2nd part: Approximation Algorithms

Preamble: NP-Hardness (a primer on)

In the 30s we started to understand what is or isn't *effectively* computable.

By the 60s computer scientists had developed fast algorithms to solve some problems, while for others the only known algorithm were very slow.

In the 70s we started to understand what is or isn't *efficiently* computable.

In 1965 Edmonds defined what efficient means: an algorithm is "efficient" if its running time is $O(n^k)$ for some constant k (n = input size)

Problems for which a polynomial-time alg. exists are called *tractable* $\rightarrow$ all the algorithms seen so far

If no polynomial-time alg. exists then the problem is called *intractable*

Examples to illustrate how perplexing questions about efficient computation can be:

1) Eulerian Circuit problem: given an undirected graph, an <u>Eulerian Circuit</u> is a cycle that crosses every edge exactly once.

almost the same

This problem can be solved in linear time.

2) Hamiltonian Circuit problem: given an undirected graph, an <u>Hamiltonian Circuit</u> is a cycle that passes through every vertex exactly once.

To date, no one knows a polynomial algorithm to solve it!

3) Minimum Spanning Trees: given a connected, undirected graph and a function $w: E \to \mathbb{R}$, output a spanning tree $T \subseteq E$ minimizing $\sum_{e \in T} w(e)$

almost the same

4) Traveling Salesperson Problem (TSP): given a complete, undirected graph and a function $w: E \to \mathbb{R}$, output a tour $T \subseteq E$ (i.e. a cycle that passes

through every vertex exactly once) minimizing

$$\sum_{e \in T} w(e).$$

To date, no one knows a polynomial algorithm to solve it!

A much easier task: given a graph and a list of vertices C, <u>check</u> if C is an Hamiltonian circuit

Easy to solve: class P ("polynomial time")

1) and 3) $\in P$

Easy to verify: class NP ("nondeterministic polynomial")

1), 3), 2), 4)

rookie mistake:

NP $\neq$ not-polynomial

To simplify the study of the complexity of problems, we limit our attention to the following class of problems:

<u>decision problems</u>: problems with a Boolean answer → YES / → NO

$\boxed{P}$ is the set of decision problems that can be solved in polynomial time

$\boxed{NP}$ is the set of decision problems with the following property: if the answer is YES then there is a proof ("certificate") of this fact that can be checked in polynomial time.

complexity classes

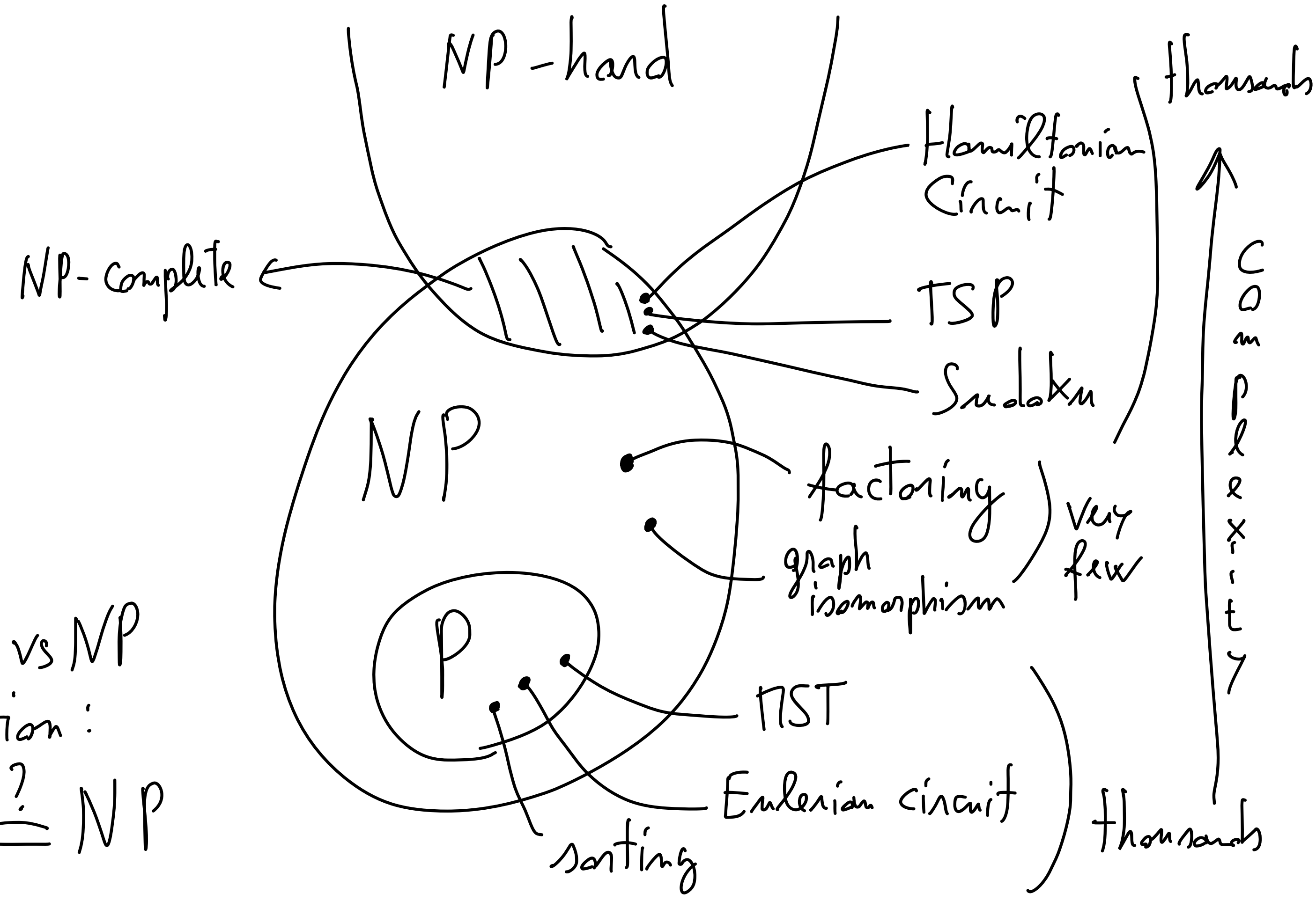
$\boxed{co\text{-}NP}$ essentially the opposite of NP: property: if the answer is NO, then there is a proof of this fact that can be checked in polynomial time

