Complexity Theory Primer

We will discuss:

- Optimization and decision problems
- The classes *P*, *EXP*, and *NP*
- *NP*-completeness, reductions, and *NP*-hardness

Source and References

This lecture is mainly based on a script by Prof. Heinz Schmitz, FH Trier.

Further reading:

- U. Schöning: *Theoretische Informatik kurz gefaßt*. B. I. Wissenschaftsverlag, Mannheim, 1992. ISBN 3-411-15641-4
- M. Garey, D. Johnson: *Computers and Intractability: a Guide to the Theory of NP-Completeness*. Freeman, San Francisco, 1979. ISBN 0-7167-1044-7
- C. Papadimitriou: *Computational Complexity*. Addison Wesley, Reading, 1994. ISBN 0-201-53082-1
- W. Cook, W. Cunningham, W. Pulleyblank, A. Schrijver: *Combinatorial Optimization*. John Wiley & Sons, New York, 1998. ISBN 0-471-55894-X

Complexity theory

Algorithms with polynomial and algorithms with exponential running time differ dramatically with respect to their applicability to practical problems:

n	20	40	60	100	300
time					
n	10^{-8} sec	10^{-8} sec	10^{-7} sec	10^{-7} sec	10^{-7} sec
n^2	10^{-7} sec	10^{-6} sec	10^{-6} sec	10^{-5} sec	10^{-4} sec
n^3	10^{-5} sec	10^{-4} sec	10^{-4} sec	10^{-3} sec	10^{-2} sec
$2^{n/\log n}$	10^{-7} sec	10^{-6} sec	10^{-5} sec	10^{-3} sec	79 *days*
2^n	10^{-3} sec	19 *min*	37 *years*	10^{13} *years*	10^{73} *years*

Assumption: 10^9 operations/sec

Complexity theory (2)

The field of *complexity theory* investigates classes of problems of different complexity and their relations.

$\Rightarrow$ Do problems exist that can be solved in time $O(n \log n)$, but not in time $O(n)$?

$\Rightarrow$ Can all problems for which we can compute a solution in $O(\log n)$ space be solved in polynomial running time?

Answers to these fundamental questions have considerable practical impact.

Optimization and decision problems

Given a certain problem $\mathcal{P}$, we denote by $I_{\mathcal{P}}$ the set of problem instances and by $S_{\mathcal{P}}$ the set of feasible solutions.

Example.

$$
\begin{aligned}
I_{\mathcal{P}} &:= \{(m, D) \mid m \in \mathbb{N} \text{ and } D \text{ is } m \times m \text{ matrix} \\
&\qquad\qquad \text{with } D(i,j) \in \mathbb{N} \text{ for } 1 \leq i, j \leq m\} \\
S_{\mathcal{P}} &:= \{(c_1, \ldots, c_k) \mid k \in \mathbb{N} \text{ and} \\
&\qquad\qquad \{c_1, \ldots, c_k\} = \{1, \ldots, k\}\}
\end{aligned}
$$

The set of feasible solutions of a certain instance $x \in I_{\mathcal{P}}$ is a subset of $S_{\mathcal{P}}$.

In the example: If $x = (m, D)$, then all m-ary tuples in $S_{\mathcal{P}}$ are feasible.

In this manner we are able to define the therm *optimization problem* in a formal way.

Optimization and decision problems (2)

Definition. An *optimization problem* $\mathcal{P}$ is a tuple $(I_{\mathcal{P}}, S_{\mathcal{P}}, sol_{\mathcal{P}}, m_{\mathcal{P}}, goal)$, where

1. $I_{\mathcal{P}}$ is the set of *instances*,
2. $sol_{\mathcal{P}} : I_{\mathcal{P}} \mapsto 2^{S_{\mathcal{P}}}$ is a function that returns the set of *feasible solutions* for a given instance $x \in I_{\mathcal{P}}$
3. $m_{\mathcal{P}} : I_{\mathcal{P}} \times S_{\mathcal{P}} \mapsto \mathbb{N}$ is a function, which, given a pair $(x, y(x))$ with $x \in I_{\mathcal{P}}$ and $y(x) \in sol_{\mathcal{P}}(x)$, returns the *value* $m_{\mathcal{P}}(x, y(x)) \in \mathbb{N}$ of solution $y(x)$ of the instance x, and
4. $goal \in \{\min, \max\}$.

