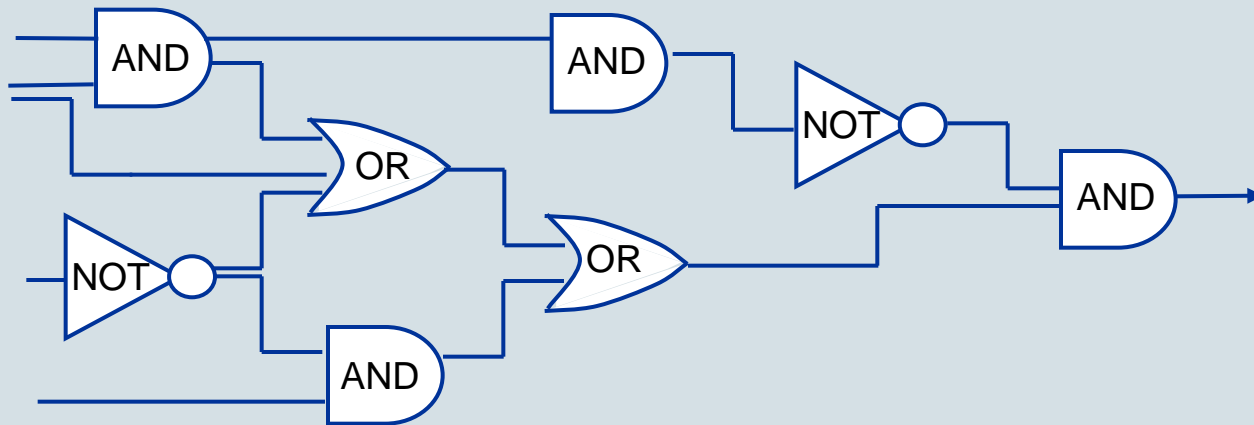


Algorithms (2IL15) – Lecture 9

NP-Completeness



Part I: Techniques for optimization

- backtracking
- greedy algorithms
- dynamic programming I
- dynamic programming II

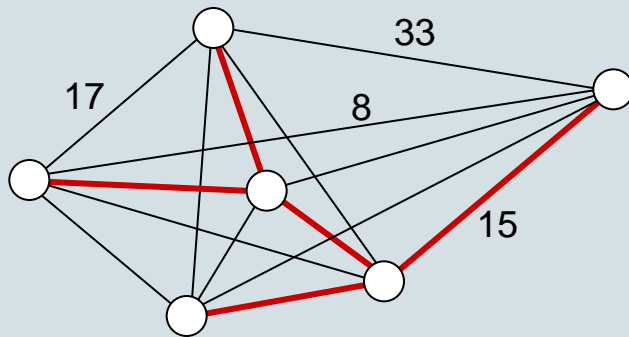
Part II: Graph algorithms

- shortest paths I
- shortest paths II
- max flow
- matching

Part III: Selected topics

- NP-completeness (**this week**)
- approximation algorithms
- linear programming

two similar (?) graph problems

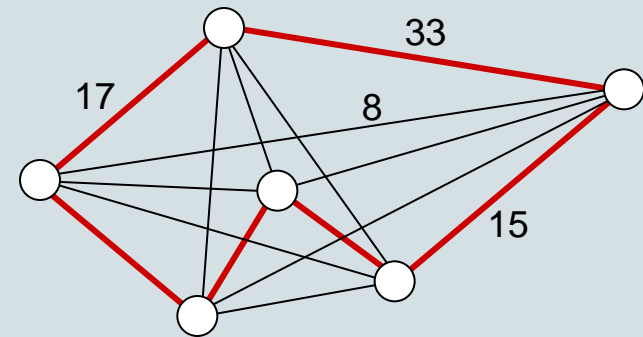


Min Spanning Tree

Input: weighted graph

Output: minimum-weight **tree**
connecting all nodes

greedy: $O(|E| + |V| \log |V|)$



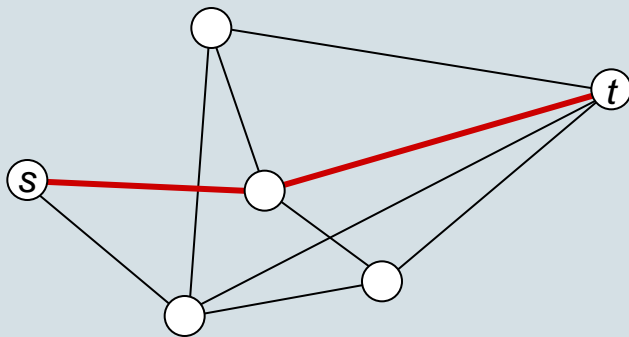
Traveling Salesman (TSP)

