Introduction to Computational Complexity (Lecture Notes for a 5-day Graduate Course)

Martin Stigge, martin.stigge@it.uu.se
Uppsala University, Sweden

July 9, 2009

Contents

1 Introduction **2**
1.1 About this document 2
1.2 What is Computational Complexity? 2

2 Basic Computability Theory **4**
2.1 Problems as Formal Languages 4
2.2 Model of Computation: Turing Machines 5
2.3 Decidability, Undecidability, Semi-Decidability 11
2.4 The Halting Problem 12
2.5 More Undecidability 14

3 Complexity Classes **19**
3.1 Landau Symbols: The $\mathcal{O}(\cdot)$ Notation 19
3.2 Time and Space Complexity 20
3.3 Relations between Complexity Classes 23

4 Feasible Computations: P vs. NP **27**
4.1 Proving vs. Verifying 27
4.2 Reductions, Hardness, Completeness 31
4.3 Natural NP-complete problems 34
4.4 Beyond NP-completeness 44

5 Advanced Complexity Concepts **47**
5.1 Non-uniform Complexity 47
5.2 Probabilistic Complexity Classes 49
5.3 Interactive Proof Systems 55

A Appendix **60**
A.1 Exercises 60
A.2 Further reading 61

1 Introduction

1.1 About this document

These lecture notes are part of a graduate course given in 2009 at Northeastern University in Shenyang, China. The course is directed at graduate students in the field of computer science or engineering who want to learn more about theoretic models of computation and complexity. The material was planned to be covered during 5 days with 2 lectures every day. It is supposed to be mostly self-contained, assuming only basic knowledge about formal notations used in mathematics and theoretical computer science. All more advanced concepts are defined and explained in detail.

In preparation for the course, lecture notes from Viggo Stoltenberg-Hansen (Uppsala University, Sweden), Oded Goldreich (Weizmann Institute of Science, Israel) and Johannes Köbler (Humboldt University Berlin, Germany) were sources of inspiration, as well as the books on Computational Complexity by Christos Papadimitriou [Pap94] and Daniel Bovet / Pierluigi Crescenzi [BC93], who should all be credited explicitly.

1.2 What is Computational Complexity?

Complexity theory addresses the study of the *intrinsic complexity* of computational tasks. The "ultimate goal" is to judge for every well-defined task, how complex it is to solve it, i.e., how many resources and how much time is needed – at most and at least – to perform the given task. While this can be seen as an *absolute* answer, also *relative* answers are interesting: How do different tasks relate to each other in their complexity? Are some always more difficult than others, no matter how we model them? How do time and resources relate to each other? Are there "most difficult" problems? To study these questions, the first step is to have a precise understanding of what it means to perform a task, to solve a problem.

This basis is provided by a field that is very closely connected and is called *computability theory*. It studies, which functions are *algorithmically computable*, i.e., given a reasonable model of computation, what can (at all) algorithmically be computed? How do we model computation? What is an algorithm? And what does it mean to "solve" a problem? Computability theory doesn't just provide the formal and mathematically rigorous foundations for related areas like complexity theory, but it provides results that are very interesting on their own. A prominent example is the existence of properties of programs that are *impossible* to verify algorithmically. We explore some of these results together with the necessary foundations in Chapter 2.

Based on these foundations, we turn to a general introduction to complexity theory in Chapter 3. The main incentive of this chapter is to introduce the reader to classical notions of complexity classes, both concerning restrictions of time and space for computations. We study how space

and time restrictions relate to each other, as well as the relation between determinism and non-determinism. The latter one concerns one of the fundamental questions of theoretical computer science: Is it more difficult to come up with a solution than to verify the validity of a given solution? This question manifests itself in the famous $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$ problem and demonstrates the failure of complexity theory to provide "absolute" answers, even though at the same time, it created strong and important frameworks for results of the "relative" kind.

This question is so fundamental and its answer so surprisingly difficult, that we study it to a deeper extend in Chapter 4. We provide alternative characterizations of the problem and introduce some classical examples to illustrate the fundamental nature of this specific problem. The perhaps most insightful result is concerned with the notion of *completeness*, which expresses the existence of "most difficult" problems. We conclude the chapter with an overview of prominent NP-complete problems, which are of importance for most areas of computer science, both in theory and in practice.

In Chapter 5, we conclude this course with a short tour through three advanced concepts that are of great interest in complexity theory: Non-uniformity, randomization and interaction. We provide some basic overview, but don't go too much into detail.

The document further contains an appendix with a list of exercises that should invite the interested reader to hands-on application of the presented knowledge. The appendix also includes selected material for deepening the knowledge in all areas presented during the course.

2 Basic Computability Theory

We start with an introduction into computability theory. The goal of this area is to find the borders of what can be algorithmically done, i.e., by some machine or someone following a precise description of a task, and what can actually not be done. To study the questions of interest in a formal way, it is necessary to first define formally what we actually mean by a "problem" and what a "computation" actually is.

2.1 Problems as Formal Languages

At the most abstract level, the setting we want to work with can be described as depicted in Figure 1: A *machine* gets a problem instance as *input* and will produce a solution at its *output*. Thus, there must be some kind of agreement about the format of input and output. In theoretical computer science, one uses the simple but powerful concept of *formal languages*.

Figure 1: Most abstract modelling level

The most basic ingredient is a finite set $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$ of symbols, the *alphabet*. A *word* w over Σ is a tuple $w = (w_1, \ldots, w_l)$ of symbols from Σ, i.e., $\forall i : w_i \in \Sigma$. We denote the *length* l of the word w with $|w|$ and the (unique) *empty word* with ε. The set of words of length n is denoted with Σ^n while $\Sigma^* := \bigcup_{n \geq 0} \Sigma^n$ is the set of all (finite) words. A formal *language* L over Σ is a set of words from Σ^*, i.e., $L \subseteq \Sigma^*$. Usual set operations can be applied to languages L_1 and L_2 (over Σ_1 and Σ_2), like union ($L_1 \cup L_2$), intersection ($L_1 \cap L_2$) and difference ($L_1 - L_2$). Additionally, we write L_1L_2 for the concatenation L_1 and L_2, containing all words w having a prefix w_1 from L_1 and the corresponding suffix w_2 from L_2:

$$L_1L_2 := \{w \mid \exists w_1 \in L_1, w_2 \in L_2 : w = w_1w_2\}$$

Note that the results of all operations are languages over $\Sigma_1 \cup \Sigma_2$.

We can now use this concept in the above setting: Input and output of the machine will be encoded into words over Σ. We consider two examples.

Example 2.1. *Let* $\Sigma := \{0, 1, \ldots, 9\}$*, and let* $[n]_{10}$ *denote the decimal representation of the natural number* n*. Then* $\mathsf{NAT} := \{[n]_{10} \mid n \in \mathbb{N}\}$ *represents all natural numbers. A machine that calculates the square of a number* n *gets* $w := [n]_{10} \in \Sigma^*$ *as input and will output a word* $v \in \Sigma^*$ *with* $v = [n^2]_{10}$*.*

Example 2.2. *As a second example, take* Σ *as before, and let* $\mathsf{PRIMES} := \{[p]_{10} \mid p \textit{ is a prime number}\}$. *Obviously*[1], $\mathsf{PRIMES} \subsetneq \mathsf{NAT}$. *A machine to check the primality of a number* n *will get* $[n]_{10}$ *as an input and will output* 1 *if* n *is prime, and* 0 *otherwise. This is our first example of a* decision problem. *The set* PRIMES *is the set of* positive instances *of the problem, out of the set* NAT *of all possible inputs. The machine is supposed to correctly distinguish positive from negative instances.*

2.2 Model of Computation: Turing Machines

After defining the format of input and output, we also need to develop a formal notion of a "machine". It should be a model that captures what we perceive as being "algorithmically computable" by a finite number of actions. But what exactly does this mean? What are actions? Over time, theoretical computer scientists (in former times still being called mathematicians), have developed many distinct models of computation. Recursive functions, rewriting systems and Turing machines are only a few prominent examples for such *unrestricted models of computation.*

It turns out that in spite of many efforts, researchers were not able to come up with models that are more expressive than these. In fact, all proposed models have been proven to be equivalent in terms of expressiveness. Even though it has not been proven that a more powerful model can not exist, it is widely believed that whatever problems are "solvable", they can be solved using any of the mentioned formalisms. This popular conjecture is called *Church's Thesis.*

The model we will focus on is the *Turing machine.* For computability theory as well as complexity theory, this model has been adopted as the standard model because of its simplicity. It can be seen as a primitive version of an idealised computer and consists of the following (c.f. Figure 2):

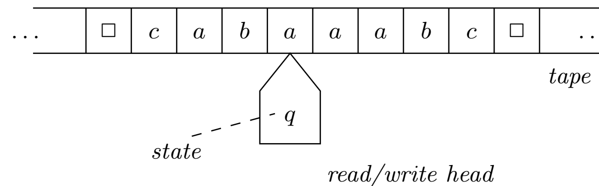


Figure 2: A Turing Machine

- A *tape* used for input, storage and output. The tape consists of squares, each with the capability to store one symbol from a working alphabet Γ. The tape is not bounded a priori.

[1]With $A \subsetneq B$ we denote strict inclusion: $A \subseteq B \wedge\ A \neq B$.

- A read/write *head* that can move stepwise across the tape, changing its symbols one at a time.
- A finite state machine. The set of states includes a designated initial state q_0 and a set of final states F.

At the beginning of the machines operation, it is in state q_0 and the tape contains an input word over an alphabet Σ. (Thus, $\Sigma \subsetneq \Gamma$.) All other positions of the tape contain a special symbol $\Box \in \Gamma$ called the *blank symbol*.[2] The machine proceeds step-wise. During each step, it reads the symbol at the current position of the tape. Depending on that symbol and the current state, the machine writes a new symbol, changes its state, and does a head move by at most one position in either direction. This procedure is repeated until the machine enters one of the final states in F. The machine then simply stops, and the contents of the tape are the output of the machine.

We now give a formal definition.

Definition 2.3 (Turing Machine)**.** *A Turing machine M is a five-tuple $M = (Q, \Gamma, \delta, q_0, F)$ where*

Q *is a finite set of* states*;*

Γ *is the* tape alphabet *including the blank:* $\Box \in \Gamma$*;*

q_0 *is the* initial state*,* $q_0 \in Q$*;*

F *is the* set of final states*,* $F \subseteq Q$*;*

δ *is the* transition function*,* $\delta : (Q - F) \times \Gamma \to Q \times \Gamma \times \{R, N, L\}$.

The transition function δ takes a non-final state and a tape symbol as arguments. Given those, it returns the action that the Turing machine will execute when this combination is encountered: the new state, the tape symbol to be written, and the movement of the head, where R, N and L symbolize "right", "no move" and "left", respectively. Note that by this definition, a machine can never "get stuck" in a non-final state, i.e., it can always execute another step.

We would also like to formally capture the global status of a Turing machine at each computation step, i.e., the state together with the contents of its tape. We call this a *configuration* of the machine and express it as a tuple $(w, q, v) \in \Gamma^* \times q \times \Gamma^*$. This is to be read as: The tape contains wv with infinitely many $\Box$ to the left and right of it; the machine's state is q and the head is over the first symbol of v. (If $v = \varepsilon$ then the head is on the first position to the right of w.) See Figure 3.

The start configuration of a machine M with an input word w is thus (ε, q_0, w) and the execution halts as soon as a configuration (v, q, z) is reached

[2] In fact, at any time, all but finitely many positions of the tape will be blank.

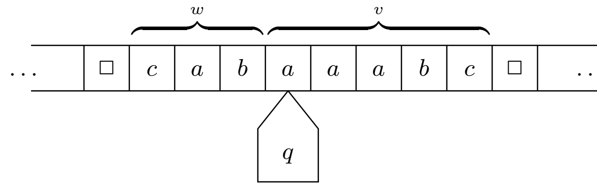


Figure 3: Turing machine configuration (w, q, v) with $w = cab$ and $v = aaabc$

with arbitrary $v, z \in \Gamma^*$ and $q \in F$. By convention, we let z be the output of M, which we also denote[3] with $M(w)$. In case M does not halt given input w, we write $M(w) = \nearrow$.

Using this notation of configurations, we may also define the *step relation* formally, i.e., the execution of one step of the machine.

Definition 2.4 (Step Relation)**.** *Given a Turing machine* $M = (Q, \Gamma, \delta, q_0, F)$*, we define* $\vdash$ *for all* $w, v \in \Gamma^*, a, b \in \Gamma$ *and* $q \in Q$ *as:*

$$(wa, q, bv) \vdash \begin{cases} (wac, p, v) & \text{if } \delta(q, b) = (p, c, R), \\ (wa, p, cv) & \text{if } \delta(q, b) = (p, c, N), \\ (w, p, acv) & \text{if } \delta(q, b) = (p, c, L). \end{cases}$$

This just formalizes the intuitive explanation of the TM's execution. We further use $\vdash^n$ to relate configurations reachable in n steps (where $n = 0$ is possible), and $\vdash^*$ for all reachable configurations.

2.2.1 The Universal Machine

We have seen that the introduced model of a Turing machine is relatively simple in the sense that it contains simple rules which – in principle – could be easily simulated by a human, given enough pencils, tape space and patience. One of the fundamental results (originally by Alan Turing) is, that there are even Turing machines that can simulate *all* other Turing machines. In a modern view, this could be regarded as the existence of a general purpose computer.

The key idea is that Turing machines are finite objects. Thus, using an appropriate coding, each Turing machine can be effectively encoded into a finite word over some fixed alphabet (just like the binary encoding of computer programs in modern computers). Also all reachable configurations are finite words. Therefore, without going too much into detail, one can imagine a machine U that gets the encoding of another machine M as an input, together with a word w that is supposed to be the input of M. It

[3]We also sometimes just write $M(z)$ to denote the computation of M on input z, but this won't introduce confusion since it will be clear from the context.

then proceeds by simulating M with input w by maintaining an encoded version of M's configuration and carefully applying each step of M.

Using $\langle M \rangle$ to denote the encoding of a Turing machine M, we can state this result as the following theorem.

Theorem 2.5 (The Universal Machine)**.** *There exists a universal Turing machine U, such that for all Turing machines M (with the same input/output alphabet Σ) and all words $w \in \Sigma^*$ the following holds:*

$$U(\langle M \rangle, w) = M(w)$$

In particular, U does not halt iff[4] *M does not halt.*

(Without proof.)

2.2.2 Extensions

We will now take a look at alternative definitions and extensions of the Turing machine model.

Transducers and Acceptors: The model as defined above gets a word as input and produces a word as output. We can therefore interpret a Turing Machine M as a function from words to words over the alphabet Σ and call it a *transducer*. A function for which such a transducer exist, is called a *computable function*. Note that for some input words, the transducer may not terminate. In that case, the function is *undefined* for these arguments and is thus a *partial function* (as opposed to a *total function* if it is defined for all arguments).

On the other hand, if we are interested in decision problems, i.e., whether a certain input word w belongs to a decision problem L or not, then it is sufficient to consider two possible outputs: a positive (in case $w \in L$) and a negative (in case $w \notin L$). Equivalently, one often defines Turing machines for this scenario as having exactly two final states: q_{yes} and q_{no}. With this definition, the tape contents after the machine halts doesn't matter – the answer is represented by the final state. Such a Turing machine is called an *acceptor*, see Figure 4, and we denote the language accepted by M with $L(M)$:

$$L(M) := \{w \in \Sigma^* \mid \exists y, z \in \Gamma^* : (\varepsilon, q_0, w) \vdash^* (y, q_{yes}, z)$$

For the rest of the course, we will mostly deal with acceptors and decision problems.

[4] "if and only if"

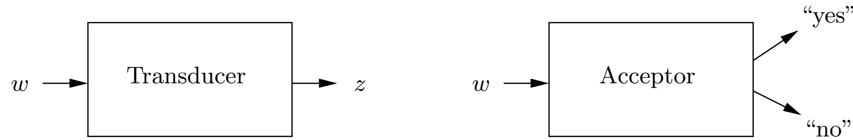


Figure 4: Transducers versus Acceptors

Multiple Tapes: We defined Turing machines with *one* tape. Sometimes the model is defined using k tapes (for any k), each one with its own tape head. It is also common to use one of them as a designated *input tape*, which is read-only and contains the input of the machine, and another one as a designated *output tape*, which is only allowed to be used for writing down the result right before halting and will then contain the output of the machine. See Figure 5.

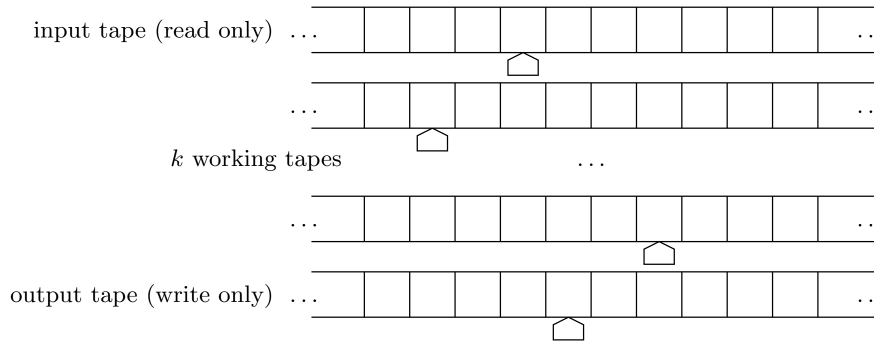


Figure 5: A Turing Machine with multiple tapes

This model is equivalent to the one we defined, in the sense that a 1-tape TM for a problem exists if and only a k-tape TM exists. That can be easily shown by encoding a "column" of tape squares from all k tapes into just one square of the 1-tape TM and using markers for the k tape heads. (Note that k is a *fixed* number and thus can not depend on the input size!) One advantage of using the extended model is that one can express certain actions more easily, like "remember this string on an extra tape before proceeding"). Another advantage is, that its easier to specify exactly, how much extra space the machine uses, apart from input and output space. This is important when we later define space-restricted complexity classes, that find their applications in modelling of memory-restricted setting.