Optimization and decision problems (3)

Example. Minimum traveling salesman (MIN-TSP)

Sets $I_{\text{MIN-TSP}}$ and $S_{\text{MIN-TSP}}$ are already defined in the previous example.

$$
\begin{aligned}
sol_{\text{MIN-TSP}}(m, D) &:= S_{\text{MIN-TSP}} \cap \mathbb{N}^m \\
m_{\text{MIN-TSP}}((m, D), (c_1, \ldots, c_m)) &:= \sum_{i=1}^{m-1} D(c_i, c_{i+1}) + D(c_m, c_1) \\
goal_{\text{MIN-TSP}} &:= \min
\end{aligned}
$$

If the context is clear, we omit the indices (e.g., MIN-TSP) and write y instead of $y(x)$.

Example. Maximum similarity multiple sequence alignment with WSOP score (MAX-MSA-WSOP).
Blackboard.

Optimization and decision problems (4)

We need some terminology to deal with optimal solutions:

Definition. A solution $y^*(x)$ is *optimal* for the instance x, if

$$m(x, y^*(x)) = goal\{v \mid v = m(x, z) \text{ and } z \in sol(x)\}.$$

The *set of all optimal solutions* $y^*(x)$ of x is referred to as $sol^*(x)$.

The *optimal value* of an instance x is denoted by $m^*(x)$.

$\Rightarrow$ it follows directly that all optimal solutions have the same value

Optimization and decision problems (5)

There are three important variants of optimization problems:

1. *constructive* problem $\mathcal{P}_C$

   input: $x \in I_{\mathcal{P}}$
   output: a $y^*(x)$ and $m^*(x)$

2. *evaluation* problem $\mathcal{P}_E$

   input: $x \in I_{\mathcal{P}}$
   output: $m^*(x)$

3. *decision* problem $\mathcal{P}_D$

   input: $x \in I_{\mathcal{P}}$ and $k \in \mathbb{N}$
   output: if $goal_{\mathcal{P}} = \max$ (the case $goal_{\mathcal{P}} = \min$ is analogous)

   1, if $m^*(x) \geq k$

   0, otherwise

Unless otherwise stated, an “optimization problem” is a constructive problem.

Optimization and decision problems (6)

We call $\mathcal{P}_D$ the *decision problem corresponding to* $\mathcal{P}$. It is "easier" than $\mathcal{P}_C$ or $\mathcal{P}_E$, since solving the constructive or evaluation problem also solves the decision problem:

- calculate $m^*(x)$ with an algorithm for $\mathcal{P}_C$ or $\mathcal{P}_E$
- compare k and $m^*(x)$

Example. TRAVELING SALESMAN (TSP) is the decision problem corresponding to MIN-TSP. Here, the input consists of additional maximal costs $k \in \mathbb{N}$ of a tour.

input: $m \in \mathbb{N}$, an $m \times m$ distance matrix with $D(i,j) \in \mathbb{N}$ *and* $k \in \mathbb{N}$.
output: 1, if there exists a permutation $(c_1, \ldots, c_m)$ of $\{1, \ldots, m\}$ with *cost* $\leq k$,
0, otherwise.

MSA-WSOP example. Blackboard.

Optimization and decision problems (7)

The following example shows a typical packing problem, where the aim is to maximize the profit of the chosen items under the restriction that they not exceed a maximum size S.

Example. MAXIMUM KNAPSACK

input: m items with sizes $s_1, \ldots, s_m \in \mathbb{N}$, profits $v_1, \ldots, v_m \in \mathbb{N}$, and a maximal size $S \in \mathbb{N}$.

output: $I \subseteq \{1, \ldots, m\}$ with $\sum_{i \in I} s_i \leq S$

measure: $\sum_{i \in I} v_i$

KNAPSACK is the corresponding decision problem. Here, an additional minimum total profit that has to be achieved is given:

input: m items with sizes $s_1, \ldots, s_m \in \mathbb{N}$ and profits $v_1, \ldots, v_m \in \mathbb{N}$, maximal size $S \in \mathbb{N}$ *and* $k \in \mathbb{N}$.

output: 1, if there exists a set $I \subseteq \{1, \ldots, m\}$ with $\sum_{i \in I} s_i \leq S$ *and* $\sum_{i \in I} v_i \geq k$,

0, otherwise.