Input: complete weighted graph

Output: minimum-weight **tour**
connecting all nodes

backtracking: $O(|V|!)$

two similar (?) graph problems

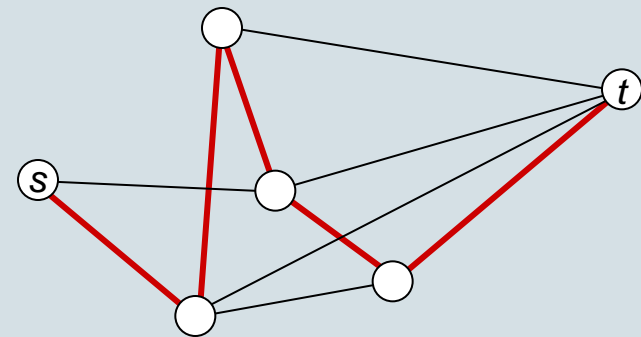


Shortest Path

Input: graph, nodes s and t

Output: simple s -to- t path with **minimum** number of edges

BFS: $O(|V| + |E|)$



Longest Path

Input: graph, nodes s and t

Output: simple s -to- t path with **maximum** number of edges

no polynomial-time algorithm known

$O(n^c)$ for some constant c

two similar (?) problems on boolean formulas

$$(x_1 \vee \neg x_3) \wedge (x_2 \vee x_5) \wedge (x_3 \vee \neg x_4)$$

$$(x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee x_3 \vee \neg x_5) \wedge (x_1 \vee x_3 \vee x_4)$$

2-SAT

Input: 2-CNF formula

Output: “yes” if formula can be satisfied, “no” otherwise

$O(\text{\#clauses})$

3-SAT

Input: 3-CNF formula

Output: “yes” if formula can be satisfied, “no” otherwise

no polynomial-time algorithm known

are we not clever enough to come up with fast (polynomial-time) algorithms for TSP, Longest Path, 3-SAT ...

... or are the problems inherently difficult ?

we don't know: $P \neq NP$?

TSP, Longest Path, 3-SAT ...are so-called **NP-hard** problems

if $P \neq NP$ (which most researchers believe to be the case)
then NP-hard problems cannot be solved in polynomial time

Complexity classes

P = class of all problems for which we can **compute** a solution in polynomial time

NP = class of problems for which we can **verify** a given solution in polynomial time

Technicalities:

- restrict attention to **decision problems**

ShortestPath: Given a graph G , nodes s and t , and a value k , is there a path from s to t of length $\leq k$?

Optimization problem is at least as hard as corresponding decision problem

Complexity classes

P = class of all problems for which we can **compute** a solution in polynomial time

NP = class of problems for which we can **verify** a given solution in polynomial time

Technicalities:

- restrict attention to **decision problems**
- what does “compute in polynomial time” mean?
 - polynomial time: $O(n^c)$ for some constant c , where n is input size
 - **input size**: number of input elements, or bit size? → **encoding**
 - **machine model**: Random Access Machine, or Turing machine?

Encodings

If we want to talk about problems being solved by a machine, then we should specify them so that the machine can understand them

→ encode input as bit string

Does it matter how we encode exactly?

- yes, for certain special problems and “stupid” encodings
- no, as long as we use “reasonable” encodings

Encodings

If we want to talk about problems being solved by a machine, then we should specify them so that the machine can understand them

→ encode input as bit string

For **decision problems** we get:

- bit strings representing “yes”-instances
- bit strings representing “no”-instances