Non-determinism: Our Turing machine model in Definition 2.3 is *deterministic*. This means that each configuration has exactly one possibility for the next step, which directly defines the next configuration. An extension of this model is to introduce *non-determinism*. At each step, the machine

is allowed to choose between different alternatives of what to do next. For that, the transition function δ doesn't just return one tuple representing the next step, but a *set* of possible steps:

$$\delta : (Q - F) \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{R, N, L\})$$

We call such a machine a *Non-deterministic Turing machine* (NTM). Using this definition, the *step relation* can be defined accordingly:

$$\begin{aligned}(wa, q, bv) &\vdash (wac, p, v) \text{ for all } (p, c, R) \in \delta(q, b),\\ (wa, q, bv) &\vdash (wa, p, cv) \text{ for all } (p, c, N) \in \delta(q, b),\\ (wa, q, bv) &\vdash (w, p, acv) \text{ for all } (p, c, L) \in \delta(q, b).\end{aligned}$$

The question remains, how a computation of such a machine actually looks like. Note that this is a theoretical model which is not supposed to resemble the behaviour of a real machine. One can imagine the computation as a *tree* in which the possible configurations are the nodes and the start configuration is the root, see Figure 6. (For a deterministic machine, this tree is just a line.)

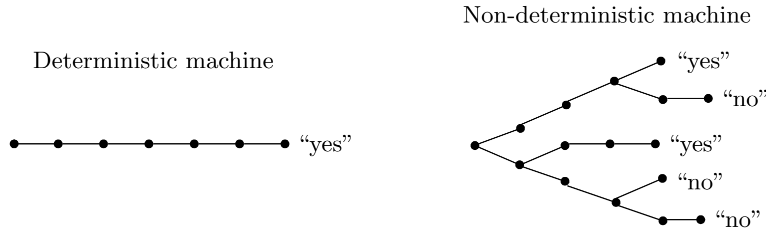


Figure 6: Computation trees of deterministic and non-deterministic computations

We only define the behaviour of a non-deterministic *acceptor*, i.e., a machine that either accepts or rejects a word. We define that a word w is accepted by a non-deterministic TM M, if there is a finite computation path in the computation tree leading to the accepting state q_{yes}. Thus, the accepted language is defined as before:

$$L(M) := \{w \in \Sigma^* \mid \exists y, z \in \Gamma^* : (\varepsilon, q_0, w) \vdash^* (y, q_{yes}, z)\}$$

It might be somewhat surprising, that also this model is still equivalent to the original Definition 2.3 in terms of expressiveness:

Theorem 2.6. *Given a non-deterministic Turing machine N, one can construct a deterministic Turing machine M with $L(M) = L(N)$. Further, if $t(w)$ is the number of steps after which N accepts an input w, there is a constant c such that M accepts w after at most $c^{t(w)}$ steps.*

Proof (Sketch). Given an input word w, we need to deterministically search the computation tree of N. This can be done using a breadth-first technique: for each $i \geq 0$, we visit all configurations of the tree up to depth i. If N accepts w, we will eventually find an accepting configuration at depth t. If d is the maximal degree of non-determinism (i.e., the maximal number of choices δ provides at each step), then the procedure will take at most $\sum_{i=0}^{t} d^i$ steps, which can be bounded from above by c^t with a suitable constant c. □

Thus, we saw that a non-deterministic Turing machine can always be simulated by a deterministic one, even though the number of steps might be exponentially higher. We note further, that if the computation tree of N on w is finite, then the procedure will terminate even if $w \notin L(N)$.

Unless stated otherwise, we will use deterministic Turing machines throughout this course.

2.3 Decidability, Undecidability, Semi-Decidability

In the previous section, we defined an acceptor M for a language L to have two final states, q_{yes} and q_{no}, and we demanded M for each input $w \in L$ to end up in the final state q_{yes}. So far we did not restrict what should happen in the case $w \notin L$. The machine M could either halt in q_{no} or not halt at all. As we will see now, this distinction actually makes an important difference.

Definition 2.7. *A language L is called* decidable*, if there exists a Turing machine M with $L(M) = L$ that* halts on all inputs*.*

In particular, we want M to halt in state q_{yes} for each input $w \in L$ and halt in state q_{no} for each input $w \notin L$. We also say M *decides* L. If L is not decidable, we call it *undecidable*. The set of all decidable languages is denoted by REC ("recursive languages", for historic reasons).

Definition 2.8. *A language L is called* semi-decidable*, if there exists a Turing machine M with $L(M) = L$.*

In particular, M needs to halt on all inputs $w \in L$ in state q_{yes}, but on inputs $w \notin L$ it can either halt in state q_{no} or not halt at all. The set of all semi-decidable[5] languages is called RE ("recursively enumerable", for historic reasons).

Example 2.9. *Recall the language* PRIMES *of all prime numbers as defined in Example 2.2. It is easy to see that* PRIMES *is decidable, since given $w = [n]_{10}$ for some n, a Turing machine M just needs to check for all numbers i with $1 < i < n$ whether n is a multiple of i. If this is the case for any i, it halts in q_{no}, otherwise in q_{yes}. Thus,* $\mathsf{PRIMES} \in \mathsf{REC}$.

[5] The name "semi-decidable" is based on the fact, that for positive instances $w \in L$, on can let M run and will eventually get a positive answer. For $w \notin L$, halting is not guaranteed, so as long as M does not halt, one does actually not know, whether w is a positive instance or not – and one might stay in this unfortunate situation infinitely.

It is immediately clear that every decidable language is also semi-decidable, i.e, $\mathsf{REC} \subseteq \mathsf{RE}$, just by definition from above. An interesting question is, whether there are languages that are semi-decidable, but not decidable. If that is the case, the subtle difference regarding the termination actually makes a difference, and we will turn to that question in the following section. Before that, we will take a look at some useful properties of the complement $\overline{L} := \Sigma^* - L$ of a language L:

Theorem 2.10. *For all languages L, the following holds:*

1. $L \in \mathsf{REC} \iff \overline{L} \in \mathsf{REC}$. *("closed under taking complements")*
2. $L \in \mathsf{REC} \iff (L \in \mathsf{RE} \wedge \overline{L} \in \mathsf{RE})$.

Proof. The first equivalence is clear, since given a Turing machine M that decides L, one can derive a machine M' deciding $\overline{L}$ by just swapping q_{yes} and q_{no}.

For the second equivalence, direction "$\Longrightarrow$" follows from $\mathsf{REC} \subseteq \mathsf{RE}$ and the closedness under complement. To prove direction "$\Longleftarrow$", let M_1 and M_2 be Turing machines with $L(M_1) = L$ and $L(M_2) = \overline{L}$. Then we can construct a TM M that, given an input w, simulates M_1 and M_2 in parallel step by step, using two tapes. Since we know that either $w \in L$ or $w \in \overline{L}$, one of the two simulations will eventually end up in an accepting state. Our machine M can then accept or reject, depending on that information. Thus, M halts on all inputs and decides L. □

2.4 The Halting Problem

We have defined decidable and semi-decidable languages, and turn now to the question whether there is actually a difference between both properties. We do that by defining a classical problem, the *Halting Problem*, and show that it is not decidable (even though being semi-decidable). The Halting Problem has great significance from a practical point of view: Given a program and an input string, is it possible in an automatic way to check if that program terminates when running it on that input? In terms of Turing machines, this can be expressed formally:

Definition 2.11 (Halting Problem)**.** *The Halting Problem H is the set of all pairs of encodings $\langle M \rangle$ of Turing machines M and words w, such that M halts on input w:*

$$H := \{(\langle M \rangle, w) \mid M(w) \neq \nearrow\}$$

We will show that the Halting Problem is not decidable, but still semi-decidable:

Theorem 2.12. $H \in \mathsf{RE} - \mathsf{REC}$

Proof. First, it is clear that H is semi-decidable, i.e., $H \in \mathsf{RE}$, since one can just use the Universal Machine from Theorem 2.5 to simulate $M(w)$. If that simulation halts, we accept in state q_{yes}. If it does not halt, we will also not halt, but that is also not necessary.

The more involved part is to show that there is no Turing machine M_H deciding H, i.e., a Turing machine that *always halts* with the right answer. Note that we can not use the Universal Machine here, since in case $M_H(w)$ does not halt, we will never know if that is really the case or if we just did not run the simulation long enough. In general, it is easier to show the existence of machines (buy just constructing one) than to show that none can exist at all – maybe we are just not clever enough to construct one? Thus, proofs are often done indirectly. The proof technique we will use is a standard method in computability and complexity theory, and is called *diagonalization.*

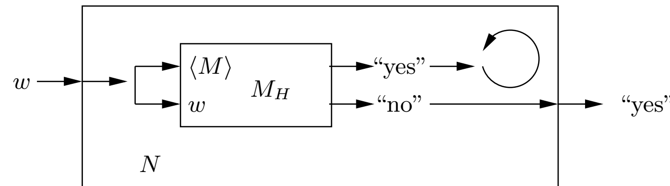


Figure 7: Machine N in the proof of $H \notin \mathsf{REC}$

Let's suppose on the contrary, that a TM M_H exists that decides H. We will use M_H to construct another machine N as follows, see Figure 7: For any input w, N simulates $M_H(w, w)$ of which we know that it will eventually halt in the accepting or rejecting state. If it halts in the accepting state, our machine N will enter an infinite loop and never halt. If, on the other hand, $M_H(w, w)$ halts in the rejecting state, N will halt in q_{yes}. Since N is a Turing machine itself, we can create an encoding $\langle N \rangle$ of it. Now, how does N behave, if it is run with input $\langle N \rangle$, i.e., its own encoding?

- Assume, $N(\langle N \rangle)$ halts. By definition of N, this means that the simulation of $M_H(\langle N \rangle, \langle N \rangle)$ came to halt in the rejecting state of M_H. But this in turn means by definition of M_H, that N with input $\langle N \rangle$ will not halt, contradicting our assumption.

- If we assume otherwise that $N(\langle N \rangle)$ does not halt, then we again know from the definition of N, that the simulation of $M_H(\langle N \rangle, \langle N \rangle)$ came to halt in the accepting state of M_H. By definition of M_H, this means that N with input $\langle N \rangle$ halts, again contradicting the assumption.

Thus, N can not exist, but since we constructed it directly from M_H, also M_H can not exist. □

Finally, we know that there are undecidable languages which are at least semi-decidable. But with a simple counting argument, it is also clear that there are also languages that are not even semi-decidable: Each semi-decidable language represents at least one Turing machine. Since they all can be encoded into strings, there are only countably many Turing machines, i.e., as many as there are natural numbers. On the other hand, there are 2^{Σ^*} languages. This is is uncountably infinite, i.e., as big as the power set of the natural numbers. Set theory tells us that there can be no one-to-one mapping from a set to its power set. Thus, we have the following corollary:

Corollary 2.13. $\mathsf{REC} \subsetneq \mathsf{RE} \subsetneq \mathcal{P}(\Sigma^*)$

Actually, one of the languages that are not even semi-decidable, we already know: Since H, the Halting Problem, is semi-decidable but not decidable, its complement $\overline{H}$ can't be semi-decidable (otherwise, one could decide H, see Theorem 2.10).

Corollary 2.14. $\overline{H} \notin \mathsf{RE}$

2.5 More Undecidability

2.5.1 Reductions

We have seen now that some problems seem to be harder than others. But is it possible to somehow directly compare problems to each other? The main concept for this in computability and complexity theory is the notion of *reductions*. Intuitively, a reduction from a problem A to a problem B expresses, that once one knows how to solve B, one can derive a method for solving A. Thus, a method to solve B induces a method to solve A. In that sense, B is "more difficult" than A, since being able to solve B implies the ability to solve A, so A can't be more difficult.

There are several concepts to capture this intuitive notion formally. The concept we will be using is called *many-one reduction*. It relies on the existence of a total computable function – recall that this is a function that is defined for all arguments and can be computed by a Turing machine.

Definition 2.15 (Many-one Reduction)**.** *A language $A \subseteq \Sigma^*$ is* many-one reducible *to a language $B \subseteq \Sigma^*$, written $A \leq_m B$, if there is a total computable function $f : \Sigma^* \to \Sigma^*$, such that*

$$\forall w \in \Sigma^* : w \in A \iff f(w) \in B$$

We call f the reduction function*.*

As we see from the definition, $A \leq_m B$ if there is a function f that can be computed and that maps all positive instances of A to positive instances of B, as well as negative instances to negative instances. Thus, to decide A

given a machine M_f that computes f and a machine M_B deciding B, one can construct a machine M_A by first simulating M_f and on its output simulating M_B. It follows that decidability and semi-decidability of B transfer to A. We state these and some additional properties of $\leq_m$ in the following lemma.

Lemma 2.16. *For all languages A, B and C the following properties hold:*

1. $A \leq_m B \wedge B \in \mathsf{REC} \Longrightarrow A \in \mathsf{REC}$ *(Closedness of* REC *under $\leq_m$)*
2. $A \leq_m B \wedge B \in \mathsf{RE} \Longrightarrow A \in \mathsf{RE}$ *(Closedness of* RE *under $\leq_m$)*
3. $A \leq_m B \wedge B \leq_m C \Longrightarrow A \leq_m C$ *(Transitivity of $\leq_m$)*
4. $A \leq_m B \iff \overline{A} \leq_m \overline{B}$

Proof. The first two by discussion above, the rest as an exercise. □

These properties can be used to directly show undecidability of languages, based on the undecidability of others: it follows directly that if A is undecidable, then B is also undecidable, provided $A \leq_m B$.

Example 2.17. *As an example, we consider* REACH*, the* reachability problem*:*

$$\mathsf{REACH} := \{(G, u, v) \mid \textit{there is a path from } u \textit{ to } v \textit{ in } G\}$$

An instance consists of a finite directed graph $G = (V, E)$ with V being the set of nodes (vertices) and E the set of edges, $E \subseteq V \times V$, and two nodes $u, v \in V$. (Note that those can easily be encoded over some fixed alphabet Σ.) The positive instances are those triples, where G contains a path from u to v, i.e., there are nodes $v_0, \ldots, v_k \in V$ such that $v_0 = u$, $v_k = v$ and $(v_i, v_{i+1}) \in E$ for all $i = 0, \ldots, i-1$. Using a standard graph search method (like depth-first search), this problem can be easily decided, since the graph G is finite. Thus, $\mathsf{REACH} \in \mathsf{REC}$*.*

The second language for our example is REG-EMPTY*, the* emptiness problem for regular languages*. Recall that regular languages are those decided by a deterministic finite automaton (DFA):*

$$\mathsf{REG\text{-}EMPTY} := \{\langle D \rangle \mid L(D) = \emptyset\}$$

An instance is the description of a DFA. The positive instances are those not accepting any word, i.e., the language they accept is empty.

It is easy to see, that the described emptiness problem can be reduced to the complement of the reachability problem: given the description of a DFA D, one can construct a graph G using the states as nodes, and the state transitions as edges. Further, we introduce a new node q_f with edges from all final states. This construction $\langle D \rangle \mapsto (G, q_0, q_f)$ establishes a reduction, since the accepted language is empty if and only if q_f is not reachable from q_0. Thus, we have $\mathsf{REG\text{-}EMPTY} \leq_m \overline{\mathsf{REACH}}$*, and it follows* $\mathsf{REG\text{-}EMPTY} \in \mathsf{REC}$ *from* $\mathsf{REACH} \in \mathsf{REC}$ *and closedness under complement.*

We take as a second and more elaborate example the *Halting Problem with empty input*. It is the set of all Turing machines, that don't halt when they start running with empty input. This seems like a special case of the general Halting Problem and might thus be easier to solve: if the general case is not decidable, then maybe there is still hope for the special case? Unfortunately, this will turn out to be a negative result, since the general case can be reduced to the special case.

Lemma 2.18. *Let H_ε be the Halting Problem with empty input, i.e.:*

$$H_\varepsilon := \{\langle M \rangle \mid M(\varepsilon) \neq \nearrow\}$$

Then $H_\varepsilon \notin \mathsf{REC}$.

Proof. We already know that $H \notin \mathsf{REC}$. If we can find a reduction $H \leq_m H_\varepsilon$, then it follows from Lemma 2.16 that $H_\varepsilon \notin \mathsf{REC}$. We will now describe the reduction function f. It takes an instance of H as input and is supposed to output an instance of H_ε, i.e.,

$$f : (\langle M \rangle, w) \mapsto \langle M' \rangle.$$

Given the pair $(\langle M \rangle, w)$ we must describe how to effectively derive a TM M' such that $M'(\varepsilon)$ halts if and only if $M(w)$ halts. This can be easily done as follows: M' ignores its input and writes w onto the tape, which is stored in its "program", i.e., in its transition function δ'. Note that w is finite and fixed, so it can be "hard-wired" in δ' beforehand. After writing w, M' proceeds by simulating M. See Figure 8.