Optimization and decision problems (8)

A *decision problem* only asks for a *yes*-*no*-answer. This can also be expressed in terms of a *set* $E \subseteq I_{\mathcal{P}}$ for which the characteristic function

$$c_E(x) := \begin{cases} 1, & \text{if } x \in E, \\ 0, & \text{otherwise} \end{cases}$$

has to be computed for a given instance $x \in I_{\mathcal{P}}$ (*membership* problem).

An equivalent, often used, definition is in terms of *languages*: Given an alphabet Σ, a language L is a subset of Σ^*. The language that corresponds to a decision problem contains those words (encodings of input instances) that yield a yes-answer.

We also say that algorithms that solve a decision problem *decide* the corresponding language.

Example.

$$\begin{aligned} \text{PRIME} &:= \{x \in \mathbb{N} \mid x \text{ prime number}\} \\ &= \{2, 3, 5, 7, 11, 13, \ldots\} \subseteq \{1, 2, 3, 4, \ldots\} \end{aligned}$$

Optimization and decision problems (9)

Example.

problem: MINIMUM PATH
input: graph $G = (V, E)$, two vertices $s, t \in V$
output: a path $(v_0, v_1, \ldots, v_k)$ from $s = v_0$ to $t = v_k$ with minimum k

problem: PATH
input: graph $G = (V, E)$, two vertices $s, t \in V$ *and* $k \in \mathbb{N}$.
output: 1, if there exists a path $(v_0, v_1, \ldots, v_l)$ from $s = v_0$ to $t = v_l$ *with* $l \leq k$,

0, otherwise.

The classes *P* and *NP*

We introduce some notation from complexity theory in order to classify optimization problems. For systematic reasons, we focus on *decision problems*.

Polynomial time.

We call algorithmic problems for which we know an algorithm that solves them in time $O(n^k)$ for some $k \in \mathbb{N}$ *efficiently solvable* (cf. table of running times at the beginning of the primer). The motivates the definition of the complexity class *P*:

$$P := \{E \text{ decision problem} \mid \text{there exists a } k \in \mathbb{N} \text{ and an algorithm } A \text{ such that } A \text{ solves } E \text{ in time } O(n^k)\}$$

An algorithm *solves* a decision problem E, if it computes c_E.

The classes *P* and *NP* (2)

Example.

We already know an algorithm that solves PATH in time $O(n^2)$:*

- Each vertex has at most $|V|$ direct successors.
- We only have to consider paths up to length $k < |V|$.

Thus, PATH $\in$ *P*.

*The algorithm sketched above is a very short summary of the Bellman-Ford algorithm to compute shortest paths in a graph.

The classes *P* and *NP* (3)

Example.

A naïve algorithm for PRIME checks, given an input $x \in \mathbb{N}$, whether one of the numbers in $\{2, \ldots, \sqrt{x}\}$ is a factor of x. This results in a running time of $\Omega(\sqrt{x}) = \Omega(2^{\frac{n-1}{2}})$.

In 2002, M. Agrawal, N. Kayal, and N. Saxena gave a polynomial time algorithm for PRIME, thus PRIME $\in$ *P*.

The classes *P* and *NP* (4)

How could we solve KNAPSACK? We have to find a subset of items $\{1, 2, \ldots, m\}$ with certain properties.

We could proceed as follows:

1. Consider every subset $I = \emptyset, \{1\}, \{2\}, \ldots$ of $\{1, 2, \ldots, m\}$
2. Check whether $\sum_{i \in I} s_i \leq S$
3. Check whether $\sum_{i \in I} v_i \geq k$.

Step 1 yields a complexity of $\Omega(2^m)$, i.e., the algorithm is only practicable for very small instances.

We cannot use the above algorithm to show that KNAPSACK is in *P*.

The classes *P* and *NP* (5)

At least we know that KNAPSACK belongs to the complexity class *EXP*:

$$EXP := \{E \text{ decision problem} \mid \text{there exists a } k \in \mathbb{N} \text{ and an algorithm } A \text{ such that } A \text{ solves } E \text{ in time } O(2^{n^k})\}$$

Since every function in $O(n^k)$ is also in $O(2^{n^k})$, we have $P \subseteq EXP$. There is even a proof that problems in *EXP* exist that do not belong to *P*. Whether KNAPSACK belongs to these problems* is an open problem.

Theorem. $P \subsetneq EXP$.