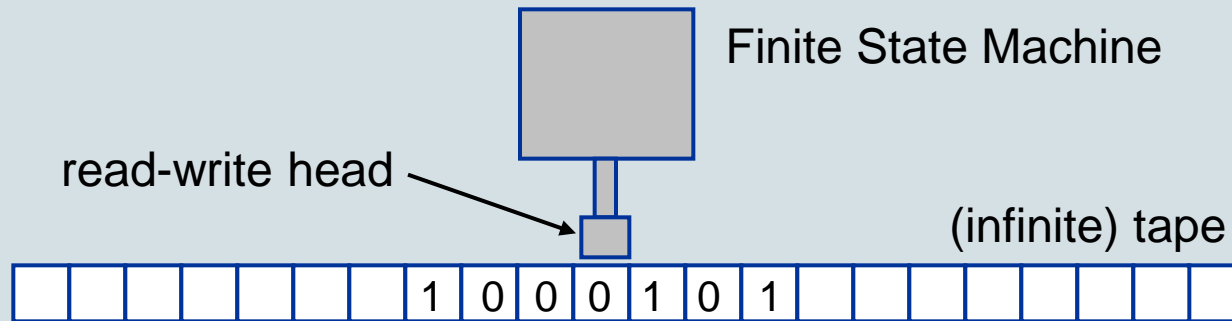
Formal language theory

Σ^* = all strings consisting of zero or more characters from alphabet Σ

language = subset of Σ^*

“yes”-instances of given decision problems \equiv language over $\{0,1\}^*$

Turing machines

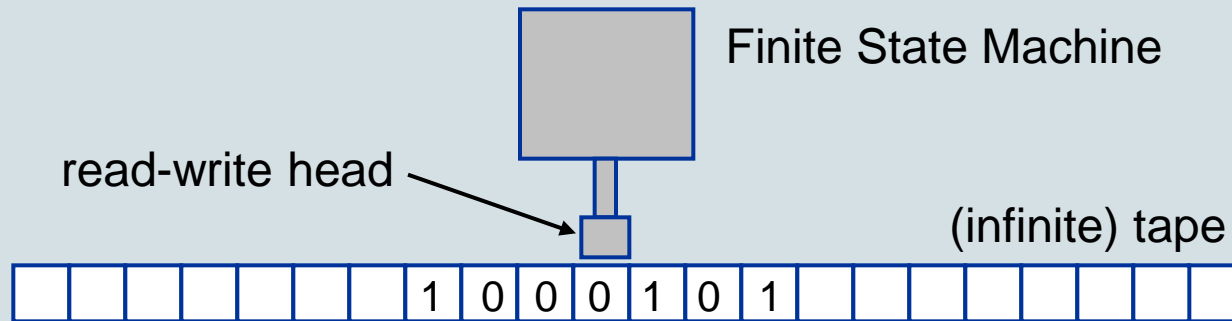


actions of Turing machine (move head, write) depend on

- current state
- symbol currently read
- transition rules specified by FSM

P = problems solvable in polynomial time on Turing machine

Turing machines



actions of Turing machine (move head, write) depend on

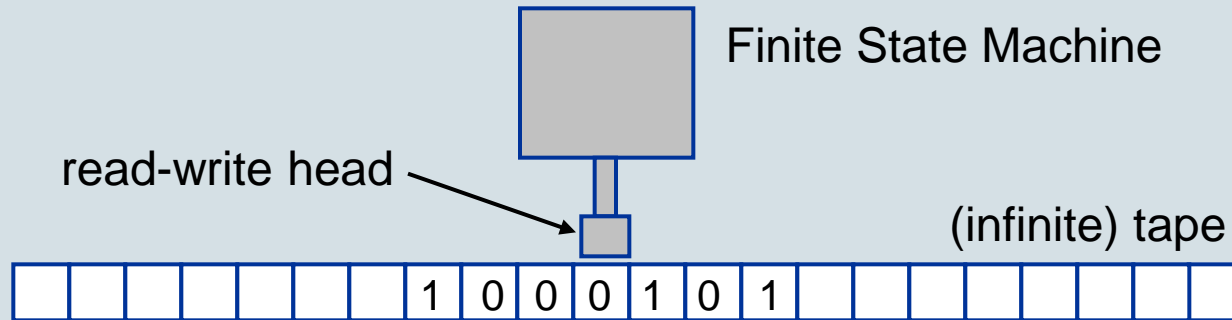
- current state
- symbol currently read
- transition rules specified by FSM

non-deterministic Turing machine

- Turing machine using non-deterministic FSM:
 - several transitions rules may apply simultaneously
- machine accepts iff one of the transitions can lead to acceptance

NP = problems solvable in polynomial time on non-det Turing machine

Turing machines



P = problems solvable in polynomial time on Turing machine

NP = problems solvable in polynomial time on non-det Turing machine

Does it matter if we choose a different machine model? No.

- anything that is computable is computable by a Turing machine
(*Church-Turing Thesis*)
- “polynomial time” on Turing machine \equiv “polynomial time” on a RAM

The (somewhat informal) way we will look at it:

machine model + running time analysis as usual:

- machine model: random access machine
- input size: number of elements

P = decision problems for which there exists polynomial-time algorithm

NP = decision problems for which there exists a polynomial-time verifier

algorithm A with two inputs

- input to the problem: x
- certificate: y

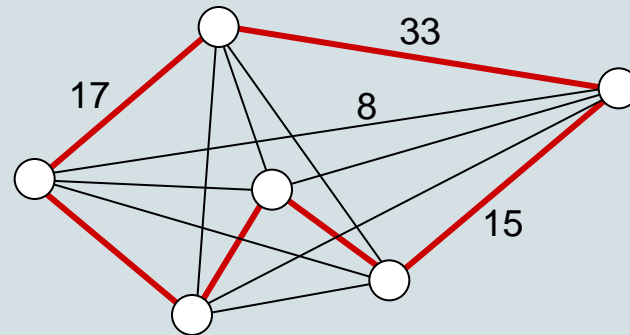
A is **polynomial-time verifier**: for any x there **exists** certificate y such that $A(x,y)$ outputs “yes” iff x is “yes”-instance, and A runs in polynomial time for such instances.

(“no”-instances do not have to be verifiable in polynomial time)

TSP

Input: complete, weighted undirected graph G , and a number k

Question: does G have a tour of length at most k visiting all nodes?



Claim: TSP is in NP

Proof: Consider “yes”-instance $x = (G, k)$.
Let y be a tour in G of length at most k .

Verifier: must check in polynomial time that

- y is a tour visiting all nodes
- $\text{length}(y) \leq k$.