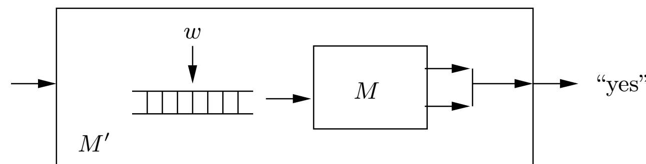


Figure 8: Machine M' in the proof of $H_\varepsilon \notin \mathsf{REC}$

This construction provides the necessary reduction function f: firstly, f can obviously be computed by a Turing machine, since it only involves basic syntactic modifications of the machine description. Secondly, by construction of the simulation, we have $M(w) = M'(\varepsilon)$. Thus, $(\langle M \rangle, w) \in H \iff \langle M' \rangle \in H_\varepsilon$. □

2.5.2 Rice's Theorem

After the introduction of the Halting Problem as a practically relevant problem that is actually undecidable, one might wonder, what other types of problems are undecidable. Maybe, after all, those are mostly just artificial

problems, and except for a few cases like the Halting Problem most problems of practical concern are actually decidable? Unfortunately, it turns out that checking *any* non-trivial (behavioural) property of a program is undecidable in the general case. This is of high practical relevance, since it means that all methods to check non-trivial properties need to either operate on a restricted, less powerful model, or are just approximate and thus erroneous.

This important result is captured by the following theorem. Intuitively, it expresses that given a Turing Machine M, one can generally not automatically check whether the accepted language of M has a certain non-trivial property, i.e., whether its language belongs to a given class $\mathcal{C}$.

Theorem 2.19 (Rice's Theorem)**.** *Let $\mathcal{C}$ be a non-trivial class of semi-decidable languages, i.e., $\emptyset \subsetneq \mathcal{C} \subsetneq \mathsf{RE}$. Then the following language $L_\mathcal{C}$ is undecidable:*

$$L_\mathcal{C} := \{\langle M \rangle \mid L(M) \in \mathcal{C}\}$$

Proof. Let us first assume that $\emptyset \notin \mathcal{C}$, i.e., the empty language does not have the property described by $\mathcal{C}$. We will reduce the Halting Problem H to $L_\mathcal{C}$.

Let further $A \in \mathsf{RE}$. Such an A must exist by assumption about $\mathcal{C}$, since $\emptyset \notin \mathcal{C}$, but $\mathcal{C} \neq \emptyset$. (Note the different levels of empty sets here!) Let further M_A be a TM that accepts A, i.e., $L(M_A) = A$.

Given a tuple $(\langle M \rangle, w)$, we construct a TM M' as follows (see Figure 9). On some input y, our new machine M' first simulates $M(w)$ on a second tape. If $M(w)$ halts, M' continues with simulating $M_A(y)$ and halting in the state in which M_A halts (if it does).

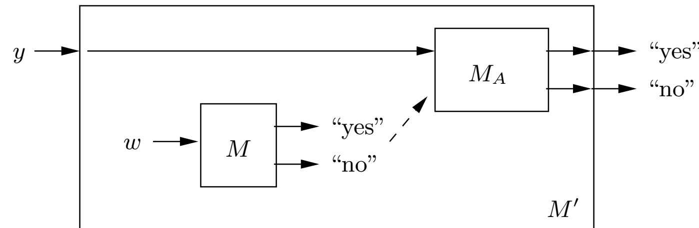


Figure 9: Machine M' in the proof of Rice's Theorem

This construction already produces the necessary reduction $H \leq_m L_\mathcal{C}$:

- If $(\langle M \rangle, w) \in H$, then M' will, on any input y, first simulate $M(w)$ which halts. M' then proceeds with a simulation of $M_A(y)$. Therefore, we have $M'(y) = M_A(y)$, so it follows $L(M') = L(M_A) = A$ and thus $\langle M' \rangle \in \mathcal{C}$.
- On the other hand, if $(\langle M \rangle, w) \notin H$, then on any input y, $M'(y)$ will not terminate (since it first simulates the non-terminating $M(w)$). Therefore, we have $M'(y) = \nearrow$, so it follows $L(M') = \emptyset$ and thus $\langle M' \rangle \notin \mathcal{C}$.

To summarize:

$$(\langle M\rangle, w) \in H \iff \langle M'\rangle \in \mathcal{C}$$

It follows $H \leq_m L_\mathcal{C}$.

We assumed in the beginning, that $\emptyset \notin \mathcal{C}$. If the opposite is the case, then the proof works exactly like above, with the only difference that the obtained reduction is $H \leq_m \overline{L_\mathcal{C}}$. We invite the reader to try out the details. In either case, it follows that $L_\mathcal{C} \notin \mathsf{REC}$. □

Note that Theorem 2.19 contains H_ε as a special case with $\mathcal{C}$ being the set of languages that include the empty word ε. Another example is the following.

Example 2.20. *The language $L := \{\langle M\rangle \mid L(M)$ contains at most 5 words$\}$ is undecidable. This follows from Rice's Theorem with $\mathcal{C}$ being the set of all languages with at most 5 words from* RE*, since this $\mathcal{C}$ is neither empty (there are semi-decidable languages with that property) nor whole* RE *(there are e.g. infinite semi-decidable languages). Thus, in general, one can not automatically find out whether a Turing machine accepts just a bounded number of inputs for any bound k.*

Beware that this undecidability is concerned with the *behaviour* of Turing machines. There are of course other properties that can easily be decided:

Example 2.21. *The language $L := \{\langle M\rangle \mid M$ contains at most 5 states$\}$ is decidable. The property can easily be checked by looking at the encoding of the given machine, which is finite. Rice's Theorem can not be applied, since the property doesn't concern the accepted language, i.e., it is not about a certain specific behaviour, but rather about the syntactical structure.*

3 Complexity Classes

So far, we only discussed what algorithms can do and where they will fail, no matter how hard we try. But in the case of decidable problems, i.e., where an algorithm always succeeds to answer our questions, there are also huge differences: How long does the algorithm run? How much memory will it need? We will now turn to these computations *with restricted resources* in a very general sense. This is the core of what complexity theory is concerned with.

3.1 Landau Symbols: The $\mathcal{O}(\cdot)$ Notation

We will be interested in resource bounds (i.e., for time and space) depending on the *size* of the input. Thus, we will consider functions $f : \mathbb{N} \to \mathbb{N}$ that will be used as a bound on the resource in question. For example, we will use $f(n) = n^2$ to place a quadratic bound on the number of steps that a Turing machine is allowed to take when run on an input w of size $|w| = n$. The point here is to categorize this as being *asymptotically* at most quadratic. So we are not really interested whether we are dealing with n^2 or $5 \cdot n^2 + 3$, or any other additive or multiplicative *constant*. The reason is, that for sufficiently large n, these constants will be eventually dominated by the term n^2. Thus, we would like to consider $c_1 \cdot n^2 + c_2$ as essentially equivalent bounds, no matter which values the constants c_1 and c_2 actually have. Further, since we are interested in the asymptotic behaviour for growing n, we will already consider a function as being “quadratic” if it only exhibits this growth from a certain point n_0 on – even if it assumes arbitrary values for the finitely many arguments which are smaller than that.

To capture this notion, it is common to use *Landau Symbols*, in particular the $\mathcal{O}(\cdot)$ symbol. Formally, it can be defined as follows:

Definition 3.1. *Let $g : \mathbb{N} \to \mathbb{N}$. With $\mathcal{O}(g)$ we denote the set of all functions $f : \mathbb{N} \to \mathbb{N}$ such that there are numbers n_0 and c with*

$$\forall n \geq n_0 : f(n) \leq c \cdot g(n).$$

It is also common to just write $f(n) = \mathcal{O}(g(n))$.

Intuitively, this means that asymptotically, function f doesn't grow faster than function g. In our example from above, we have $f(n) = 5 \cdot n^2 + 3$ and $g(n) = n^2$, and it is easy to verify $f \in \mathcal{O}(g)$. There is another characterization of $f \in \mathcal{O}(g)$, that is sometimes more convenient:

Lemma 3.2. *For $f, g : \mathbb{N} \to \mathbb{N}_{>0}$, we have the following equivalence:*

$$f \in \mathcal{O}(g) \iff \exists c > 0 : \limsup_{n \to \infty} \frac{f(n)}{g(n)} \leq c$$

Proof. Left as an exercise. □

In the alternative notation $f(n) = \mathcal{O}(g(n))$, the example reads $5 \cdot n^2 + 3 = \mathcal{O}(n^2)$, but this is a slight misuse of notation, since the right hand side is actually a *set*. It is even common to write $\mathcal{O}(h(n)) = \mathcal{O}(g(n))$ if for all functions $f \in \mathcal{O}(h)$ it holds that $f \in \mathcal{O}(g)$. Note that the "=" sign actually represents an "element of" or a "subset of" relation (depending on the context) and is thus *not symmetric*. For example it holds that $\mathcal{O}(n) = \mathcal{O}(n^2)$, but not $\mathcal{O}(n^2) = \mathcal{O}(n)$.

Example 3.3. *Some more, important examples are the following:*

- $n \cdot \log(n) = \mathcal{O}(n^2)$
- $n^c = \mathcal{O}(2^n)$ *for all constants* c
- $\mathcal{O}(1)$ *are the bounded functions*
- $n^{\mathcal{O}(1)}$ *are the functions bounded by a polynomial*

In addition to $\mathcal{O}(\cdot)$ to represent an upper bound of the growth, there are also other Landau symbols to represent strict upper bounds ($o(\cdot)$), lower bounds ($\Omega(\cdot)$), strict lower bounds ($\omega(\cdot)$) and "grows asymptotically equally" ($\Theta(\cdot)$). We skip the details for these.

3.2 Time and Space Complexity

We introduced some kind of "classification" to the complexity functions that will be useful to reason about complexity in a little more abstract way. But what functions do we actually consider? For mostly technical reasons, we need to impose restrictions on which complexity functions we want to allow. They should not be arbitrarily "wild", but they should computable and sometimes we even have to explicitly demand one of the following "stopwatch" properties:

Definition 3.4. *Let* $f : \mathbb{N} \to \mathbb{N}$ *be a computable function.*

1. f *is* time-constructible *if there exists a TM which on input*[6] 1^n *stops after* $\mathcal{O}(n + f(n))$ *steps.*
2. f *is* space-constructible *if there exists a TM which on input* 1^n *outputs* $1^{f(n)}$ *and does not use more than* $\mathcal{O}(f(n))$ *space.*

[6] We write a^n for the string $\underbrace{aaa \cdots a}_{n}$.

In fact, all common "natural" functions do have these properties, including the polynomials and the exponential and logarithm functions. Thus, we don't restrict ourselves unnecessarily.

We continue with central definitions for complexity classes. Recall that we defined Turing machines to be *deterministic*, but that we introduced the extension of *non-determinism* in Section 2.2.2. The difference is, that a non-deterministic machine may have the opportunity to choose between several possible configurations for certain steps, which gives rise to a *computation tree.*

Definition 3.5. *We define the resource measures as follows:*

1. *The* runtime $\mathsf{time}_M(w)$ *of a TM M with input w is defined as:*

$$\mathsf{time}_M(w) := \max\{t \geq 0 \mid \exists y, z \in \Gamma^*, q \in F : (w, q_0, \varepsilon) \vdash^t (y, q, z)\}$$

2. *If, for all inputs w and a $t : \mathbb{N} \to \mathbb{N}$ it holds that $\mathsf{time}_M(w) \leq t(|w|)$, then M is $t(n)$-*time-bounded*. We further define:*

$$\mathsf{DTIME}(t(n)) := \{L(M) \mid M \textit{ is } t(n)\textit{-time-bounded}\}$$

3. *The* required space $\mathsf{space}_M(w)$ *of a TM M with input w is defined as:*

$$\mathsf{space}_M(w) := \max\{n \geq 0 \mid M \textit{ uses } n \textit{ squares on a working tape}\}$$

4. *If for all inputs w and an $s : \mathbb{N} \to \mathbb{N}$ it holds that $\mathsf{space}_M(w) \leq s(|w|)$, then M is $s(n)$-*space-bounded*. We further define:*

$$\mathsf{DSPACE}(s(n)) := \{L(M) \mid M \textit{ is } s(n)\textit{-space-bounded}\}$$

5. *For functions, we have:*

$$\mathsf{FTIME}(t(n)) := \{f \mid \exists M \textit{ being } t(n)\textit{-time-bounded and computing } f\}$$

6. *For non-deterministic M,* time *and* space *are defined as above, and we have:*

$$\mathsf{NTIME}(t(n)) := \{L(M) \mid M \textit{ is non-det. and } t(n)\textit{-time-bounded}\}$$
$$\mathsf{NSPACE}(s(n)) := \{L(M) \mid M \textit{ is non-det. and } s(n)\textit{-space-bounded}\}$$

Note that we talk about multiple tapes for $\mathsf{space}_M(w)$, and in particular only about the space requirement the working tapes. The reason for this is that the machine could be restricted to use even less space than the size of the input, e.g., at most $\log(n)$ extra squares for inputs of size n. This implies that the machine is not allowed to write on the input tape and to read from the output tape.

We can now characterize the following deterministic time complexity classes:

$$\mathsf{LINTIME} := \bigcup_{c\geq 1} \mathsf{DTIME}(cn+c) = \mathsf{DTIME}(\mathcal{O}(n)) \qquad \text{(Linear time)}$$

$$\mathsf{P} := \bigcup_{c\geq 1} \mathsf{DTIME}(n^c+c) = \mathsf{DTIME}(n^{\mathcal{O}(1)}) \qquad \text{(Polynomial time)}$$

$$\mathsf{FP} := \bigcup_{c\geq 1} \mathsf{FTIME}(n^c+c) = \mathsf{FTIME}(n^{\mathcal{O}(1)}) \qquad \text{(Polyn.-time functions)}$$

$$\mathsf{EXP} := \bigcup_{c\geq 1} \mathsf{DTIME}(2^{n^c+c}) = \mathsf{DTIME}\left(2^{n^{\mathcal{O}(1)}}\right) \qquad \text{(Exponential time)}$$

For deterministic space complexity classes, we have the following:

$$\mathsf{L} := \mathsf{DSPACE}(\mathcal{O}(\log(n))) \qquad \text{(Logarithmic space)}$$

$$\mathsf{PSPACE} := \mathsf{DSPACE}(n^{\mathcal{O}(1)}) \qquad \text{(Polynomial space)}$$

$$\mathsf{EXPSPACE} := \mathsf{DSPACE}\left(2^{n^{\mathcal{O}(1)}}\right) \qquad \text{(Exponential space)}$$

The non-deterministic classes NLINTIME, NP, NEXP, NL, NPSPACE and NEXPSPACE are defined in a similar way.

Example 3.6. *As a classical example, we consider again* REACH*, the* reachability problem*, which is defined as before in Example 2.17:*

$$\mathsf{REACH} := \{(G, u, v) \mid \textit{there is a path from } u \textit{ to } v \textit{ in } G\}$$

We saw, that this problem is decidable, but how much space does it take if we are careful?

We first note that REACH *can be non-deterministically decided in logarithmic space, i.e.,* $\mathsf{REACH} \in \mathsf{NL}$*: A machine M iteratively explores the graph beginning from u by non-deterministically choosing an edge to traverse. It uses the working tape to remember the number of the current node, together with a step counter. The machine accepts, if it encounters node v, but rejects, if this did not happen after n steps. This works since if there is a path, then there is one of length at most n, and at least one possible computation of M will find it. Both the maintained node number and the step counter require logarithmic space, thus the space requirement of M is* $\mathcal{O}(\log(n))$*.*

Deterministically, this can be done using $\mathcal{O}(\log(n)^2)$ *space,* $\mathsf{REACH} \in \mathsf{DSPACE}(\mathcal{O}(\log(n)^2))$*. The construction for doing that is a bit more involved. Without going into detail, the problem for a path of length n between u and v is recursively split at an intermediate node w into two paths of length $n/2$, iterating over all intermediate nodes w. The recursion stack for doing this is bounded by a logarithmic number of elements, each containing two node numbers and a length (which take logarithmic space). Together, this results in the* $\mathcal{O}(\log(n)^2)$ *space bound.*

3.3 Relations between Complexity Classes

The following relations between the above classes are immediately clear from their definitions:

$$\mathsf{LINTIME} \subseteq \mathsf{P} \subseteq \mathsf{EXP}$$

The same holds for the non-deterministic variants.

So far, we only considered inclusions, without separating the classes, i.e., without showing that classes are actually different. Such a result is the following, which establishes a quite "fine-grained" separation of complexity classes:

Theorem 3.7 (Hierarchy Theorem)**.**

- *Let $f : \mathbb{N} \to \mathbb{N}$ be time-constructible and $g : \mathbb{N} \to \mathbb{N}$ with*

$$\liminf_{n\to\infty} \frac{g(n) \cdot \log(g(n))}{f(n)} = 0.$$

  Then there is a language L with $L \in \mathsf{DTIME}(f(n)) - \mathsf{DTIME}(g(n))$.

- *Let $f : \mathbb{N} \to \mathbb{N}$ be space-constructible and $g : \mathbb{N} \to \mathbb{N}$ with*

$$\liminf_{n\to\infty} \frac{g(n)}{f(n)} = 0.$$

  Then there is a language L with $L \in \mathsf{DSPACE}(f(n)) - \mathsf{DSPACE}(g(n))$.

(Without proof.)

Note the analogy of the first part of the theorem concerning the time hierarchy to the classes of decidable and semi-decidable languages in Section 2.3: Recall that $\mathsf{REC} \subsetneq \mathsf{RE}$, which means that we gain expressiveness by just letting a Turing machine reject an input by a diverging computation, i.e., by not halting after finitely many steps. One of the new problems we gained by allowing that was the Halting Problem H. The above result is a "quantitative" variant, where longer runtime also yields new languages.