*As we will argue later, most complexity theoreticians think so

The classes *P* and *NP* (6)

Non-deterministic algorithms.

The knowledge that problems such as KNAPSACK belong to the class *EXP* is not very satisfactory, because *EXP* also contains problems whose solution requires exponential running time due to tedious/long-winded calculations.

But KNAPSACK exhibits a special *problem structure* that differs from other problems in *EXP*:

- The size of the solution space is exponential.
  (KNAPSACK: all 2^m subsets are solution candidates.)

- We can check in *polynomial time* whether a solution candidate is feasible.
  (KNAPSACK: we just have to compute the sums

$$\sum_{i \in I} s_i \quad \text{and} \quad \sum_{i \in I} v_i$$

  and compare them to S and k.)

The classes *P* and *NP* (7)

The reason for the complexity of KNAPSACK are thus not expensive calculations but the difficulty of finding a solution in the exponentially large solution space.

We classify problems that exhibit such a structure by establishing the complexity class *NP*.

The problem structure "*large solution space* / *easy candidate test*" suggests to search the solution space *in parallel*.

$\Rightarrow$ If we were able to check all candidates $I \subseteq \{1, \ldots, m\}$ in parallel, we could solve KNAPSACK in in polynomial time.

The classes *P* and *NP* (8)

To be able to do parallel calculations in our pseudocode model, we add the possibility of *non-deterministic branching*:

begin $\text{expr}_1 \mid \text{expr}_2 \mid \cdots \mid \text{expr}_k$ **end**

Executing such a statement splits the computation path in k independent computation paths, which cannot communicate with each other. Each such path computes an independent result.

Non-deterministic branching changes the traditional linear computation path into a *computation tree*. We can use this tree to generate and check candidate solutions.

The classes *P* and *NP* (9)

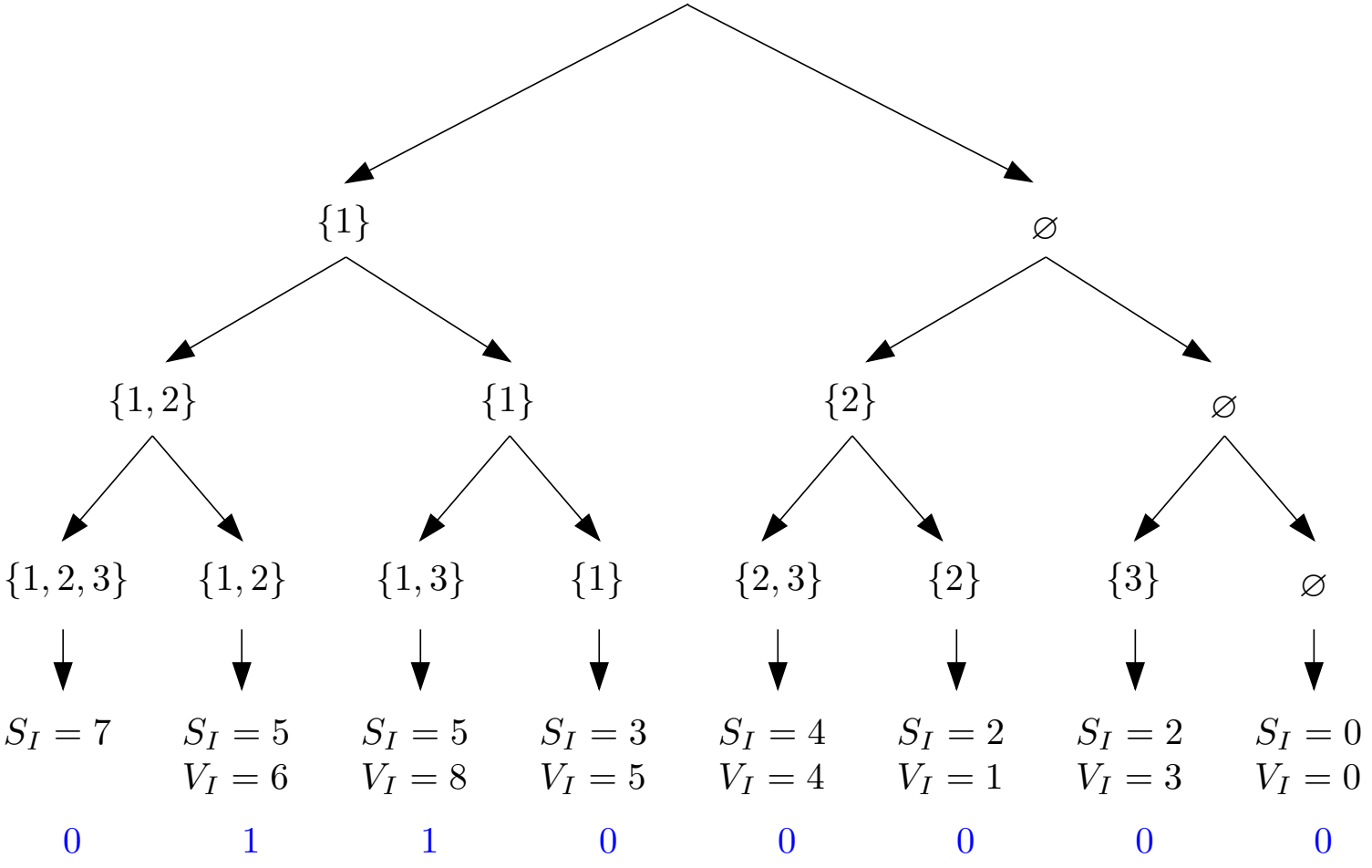
Example.

