Complexity Theory

Lecture Notes

October 27, 2016

Prof. Dr. Roland Meyer

TU Braunschweig
Winter Term 2016/2017

Preface

These are the lecture notes accompanying the course *Complexity Theory* taught at TU Braunschweig. As of October 27, 2016, the notes cover 11 out of 28 lectures. The handwritten notes on the missing lectures can be found at `http://concurrency.cs.uni-kl.de`. The present document is work in progress and comes with neither a guarantee of completeness wrt. the content of the classes nor a guarantee of correctness. In case you spot a bug, please send a mail to `roland.meyer@tu-braunschweig.de`.

Peter Chini, Judith Stengel, Sebastian Muskalla, and Prakash Saivasan helped turning initial handwritten notes into LaTeX, provided valuable feedback, and came up with interesting exercises. I am grateful for their help!

Enjoy the study!

Roland Meyer

Introduction

Complexity Theory has two main goals:

- Study computational models and programming constructs in order to understand their power and limitations.
- Study computational problems and their inherent complexity. Complexity usually means time and space requirements (on a particular model). It can also be defined by other measurements, like randomness, number of alternations, or circuit size. Interestingly, an important observation of computational complexity is that the precise model is not important. The classes of problems are robust against changes in the model.

Background:

- The *theory of computability* goes back to Presburger, Gödel, Church, Turing, Post, Kleene (first half of the 20^{th} century).

  It gives us methods to decide whether a problem can be computed (decided by a machine) at all. Many models have been proven as powerful as Turing machines, for example while-programs or C-programs. The *ChurchTuring conjecture* formalizes the belief that there is no notion of computability that is more powerful than computability defined by Turing machines. Phrased positively, Turing machines indeed capture the idea of computability.

- *Complexity theory* goes back to "On the computation complexity of algorithms" by Hartmanis and Stearns in 1965 [**?**]. This paper uses multi-tape Turing machines, but already argues that concepts apply to any reasonable model of computation.

Contents

Chapter 1

Crossing Sequences and Unconditional Lower Bounds

Goal: We establish an *unconditional lower bound* on the power of a *uniform complexity class.*

- The term *unconditional* is best explained by its opposite. If we show a *conditional* lower bound — say NP-hardness — we mean that the problem is hard *under the assumption* that $\mathsf{P} \neq \mathsf{NP}$. An unconditional lower bound does not need such an assumption. Unconditional lower bounds are rare, even proving $\mathsf{SAT} \notin \mathsf{DTIME}(n^3)$ seems out of reach.
- *Uniformity* means that we use one algorithm (one Turing Machine) to solve the problem for all inputs. *Non-uniform* models may use different algorithms (popular in circuit complexity) for different instances.

To get the desired lower bound, we employ a counting technique called *crossing sequences.* Crossing sequences are vaguely related to fooling sets in automata theory.

1.1 Crossing Sequences

Let $COPY = \{\, w\#w \mid w \in \{a,b\}^* \,\}$. This language is not context free and hence not regular.

Goal 1.1. Show an upper and a lower bound for $COPY$:

- *Upper bound*: $COPY$ can be decided in quadratic time.
- *Lower bound*: $COPY$ cannot be decided in subquadratic time (on a 1-tape DTM (deterministic Turing machine)).

Recall 1.2.

1. When we refer to a problem written as a set, deciding the problem means deciding *membership* for that set. Given $x \in \{a, b\}^*$, does $x \in COPY$ hold?

2. We assume that all tapes of a Turing machine are *right-infinite:* To the left, they are marked by \$. From this marker, one can only move to the right and \$ cannot be overwritten. Unless otherwise stated, we will assume that the TM only halts on \$.

3. The time and the space requirements of a TM are measured relative to the size of the input. Without further mentioning, this size will be referred to as n.