Example 3.8. *Let $\mathcal{C}_k := \mathsf{DTIME}(\mathcal{O}(n^k))$ be the set of problems that can be solved by a machine that is time-bounded by a polynomial of degree k. Then it follows from Theorem 3.7:*

$$\mathcal{C}_1 \subsetneq \mathcal{C}_2 \subsetneq \mathcal{C}_3 \subsetneq \dots$$

In other words: For each two polynomials $p(n)$ and $q(n)$ of different degrees $\deg p < \deg q$, there is a language L that can be decided in $\mathcal{O}(q(n))$ time but not in $\mathcal{O}(p(n))$ time. Thus, this establishes a lower time complexity bound *for L.*

We turn now to the comparison between deterministic and non-deterministic classes. To begin with, we have the following relations:

Theorem 3.9. *For each space-constructible function $f : \mathbb{N} \to \mathbb{N}$, the following holds:*

$$\mathsf{DTIME}(f) \subseteq \mathsf{NTIME}(f) \subseteq \mathsf{DSPACE}(f) \subseteq \mathsf{NSPACE}(f)$$

Proof (Sketch). The first and third inclusion are clear, since deterministic machines can be seen as special cases of non-deterministic machines. Thus, we only need to prove $\mathsf{NTIME}(f) \subseteq \mathsf{DSPACE}(f)$.

The key observation is, that a machine that is time-bounded by a function f is as well space-bounded by that function, since in each step, a machine can only reach at most one additional square. Further, we can take care of the non-determinism by a deterministic simulation in the following way: We know, that there is a non-deterministic machine N with a runtime bound $f(n)$ for inputs of size n. Thus, N does at most $f(n)$ non-deterministic choices during its computation. Further, let d be the maximal degree of the choices, i.e., at each step, N can choose only among d different next steps. (This number can be derived from the transition function δ).

We can now conduct a deterministic simulation of N: given an input w of size n, we iteratively generate all words of length $f(n)$ over the alphabet $\{1, \dots, d\}$ on a working tape. Each symbol represents the choice of N's decisions for the non-deterministic choices, see Figure 10.

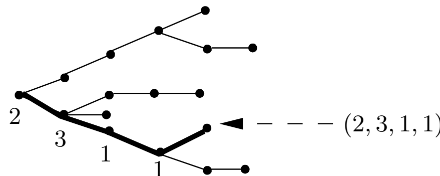


Figure 10: Deterministic simulation of a non-deterministic computation tree: Numbering all non-deterministic choices. The illustrated computation path is represented by the "choice word" $c = 2311$.

For each of these "choice words" c, we simulate N and at each choice, we act according to the current symbol in c. For the simulation itself, we use another working tape. Note that this simulation is deterministic, since we iteratively simulate all paths in the computation tree of N. Now we know, that if there is an accepting computation of N, then we will eventually simulate it. Further, we don't use more than $f(n)$ space, neither for the choices c nor for the working tape of N. □

Thus, it directly follows:

$$\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE} \subseteq \mathsf{NPSPACE}$$

In the context of polynomial complexity, this means that not only is non-determinism (not surprisingly) at least as expressive as determinism, but also that a polynomial space restrictions are less restrictive than polynomial time restrictions. It is actually unknown, whether the first two inclusions are strict, and we will turn to that question in the next chapter. For the third inclusion, it is known that both classes are equal, which means that non-determinism does not increase expressiveness. This is a direct consequence of the following theorem:

Theorem 3.10 (Savitch)**.** *For each space-constructible function $f : \mathbb{N} \to \mathbb{N}$, the following holds:*

$$\mathsf{NSPACE}(f) \subseteq \mathsf{DSPACE}(f^2)$$

Proof (Sketch). Let $L \in \mathsf{NSPACE}(f)$ and M_L be a non-deterministic Turing machine deciding L with space bound f. The proof is based on the *configuration graph* $G_{M,w}$ of M given an input w. A node in the graph represents one configuration of M, in particular the state, the positions of the heads and the contents of the working tapes. It follows, that there is a constant c such that the number of configurations is bounded by $c^{f(n)}$ if we let n denote the size of w.

For a deterministic decision procedure, we now only need to find out if from the initial configuration an accepting configuration is reachable. Further, we can assume that there is only one accepting configuration, since M can be trivially extended to clean up the working tapes before finally accepting. Thus, we are left with an instance of the *reachability problem* in graph $G_{M,w}$. We saw in Example 3.6 that for a graph with m nodes it can be solved deterministically in $\mathcal{O}(\log(m)^2)$ space. Since the configuration graph has at most $c^{f(n)}$ nodes, we can conduct the deterministic decision of L deterministically using $\mathcal{O}\left(\log\left(c^{f(n)}\right)^2\right) = \mathcal{O}(f(n)^2)$ space[7]. Further, we can get rid of the constant factor that is denoted by the $\mathcal{O}$ symbol by encoding the contents of several tape squares into one (using a larger alphabet). The theorem follows. □

Corollary 3.11. $\mathsf{PSPACE} = \mathsf{NPSPACE}$.

The last relation of complexity classes we want to introduce in this chapter is the relation of a class $\mathcal{C}$ of languages to the class $\mathsf{co}\text{-}\mathcal{C}$ of their complements:

Definition 3.12. *Let $\mathcal{C} \subseteq \mathcal{P}(\Sigma^*)$ be a class of languages. Then we define:*

$$\mathsf{co}\text{-}\mathcal{C} := \{\overline{L} \mid L \in \mathcal{C}\}$$

[7]Note that a logarithm base change only imposes an additional constant factor: $\log_a(y) = \log_b(y) \cdot \log_a(b)$.

Example 3.13. *Recall that* RE *is the set of languages* L *for which a Turing machine* M *exists which halts for exactly the positive instances of* L*. Otherwise it does not halt. In these terms,* co- RE *is the set of languages such that a machine* M *exists that always halts for exactly the negative instances.*

Complexity class NP *exhibits a similar situation: There is a non-deterministic machine* M *such that exactly for the positive instances of a language* $L \in \mathsf{NP}$ *a computation path exists that leads to an accepting state. Likewise, for an* $L \in \text{co-}\mathsf{NP}$*, there must be a non-deterministic* M *with an accepting computation path exactly for the negative instances.*

We already know from Theorem 2.10, that the class of decidable languages is closed under taking the complement. Thus, we know $\mathsf{REC} = \text{co-}\mathsf{REC}$. Further, we know that L and its complement are semi-decidable, if and only if L is decidable. Thus, we have $\mathsf{REC} = \mathsf{RE} \cap \text{co-}\mathsf{RE}$. This can be seen as a compensation of the *asymmetry* of RE between positive and negative instances: REC is the intersection of problems with identifiable positive instances (RE) and problems with identifiable negative instances (co- RE). Thus, the symmetry is restored in REC.

With the polynomially time-bounded classes, P, NP and co- NP, the situation might look suspiciously similar, but it is not that clear: As described above, NP exhibits a similar asymmetry. But in contrast to the case with the decidable and semi-decidable languages, the relation of the three sets is not known in detail so far. We will turn to that question in the following chapter.

Fortunately, we are in a much better situation for the polynomially space-bounded classes. As shown above, $\mathsf{PSPACE} = \mathsf{NPSPACE}$, and since deterministic complexity classes are closed under taking the complement (via swapping q_{yes} and q_{no}), we have $\text{co-}\mathsf{PSPACE} = \mathsf{PSPACE}$ and thus $\mathsf{NPSPACE} = \text{co-}\mathsf{NPSPACE}$.

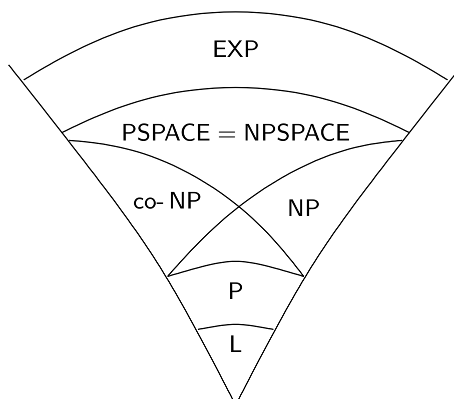


Figure 11: Overview of the most important complexity classes

4 Feasible Computations: P vs. NP

We gained now a basic understanding of the notions of computability and complexity. In this chapter, we will focus on the two most prominent complexity classes, P and NP, and their relation. Their importance is rooted in the notion that polynomial time bounds are considered "feasible", "tractable", or "efficient". We will introduce an alternative characterization of NP that allows for a more convenient perspective on the practical importance of the class. Following that, we explore techniques for classifying certain problems as being "more difficult" than others. This will lead to the popular notion of NP-completeness.

4.1 Proving vs. Verifying

In the previous chapter, we introduced the complexity classes P and NP as the classes of languages that can be decided by deterministic and non-deterministic Turing machines, respectively, which have a polynomial runtime bound. In particular, for each $L_1 \in \mathsf{P}$, we demanded the existence of a Turing machine M and a polynomial $p_M(n) = n^{O(1)}$ such that for each input $w \in \Sigma^*$, the runtime of M with input w is bounded by $p_M(|w|)$. Similarly, for $L_2 \in \mathsf{NP}$, there must be a non-deterministic Turing machine N and a polynomial $p_N(n)$, such that for each input w, the length of *all* computation paths of N with input w is bounded by $p_N(|w|)$. While a practical intuition for the significance of P is immediately clear from that description, it might still lack a good intuition for NP. We will therefore characterize NP in a different way now.

We begin with a definition regarding relations as sets of tuples.

Definition 4.1. *Let $R \in \Sigma^* \times \Sigma^*$ be a binary relation. R is* polynomially bounded*, if there exists a polynomial $p(n)$, such that for all $(x, y) \in R$, it holds that $|y| \leq p(|x|)$.*

Thus, the second component of the pairs in such a relation is always bounded by a polynomial in the length of the first. Using this notion, we can characterize NP as follows:

Lemma 4.2. *The complexity class* NP *is the class of all languages L for which there exists a polynomially bounded relation $R_L \in \Sigma^* \times \Sigma^*$, such that*

- *R_L is decidable in polynomial time, i.e., $R_L \in \mathsf{P}$, and*
- *$x \in L$ if and only if there exists a w with $(x, w) \in R_L$.*

We call w a witness *(or* proof*) for $x \in L$ and R_L the* witness relation*.*

Before supplying a proof of this, we proceed with a short discussion about the practical meaning of this characterization, which is in fact the central reason for the high significance of NP.

We recall that for languages L in P, a machine must be able to decide membership of a word x to L in polynomial time. One can see this as the task of *proving* the membership: once given a problem instance, the machine must work out a proof within a reasonable time. In these terms and using the new characterization, the languages $L \in \mathsf{NP}$ relieve the machine from finding the proof. A proof is supplied in form of the witness w, and all that is left for the machine is to *verify* the validity of the proof. By the above characterization, this verification procedure is supposed to be efficient (since $R_L \in \mathsf{P}$) and the proof has a "reasonable" size (since R_L is polynomially bounded). We may thus characterize languages $L \in \mathsf{NP}$ as follows:

$$L = \{x \in \Sigma^* \mid \exists w \in \Sigma^* : (x, w) \in R_L\}$$

Therefore, one can interpret P as the set of problems that can be *solved* efficiently, whereas NP are the problems for which a solution can be *checked* efficiently. See Figure 12 for the setting: A machine deciding a P-problem is "on its own", while a machine deciding a NP-problem can rely on a supplied witness. There just needs to exist such a witness for exactly the positive instances, the machine does not have to come up with one itself.

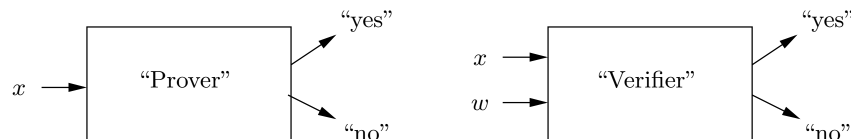


Figure 12: Left: A "prover" for deciding a P-problem instance x; Right: A "verifier" for deciding an NP-problem instance x with supplied "witness" w

Now, in order to be able to (efficiently) find a solution for a problem, it seems a natural necessity to be able to (efficiently) recognize a solution as being one. It is one of the most fundamental questions of computer science, whether there is "more" to it: whether there are problems, whose solutions can be efficiently *verified*, but which can't be *found* efficiently. In formal terms, this is the famous $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$ question. Although it may seem intuitively reasonable to believe that the two classes should be unequal, a proof of this conjecture has so far turned out to be out of sight. In fact, many "proofs" of $\mathsf{P} \neq \mathsf{NP}$ as well as $\mathsf{P} = \mathsf{NP}$ have been proposed over time, but were so far all being shown to be flawed. (See for example [Woe] for a collection of proof attempts.)

We note that also $\mathsf{co\text{-}NP}$ can be characterized using the witness-based approach. While for an $L \in \mathsf{NP}$ there must be a witness relation R_L supplying witnesses for all *positive instances*, an $L' \in \mathsf{co\text{-}NP}$ has a witness relation $R_{L'}$ with witnesses for all *negative instances*. Thus, NP is characterized

by efficiently checkable *proofs*, whereas co- NP is characterized by efficiently checkable *disproofs*. We can therefore write L' in these terms as follows:

$$L' = \{x \in \Sigma^* \mid \forall w \in \Sigma^* : (x, w) \notin R_{L'}\}$$

We now sketch proof of Lemma 4.2, stating the equivalence of both characterizations of NP.

Proof sketch of Lemma 4.2. We wish to prove the equivalence of the "traditional" NP definition which is based on the notion of a non-deterministic Turing machine and the above characterization which is based on the notion of a witness relation.

Let's first take a language $L \in \mathsf{NP}$ using the traditional definition from Section 3.2. This means that there exists a non-deterministic TM N and a polynomial $p_N(n)$ such that N decides L and all computation paths of N for an input of size n are bounded by $p_N(n)$. We want to show the existence of a witness relation $R_L \in \mathsf{P}$ that is polynomially bounded. And indeed, it exists: since given an input x of size n, N does at most $p_N(n)$ steps, it also does at most $p_N(n)$ non-deterministic choices. We can encode these choices into a string that is polynomially bounded in n. For $x \in L$, there must be at least one accepting computation path, and we can use the encoding of this path as the witness w of membership $x \in L$. The relation R_L we get is by construction polynomially bounded. It can also be decided in polynomial time, since to verify $(x, w) \in R_L$, one only needs to (deterministically) simulate N with the non-deterministic choices w, i.e., only the computation path represented by w needs to be simulated.

For the other direction, let's assume the existence of a witness relation R_L for a language L. Recall that $R_L \in \mathsf{P}$, so there is a deterministic TM M deciding R_L and a polynomial $p(n)$ bounding the size of the second component. We want to show the existence of a polynomially bounded non-deterministic TM N deciding L. This machine N can easily be constructed: given an input x, the machine N can non-deterministically write a witness w on an extra tape. We know, that $|w|$ must be bounded by $p(|x|)$, so this takes only polynomially long. After that, N simulates the machine M that decides R_L with input (x, w). This will also take only polynomial time by assumption. N accepts if and only if M accepts: by construction, given input x, there is an accepting computation of N if and only if $x \in L$. Further, the runtime of N is bounded polynomially, thus $L \in \mathsf{NP}$. □

We give an example of a natural problem in the class NP called the *satisfiability problem*. It will turn out that this language plays a fundamental role in later proofs. For the example, we need the notion of *boolean formulas*:

Definition 4.3. *Let $X = \{x_1, \ldots, x_N\}$ be a set of variable names.*

- *A* boolean formula *over X is defined inductively as follows:*

  - *Every variable x_i is a boolean formula.*
  - *If φ_1 and φ_2 are boolean formulas, then also $\neg\varphi_1$, the* negation *of φ_1, and $\varphi_1 \wedge \varphi_2$, the* conjunction *of φ_1 and φ_2.*

- *We denote the* set of all boolean formulas *with* BOOL.
- *A* truth assignment *for the variables in X is a word $\alpha = \alpha_1 \dots \alpha_N \in \{0,1\}^N$. The* value *$\varphi(\alpha)$ of φ under α is defined inductively over the structure of φ:*

$\varphi:$	x_i	$\neg\psi$	$\psi_1 \wedge \psi_2$
$\varphi(\alpha):$	α_i	$1 - \psi(\alpha)$	$\psi_1(\alpha) \cdot \psi_2(\alpha)$

- *We use $\varphi_1 \vee \varphi_2$ (disjunction), $\varphi_1 \to \varphi_2$ (implication) and $\varphi_1 \leftrightarrow \varphi_2$ (equivalence) as shorthand notations for $\neg(\neg\varphi_1 \wedge \neg\varphi_2)$, $\neg\varphi_1 \vee \varphi_2$ and $(\varphi_1 \to \varphi_2) \wedge (\varphi_2 \to \varphi_1)$, respectively.*

Example 4.4. *As another shorthand notation, consider the* exclusive or XOR *with m arguments:*

$$\mathsf{XOR}(z_1, \dots, z_m) := \bigvee_{i=1}^{m} z_i \wedge \bigwedge_{1 \le i < j \le m} \neg(z_i \wedge z_j)$$

Example 4.5. *Consider $\psi_1 = (x_1 \vee \neg x_2) \wedge x_3$ and $\alpha = 011$, then we have $\psi_1(\alpha) = 1$. Other formulas, like $\psi_2 = (x_1 \wedge \neg x_1)$, always evaluate to 0.*

We call a boolean formula φ satisfiable*, if there is a truth assignment α for its variables such that $\varphi(\alpha) = 1$. Using an appropriate encoding, boolean formulas can be encoded into words over some fixed alphabet Σ. Thus, we can define the language of all encodings of satisfiable formulas as follows:*

$$\mathsf{SAT} := \{\langle\varphi\rangle \mid \varphi \in \mathsf{BOOL} \text{ is satisfiable}\}$$

This is a decision problem called the satisfiability problem. *It turns out, that $\mathsf{SAT} \in \mathsf{NP}$: For each positive instance, a witness for the satisfiability of the formula is a satisfying assignment of truth values. The size of such a witness is linearly bounded in the representation of the formula, and the validity can be easily checked in polynomial time.*

As we notice from this example, there are natural problems, for which it is easy to see that they belong to the class NP. In fact, it is unknown whether $\mathsf{SAT} \in \mathsf{P}$, i.e., whether in general the existence of a satisfying truth assignment can be checked deterministically in polynomial time. In fact, as we will see in the following sections, a proof for that would be sufficient for showing $\mathsf{P} = \mathsf{NP}$ – or, conversely, $\mathsf{P} \neq \mathsf{NP}$ would imply $\mathsf{SAT} \notin \mathsf{P}$. This can be seen as a clear indication that an efficient way to solve the satisfiability problem does not exist.