algorithm: *non-deterministic knapsack*
input: $m, s_1, \ldots, s_m, v_1, \ldots, v_m, S, k \in \mathbb{N}$

$I := \emptyset$;
for $i :=$ 1 to m **do**
  begin $I := I \cup \{i\} \mid I := I$ **end**; // non-deterministic branching
if $(\sum_{i \in I} s_i \leq S)$ & $(\sum_{i \in I} v_i \geq k)$ **return** 1; // check candidates
return 0;

The classes *P* and *NP* (10)

Let's look at the instance $m := 3$, $s := (3, 2, 2)$, $v := (5, 1, 3)$, $S := 6$, $k := 6$.



By means of the non-deterministic branching operator we can adequately capture the problem structure of KNAPSACK.

The classes *P* and *NP* (11)

Non-deterministic computations yield a 1-answer if and only if *at least one* of the computation paths yields a 1-answer.

Remark. Non-determinism is a *theoretical model*, and currently no computer can realize this model for realistic applications. This is due to the fact that the number of independent computation paths grows exponentially with the input size. In a computer, however, the number of parallel processes or processors is bounded by a constant.

Example. One million processors are not sufficient to simulate all independent computation paths for the KNAPSACK problem for an instance with only 20 items.

The classes *P* and *NP* (12)

$$NP := \{E \text{ decision problem} \mid \text{there exists a } k \in \mathbb{N} \text{ such that a } \textit{non-deterministic} \text{ algorithm } A \text{ solves } E \text{ in time } O(n^k)\}$$

The definition of the class *NP* (= **N**on-deterministic **P**olynomial time) captures the following aspects:

- We allow non-determinism by adding the non-deterministic branching operator to the pseudocode instruction set.
- The running time has to be polynomially bounded on each path of the computation tree.
- The whole computation returns 1 if and only if one path yields a 1-answer.

We have already seen that KNAPSACK $\in$ *NP*.

The classes *P* and *NP* (13)

We can characterize *NP* also in terms of languages:

Theorem. Let $L \subseteq \Sigma^*$ be a language. $L \in$ *NP* if and only if there is a polynomial-time decidable relation R such that

$$L = \{x \mid (x, y) \in R \text{ for some } y \text{ and } |y| \leq |x|^k\} \ .$$

Merkregel: Problems in P are *"easy to solve"*, problems in *NP* are *"easy to check"*.

Proof sketch. ($\Leftarrow$) Assume such an R exists. Then L can be decided by a non-deterministic algorithm that, given an input x, "guesses" a y of length at most $|x|^k$ and then decides whether $(x, y) \in R$ in polynomial time.

The forward direction ($\Rightarrow$) of the proof is similar.

A detailed proof is part of the assignments (Ex. 7.3).

The classes *P* and *NP* (14)

The class *NP* contains all problems with the mentioned structure:

- the size of the solution space is bounded by $2^{p(n)}$ (for a polynomial p)
- checking a candidate needs polynomial time

The class is very interesting since the above characterization comprises many practically relevant problems.

Theorem. $P \subseteq NP \subseteq EXP$.

