configuration to another. The computer hardware that accomplishes this mapping can be implemented as a boolean combinational circuit, which we denote by M in the proof of the following lemma.

Proof of Lemma 2

- 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 $x \in L$ if and only if $C \in$ CIRCUIT-SAT.
- Since $L \in NP$, there must exist an algorithm A that verifies L in polynomial time. The algorithm F that we shall construct uses the two-input algorithm A to compute the reduction function f.
- Let $T(n)$ denote the worst-case running time of algorithm A on length-n input strings, and let $k \geq 1$ be a constant such that $T(n) = O(n^k)$ and the length of the certificate is $O(n^k)$. (The running time of A is actually a polynomial in the total input size, which includes both an input string and a certificate, but since the length of the certificate is polynomial in the length n of the input string, the running time is polynomial in n.)

Proof of Lemma 2 (cont.)

- The basic idea of the proof is to represent the computation of A as a sequence of configurations.
- As the figure on the next slide illustrates, we can break each configuration into parts consisting of the program for A, the program counter and auxiliary machine state, the input x, the certificate y, and working storage. The combinational circuit M, which implements the computer hardware, maps each configuration c_i to the next configuration c_{i+1}, starting from the initial configuration c_0.
- Algorithm A writes its output (0 or 1) to some designated location by the time it finishes executing, and if we assume that thereafter A halts, the value never changes.
- Thus, if the algorithm runs for at most $T(n)$ steps, the output appears as one of the bits in the configuration $c_{T(n)}$.


c_0
A
PC
aux machine state
x
y
working storage
M
c_1
A
PC
aux machine state
x
y
working storage
M
c_2
A
PC
aux machine state
x
y
working storage
M
M
$c_{T(n)}$
A
PC
aux machine state
x
y
working storage
0/1 output

Proof of Lemma 2 (cont.)

- The reduction algorithm F constructs a single combinational circuit that computes all configurations produced by a given initial configuration. The idea is to paste together T (n) copies of the circuit M. The output of the ith circuit, which produces configuration c_i, feeds directly into the input of the (i+1)st circuit. Thus, the configurations, rather than being stored in the computer's memory, simply reside as values on the wires connecting copies of M.
- What the polynomial-time reduction algorithm F must do is as follows. 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 first 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 $c_{T(n)}$.

Proof of Lemma 2 (cont.)

- Algorithm F modifies circuit C' slightly to construct the circuit $C = f(x)$. First, it wires the inputs to C' corresponding to the program for A, the initial program counter, the input x, and the initial state of memory directly to these known values. Thus, the only remaining inputs to the circuit correspond to the certificate y. Second, it ignores all outputs from C', except for the one bit of $c_{T(n)}$ corresponding to the output of A. This circuit C, so constructed, computes $C(y) = A(x,y)$ for any input y of length $O(n^k)$. The reduction algorithm F, when provided an input string x, computes such a circuit C and outputs it.
- We need to prove two properties. First, we must show that F correctly computes a reduction function f. That is, we must show that C is satisfiable if and only if there exists a certificate y such that $A(x,y) = 1$. Second, we must show that F runs in polynomial time.

Proof of Lemma 2 (cont.)

- To show that F correctly computes a reduction function, let us suppose that there exists a certificate y of length $O(n^k)$ such that $A(x,y) = 1$. Then, if we apply the bits of y to the inputs of C, the output of C is $C(y) = A(x,y) = 1$. Thus, if a certificate exists, then C is satisfiable.
- For the other direction, suppose that C is satisfiable. Hence, there exists an input y to C such that $C(y) = 1$, from which we conclude that $A(x,y) = 1$. Thus, F correctly computes a reduction function.

- To complete the proof, we need to show that F runs in polynomial time w.r.t. to $n = |x|$.
- The first observation is that the number of bits required to represent a configuration is polynomial in n. The program for *A* itself has constant size, independent of the length of its input x. The length of the input x is n, and the length of the certificate y is $O(n^k)$. Since the algorithm runs for at most $O(n^k)$ steps, the amount of working storage required by *A* is polynomial in n as well.
- The combinational circuit M implementing the computer hardware has size polynomial in the length of a configuration, which is $O(n^k)$; hence, the size of M is polynomial in n. Most of this circuitry implements the logic of the memory system.
- The circuit *C* consists of at most $t = O(n^k)$ copies of M, and hence it has size polynomial in n.
- The reduction algorithm F can construct *C* from x in polynomial time, since each step of the construction takes polynomial time. **QED**