4.2 Reductions, Hardness, Completeness

As a next step on our quest to capture the expressiveness of NP in comparison to P is the wish to be able to directly compare the complexity of concrete problems. This is not easily possible by just showing the membership to a certain complexity class (by, e.g., providing a sufficiently bounded Turing machine): how do we know in what way two problems A and B relate to each other, if they are both in NP? Which one is "harder"? Even more, what about a language $C \in \mathsf{P}$, is it "easier" than A and B? That is not necessarily the case, since deterministic polynomial time algorithms for A and B might also exist, we just did not find them. Their time bounds might even be of lower degrees! Thus, we see that providing membership to a class by giving a Turing machine description provides only an upper complexity bound. For direct comparison, a more sophisticated concept is necessary, as well as for showing lower complexity bounds, since these are usually quite difficult to prove directly. (How can you show that a machine with certain resource bounds *can not exist* for a given problem?)

We recall the concept of *reductions* from Section 2.5.1. A problem A can be reduced to a problem B, if instances from A can be solved by being able to solve instances from B. The concept we introduced was the *many-one reduction*, where a reduction function f is provided, such that positive instances from A map to positive instances from B, and likewise for negative instances. Thus, to solve A, one simply applies f and then decides the membership of the result to B.

This worked well for decidability questions and distinguishing decidable from undecidable problems. We will use the same idea for the comparison of problems from a complexity point of view. Since in this setting, the resources for deciding A and B are bounded, we also need to impose bounds on the reduction function itself – otherwise, all the necessary work could be done by computing f and we would not get a useful complexity comparison.

The reduction concept we use is sometimes called "Cook reduction" and is a bounded variant of the many-one reduction we introduced before. We define it as follows:

Definition 4.6. *A language $A \subseteq \Sigma^*$ is* polynomially reducible *to a language $B \subseteq \Sigma^*$, written $A \leq_m^p B$, if there is a total function $f \in \mathsf{FP}$, such that*

$$\forall w \in \Sigma^* : w \in A \iff f(w) \in B$$

This polynomial reduction has properties similar to the many-one reduction as in Lemma 2.16:

Lemma 4.7. *For all languages A, B and C the following properties hold:*

1. $A \leq_m^p B \land B \in \mathsf{P} \Longrightarrow A \in \mathsf{P}$ *(Closedness of P under $\leq_m^p$)*
2. $A \leq_m^p B \land B \in \mathsf{NP} \Longrightarrow A \in \mathsf{NP}$ *(Closedness of NP under $\leq_m^p$)*

3. $A \leq_m^p B \wedge B \leq_m^p C \Longrightarrow A \leq_m^p C$ *(Transitivity of $\leq_m^p$)*
4. $A \leq_m^p B \iff \overline{A} \leq_m^p \overline{B}$

Proof. Left as an exercise. Note that the composition $p \circ q(n) := p(q(n))$ of two polynomials $p(n)$ and $q(n)$ is again a polynomial. □

We further define a notion of "hard" problems for a class $\mathcal{C}$. These problems are "more difficult" than everything in the whole class. This approach can be used in practise to convince oneself, that a certain problem is for sure difficult to solve. The notion can further be used as a first step on the way to discover a difference between P and NP: if there exists some problem in $\mathsf{NP} - \mathsf{P}$, then it certainly has to be one of the most difficult ones in NP.

Definition 4.8.

- *A language A is called* $\mathcal{C}$-hard*, if*

$$\forall L \in \mathcal{C} : L \leq_m^p A.$$

- *A $\mathcal{C}$-hard language A belonging to $\mathcal{C}$ is called* $\mathcal{C}$-complete*.*
- NPC *is the class of all* NP*-complete languages.*

This definition allows us to talk about the "most difficult" problems in NP by just referring to the set of NP-complete languages, NPC. One can, by definition, solve any problem in NP, if one knows how to solve just one single problem in NPC. If indeed just one of them can be solved *efficiently*, then $\mathsf{P} = \mathsf{NP}$ would follow, as the second part of the following lemma states:

Lemma 4.9.

1. *A is $\mathcal{C}$-complete if and only if $\overline{A}$ is co-$\mathcal{C}$-complete.*
2. $\mathsf{P} \cap \mathsf{NPC} \neq \emptyset \Longrightarrow \mathsf{P} = \mathsf{NP}$
3. $A \in \mathsf{NPC} \wedge A \leq_m^p B \wedge B \in \mathsf{NP} \Longrightarrow B \in \mathsf{NPC}$

Proof. For the first part, let A be $\mathcal{C}$-complete and $L \in \mathsf{co}$-$\mathcal{C}$. We want to show $L \leq_m^p \overline{A}$. Since $\overline{L} \in \mathcal{C}$, we have $\overline{L} \leq_m^p A$. This is equivalent to $L \leq_m^p \overline{A}$, see Lemma 4.7.

For the second part, assume there is an $A \in \mathsf{P} \cap \mathsf{NPC}$ and let $L \in \mathsf{NP}$. We want to show that $L \in \mathsf{P}$. And indeed: since $A \in \mathsf{NPC}$, we have $L \leq_m^p A$ by definition. But since also $A \in \mathsf{P}$, it follows from Lemma 4.7 – namely the closedness of P under $\leq_m^p$ – that also $L \in \mathsf{P}$.

For the third part, assume $A \in \mathsf{NPC}$, $A \leq_m^p B$ and $B, L \in \mathsf{NP}$. We want to show $L \leq_m^p B$, since this establishes the NP-completeness of B. From $A \in \mathsf{NPC}$ it follows $L \leq_m^p A$. Using transitivity of $\leq_m^p$, we immediately get $L \leq_m^p B$. □

The first part of the lemma tells us that "hardest" problems in a complexity class $\mathcal{C}$ directly give rise to "hardest" problems in the "parallel world" co-$\mathcal{C}$ of the complements. The second part states, that if the conjecture $\mathsf{P} \neq \mathsf{NP}$ really holds, then none of the NP-complete problems can be solved efficiently. Further, the third part tells us, that all NP-complete problems are "equally difficult", they can all be reduced to each other.

But so far, we didn't answer an important question yet: Do NP-complete languages actually exist? Indeed, they do:

Lemma 4.10. *The following language is* NP*-complete:*

$$\mathsf{NPCOMP} := \{(\langle M \rangle, x, 1^n) \mid M \textit{ is NTM and accepts } x \textit{ after} \leq n \textit{ steps}\}$$

(1^n denotes unary encoding of n, and "NTM" means "non-deterministic Turing machine".)

Before providing the proof, we should point out the two proof obligations in order to show that a language A is indeed NP-complete:

1. **Membership:** Show $A \in \mathsf{NP}$.
   (Directly or via $A \leq_m^p B$ for a $B \in \mathsf{NP}$.)

2. **Hardness:** Show $L \leq_m^p A$ for all $L \in \mathsf{NP}$.
   (Directly or via $C \leq_m^p A$ for a C which is NP*-hard.)*

The first part can be done by either describing a witness relation R_A, or showing that $A \leq_m^p B$ for a language B that is already known to be in NP. The second part can be done by either reducing an arbitrarily chosen L to A, or, as soon as NP-hard problems are known, reducing another NP-hard problem to A. The latter is usually easier, since only a particular reduction needs to be proven. We will do that in all future proofs after the following one.

Proof of Lemma 4.10. In the following, we assume a fixed alphabet Σ for all languages involved in the proof. This does not loose generality, since recoding can be done easily.

It is clear that $\mathsf{NPCOMP} \in \mathsf{NP}$, since a witness w for $(\langle M \rangle, x, 1^n) \in \mathsf{NPCOMP}$ is just a sequence of the non-deterministic choices M has to take for accepting x within n steps, so w is at most n symbols long and therefore linearly bounded. Verifying the validity of w is in P since only the computation path of M which is represented by w has to be checked.

To show also NP-hardness of NPCOMP, let $L \in \mathsf{NP}$. We want to show that $L \leq_m^p \mathsf{NPCOMP}$. For doing that, let M_L be a non-deterministic Turing machine deciding L, time bounded by a polynomial $p(n)$. The following function $f \in \mathsf{FP}$ establishes the reduction $L \leq_m^p \mathsf{NPCOMP}$:

$$f : x \mapsto (\langle M_L \rangle, x, 1^{p(|x|)})$$

(Note that we used the "classical" definition of NP for this part of the proof. This is not a problem, since both characterizations are equivalent.) □

4.3 Natural NP-complete problems

We note that NPCOMP is a quite "artificial" problem, which seems to be designed just for the purpose of being NP-complete and without practical relevance. This is certainly true, but as we will see in this section, there are very natural problems, which can all be proven NP-complete using NPCOMP.

4.3.1 NP-completeness of SAT

To study the first natural problem of interest, we turn back to the *satisfiability problem* introduced in Example 4.5, that consisted of all satisfiable boolean formulas:

$$\mathsf{SAT} := \{\langle\varphi\rangle \mid \varphi \in \mathsf{BOOL} \text{ is satisfiable}\}$$

We already saw that SAT $\in$ NP: Once we are given a satisfying truth assignment to the variables, it is easy to efficiently check its validity. In fact, as discussed in the beginning of this chapter, this is the defining property of NP-problems: Once a witness for a positive instance of a problem is known, it can be efficiently verified. In that regard, SAT does not differ from other NP-problems. What makes it special is the fact that NPCOMP from above can be directly reduced to it, which turns SAT into the first natural NP-complete problem we consider in this course. The details are given in the proof to the following theorem:

Theorem 4.11 (Cook, Levin)**.** *SAT is NP-complete.*

Proof. Since we already know SAT $\in$ NP, it is sufficient to show NPCOMP $\leq_m^p$ SAT. For doing that, we need to supply a reduction function $f \in \mathsf{FP}$ that transforms a tuple $(\langle M\rangle, x, 1^n)$ of an NTM encoding $\langle M\rangle$, an input word x and a runtime bound in unary 1^n, into a formula ψ such that

$$(\langle M\rangle, x, 1^n) \in \mathsf{NPCOMP} \iff \psi \in \mathsf{SAT}\,.$$

Without loss of generality we may assume that M has only one single tape, since a k-tape TM can be transformed into just using one single tape while being slowed down only polynomially.

We know that if M accepts x, then there is an accepting computation path of length at most n. Thus, there are at most $2n + 1$ tape positions reached and therefore only that many positions can contain anything else than the blank symbol. Thus, we can imagine a configuration of the machine at each step as a line with $2n + 1$ symbols, plus the information on which position the tape head is and in which state the machine is. We know that we need to consider at most n steps, so we can imagine a whole (single) computation of the machine as a matrix with n rows of $2n+1$ entries, each row representing the computation in one step. See Figure 13. The matrix has $n \cdot (2n + 1) = \mathcal{O}(n^2)$ entries.

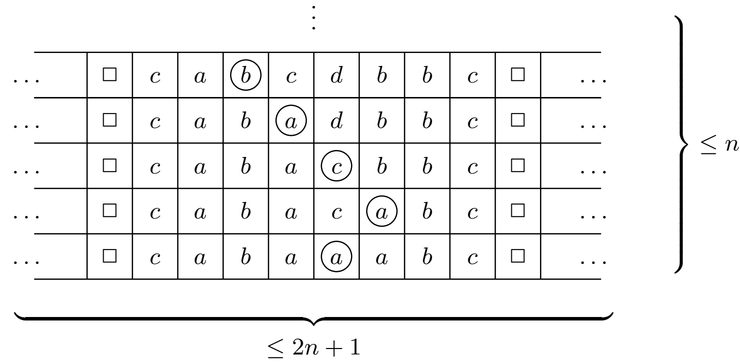


Figure 13: Matrix representing a computation. The position of the tape head in each row is represented by the circle.

Note that since M is non-deterministic, there are several different computations possible for each input x. Each computations corresponds to one path in the computation tree. M accepts x in at most n steps if there is a computation path of length $\leq n$ that leads to the accepting state q_{yes}. This is the case if and only if there exists one of the described computation matrices in which one of the rows represents an accepting configuration.

The key idea to the proof is now to exactly describe such a matrix using one (gigantic) boolean formula ψ. Boolean variables will be used to describe the tape contents, the position of the tape head, the machine state in each row, and the relation of different rows to each other. The constructed formula ψ is satisfiable if and only if the encoded matrix represents an accepting computation, since every satisfying assignment to the variables represents a valid and accepting computation of the machine, and vice versa. Further, each matrix entry can be encoded into boundedly many boolean variables, which makes the resulting formula, even though huge, of size $\mathcal{O}(n^2)$ and thus only polynomially big.

We proceed by providing details of the encoding. Recall that we are given the description of Turing machine M, with the states $Q = \{q_0, \dots, q_k\}$ and tape alphabet $\Gamma = \{a_1, \dots, a_l\}$. One of the states is the accepting state q_{yes}. We are further given input x and the step count n. The construction uses the following variables:

- $Q_{t,q}$ for all[8] $t \in [0, n]$ and $q \in Q$.
  Interpretation: After step t, the machine is in state q.
- $H_{t,i}$ for all $t \in [0, n]$ and $i \in [-n, n]$.
  Interpretation: After step t, the tape head is at position i.
- $T_{t,i,a}$ for all $t \in [0, n]$, $i \in [-n, n]$ and $a \in \Gamma$.
  Interpretation: After step t, the tape contains symbol a at position i.

[8] We use $[a, b]$ for the set $\{a, a + 1, \dots, b\}$ of integers.

It is now clear, that these are only $\mathcal{O}(n^2)$ variables. Next, we describe the formula ψ. It is composed of different parts:

$$\psi := \mathit{Conf} \wedge \mathit{Start} \wedge \mathit{Step} \wedge \mathit{End}$$

We describe the parts in detail:

Conf**:** This formula will make sure that a satisfying truth assignment actually represents valid configurations at each step: at each time point t, the machine is in exactly one state $q \in Q$, the head is at exactly one tape position, and each tape position contains exactly one symbol from Γ. Thus, we have three parts:

$$\mathit{Conf} := \mathit{Conf}_Q \wedge \mathit{Conf}_H \wedge \mathit{Conf}_T$$

$$\mathit{Conf}_Q := \bigwedge_{t=0}^{n} \mathsf{XOR}(Q_{t,q_0}, \ldots, Q_{t,q_k})$$

$$\mathit{Conf}_H := \bigwedge_{t=0}^{n} \mathsf{XOR}(H_{t,-n}, \ldots, H_{t,n})$$

$$\mathit{Conf}_T := \bigwedge_{t=0}^{n} \bigwedge_{i=-n}^{n} \mathsf{XOR}(T_{t,i,a_1}, \ldots, T_{t,i,a_l})$$

To ease notation, we used a shorthand notation XOR to denote that exactly one of the variables should be true:

Start**:** At $t = 0$, the machine is in the start configuration:

$$\mathit{Start} := Q_{0,q_0} \wedge H_{0,0} \wedge \bigwedge_{i=-n}^{-1} T_{0,i,\Box} \wedge \bigwedge_{i=0}^{|x|-1} T_{0,i,x_{i+1}} \wedge \bigwedge_{i=|x|}^{n} T_{0,i,\Box}$$

Step**:** At each step, the machine executes a legal action, i.e., it changes only one tape field, moves the head by at most one position, and all these actions together with the state change are provided by the transition function δ. We compose it of two parts Step_1 and Step_2. Step_1 makes sure that the tape doesn't change at positions other than the head position. Step_2 encodes the actual executed action:

$$\mathit{Step} := \mathit{Step}_1 \wedge \mathit{Step}_2$$

$$\mathit{Step}_1 := \bigwedge_{t=0}^{n-1} \bigwedge_{i=-n}^{n} \bigwedge_{a \in \Gamma} ((\neg H_{t,i} \wedge T_{t,i,a}) \rightarrow T_{t+1,i,a})$$

$$\mathit{Step}_2 := \bigwedge_{t=0}^{n-1} \bigwedge_{i=-n}^{n} \bigwedge_{a \in \Gamma} \bigwedge_{p \in Q} \Big((Q_{t,p} \wedge H_{t,i} \wedge T_{t,i,a}) \rightarrow \bigvee_{(q,b,D) \in \delta(p,a)} (Q_{t+1,q} \wedge H_{t+1,i+D} \wedge T_{t+1,i,b}) \Big)$$

End: At some point, the machine reaches an accepting configuration:

$$End := \bigvee_{t=0}^{n} Q_{t,q_{yes}}$$

This completes the description of formula ψ. It is easy to verify, that the formula as well as the time complexity for its construction are polynomial in n.

□

We note that the NP-completeness of SAT directly establishes that its complement UNSAT, the set of all *unsatisfiable* formulas, is co- NP-complete:

$$\mathsf{UNSAT} := \{\langle\varphi\rangle \mid \varphi \in \mathsf{BOOL} \text{ is } \textit{not} \text{ satisfiable}\} = \overline{\mathsf{SAT}}$$

The example SAT from above shows that there are indeed natural problems that are NP-complete. We will study some more of them in this section. Recall that it is sufficient to provide a polynomial time reduction from a known NPC problem A to a so far unknown problem $B \in \mathsf{NP}$ to show the NP-completeness of B. Thus, we will start reducing SAT to new problems, which we can then add to our "pool" of known NP-complete problems. For every new problem, the whole variety of the "pool" can be used for new reductions.