Proof. This is easy to see: Every deterministic polynomial-time algorithm is also a non-deterministic polynomial time algorithm.

On the other hand we can fully deterministically simulate a computation tree of polynomial depth in time $O(2^{p(n)})$ (e.g., with DFS) to check whether one of the leaves returns a 1-answer.

The classes *P* and *NP* (15)

The question to find an efficient algorithm for KNAPSACK and many other problems in *NP* can be formulated shortly as

> ***P-NP* question**: Does $P = NP$ hold or does $P \neq NP$ hold?*

This is one of the seven *millenium problems* identified by the Clay Mathematics Institute. There is a video (see Section "Links" on the class website) with a very nice lecture by Vijaya Ramachandran on this problem.

*proven answer worth 10^6 $

Reductions

$\Rightarrow$ Comparing problems with respect to their complexity

The concept of *reducability* captures the following intuitive idea:

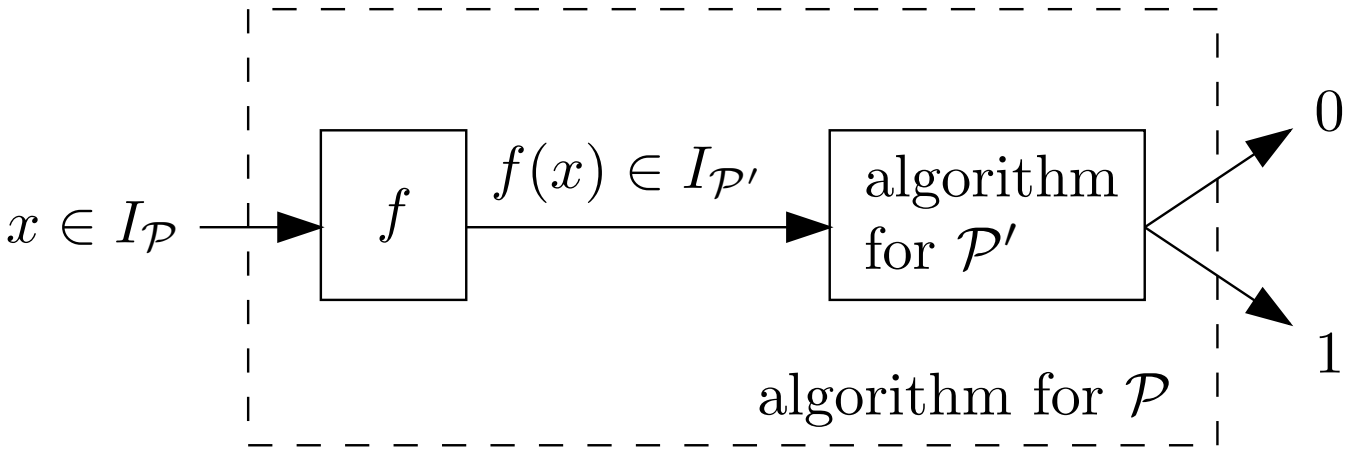
> If we can solve a problem $\mathcal{P}$ by means of an algorithm for a problem $\mathcal{P}'$ **without "significant" additional effort**, we call $\mathcal{P}$ *reducible to* $\mathcal{P}'$.

Thus, we relate and compare problems $\mathcal{P}$ and $\mathcal{P}'$ with respect to their algorithmic tractability. An algorithm A for $\mathcal{P}$ can use an algorithm A' for $\mathcal{P}'$ as a subroutine.

If $\mathcal{P}$ is reducible to $\mathcal{P}'$, it cannot be *algorithmically harder* than $\mathcal{P}'$, which we express by writing $\mathcal{P} \leq \mathcal{P}'$.

Reductions (2)

Let $\mathcal{P}$ and $\mathcal{P}'$ be two decision problems. The following picture illustrates the relation $\mathcal{P} \leq \mathcal{P}'$:



Definition. Problem $\mathcal{P}$ is *reducible* to $\mathcal{P}'$ if there exists a *polynomial-time* function $f : I_{\mathcal{P}} \mapsto I_{\mathcal{P}'}$ with the property

$$x \in \mathcal{P} \longleftrightarrow f(x) \in \mathcal{P}' \ .$$

Reductions (3)

To show $\mathcal{P} \leq \mathcal{P}'$ we have to find a suitable *reduction* f and show

$$x \in \mathcal{P} \longleftrightarrow f(x) \in \mathcal{P}' \ .$$

- Note that we do not necessarily need to know an algorithm for $\mathcal{P}'$ to show $\mathcal{P} \leq \mathcal{P}'$.