NP-hardness: a computational problem is **NP-hard** if a polynomial-time algorithm for it would imply a polynomial-time algorithm for every problem in NP.

→ are the hardest problems in NP

A problem is **NP-complete** if it is both NP-hard and in NP.

What (today) we think the world looks like:



The P vs NP question:

$P \overset{?}{=} NP$

Why study NP-hardness:

- being NP-hard is <u>strong evidence</u> that a problem is intractable

- it suggests you should use a $\neq$ approach, such as
  - identify tractable special cases
  - compromise on correctness: $\longrightarrow$ <u>approximation algorithms</u>

Cook-Levin Theorem ('71): 3-SAT is NP-hard

<u>SAT</u>: formula satisfiability

input: a Boolean formula like $\underbrace{(b \wedge \bar{c})}_{\text{clause}} \vee (\bar{a} \wedge b)$

AND ($\wedge$), NOT ($\bar{\ }$), OR ($\vee$)

output: it is possible to assign Boolean values to the variables $a, b, c, \ldots$ so that the entire formula evaluates to TRUE?

A special case of SAT:

3-SAT : a Boolean formula is in conjunctive normal form (CNF) if it is a conjunction (AND) of several clauses, each of which is the disjunction (OR) of several literals, each of which is either a variable or its negation

↓

a.k.a.
3-CNF-SAT

example: $(a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d)$

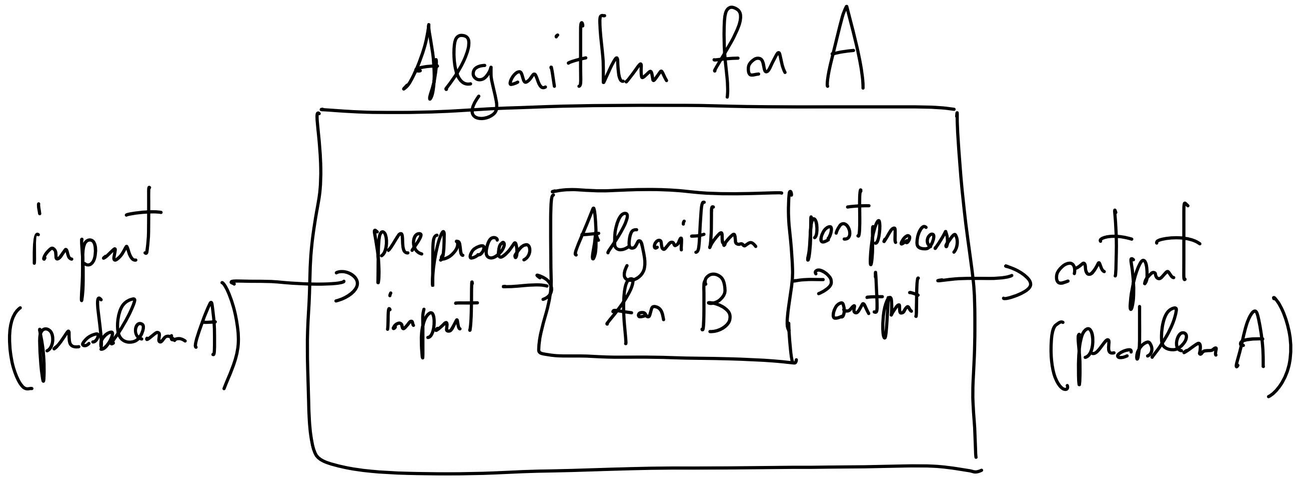
a 3-CNF formula is a CNF formula with <u>exactly 3 literals per clause</u>

How to show a problem is NP-hard?

<u>Reductions</u> (general scheme/concept)

A <u>reduction</u> is an algorithm for transforming one problem into another. It is the way we compare the computational complexity of 2 problems, A and B

A problem A <u>reduces</u> to problem B if an algorithm that solves B can be translated into one that solves A:



notation: $A \leq B$

"reduces to"

If the reduction is "efficient" then B is as hard as A (equiv. A is not harder than B)