We will introduce and discuss in this section more NP-complete problems that are relevant in practice, and provide proofs of NP-completeness for some of them.

4.3.2 CIRSAT

CIRSAT is the satisfiability problem for boolean circuits.

Definition 4.12 (Boolean Circuit)**.** *Let $X = \{x_1, \dots, x_N\}$ be a set of variable names.*

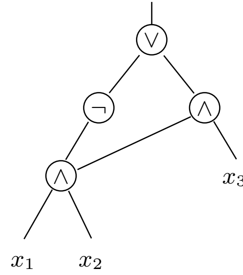
- *A* boolean circuit *over variables X is a sequence $c = (g_1, \dots, g_m)$ of* gates[9]*:*
$$g_i \in \{\bot, \top, x_1, \dots, x_N, (\neg, j), (\wedge, j, k)\}_{1 \le j,k < i}$$
- *The boolean function $f_c^{(i)}$ represented by gate g_i is N-ary, i.e., it takes an input $\alpha \in \{0,1\}^N$, and is defined inductively:*

$g_i(\alpha)$:	$\bot$	$\top$	x_i	$(\neg, j)$	$(\wedge, j, k)$
$f_c^{(i)}(\alpha)$:	0	1	α_i	$1 - f_c^{(j)}(\alpha)$	$f_c^{(j)}(\alpha) \cdot f_c^{(k)}(a)$

[9] As with boolean formulas, we omit a $\vee$ gate in the definition for simplicity reasons. It can easily be simulated by a $\wedge$ gate and three $\neg$ gates, enabling us to use it anyway as shorthand notation.

- *The boolean function f_c represented by c is defined as $f_c(\alpha) := f_c^{(m)}(\alpha)$.*
- *c is* satisfiable *if there is an input $\alpha \in \{0,1\}^N$ such that $f_c(\alpha) = 1$.*

Example 4.13. *The circuit $c = (x_1, x_2, x_3, (\wedge, 1, 2), (\neg, 4), (\wedge, 4, 3), (\vee, 5, 6))$ can be represented graphically as follows:*



It corresponds to the boolean formula $\varphi = \neg(x_1 \wedge x_2) \vee (x_1 \wedge x_2 \wedge x_3)$.

Apparently, there is a one-to-one correspondence between boolean formulas and boolean circuits. Further, just like boolean formulas, circuits can be encoded using a fixed alphabet Σ. Thus, we have a new decision problem CIRSAT containing all encodings of satisfiable circuits, and it is also NP-complete.

Definition 4.14. *The* circuit satisfiability problem *is defined as follows:*

$$\mathsf{CIRSAT} := \{\langle c \rangle \mid c \text{ is a satisfiable circuit}\}$$

Lemma 4.15. CIRSAT *is* NP*-complete.*

Proof. A boolean formula φ can efficiently be transformed into an equivalent circuit c, thus we have SAT $\leq_m^p$ CIRSAT. Further, CIRSAT $\in$ NP is clear, since a satisfying input witnesses the satisfiability of a circuit c, is bounded by the size of the circuit description and can be verified efficiently. □

Note that in the other direction, a transformation of circuits to boolean formulas is not necessarily efficient, since circuits can "reuse" sub-circuits multiple times. But it is true nevertheless, that CIRSAT $\leq_m^p$ SAT – this reduction produces a formula that is only equivalent regarding satisfiability (and not necessarily regarding all values).

4.3.3 3-SAT

We defined satisfiability for the general class of all boolean formulas. Since this is a very important problem in practice, researchers try to find efficient ways to decide it at least for important sub-classes of boolean formulas. Unfortunately, it turns out that the satisfiability problem of already a quite restricted and simplified form of boolean formulas is NP-complete:

Definition 4.16 (CNF). *Let $X = \{x_1, \ldots, x_N\}$ be a set of variable names.*

- *A* literal *l is either a variable name x_i or the negation*[10] $\neg x_i$.
- *A disjunction $C = l_1 \vee \ldots \vee l_k$ of literals is called a* clause.
- *A conjunction $\varphi = C_1 \wedge \ldots \wedge C_m$ of clauses is a boolean formula in* conjunctive normal form *(CNF).*
- *The set of all CNF formulas is denoted with* CNFBOOL*:*

$$\mathsf{CNFBOOL} := \left\{ \bigwedge_{i=1}^{m} \bigvee_{j=1}^{k(i)} \sigma_{i,j} \mid \sigma_{i,j} \text{ are literals} \right\}$$

- *We use k-CNF for CNF formulas where the clauses only contain k literals (for fixed k) and denote the set of all k-CNF with k-*CNFBOOL*. Thus, we have:*

$$k\text{-}\mathsf{SAT} := \{\langle\varphi\rangle \mid \varphi \in k\text{-}\mathsf{CNFBOOL} \text{ is satisfiable}\}$$

Even though it is easy to see that 1-SAT $\in$ P and also 2-SAT $\in$ P (see Exercise 6), the situation is already different for 3-SAT:

Lemma 4.17. 3-SAT *is* NP*-complete.*

Proof. Since 3-SAT $\leq_m^p$ SAT via the identity function, we have 3-SAT $\in$ NP. This is a good example of the fact that a special case trivially reduces to the general case.

To further show NP-hardness, we will show CIRSAT $\leq_m^p$ 3-SAT. Let $c = (g_1, \ldots, g_m)$ be a boolean circuit with N inputs. We want to transform it into a 3-CNF formula ψ_c which should be satisfiable iff c is satisfiable. This formula will contain variables $x_1, \ldots, x_N$ representing the inputs of c, and additionally variables $y_1, \ldots, y_m$ representing the values at the gates of c. In particular, for each gate g_i, we create the following clauses:

Gate g_i	Clause	Semantics
$\bot$	$\{\overline{y_i}\}$	$y_i = 0$
$\top$	$\{y_i\}$	$y_i = 1$
x_j	$\{\overline{y_i}, x_j\}, \{\overline{x_j}, y_i\}$	$y_i \leftrightarrow x_j$
$(\neg, j)$	$\{\overline{y_i}, \overline{y_j}\}, \{y_i, y_j\}$	$y_i \leftrightarrow \overline{y_j}$
$(\wedge, j, k)$	$\{\overline{y_i}, y_j\}, \{\overline{y_i}, y_k\}, \{\overline{y_j}, \overline{y_k}, y_i\}$	$y_i \leftrightarrow (y_j \wedge y_k)$

Finally, we also add the clause $\{y_m\}$.

Using this construction, it is clear that c is satisfiable iff ψ_c is:

[10] We also write $\overline{x_i}$ as shorthand notation for $\neg x_i$.

- If there is $\alpha \in \{0,1\}^N$ with $f_c(\alpha) = 1$, then we use this α as an assignment for the variables $x_1, \ldots, x_N$. Further, we use the value $f_c^{(j)}(\alpha)$ at each gate g_j as an assignment to variable y_j. By construction, this makes all clauses true. In particular, $\{y_m\}$ will be true, since $f_c^{(m)}(\alpha) = f_c(\alpha) = 1$. Thus, ψ_c is satisfied.
- If $f_c(\alpha) = 0$ for all $\alpha \in \{0,1\}^N$, then $\{y_m\}$ will never be true, since by construction, the value of y_m corresponds to $f_c^{(m)}(\alpha)$. Thus, ψ_c can not be satisfied.

It follows, that $c \in \mathsf{CIRSAT} \iff \psi_c \in \mathsf{3\text{-}SAT}$, which is the desired reduction function, since ψ_c can be constructed efficiently. □

4.3.4 INDEPSET

We now turn to the class of *graph-theoretic problems*. The one we consider here, INDEPSET, is concerned with *undirected* graphs. Such a graph $G = (V, E)$ consists of a finite set of nodes V and a set of edges[11] $E \subseteq \binom{V}{2}$, where each edge is a set of two (different) nodes, connecting those two[12]. As with directed graphs in earlier examples, it is clear that efficient encodings over a fixed alphabet Σ exist.

One natural question to ask is, whether for a $k \in \mathbb{N}$ there is a subset $I \subseteq V$ of size k such that no two nodes of I are connected by an edge:

Definition 4.18. *The* independent set problem *is defined as follows:*

$$\mathsf{INDEPSET} := \left\{ (G, k) \mid \exists I \subseteq V(G) : \|I\| = k \wedge \binom{I}{2} \cap E(G) = \emptyset \right\}$$

Lemma 4.19. INDEPSET *is* NP*-complete.*

Proof. It is clear that $\mathsf{INDEPSET} \in \mathsf{NP}$, since a set I with the property of being an independent set of size k is a witness for a positive instance $(G, k) \in \mathsf{INDEPSET}$ and can be verified efficiently by checking for the existence of edges between its nodes.

To prove the NP-hardness, we provide a reduction $\mathsf{3\text{-}SAT} \leq_m^p \mathsf{INDEPSET}$. Given a 3-SAT-formula $\varphi = \bigwedge_{i=1}^k \left(\bigvee_{j=1}^3 \sigma_{i,j} \right)$ with k clauses of size 3, we construct a graph G such that it has an independent set of size k iff φ is satisfiable.

Each literal $\sigma_{i,j}$ is a node in the graph:

$$V := \{\sigma_{i,j}\}$$

[11] We write $\binom{A}{n}$ for the set of subsets of A with n elements.
[12] We also write $V(G)$ and $E(G)$ for the nodes and edges of G, respectively.

We will connect all literals in the same clause with an edge:

$$E_1 := \big\{\{\sigma_{i,j}, \sigma_{i,m}\} \mid i \in [1,k], j, m \in \{1,2,3\}\big\}$$

Further, we connect all literals that are complementary, i.e., all literals α_1 and α_2 with $\alpha_1 = x$ and $\alpha_2 = \neg x$ for some variable x.

$$E_2 := \big\{\{\sigma_{i,j}, \sigma_{l,m}\} \mid \sigma_{i,j} \text{ and } \sigma_{l,m} \text{ are complementary}\big\}$$

Figure 14 illustrates the construction.

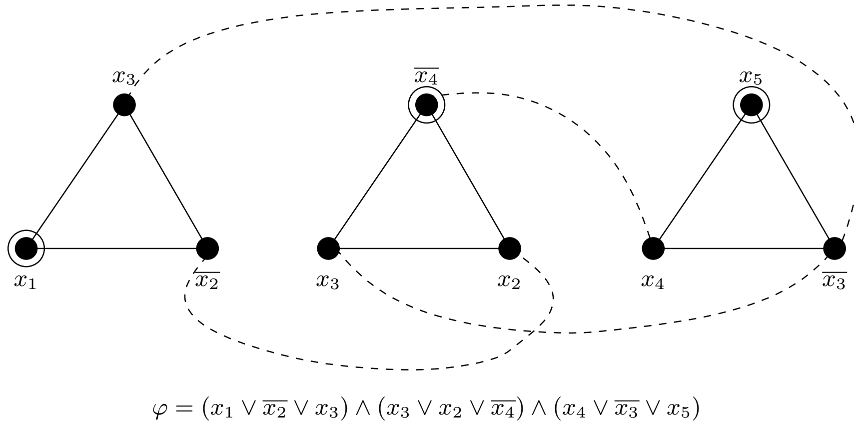


Figure 14: Example for the construction of graph $G = (V, E_1 \cup E_2)$ for given boolean 3-CNF formula φ; The triangles are the edges from E_1, connecting literals from the same clause. The dashed lines are the edges from E_2, connecting complementary literals. The marked nodes form an independent set, representing the (partial) truth value assignment $x_1 = x_5 = 1$, $x_4 = 0$ that leads to a satisfying assignment for φ.

To summarize, the reduction function can be expressed as follows:

$$f : \varphi \mapsto (\underbrace{(V, E_1 \cup E_2)}_{G}, k)$$

This can easily be computed in polynomial time.

The reduction property holds: If there is a satisfying truth assignment for the variables in φ, there is at least one literal in each clause true. If we chose one of those literals for each clause, then the corresponding nodes in the graph form an independent set of size k, since they are pairwise in different clauses (thus no connections from edges in E_1) and are not pairwise complementary (thus no connections from edges in E_2). On the other hand, if there is an independent set of size k in the resulting graph, no two of them can represent literals from the same clause in φ because of the edges in E_1. Further, no two can represent complementary literals because of the edges in E_2. Thus, there is a truth assignment satisfying all clauses of φ. □

4.3.5 More NP-complete Problems

Many other graph-theoretic problems can be shown NP-complete, using INDEPSET (or SAT) as a starting point (see also Exercise 7). We will define a selection of important ones.

Definition 4.20 (Clique Problem)**.**

- *A* k-clique *in a graph is a subset* $C \subseteq V$ *of size* k *of the nodes such that all nodes in* C *are pairwise connected.*
- *The* clique problem *is defined as follows:*

$$\mathsf{CLIQUE} := \left\{ (G,k) \mid \exists C \subseteq V(G) : \|C\| = k \wedge \binom{C}{2} \subseteq E \right\}$$

(Intuitive problem statement: "Is there a k*-clique in* G*?")*

Definition 4.21 (Node Cover Problem)**.**

- *A* k-node cover *in a graph is a subset* $N \subseteq V$ *of size* k *of the nodes such that of each edge in* E*, at least one end node is in* N*.*
- *The* node cover problem *is defined as follows:*

$$\mathsf{NODECOVER} := \{(G,k) \mid \exists N \subseteq V(G) : \|N\| = k \wedge \forall e \in E(G) : e \cap N \neq \emptyset\}$$

(Intuitive problem statement: "Is there a k*-node cover in* G*?")*

Definition 4.22 (Hamilton Path Problem)**.**

- *A* Hamilton path $p = (p_0, \ldots, p_k)$ *of length* k *in a graph* G *is a path that visits all nodes in* $V(G)$ *exactly once.*
- *The* Hamilton Path Problem *is defined as follows:*

$$\mathsf{HAMILTONPATH} := \{G \mid \exists p : p \textit{ is Hamilton path in } G\}$$

We also define some other examples, which are not directly graph-theoretic problems, but are related in the sense that they often can be expressed as graph problems – and in fact, this is often the key to proving their NP-completeness.

Definition 4.23 (Hitting Set Problem)**.**

- *A* k-hitting set *in a set* A *for a collection* $C = (C_1, \ldots, C_m)$ *of subsets of* A *is a set* H *of size* k *which contains at least one element of each* C_i*.*

- *The* hitting set problem *is defined as follows:*

$$\mathsf{HITTINGSET} := \{(A, C, k) \mid \exists H \subseteq A : \|H\| = k \wedge \forall C_i \in C : H \cap C_i \neq \emptyset\}$$

(Intuitive problem statement: "Is there a k-hitting set for C?")

Definition 4.24 (Travelling Salesman Problem)**.**

- *A* distance matrix D *of size n is a matrix with $n \times n$ over the natural numbers $\mathbb{N}_{\geq 0}$.*
- *A* permutation $\pi : [1, n] \to [1, n]$ *is an injective mapping of $[1, n]$ to itself, i.e., $i \neq j \implies \pi(i) \neq \pi(j)$.*
- *The* Travelling salesman problem *is defined as follows:*

$$\mathsf{TSP} := \left\{ (D, k) \mid \exists \pi : \sum_{i=1}^{n} D[\pi(i), \pi(i+1)] \leq k \right\}$$

(Intuitive problem statement: "Is there a route through the n cities of length at most k?")

Definition 4.25 (Knapsack problem)**.**

- *Let $V = (v_1, \ldots, v_n)$ be* values *and $W = (w_1, \ldots, w_n)$* weights*, all natural numbers.*
- *The* Knapsack problem *is defined as follows:*

$$\mathsf{KNAPSACK} := \left\{ (V, W, l, m) \mid \exists S \subseteq [1, n] : \sum_{i \in S} w_i \leq l \wedge \sum_{i \in S} v_i \geq m \right\}$$

(Intuitive problem statement: "Is there a selection S of the n items, such that there total weight is at most the limit l and their total value is at least m?")

Definition 4.26 (Integer Linear Programming)**.**

- *Let A be a matrix of size $n \times n$ with integer coefficients and b a vector with n integers:*

$$A \in \mathbb{Z}^{n \times n} \quad \wedge \quad b \in \mathbb{Z}^n$$

- *The* Integer linear programming problem *is defined as follows:*

$$\mathsf{ILP} := \{(A, b) \mid \exists x \in \mathbb{Z}^n : Ax \leq b\}$$

(Intuitive problem statement: "Is there an integer solution to the system of linear inequalities represented by A and b?")

Definition 4.27 (Bin Packing)**.**

- *Let* $A = (a_1, \ldots, a_n) \in \mathbb{Z}^n$ *denote the* item sizes *and* $b, c \in \mathbb{Z}$ *the* number of bins *and the* capacity*.*
- *The* bin packing problem *is defined as follows:*

$$\mathsf{BINPACK} := \left\{ (A, b, c) \mid \exists \text{ partition } S_1, \ldots, S_b \text{ of } [1, n] \text{ s.t. } \forall i : \sum_{j \in S_i} a_j \leq c \right\}$$

(Intuitive problem statement: "Given b bins of capacity c and n items with sizes given by A, is it possible to pack all items into the bins?")

Lemma 4.28. *The problems* CLIQUE*,* NODECOVER*,* HITTINGSET*,* TSP*,* HAMILTONPATH*,* KNAPSACK*,* ILP *and* BINPACK *are* NP*-complete.*

(Without proof.)

4.4 Beyond NP-completeness

We now explored a variety of natural problems with practical importance that are NP-complete. In this last section of the current chapter, we shortly state some related results and concepts.