Notes:

- verifying optimization problem is much harder
- verifying “no”-instance is much harder

3-SAT

Input: 3-CNF formula F

Question: is there a truth assignment to variables that makes F true?

$$(x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee x_3 \vee \neg x_5) \wedge (x_1 \vee x_3 \vee x_4)$$

Claim: 3-SAT is in NP

Proof: Consider “yes”-instance $x = (G, k)$.

Let y be a description of a satisfying truth assignment.

($x_1 = \text{TRUE}$, $x_2 = \text{TRUE}$, $x_3 = \text{FALSE}$, etc.)

Verifier: must check in polynomial time that F evaluates to TRUE for the given assignment



Theorem: $P \subseteq NP$

Proof:

Consider problem in P , let ALG be polynomial-time algorithm for problem.

Let x be “yes”-instance.

Take $y = \text{empty}$.

Verifier: just run ALG on x , and ignore y



One-million dollar(*) question:

$P = NP ?$

almost all researchers think $P \neq NP$

(*) One of 7 Millenium Problems for which Clay Math Institute awards \$1,000,000

NP-complete problems: the most difficult problems in NP

if you can solve **any** NP-complete problem in polynomial time,
then you can solve **every** problem in NP in polynomial time

Why is it important to know about NP-completeness?

- if a problem is NP-complete, then it cannot be solved in polynomial time (unless $P = NP$)

You should know

- what the complexity classes P and NP are
- what an NP-complete problem is
- a few important problems that are NP-complete
- how you can prove a new problem to be NP-complete

NP-complete problems: the most difficult problems in NP

if you can solve **any** one of the NP-complete problems in polynomial time,
then you can solve **every** problem in NP in polynomial time

because algorithm for NP-complete problem can be used to solve any
other problem in NP, after suitable preprocessing (**reduction**)

Reductions

problem A is **polynomial-time reducible** to problem B if there is a reduction algorithm mapping instances of A to instances of problem B such that

- “yes”-instances of A are mapped to “yes”-instances of B
- “no”-instances of A are mapped to “no”-instances of B
- the reduction algorithm runs in polynomial time

Notation: problem A \leq_p problem B

Example: Hamiltonian Cycle \leq_p TSP

Hamiltonian Cycle

Input: undirected graph $G=(V,E)$

Question: is there a tour in G ? (tour = cycle visiting every node exactly once)

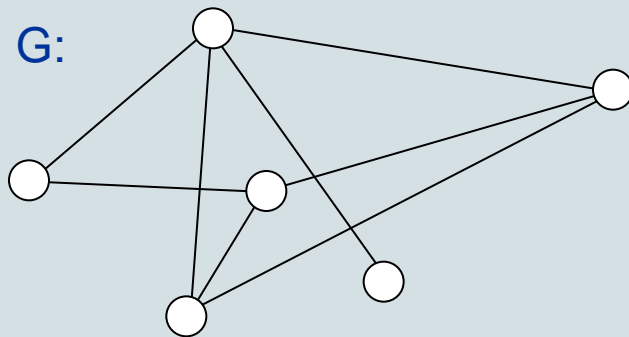
TSP

Input: complete, weighted undirected graph $G=(V,E)$, and a number k

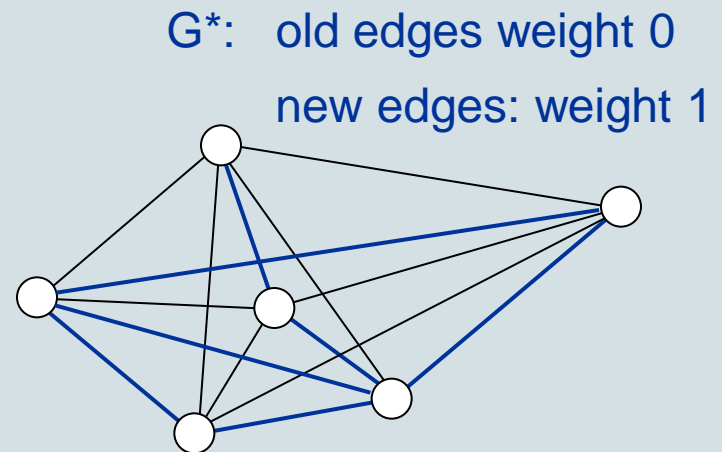
Question: is there a tour of length at most k ?

Theorem: Hamiltonian Cycle \leq_p TSP

Proof.