4. $\mathcal{O}$-notation = *asymptotic upper bound* ($\leq$, read as *no more than*):

   $$\mathcal{O}(g(n)) := \{ f : \mathbb{N} \to \mathbb{N} \mid \exists c \in \mathbb{R}_+ \ \exists n_0 \in \mathbb{N} \ \forall n \geq n_0 : f(n) \leq c \cdot g(n) \} .$$

   o-notation = *asymptotic strict upper bound* ($<$, read as *less than*):

   $$o(g(n)) := \{ f : \mathbb{N} \to \mathbb{N} \mid \forall c \in \mathbb{R}_+ \ \exists n_0 \in \mathbb{N} \ \forall n \geq n_0 : f(n) \leq c \cdot g(n) \} .$$

Lemma 1.3. *$COPY \in \mathsf{DTIME}(n^2)$.*

The proof is left as an exercise to the reader.

Definition 1.4. The *crossing sequence of M on input x at position i* is the sequence of states of M when moving its head from cell i to cell $i+1$ or from cell $i+1$ to cell i. To be precise, we mean the state that is reached after the transition. We denote the crossing sequence by $CS(x, i)$.

Picture missing If q is a state in an odd position of the crossing sequence, then M moves its head from left to right. If it is a state in an even position, M moves from right to left. Transitions staying in cell i do not contribute to the crossing sequence.

Lemma 1.5 (Mixing Lemma)**.** *Consider words $x = x_1.x_2$ and $y = y_1.y_2$. If $CS(x, |x_1|) = CS(y, |y_1|)$, then*

$$x_1.x_2 \in L(M) \quad \textit{if and only if} \quad x_1.y_2 \in L(M).$$

Sketch. Intuitively, the crossing sequence is all that M remembers about x_2 resp. y_2 when operating on the left part x_1. Since the crossing sequences are assumed to coincide, M will behave the same on x_1, independent of whether x_2 or y_2 is on the right. Since the \$ symbol (on which we accept) is on the left, we are done. □

The next lemma states that the crossing sequences (their length) form a lower bound on the time complexity of a TM. If the TM is guaranteed to move on every transition, even equality holds. The proof is immediate by the definition of crossing sequences.

Lemma 1.6 (Fundamental Inequality)**.** $\mathrm{Time}_M(x) \geq \sum_{i\geq 0} |CS(x,i)|$.

We are now prepared to show the unconditional lower bound.

Theorem 1.7. *COPY* $\notin \mathsf{DTIME}(o(n^2))$.

Proof. Let M be a 1-tape DTM for *COPY*. Consider inputs of the form

$$w_1.w_2\#w_1.w_2 \quad \text{with } |w_1| = |w_2| = n.$$

We have:

$$\begin{aligned}
&\sum_{w_2\in\{a,b\}^n} \mathrm{Time}_M(w_1.w_2\#w_1.w_2) \\
(\text{ Lemma 1.6 }) \;\geq\; &\sum_{w_2\in\{a,b\}^n} \sum_{\nu=2n+1}^{3n} |CS(w_1.w_2\#w_1.w_2,\nu)| \qquad (1.1)\\
=\; &\sum_{\nu=2n+1}^{3n} \sum_{w_2\in\{a,b\}^n} |CS(w_1.w_2\#w_1.w_2,\nu)|\,. \qquad (1.2)
\end{aligned}$$

Consider ν with $2n+1 \leq \nu \leq 3n$. Denote the average length of the crossing sequences from the set $Cross_\nu := \{\, CS(w_1.w_2\#w_1.w_2,\nu) \mid w_2 \in \{a,b\}^n \,\}$ by

$$l_\nu \;:=\; \frac{\sum_{w_2\in\{a,b\}^n} |CS(w_1.w_2\#w_1.w_2,\nu)|}{2^n}\;.$$

Claim 1.8. At least half (which means $\frac{2^n}{2} = 2^{n-1}$) of the crossing sequences from $Cross_\nu$ have length $\leq 2l_\nu$.

Claim 1.9. The crossing sequences from $Cross_\nu$ are pairwise distinct.

Claim 1.10. The number of crossing sequences of length $\leq 2l_\nu$ is bounded by $(|Q|+1)^{2l_\nu}$, where Q is the set of states of M.

We add $+1$ because the sequence may be shorter.