4.4.1 Between P and NPC

After so many NP-complete problems, one might get the impression that each problem in NP is either in P or is NP-complete (if $\mathsf{P} \neq \mathsf{NP}$, otherwise, the classes are the same anyway). Surprisingly, this is not the case: If $\mathsf{P} \neq \mathsf{NP}$, then there are problems *between* the classes: They are "strictly easier" than the NP-complete ones, but still "too difficult" to be efficiently solved in polynomial time:

Lemma 4.29. *If* $\mathsf{P} \neq \mathsf{NP}$*, then there is a language* $L \in \mathsf{NP} - (\mathsf{P} \cup \mathsf{NPC})$.

(Without proof.)

4.4.2 Pseudo-polynomial complexity

An important observation is, that the property of NP-completeness is often quite dependent on the precise formulation and representation of the problem. It is for example crucial, that the *integer linear programming problem* ILP is restricted to integers. It might look like a special case of LP, the *linear programming problem* for rational numbers, but in fact, $\mathsf{LP} \in \mathsf{P}$. Thus, this "general case" is easier (and in that sense not a generalization).

Apart from formulation details, also the *representation* of problem instances plays an important role. Consider for example the *Knapsack problem* KNAPSACK which we claimed to be NP-complete. While this is true (even though we did not provide a proof), there is an algorithm based on dynamic programming, that can solve the problem in time $\mathcal{O}(n \cdot l)$ given n items and

the weight limit l. This is in fact only linearly related to the weight limit l and the number of items n. It still is not a polynomial algorithm because $n \cdot l$ is not polynomial in the *size of the input* which is $\mathcal{O}(n \cdot \log(l))$. This is because even though we are given n items, the weight limit is expected to be given in a binary (or other k-ary) representation. Therefore, such an algorithm is called *pseudo-polynomial*. This subtle difference is often a source of confusion.

If a problem remains NP-complete even if instances of size n are only allowed to contain values bounded by a polynomial $p(n)$, then we call them *strongly* NP*-complete*. These can not even be solved by a pseudo-polynomial algorithm (unless $\mathsf{P} = \mathsf{NP}$). In fact, all problems we presented – except KNAPSACK – are strongly NP-complete.

4.4.3 Unknown relations

We saw an example of an co- NP-complete problem: UNSAT, the set of all unsatisfiable boolean formulas. Since co- NP is the "symmetric cousin" of NP, the problems that are co- NP-complete can be regarded as equally difficult as the NP-complete ones. But how do NP and co- NP actually relate to each other?

It is unknown whether NP is closed under taking complements, i.e., whether $\mathsf{NP} = \text{co-}\,\mathsf{NP}$, even though that would be implied by $\mathsf{P} = \mathsf{NP}$. Thus, $\mathsf{NP} \neq \text{co-}\,\mathsf{NP}$ is stronger than the conjectured $\mathsf{P} \neq \mathsf{NP}$, but it is actually also widely believed. The intuitive notion of this conjecture is, that there are problems with efficiently verifiable proofs that don't have efficiently verifiable disproofs. Further, even $\mathsf{P} \subsetneq \mathsf{NP} \cap \text{co-}\,\mathsf{NP}$ is not known, i.e., whether there are problems with efficient proofs and disproofs that can not be solved efficiently.

At the "upper end", it is also unknown whether $\mathsf{NP} \subsetneq \mathsf{PSPACE}$, i.e., whether there are problems (deterministically) solvable with polynomial space requirements, that don't have efficiently verifiable witnesses. Actually, even $\mathsf{P} \subsetneq \mathsf{PSPACE}$ is unknown.

As a last open question in this chapter, we wish to briefly discuss the class of languages between P and NPC. Since $\mathsf{P} \neq \mathsf{NP}$ is not known, no language is actually proven to be within $\mathsf{NP} - (\mathsf{P} \cup \mathsf{NPC})$. There are a few natural languages which still are candidates for this, i.e., no one has yet discovered a polynomial time algorithm or has established a NP-completeness proof. One of these candidates was the problem PRIMES of deciding whether a number is a prime number, as introduced in Example 2.2. (It is actually also clearly in co- NP and thus was a candidate for showing $\mathsf{P} \subsetneq \mathsf{NP} \cap \text{co-}\,\mathsf{NP}$.) Though, in a recent result from 2002, Agrawal, Kayal, and Saxena proved elegantly that $\mathsf{PRIMES} \in \mathsf{P}$. Thus, there is one candidate less. Another prominent candidate that remains, is the *graph isomorphism problem* GI. It

states the task, given two graphs G_1 and G_2, to decide whether there is an isomorphism between them, i.e., if they "look" the same:

$$\mathsf{GI} := \{(G_1, G_2) \mid \exists \pi : (e \in E(G_1) \iff \pi(e) \in E(G_2))\}$$

This problem is of high practical relevance, and efficient approximations and solutions for subproblems have been intensively studied (e.g. for trees or planar graphs). Also in complexity theory, this general problem has gained so much attention, that researchers defined a whole new complexity class (also with the name GI) with all problems that can be reduced to the graph isomorphism problem.

5 Advanced Complexity Concepts

In this last chapter, we will widen the view to other concepts that were developed withing complexity theory and proved useful for gaining new insights.

5.1 Non-uniform Complexity

In our computation model, the Turing machine, we so far always assumed *one fixed* machine of finite size to be able to handle problem instances of arbitrary size. This is called a *uniform* model of computation, a "one size fits it all" approach. There are natural situations, in which one wants to model a device which has more "hardwired information" accessible when the size of the input grows. Consider for example a cryptographic scheme in which the security scales with the size of some key and which is supposed to be proven secure against any attacker. One may want to model the attacker to be prepared in a way that he might have some fixed precomputed knowledge available for each key size he encounters. Even against such an attacker, one wants the scheme to be proven secure. The model also gives a new approach to the $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$ problem.

To capture this notion of *non-uniformity*, we introduce the concept of *advice*. An advice string a_n is a string that a machine, given an input of size n, will be given as an additional input.

Definition 5.1 (Turing Machine with Advice)**.**

- *A tuple* $M = (Q, \Gamma, \delta, q_0, F, A)$ *with* $Q, \Gamma, \delta, q_0, F$ *as defined before (see Definition 2.3) and* $A = \{a_n\}_{n \geq 0}$ *is a* Turing machine with advice*.*
- *The set a* A *is called the* advice*.*
- *The language over an alphabet* $\Sigma \subseteq \Gamma$ *accepted by* M *is defined as:*

$$L(M) := \{x \in \Sigma^* \mid \exists y, z \in \Gamma^* : (\varepsilon, q_0, x\#a_{|x|}) \vdash^* (y, q_{yes}, z)$$

(We assume $\# \in \Gamma$ *to be some dedicated separation symbol.)*

Note that the Turing machine itself is a *finite* object, i.e., a finite representation of the algorithm. The advice however can be *infinite*: For each n, a_n can be completely different, supplying more information to the machine for its decision. Therefore, it can't be encoded directly into the machines states, it is external information.

Note also the difference to the concept of a *witness* we had in the previous chapter to characterize NP: A witness is (possibly) a different string w for each input x. Recall that the property is that (at least) one w exists for each $x \in L$ and none exist for $x \notin L$. In contrast to this, an advice is *static*

for each n. This means that even though different n may result in different a_n, but for each length, a_n is *fixed*. Thus, for all inputs x of size n, M gets *the same* a_n as advice.

We note further that without additional restrictions, this is a very powerful concept: For any arbitrary language L (even an undecidable one!), we could construct a TM with advice deciding that language by encoding into each a_n a table with information about all possible input words of size n. The machine then would only have to look up the input word in a_n and halt with the encoded answer it found in the table.

To get a more reasonable model, we impose restrictions on the size of the advice. If a_n is for example bounded by a polynomial $p(n)$, the above construction would not be possible anymore. This gives rise to a new language class:

Definition 5.2. $\mathsf{P/poly}$ *is the set of all languages L such that a Turing machine M with advice A exists, such that* $\forall n : |a_n| \leq p(n)$ *for some polynomial $p(n)$.*

This can be seen as *non-uniform* polynomial time complexity, and it is clear that it is at least as powerful as the uniform variant: $\mathsf{P} \subseteq \mathsf{P/poly}$ since for P, there is no advice necessary. Because of this inclusion, it would be sufficient to have a language $L \in \mathsf{NP}$ which is not in $\mathsf{P/poly}$, for proving $\mathsf{P} \neq \mathsf{NP}$. An intuition of why this might be more difficult than it seems at the first sight, is the following result:

Lemma 5.3. *Even* $\mathsf{P/poly}$ *contains undecidable problems.*

Proof (Sketch). This can be shown by first demonstrating the existence of undecidable languages over an alphabet with just one symbol, so called *unary languages*. This is immediately clear by encoding the instances of the Halting Problem in unary. Second, one can show that each unary language L is in $\mathsf{P/poly}$, since the advice a_n for each length only needs to indicate, whether the only possible unary word with that size is in L or not. □

Still, researchers have been able to show $\mathsf{NP} - \mathsf{P/poly} \neq \emptyset$ under reasonable assumptions.

We note that there is another characterization of that class in terms of *circuits*. Recall (from Definition 4.12) that a boolean circuit consists of binary inputs and gates and delivers one output bit. This can be also used as a model of computation. Let C_n be a boolean circuit with n inputs, then we can define its accepted language as follows:

$$L(C_n) := \{\alpha \in \{0,1\}^n \mid f_{C_n}(\alpha) = 1\}$$

Since one circuit can only deal with words of a fixed size[13], the computation model is extended to a *circuit family* $C = \{C_n\}_{n \geq 0}$, supplying a different circuit for each input size. Thus, the accepted language is:

$$L(C) := \{\alpha \in L(C_{|\alpha|})\}$$

Intuitively, this also captures the notion of *non-uniformity*: For each input size, a possibly completely new device is supplied for solving a given problem. A circuit family does not need to have a finite representation, in contrast to a Turing machine. As with the advice, the model is very powerful if it is not restricted. Thus, one considers a size bound $\text{size}(C_n)$ on the circuits C_n, which is defined via the number of their gates. This gives rise to a class of languages:

$$\mathsf{DSIZE}(s(n)) := \{L(C) \mid \forall n \geq 0 : \text{size}(C_n) \leq s(n)\}$$

It is known, that a language decidable in $t(n)$ time can be decided by circuits of size $\mathcal{O}(t(n)^2)$:

Lemma 5.4. $\mathsf{DTIME}(t(n)) \subseteq \mathsf{DSIZE}(\mathcal{O}(t(n)^2))$ *(Without proof.)*

Finally, we can characterize $\mathsf{P/poly}$ as the set of languages with polynomial circuits:

Lemma 5.5. $\mathsf{P/poly} = \mathsf{DSIZE}(n^{O(1)})$

Proof (Sketch). First, let $L \in \mathsf{P/poly}$ decided by a Turing machine with advice M. The construction of a circuit family C works by constructing for each n a circuit C_n that captures the actions of M given an input of size n. This works thanks to Lemma 5.4 from above with only quadratic growth. Further, the advice a_n is only polynomially big, so it can easily be incorporated into the construction.

Second, let $L \in \mathsf{DSIZE}(n^{O(1)})$ decided by a circuit family C. One can use a Turing machine that evaluates a circuit in polynomial time, and give it a description of the circuit family as advice. □

5.2 Probabilistic Complexity Classes

The second advanced concept we want to look at in this chapter deals with *randomized computations*: So far, the computation models that we used had the property, that for each input x, there was one single answer, one clear outcome of the computation. This even holds for the non-deterministic computations, in the light of their witness-based characterization: Either there exists a witness, or it does not – but in either case, provided a "potential"

[13] The alphabet is also fixed to $\Sigma = \{0, 1\}$, but this is not a practical problem, since – as all modern computers do – other alphabets can be easily encoded into binary form.

witness, the verification procedure is deterministic, producing always the same answer. What we do now is to relax this requirement: At each step, we allow the machine to *randomly* choose between different possibilities. One can imagine this as *tossing a coin*: with equal probability, "heads" or "tails" will occur, which will guide the next step of the machine. This introduces *uncertainty* into the answer of the machine: with the same input x, it will sometimes classify x as a positive instance, and sometimes as a negative one.

But how can we use this kind of machine in order to obtain information about an instance? The difference between positive and negative instances will be in the probability with which the machine will end up in the final states q_{yes} and q_{no}. For example, if we have a machine with the property that for $x \in L$, the vast majority of all possible coin toss outcomes leads to q_{yes}, and for $x \notin L$ they lead to q_{no}, then just one run of the machine has a notable tendency to give the right answer – even though sometimes, it might be wrong. Depending on the application scenario, this could be enough: Imagine a cryptographic attack, where the attacker is already happy if he can recover some secret just with a certain probability.

Syntactically, a probabilistic Turing machine looks the same as a non-deterministic one: the transition function δ may offer a variety of choices. (For simplicity and without loosing generality, we may assume that there are always at most two choices.) The difference to the classical non-deterministic definition is how we define the accepted language. Classically, the question was whether there is *at least one* computation leading to q_{yes}. In the probabilistic setting, it matters *how many* there are. This is also nicely illustrated in terms of the computation tree, see Figure 15. Note that this is a *realistic* model of computation, compared to the quite *theoretical* nature of the classical non-deterministic machine.

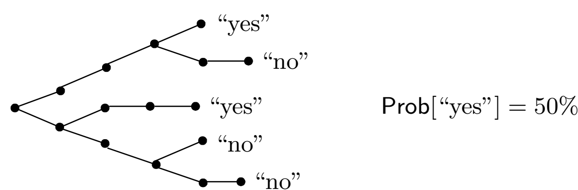


Figure 15: A computation tree: In the probabilistic model, each branch is a "coin toss" with equal probability for both possible steps. The acceptance probability is the probability of reaching an accepting state, in the example 50%.

5.2.1 Notation

Our notation is as follows: For a language $L \subseteq \Sigma^*$, we let $\chi_L : \Sigma^* \to \{0, 1\}$ be the *characteristic function* of L, i.e.:

$$\chi_L(x) := \begin{cases} 1 & \text{if } x \in L \\ 0 & \text{if } x \notin L \end{cases}$$

Further, we let a machine M output 1 if it halts in q_{yes} and 0 if it halts in q_{no}. (So we can assume a generic q_{halt} final state.) This allows expressions like "$M(x) = \chi_L(x)$" to describe the behaviour of a machine M that deterministically decides L. (This is a quite general notation that can be applied throughout all of the previous chapters.)

To also include the probabilistic behaviour into the notation, we may regard the *coin tosses* as one string $r \in \{0, 1\}^*$ that is given to the machine as an auxiliary input, which is a second argument in the notation. This allows for notation like:

$$\mathsf{Prob}_r[M(x, r) = 1]$$

If it is clear that we take the probability over r and talk about a probabilistic M, we also simply write:

$$\mathsf{Prob}[M(x) = 1]$$

5.2.2 One-sided Error

We first allow the machine to only have a *one-sided error*. This means that for positive instances, it might give a wrong answer (0 instead of 1), as long as the probability for this is low. For negative answers, we demand the machine to always answer correctly, i.e., with a 0. Languages for which efficient machines with such a behaviour exist are grouped into the following complexity class:

Definition 5.6. *The class* RP *contains all languages* L *for which a polynomially time-bounded, probabilistic Turing machine* M *exists, such that:*

$$\begin{aligned} x \in L &\implies \mathsf{Prob}[M(x) = 1] \geq 1/2 \\ x \notin L &\implies \mathsf{Prob}[M(x) = 1] = 0 \end{aligned}$$

While RP allows the one-sided error for positive instances, we have symmetrically that $\mathsf{co\text{-}RP}$ allows a one-sided error for negative instances.

It is immediately clear, that $\mathsf{P} \subseteq \mathsf{RP}$, since a deterministic machine is a special case of a probabilistic one (discarding the coin tosses). Further, we have $\mathsf{RP} \subseteq \mathsf{NP}$ as already discussed. This can also be compared in terms of the *witness relation*: recall that NP languages L can be characterized by a polynomially bounded relation R_L such that for each $x \in L$, there is a w with $(x, w) \in R_L$. In these terms, the coin tosses leading to acceptance can

also be written as a relation R_L such that for each $x \in L$, at least half of all r have $(x, r) \in R_L$. In short:

	NP	RP
$x \in L$:	$\exists w : (x, w) \in R_L$	$\mathsf{Prob}_r[(x, r) \in R_L] \geq 1/2$
$x \notin L$:	$\forall w : (x, w) \notin R_L$	$\forall r : (x, r) \notin R_L$

An interesting question is, whether expressiveness is invariant under changes of the constant 1/2 in the definition. Consider a machine M that fulfills all requirements, but can only guarantee a success rate of 1/3 for all positive instances. It turns out, that one can transform M into a machine M' which will again have a 1/2 constant: M' will simulate M twice and output 1 exactly if one of the simulations did. Otherwise it outputs 0. For an input $x \notin L$, both simulations of M will output 0, so M' also does. For an input $x \in L$, the probability for each of the simulations to output 0 is at most 2/3. Thus, it is $(2/3)^2 = 4/9$ that both will, which implies that M' will output 1 with probability $5/9 > 1/2$.

In fact, this robustness can not just be expanded to all other constants strictly larger than 0, it works even if the acceptance ratio depends polynomially on the size of n. The construction for that is similar to the above, but uses polynomially many runs of the original machine.

Lemma 5.7. *If there is a polynomially time-bounded probabilistic TM M for a language L and a polynomial $p(n)$ such that the following holds:*

$$x \in L \implies \mathsf{Prob}[M(x) = 1] \geq 1/p(|x|)$$
$$x \notin L \implies \mathsf{Prob}[M(x) = 1] = 0$$

Then $L \in \mathsf{RP}$. *(Without proof.)*

This very low acceptance probability can not just be "boosted" back to 1/2, it can even be brought very close to 1 using basically the same technique:

Lemma 5.8. *For each $L \in \mathsf{RP}$ and each polynomial $p(n)$, there is a polynomially time-bounded probabilistic TM M such that:*

$$x \in L \implies \mathsf{Prob}[M(x) = 1] \geq 1 - 2^{-p(|x|)}$$
$$x \notin L \implies \mathsf{Prob}[M(x) = 1] = 0$$

(Without proof.)