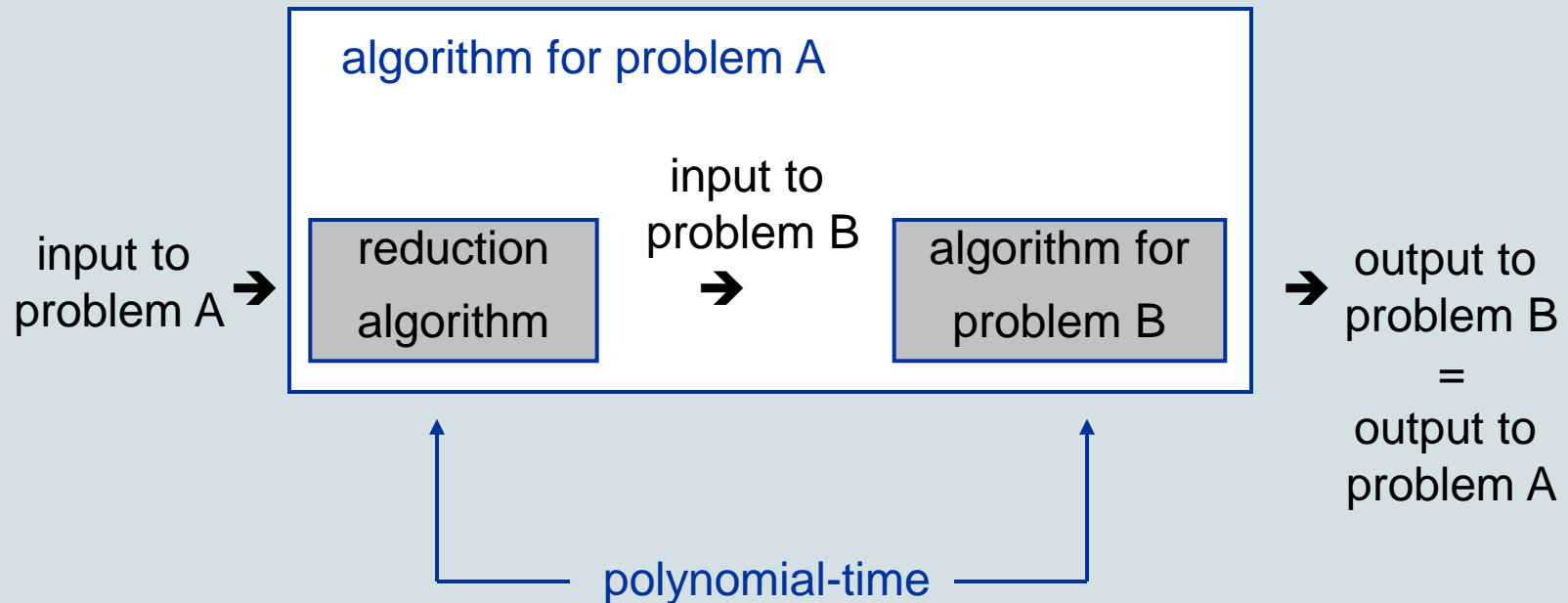
reduction
algorithm



G has Hamiltonian cycle iff G^* has tour of length at most 0

Theorem: If problem $A \leq_p$ problem B and problem B is in P then problem A is also in P .

Proof.



NP-hardness and NP-completeness

Problem A is **NP-hard** if problem $B \leq_p$ problem A for all problems B in NP.

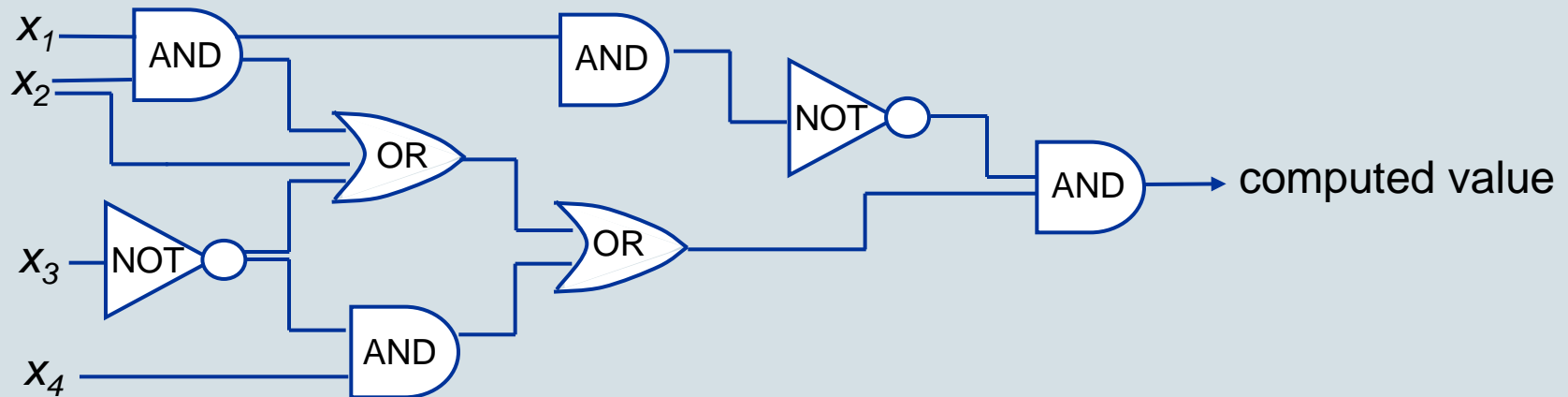
Problem A is **NP-complete** if (i) problem A is NP-hard, (ii) problem A is in NP.