- If, however, this is the case (now or later), we immediately get an algorithm for $\mathcal{P}$ (with a bit of* overhead).

Lemma. The relation $\leq$ is reflexive and transitive.

Proof. Blackboard.

*polynomial

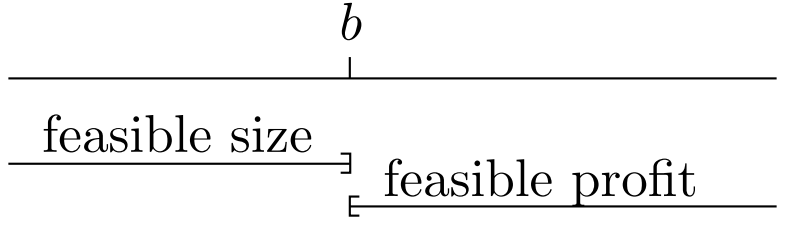
Reductions (4)

Example. Let $\mathcal{P}$ = SUM OF SUBSETS and $\mathcal{P}'$ = KNAPSACK.

problem: SUM OF SUBSETS
input: $a_1, \ldots, a_m, b \in \mathbb{N}$
question: Does an $I \subseteq \{1, \ldots, m\}$ exists with $\sum_{i \in I} a_i = b$?

Let $x = (a_1, \ldots, a_m, b) \in I_{\mathcal{P}}$. How can we *decide* this instance of SUM OF SUBSETS with an algorithm for KNAPSACK?

Idea: Treat every number $a_i \in \mathbb{N}$ as an item of size a_i and profit a_i. The algorithm for KNAPSACK should choose a subset of items (numbers) whose values sum up *exactly* to b. The trick is to split the equality into two inequalities. Thus, we require the sizes of the choice to be *at most* b and the profits to be *at least* b.



Reductions (5)

Formally, we define the reduction f as

$$f(a_1, \ldots, a_m, b) := (a_1, \ldots, a_m, a_1, \ldots, a_m, b, b).$$

Obviously, f runs in polynomial time since we just have to copy the input. Now we show (blackboard) the property

$$x \in \text{SUM OF SUBSETS} \longleftrightarrow f(x) \in \text{KNAPSACK} \ .$$

Reductions (6)

The example shows

- If we have an *efficient* algorithm for KNAPSACK, we can use it to solve SUM OF SUBSETS efficiently.
- Vice versa: If we can prove that is impossible to solve SUM OF SUBSETS efficiently this must also hold for KNAPSACK.

This is a *general* observation:

Theorem. (proof omitted, it is easy)

1. If $\mathcal{P} \leq \mathcal{P}'$ and $\mathcal{P}' \in P$ then $\mathcal{P} \in P$.
2. If $\mathcal{P} \leq \mathcal{P}'$ and $\mathcal{P}' \in NP$ then $\mathcal{P} \in NP$.

Another way to put this is to say that P and NP are *closed* under reduction.

NP-completeness

Using reductions we can compare all problems in *NP*. The following natural question arises:

> Does *NP* contain “hardest” problems? That is, problems, to which *all* other problems in *NP* are reducible.

These are extremely interesting with respect to the *P*-*NP*-question.

Definition. A decision problem $\mathcal{P}$ is *NP-complete*, if

1. $\mathcal{P} \in NP$ and
2. for all $\mathcal{P}' \in NP$ holds $\mathcal{P}' \leq \mathcal{P}$.

NP-completeness (2)

Thus, an algorithm for an *NP*-complete problem $\mathcal{P}$ can solve every other problem in *NP* without significant overhead.

Such a problem $\mathcal{P}$ can be seen as a representative for the whole class *NP* in terms of complexity.

In particular: A *polynomial time* algorithm only for one single *NP*-complete problem leads to an efficient algorithm for every problem in *NP*, and we have $P = NP$.

Theorem. Let $\mathcal{P}$ be an *NP*-complete problem. If $\mathcal{P} \in P$, then also $P = NP$.

Proof. Blackboard

NP-completeness (3)

problem: SAT
input: Boolean expression ϕ over variables $x_1, \ldots, x_n$ in conjunctive normal form, e.g., $\big((x_1 \vee \neg x_2) \wedge \neg x_1\big)$
question: Is ϕ *satisfiable*?