The downside of this *probability amplification* (for both lemmas) is, that the running time increases polynomially, which is formally fine, but might turn out to be a problem in practice. On the positive side, this amplification may increase the chance of being right to "almost always".

5.2.3 Two-sided Error

Now, we allow the machine to be wrong for positive *and* negative instances, with a probability of 1/3:

Definition 5.9. *The class* BPP *contains all languages L for which a polynomially time-bounded, probabilistic Turing machine M exists, such that:*

$$\forall x \in L : \mathsf{Prob}[M(x) = \chi_L(x)] \geq 2/3$$

Using the notation before, we may express this as:

$$\begin{aligned} x \in L &\implies \mathsf{Prob}[M(x) = 1] \geq 2/3 \\ x \notin L &\implies \mathsf{Prob}[M(x) = 1] < 1/3 \end{aligned}$$

We notice that this class is closed under taking complements, i.e., BPP = co- BPP, since the definition is symmetric. Just as with RP, the constant 2/3 is arbitrary and can be replaced. We notice that there are actually two constants involved: One is a probability such that the right answers are strictly more likely (1/2 in this example), and the other one is the "gap" between the probabilities for right and wrong answers (1/6 in this example). Both may be chosen quite freely, even depending on $|x|$:

Lemma 5.10. *If there is a polynomially time-bounded probabilistic TM M for a language L, a polynomial $p(n)$ and a computable function $f(n)$ such that the following holds:*

$$\begin{aligned} x \in L &\implies \mathsf{Prob}[M(x) = 1] \geq f(|x|) + 1/p(|x|) \\ x \notin L &\implies \mathsf{Prob}[M(x) = 1] < f(|x|) - 1/p(|x|) \end{aligned}$$

Then $L \in$ BPP. *(Without proof.)*

Again, this can be "boosted" quite strongly:

Lemma 5.11. *For each $L \in$ BPP and each polynomial $p(n)$, there is a polynomially time-bounded probabilistic TM M such that:*

$$\forall x \in L : \mathsf{Prob}[M(x) = \chi_L(x)] \geq 1 - 2^{-p(|x|)}$$

(Without proof.)

Clearly, RP $\subseteq$ BPP, since a one-sided error is just a special case of a two-sided one. For BPP, the relation to NP is actually unknown. But it is important to note that the probability of wrong answers for languages in $L \in$ BPP can be made exponentially small. Thus, BPP has actually widely replaced P in representing the intuitive notion of "efficiently solvable problems". Indeed, it is widely believed that P = BPP, but there is no proof.

5.2.4 Monte Carlo vs. Las Vegas

The machines considered so far always answered, but the answers might be wrong sometimes. Algorithms with this property are called *Monte Carlo* algorithms. In contrast to that, *Las Vegas* algorithms always answer right, but might sometimes, i.e., with a certain but not too high probability, fail to deliver an answer.

We let a machine denote with "$\perp$" the answer "I don't know", in addition to the positive and negative answers 1 and 0. This is used in the following definition:

Definition 5.12. *The class* ZPP *contains all languages* L *for which a polynomially time-bounded, probabilistic Turing machine* M *exists, such that:*

- $\forall x \in L : \mathsf{Prob}[M(x) = \perp] \leq 1/2$*, and*
- $\forall x \in L, r : M(x, r) \neq \perp \implies M(x, r) = \chi_L(x)$

To conclude this section about probabilistic complexity, we state some known relations between the classes:

Lemma 5.13.

1. $\mathsf{P} \subseteq \mathsf{ZPP} \subseteq \mathsf{RP} \subseteq \mathsf{BPP}$
2. $\mathsf{ZPP} = \mathsf{RP} \cap \mathsf{co\text{-}RP}$
3. $\mathsf{BPP} \subseteq \mathsf{P/poly}$
4. $\mathsf{BPP} = \mathsf{P}$ *if pseudo random number generators exist.*
   (Efficient derandomization)

(Without proof.)

The last part of the lemma is concerned with a branch within complexity theory that deals with *derandomization*. The idea here is to simulate "truly" random choices using "artificial" randomness that is created from a small initial "seed" of random input. Devices for doing that are called *pseudo random number generators* (PRNGs). Their output is not really random, but it *looks* random. This means that no efficient observer is able to tell the difference, or to predict future output from past output. (Note that this is a quite strict and rather theoretical notion and should not be confused with PRNGs you encounter in practice nowadays.) Under certain assumptions, one can show that these generators of different strength actually exist, and they can even be used to completely remove the dependence on a random source, thus making the decision procedure deterministic.

5.3 Interactive Proof Systems

We start this last section with the notion of a *proof*: Involved in a proof are usually two entities, the *prover* and the *verifier*. In mathematics, a proof for an assertion is a sequence of steps and intermediate claims, written down by a prover and to be checked by a verifier. The writer of the proof wants to convince the reader of the assertion's validity. In general, a proof is more than that: prover and verifier may want to *interact*, involving a dialog with questions over questions by the verifier, carefully chosen and possibly adapted to earlier answers given by the prover to clarify or to point out inconsistencies in the proof. Eventually, if the assertion is actually true and the verifier has been convinced, the process will end. However, a careful verifier will only be convinced by such a process, if the assertion indeed holds – he will refuse to believe what he is being told otherwise.

Interactive proof systems are a formalisation of this intuitive concept. We can interpret the class NP as the restrictive, *non-interactive* version: The prover supplies his proof (we called it "witness") to the verifier, that can check it and has to do so efficiently. There is no interaction allowed, the verifier can not ask additional questions. We call this an NP *proof system*. This concept will be generalized now: An *interactive proof system* is composed of two machines: a prover, and a verifier. Given an input x, they start an interaction after which the verifier either accepts or rejects the input, see Figure 16.

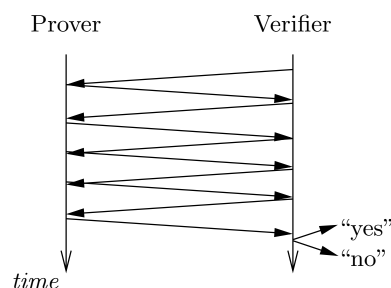


Figure 16: A prover and a verifier interact during an interactive proof.

Such a proof system should have the following properties:

1. Efficiency of the verifier.
2. Correctness requirement:

   Completeness: Each true assertion has a convincing proof strategy.

   Soundness: No false assertion has a convincing proof strategy.

Clearly, NP proof systems have this property: The witness relation R_L is decidable in polynomial time, so the verifier works efficiently. The correctness is ensured by the fact that exactly for the positive problem instances, there is a witness. Our generalized model of an interactive proof system should also adhere to these properties.

We now provide a formal definition for this concept. It is based on the notion of an *interactive Turing machine*: This is a Turing machine with an extra *communication tape* and two extra *communication states* $q_?$ and $q_!$. Two of those machines M_1, M_2 can be composed to $\langle M_1, M_2 \rangle$ such that they share the communication tape. M_1 starts in q_0 and M_2 in $q_?$, waiting for M_1 to hand over control. After M_1 did some computations and possibly wrote a message to the communication tape, it hands over control to M_2 by switching to $q_?$. This switches M_2 to $q_!$ and lets M_2 do computations. The machines then hand the control over back and forth, until finally, M_2 halts. Its output tape contains then the result of the interaction denoted with $\langle M_1, M_2 \rangle(x)$ if the common input was x. The period between two control switches is called a *round*. Since we are working on decision problems, we assume – as above – output 1 for a positive answer "$x \in L$" and 0 for a negative answer "$x \notin L$". This model is now used as follows:

Definition 5.14 (Interactive Proof System)**.** *Let L be a language. An* interactive proof system *for L is a pair $\langle P, V \rangle$ of interactive Turing machines with the following properties:*

1. *V is probabilistic and polynomially time-bounded.*
2. *Correctness:*

   Completeness: $\forall x \in L : \mathsf{Prob}[\langle P, V \rangle(x) = 1] \geq 2/3$

   Soundness: $\forall x \notin L : \forall P^* : \mathsf{Prob}[\langle P^*, V \rangle(x) = 1] < 1/3$

We note that the prover is not bounded computationally. We further note that for negative answers, we quantify over *all* prover strategies, i.e., also over all dishonest ones. This model closely resembles requirements used in cryptology, and it is widely used as a standard model for exploring properties of cryptographic protocols. We use it here for the sake of complexity theory and thus create a complexity class for all languages that can be decided with this model:

Definition 5.15 (IP hierarchy)**.**

- *Let $r : \mathbb{N} \to \mathbb{N}$. The class $\mathsf{IP}(r(n))$ contains all languages L for which an interactive proof system $\langle P, V \rangle$ exists such that on common input x, at most $r(|x|)$ rounds are used.*

- *The class* IP *contains all languages* L *having interactive proof systems:*

$$\mathsf{IP} := \bigcup_r \mathsf{IP}(r(n))$$

We note that clearly we have $\mathsf{NP} \subseteq \mathsf{IP}$, since that is the special case of just one round: the prover writes the witness on the tape, hands over control to the verifier, which proceeds (even deterministically).

Indeed, not just the introduction of interaction to this concept, but also of randomness is essential: If the verifier V were to be deterministic, we could construct a new prover P' which would know each query of the verifier beforehand (because it could simulate V). Thus, it would only have to send the final message to the verifier which then would – deterministically – run a check to produce the output. This is an NP proof system, so without randomization, IP would clearly be the same as NP.

We note further that we have $\mathsf{IP} = \mathsf{IP}(n^{\mathcal{O}(1)})$, i.e., there are only polynomially many rounds, since the verifier is time-bounded by a polynomial. Also, just as for BPP, it can be shown that the error bounds can be made very small:

Lemma 5.16. *For each* $L \in \mathsf{IP}$ *and each polynomial* $p(n)$*, there is an interactive proof system* $\langle P, V \rangle$ *such that the correctness properties satisfy:*

Completeness: $\forall x \in L : \mathsf{Prob}[\langle P, V \rangle(x) = 1] \geq 1 - 2^{-p(|x|)}$

Soundness: $\forall x \notin L : \forall P^* : \mathsf{Prob}[\langle P^*, V \rangle(x) = 1] < 2^{-p(|x|)}$

(Without proof.)

This, again, comes with the price of a polynomially increased runtime. In particular, either the number of rounds or the size of the messages increases rapidly.

We now provide an example problem that is known to be decidable by an interactive proof system. It is GNI, the *graph non-isomorphism problem*, which is the complement of the *graph isomorphism problem*, and thus contains all pairs of graphs which are not isomorphic to each other:

$$\mathsf{GNI} := \{(G_1, G_2) \mid \neg\exists\pi : (e \in E(G_1) \iff \pi(e) \in E(G_2))\}$$

Theorem 5.17. $\mathsf{GNI} \in \mathsf{IP}$.

Proof (Sketch). The idea of the proof is that any "shuffling" of one of the two graphs G_1 and G_2, i.e., a random permutation of its nodes, produces a new graph H that is isomorphic to *exactly one* of the two, if G_1 and G_2 are not isomorphic. On the other hand, if they are isomorphic, such a shuffling will always be isomorphic to both. Thus, the verifier can choose one of G_1 and G_2 randomly, let's say G_i, $i \in \{1, 2\}$. It also chooses a random

permutation π over the nodes and creates $H := \pi(G_i)$. It then sends H to the prover P and asks, which of the given graphs it used for the shuffling. Finally, P supplies the answer for that and V checks if the answer is correct.

This protocol has the desired properties: First, the verifier is *efficient*, since it is only randomly choosing a permutation and applying it to a graph. Second, the *completeness* property is satisfied, since we can construct a P that finds out the right answer if indeed G_1 and G_2 are not isomorphic (recall that P is not bounded). Third, also the *soundness* property is satisfied, since when G_1 and G_2 are isomorphic, whatever a prover is trying to do, its chance to guess the correct i is just $1/2$ since i was randomly chosen and can not be derived from H.

Note that the soundness bound is only $1/2$, even though in the definition we demanded $1/3$. This is easily fixed by conducting the protocol twice (sequentially or in parallel) and only accepting if the prover was right both times. By this simple probability amplification, the soundness bound drops to $1/4$. □

We note that there are only two rounds involved in the above protocol, thus $\mathsf{GNI} \in \mathsf{IP}(2)$. This gives an indication that IP is a quite strong class, since GNI (which is in $\mathsf{co\text{-}NP}$) is not known to be in P or NP. Indeed, the following relations can be shown:

Theorem 5.18.

- $(\mathsf{NP} \cup \mathsf{co\text{-}NP}) \subseteq \mathsf{IP}$
- $\mathsf{IP} = \mathsf{PSPACE}$

(Without proof.)

Note that the first one is weaker version of the second relation. It can be shown by constructing an interactive proof system for $\overline{\mathsf{3\text{-}SAT}}$, which is $\mathsf{co\text{-}NP}$-complete (since $\mathsf{3\text{-}SAT}$ is NP-complete).

There are a couple of extensions to the concept of interactive proof systems. The perhaps most important one, which is also widely used in cryptography, are the *zero knowledge proofs*. These are special variants where it is guaranteed that the prover does not provide “knowledge” to the verifier. It is surely important to define what knowledge actually is, but one can see it as information that can not be efficiently computed by the verifier itself. Thus, a prover could for example prove its identity to the verifier by convincing him that he possesses some secret information that only he has (like an isomorphism between two very big graphs or the factorization of two very big numbers). The properties of interactive proof systems will allow the verifier to get convinced that the prover actually has this information and is therefore who he claims to be. This alone is not enough for a desired cryptographic protocol, since the prover also does not want anyone

to “steal” his identity: he could easily prove the possession of the secret by telling it, but this would enable the verifier to later act like he was the prover towards third parties. The problem is that he *gained knowledge*. Thus, the zero knowledge property is an additional guarantee: It protects the verifier from this “theft”. Unfortunately, going into the details of this is out of scope for us and material for another course.

A Appendix

A.1 Exercises

Exercise 1. Let Σ be an alphabet ordered under "$<$". We define a *lexicographical order* $<$ on Σ^* as follows: We have $x < y$ if:

- $|x| < |y|$ or
- $|x| = |y|$ and $\exists i \leq |x| : x_1 \dots x_{i-1} = y_1 \dots y_{i-1} \wedge x_i < y_i$

Further, we call a function $f : \Gamma^* \to \Sigma^*$ *monotonic*, if $f(x) \leq f(y)$ for all $x \leq y$. A language L is *recursively enumerable in lexicographical order*, if it is the image[14] of a monotonic, computable function. Show that a language L is decidable if and only if it is either recursively enumerable in lexicographical order or empty.

Exercise 2. Show the last two properties from Lemma 2.16, i.e., show that for all A, B, C:

1. $A \leq_m B \wedge B \leq_m C \Longrightarrow A \leq_m C$ (Transitivity of $\leq_m$)
2. $A \leq_m B \iff \overline{A} \leq_m \overline{B}$

Exercise 3. Show that the following language is either inside or outside one or both of the classes REC and RE:

$$Eq := \{(\langle M_1 \rangle, \langle M_2 \rangle) \mid L(M_1) = L(M_2)\}$$

What about $\overline{Eq}$?

Exercise 4. Show Lemma 4.7, i.e., show that for all A, B, C:

1. $A \leq_m^p B \wedge B \in \mathsf{P} \Longrightarrow A \in \mathsf{P}$ (Closedness of P under $\leq_m^p$)
2. $A \leq_m^p B \wedge B \in \mathsf{NP} \Longrightarrow A \in \mathsf{NP}$ (Closedness of NP under $\leq_m^p$)
3. $A \leq_m^p B \wedge B \leq_m^p C \Longrightarrow A \leq_m^p C$ (Transitivity of $\leq_m^p$)
4. $A \leq_m^p B \iff \overline{A} \leq_m^p \overline{B}$

Exercise 5. We defined $\leq_m^p$ to be the *polynomial time reduction* and showed that there are NP-complete problems under $\leq_m^p$ reduction. One can also show that there are P-complete problems under *logarithmic space reduction* $\leq_m^l$. Let $\leq_m^{lin}$ be *linear time reduction*, i.e., defined like $\leq_m^p$ but the reduction function is now restricted to linear time. Show that there is no P-complete problem under $\leq_m^{lin}$, i.e., that there is no problem $A \in \mathsf{P}$ such that $\forall L \in \mathsf{P} : L \leq_m^{lin} A$. *(Hint: Hierarchy Theorem)*

[14] The *image* of a function $f : X \to Y$ is the set $f(X) = \{f(x) \mid x \in X\}$

Exercise 6. Show that 1-SAT $\in$ P and 2-SAT $\in$ P. *(Hint: For* 2-SAT*, write each clause $l_1 \vee l_2$ as an implication and interpret the set of all those implications as edges in a directed graph over the variables and their negations. How does the graph look like if the original formula is satisfied?)*

Exercise 7. Show that NODECOVER and CLIQUE are NP-complete. *(Hint: Reduce* INDEPSET*.)*

Exercise 8. Show that HITTINGSET is NP-complete.

A.2 Further reading

References

[BC93] Daniel Pierre Bovet and Pierluigi Crescenzi. *Introduction to the Theory of Complexity.* Prentice Hall International (UK) Limited, 1993.

[Pap94] Christos H. Papadimitriou. *Computational Complexity.* Addison-Wesley Publishing Company, Inc., 1994.

[Woe] Gerhard J. Woeginger. The P-versus-NP page . http://www.win.tue.nl/~gwoegi/P-versus-NP.htm.