Theorem: If any one NP-complete problems can be solved in polynomial time, then every problem in NP is solvable in polynomial time and $P = NP$.

Theorem: If problem A is NP-hard and problem $A \leq_p$ problem B, then problem B is also NP-hard.

now we know how to prove a problem is NP-complete ...
... if we have an NP-complete problem to start from

A first NP-complete problem: Circuit-SAT

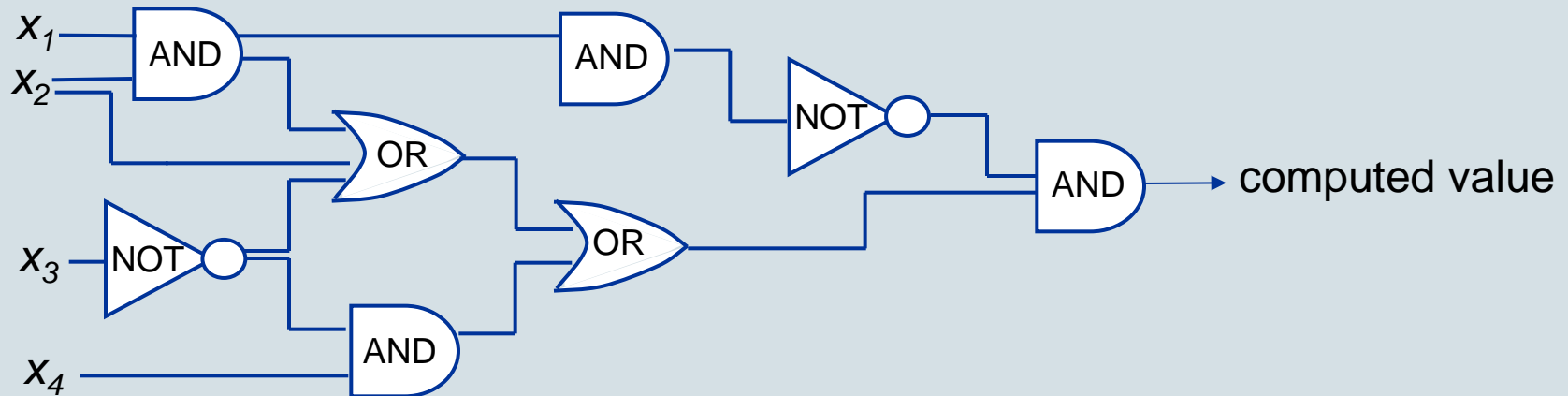


Circuit-SAT

Input: Boolean combinational circuit

Question: Can variables be set such that circuit evaluates to true ?

A first NP-complete problem: Circuit-SAT



Lemma 1: Circuit-SAT is in NP

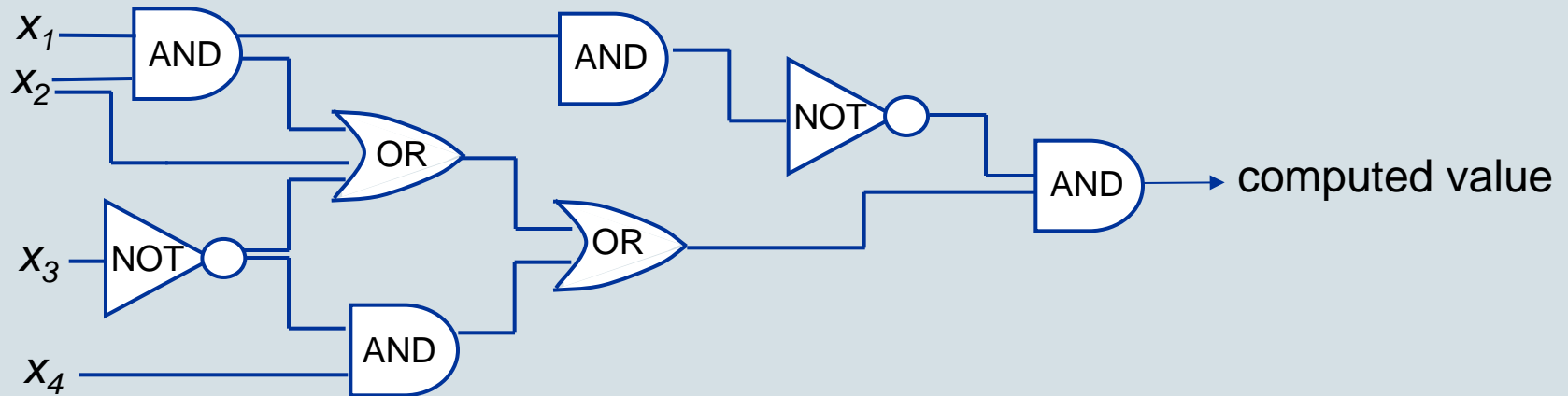
Proof. Must show that there exists a polynomial-time verifier.