By Claim 1.8 to Claim 1.10, we have

$$(|Q|+1)^{2l_\nu} \geq 2^{n-1}\;.$$

Note that we need the fact that the crossing sequences are different to enforce the inequality. Since

$$(|Q|+1)^{2l_\nu} = (2^{\log(|Q|+1)})^{2l_\nu} = 2^{\log(|Q|+1)2l_\nu}\;,$$

monotonicity of logarithms yields $\log(|Q|+1)2l_\nu \geq n-1$. Hence, there is a constant c (depending on Q but not depending on n and not on ν) so that

$$l_\nu \;\geq\; cn\;. \qquad (1.3)$$

With this lower bound,

Claim 1.11. $\sum_{w_2 \in \{a,b\}^n} \mathrm{Time}_M(w_1.w_2\#w_1.w_2) \geq 2^n cn^2$.

Since there are only 2^n words w_2, Claim 1.11 implies

$$\mathrm{Time}_M(w_1.w_2\#w_1.w_2) \geq cn^2$$

for all least one w_2. This concludes the proof of Theorem 1.7. $\square$

Claim 1.8 is the following statement.

Lemma 1.12. *If $\frac{1}{n}\sum_{i=1}^{n} w_i = d$, at least half of the w_i have a value $\leq 2d$.*

Proof. Assume at least half of the w_i have a value $> 2d$. Then

$$\sum_{i=1}^{n} w_i > \frac{n}{2} \cdot 2d = nd \ .$$

From this we can deduce

$$\frac{1}{n}\sum_{i=1}^{n} w_i > \frac{nd}{n} = d \ .$$

This contradicts the assumption. $\square$

Proof of Claim 1.9. Towards a contradiction, consider two words $u \neq v$ of length $|u| = |v| = n$ and assume $CS(w_1.u\#w_1.u, \nu) = CS(w_1.v\#w_1.v, \nu)$. Recall that $2n+1 \leq \nu \leq 3n$. By Lemma 1.5, we have

$$w_1.u\#w_1.u \in L(M) \quad \text{iff} \quad w_1.u\#w_1.v \in L(M) \ .$$

Since the former word is a member of $COPY$ but the latter is not, and since M is assumed to accept $COPY$, this is a contradiction. $\square$

Proof of Claim 1.11.

$$\begin{aligned}
& \sum_{w_2 \in \{a,b\}^n} \mathrm{Time}_M(w_1.w_2\#w_1.w_2) \\
\text{((In)equalities (1.1) and (1.2))} \geq & \sum_{\nu=2n+1}^{3n} \sum_{w_2 \in \{a,b\}^n} |CS(w_1.w_2\#w_1.w_2, \nu)| \\
\text{(Definition } l_\nu \text{)} = & \sum_{\nu=2n+1}^{3n} 2^n l_\nu \\
\text{(Inequality (1.3))} \geq & \sum_{\nu=2n+1}^{3n} 2^n cn \\
= & \ 2^n cn^2 \ .
\end{aligned}$$

$\square$

1.2 A Gap Theorem for Deterministic Space Complexity

Recall that the input tape of a space-bounded TM is *read only.* The TM can only write to (and read from) its work tape. If $s(n) : \mathbb{N} \to \mathbb{N}$ is a space bound for the work tape, we will assume $s(n) > 0$ for all $n \in \mathbb{N}$. The reason is that we need a position for the head. The function $s(n)$ is *unbounded* if for all $m \in \mathbb{N}$ there is an $n \in \mathbb{N}$ so that $m < s(n)$. Note that $s(n)$ is unbounded if and only if $s(n) \notin \mathcal{O}(1)$.

Goal 1.13. Show that $o(\text{loglog}(n))$ work tape is no better than having no tape at all.

Theorem 1.14. $\mathsf{DSPACE}(o(\text{loglog}(n))) = \mathsf{DSPACE}(\mathcal{O}(1))$.

The inclusion from right to left is left as an exercise to the reader. We will prove the reverse inclusion.

Definition 1.15. Consider a TM $M = (Q, \Sigma, \Gamma, \$, \llcorner\!\lrcorner, \delta, q_0, q_{acc}, q_{rej})$ (the precise definition will be given in Section 2.1). A *small configuration* of M consists of:

- the current state (in Q),
- the content of the work tape (in Γ^*), and
- the head position on the work tape.