Theorem (Cook, 1970). SAT is *NP*-complete.

Rough proof sketch: blackboard.

Not every *NP*-completeness proof has to be this ingenuous. In most cases, the best way to show *NP*-completeness of a problem is to use reductions.

Let $\mathcal{P}$ be an *NP*-complete problem. If we can show

$$\mathcal{P} \leq \mathcal{P}' \quad \text{and} \quad \mathcal{P}' \in NP$$

then $\mathcal{P}'$ is *NP*-complete as well.*

*why?

Complexity of MSA

And what about multiple sequence alignment (MSA)?

Theorem. MSA is *NP*-complete.

Proof.

(a) MSA $\in$ *NP*: Ex. 7.4.

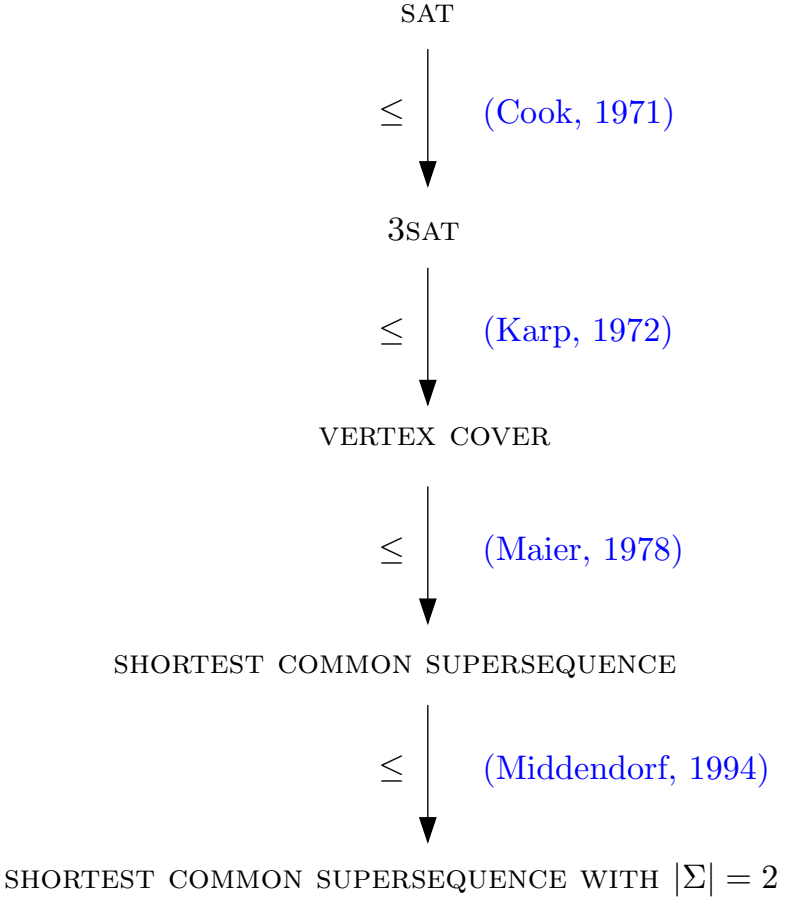
(b) Reduction from the following problem:

problem: SHORTEST COMMON SUPERSEQUENCE WITH $|\Sigma| = 2$ (SCS_2)

input: Finite set S of sequences over alphabet Σ with $|\Sigma| = 2$ and positive integer m.

question: Is there a sequence s with $|s| \leq m$ such that each $t \in S$ is a subsequence of s?

Remainder of proof: blackboard.



NP-hardness

A language/problem L' is *NP*-hard if $\forall L \in NP : L \leq L'$. This class contains thus hard problems that do not even have to be in *NP*.

A famous example of an *NP*-hard but not *NP*-complete* problem is the *halting problem* $\mathcal{H}$ "given a program and its input, will it terminate?".

E.g., SAT $\leq \mathcal{H}$, because we can take the non-deterministic code for SAT and modify it in such a way that we

- output 1 if the input formula is satisfiable and
- go into an infinite loop otherwise.

The term *NP*-hardness is often extend to the corresponding constructive or evaluation problems of a decision problem (in the sense of Ex. 6.3)

*Remark: $\mathcal{H} \neq NP$, because all problems in *NP* are *decidable*, but the halting problem is not.