Consider “yes”-instance x (so: x = description of satisfiable circuit)

certificate y : take satisfying truth assignment

We can easily check in polynomial time if truth assignment produces TRUE.

A first NP-complete problem: Circuit-SAT



Lemma 2: For every problem A in NP, we have: $A \leq_p \text{Circuit-SAT}$

Lemma: For every problem A in NP, we have: $A \leq_p \text{Circuit-SAT}$

Proof: (sketch of sketch of ...)

Step 1: show the following:

if an algorithm runs in polynomial time, then there is a polynomial-size Boolean circuit that “implements” the algorithm, and such a circuit can be constructed in polynomial time

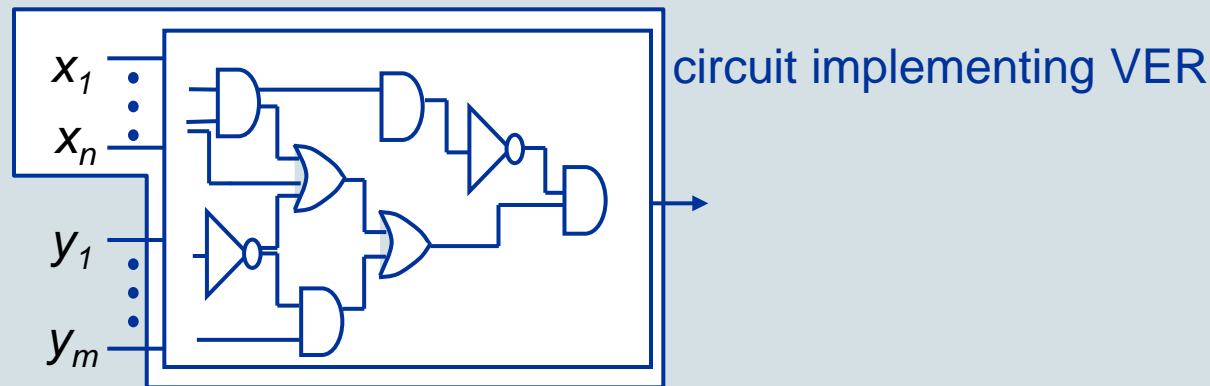
(idea: algorithm runs on computer that is essentially a Boolean circuit)

Lemma: For every problem A in NP, we have: $A \leq_p \text{Circuit-SAT}$

Proof: (sketch of sketch of ...)

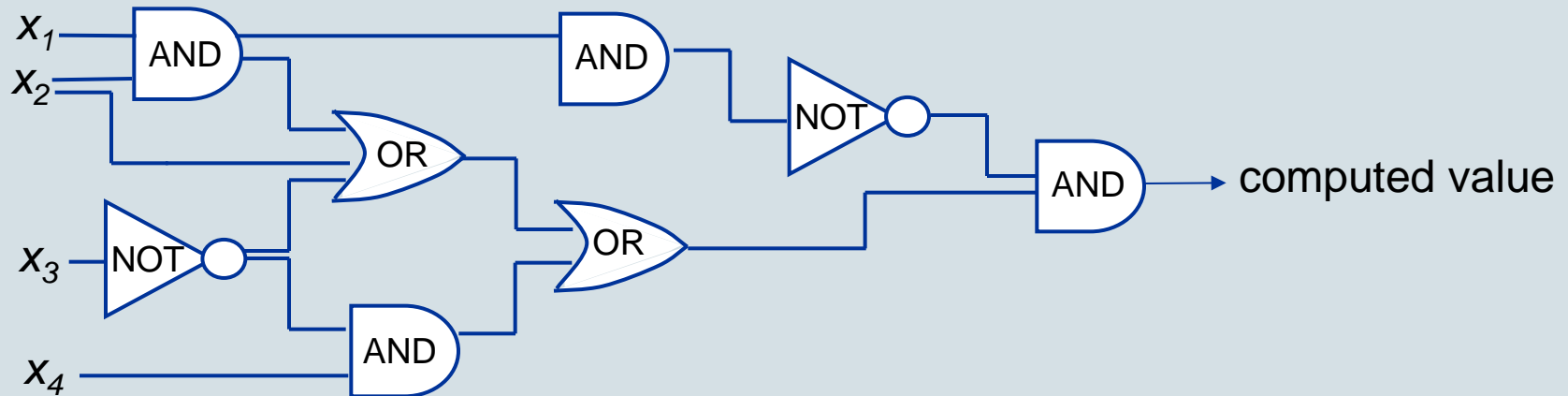
Step 2: Describe algorithm that performs reduction:

- A in NP, so A has verification algorithm $\text{VER}(x,y)$ that checks if x is “yes”-instance using certificate y . Construct circuit implementing VER .



- “Fix” the variables corresponding to x according to the given input
 - Run Circuit-SAT
- Circuit-SAT returns “yes” iff x is “yes” instance for problem A

A first NP-complete problem: Circuit-SAT



Lemma 1: Circuit-SAT is in NP

Lemma 2: For every problem A in NP, we have: $A \leq_p \text{Circuit-SAT}$
(in other words: Circuit-SAT is NP-hard).

Theorem: Circuit-Sat is NP-complete.

Proving NP-completeness of other problems

Theorem: If problem A is NP-hard and problem $A \leq_p$ problem B, then problem B is also NP-hard.

General strategy to prove that a problem B is NP-complete

1. Select problem A that is known to be NP-complete.
2. Prove that $A \leq_p B$:
 - i. Describe reduction algorithm, which maps instances x of A to instances $f(x)$ of B.
 - ii. Prove that x is “yes”-instance for A iff $f(x)$ is “yes”-instance for B
 - iii. Prove that reduction algorithm runs in polynomial time

(Now you have shown that B is NP-hard.)

3. Prove that B is in NP by giving polynomial-time verification algorithm.

Proving NP-completeness of other problems: example

SATISFIABILITY

Input: A Boolean formula

Question: Is the formula satisfiable ?

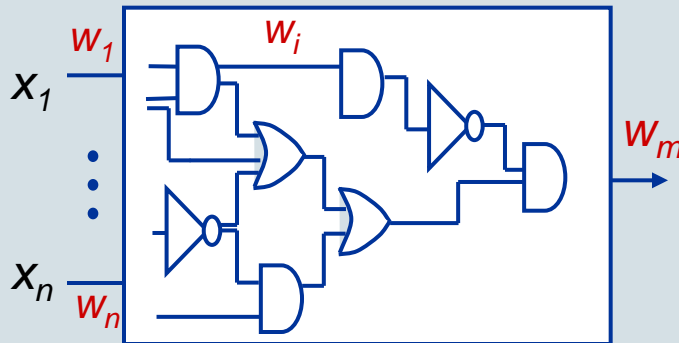
$$((x_1 \rightarrow \neg x_3) \leftrightarrow (x_1 \vee \neg x_2 \vee x_3)) \wedge (\neg (x_2 \vee x_3 \vee x_5) \rightarrow (x_1 \vee x_3 \vee x_4))$$

Theorem: SATISFIABILITY is NP-complete.

Proof:

1. Select known NP-complete problem: **Circuit-SAT**
2. Prove that $\text{Circuit-SAT} \leq_p \text{SATISFIABILITY}$
 - i. Describe **reduction algorithm**, which maps instances x of A to instances $f(x)$ of B .
 - ii. Prove that x is “yes”-instance for A iff $f(x)$ is “yes”-instance for B
 - iii. Prove that reduction algorithm runs in polynomial time
3. Prove that SATISFIABILITY is in NP by giving polynomial-time verification algorithm.

Reduction algorithm: convert Boolean circuit into Boolean formula

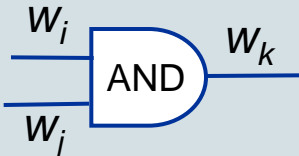


problem: shared “sub-circuits”



cannot convert in straightforward top-down manner

- introduce variable for every wire w_i (including input and output wires)

- write clause for every gate:  $\rightarrow (w_i \wedge w_j) \leftrightarrow w_k$

- Boolean formula: (output-wire variable) \wedge conjunction of gate-clauses

check: – reduction uses polynomial time

– Boolean circuit satisfiable iff Boolean formula satisfiable

Theorem: SATISFIABILITY is NP-complete.

Proof:

1. Select known NP-complete problem: **Circuit-SAT**
2. Prove that $\text{Circuit-SAT} \leq_p \text{SATISFIABILITY}$
 - i. Describe reduction algorithm, which maps instances x of A to instances $f(x)$ of B .
 - ii. Prove that x is “yes”-instance for A iff $f(x)$ is “yes”-instance for B
 - iii. Prove that reduction algorithm runs in polynomial time

DONE

3. Prove that SATISFIABILITY is in NP by giving polynomial-time verification algorithm.

EASY



Are NP-complete problems the most difficult problems ?

No.

- there many other complexity classes besides P and NP:

co-NP, PSPACE, EXPTIME, EXPSPACE, P^{NP} , ...

some of these (e.g. EXPSPACE) are known to contain problems not in NP, many relations between these classes are unknown

- there are also **undecidable problems**:

Halting Problem: decide for a given computer program and input whether the program will halt on the given input.

It is impossible to give an algorithm for the Halting Problem that is guaranteed to terminate for every (program, input) pair.

Summary

- **P**: class of decision problems that can be solved in polynomial time
- **NP**: class of decision problems that can be verified in polynomial time
- **NP-complete problems**: problems A in NP such that $B \leq_p A$ for any B in NP

if a problem is NP-complete, then it cannot be solved in polynomial time (unless $P = NP$)

- **Circuit-SAT** and **SATISFIABILITY** are NP-complete

Friday: more NP-complete problems