The small configuration omits the input word and the head position on the input tape.

Definition 1.16. The *extended crossing sequence of M on input x at position i, $ECS(x, i)$*, is the sequence of small configurations of M when moving its head from cell i to $i + 1$ or from cell $i + 1$ to i on the input tape.

Lemma 1.17. *Let M be $s(n)$-space bounded. The number of extended crossing sequences on inputs of length n is at most $2^{2^{ds(n)}}$, where d is a constant (depending on M but not on the input).*

Proof. Let $M = (Q, \Sigma, \Gamma, \$, \llcorner\!\lrcorner, \delta, q_0, q_{acc}, q_{rej})$ be the $s(n)$-space-bounded TM of interest. The number of small configurations on inputs of length n is bounded by $|Q|\,|\Gamma|^{s(n)}\,s(n)$. Since $s(n) > 0$, we have

$$|Q|\,|\Gamma|^{s(n)}\,s(n) \leq c^{s(n)},$$

where c is a constant depending only on Q and Γ but not on n.

In an extended crossing sequence, no small configuration may appear twice in the same direction. Otherwise, a (large) configuration of M would appear twice in the computation. As M is deterministic it would be stuck in an infinite loop. Since M is assumed to halt, this cannot happen. Thus, there are at most

$$\underbrace{(c^{s(n)}+1)^{c^{s(n)}}}_{left} \cdot \underbrace{(c^{s(n)}+1)^{c^{s(n)}}}_{right} = (c^{s(n)}+1)^{2c^{s(n)}} \leq 2^{2^{ds(n)}}$$

different extended crossing sequences on inputs of length n, where $d > 0$ is a constant (again dependent on M but not on the input). □

Proof of Theorem 1.14. Towards a contradiction, assume there was a language $L \in \mathsf{DSPACE}(o(\mathrm{loglog}(n)))$ with $L \notin \mathsf{DSPACE}(\mathcal{O}(1))$. Then there is a 1-tape DTM M with $L = L(M)$ and space bound $s(n) \in o(\mathrm{loglog}(n)) \setminus \mathcal{O}(1)$. We will show that M cannot exist.

Since $s(n) \in o(\mathrm{loglog}(n))$, there is an $n_0 \in \mathbb{N}$ so that for all $n \geq n_0$ we have

$$s(n) < \frac{1}{2d} \mathrm{loglog}(n) \ .$$

Here, d is the constant from Lemma 1.17. With this, we can deduce

$$2^{2^{ds(n)}} < 2^{2^{d\frac{1}{2d}\mathrm{loglog}(n)}} = \left(2^{2^{\mathrm{loglog}(n)}}\right)^{\frac{1}{2}} = n^{\frac{1}{2}} \leq \frac{n}{2} \ . \tag{1.4}$$

By the fact that $s(n)$ is unbounded, there is an input x' so that

$$s_0 := \mathrm{Space}_M(x') > \max\{\, s(n) \mid 0 \leq n \leq n_0 \,\}.$$

Let x be the shortest input with $s_0 = \mathrm{Space}_M(x)$. By the definition of s_0 we have $|x| > n_0$. Otherwise, $\mathrm{Space}_M(x) = s_0 > s(|x|) \geq \mathrm{Space}_M(x)$, which is a contradiction.
By Lemma 1.17 and Inequality (1.4), the number of extended crossing sequences is smaller than $\frac{|x|}{2}$. Hence, there are positions $i < j < k$ with

$$ECS(x,i) = ECS(x,j) = ECS(x,k).$$

Indeed, if there were at most two positions for each extended crossing sequence, we would get $|x| < 2 \cdot \frac{|x|}{2} = |x|$, which is a contradiction.

Now we shorten the input x by cutting out either the sequence from i to j or the sequence from j to k. Since every small configuration on x appears in at least one of the two shortened strings, M will use the same space on at least one of the two strings. This contradicts the choice of x as the shortest input with space bound s_0. □

The above theorem can be phrased as follows: *If M runs in $o(\text{loglog}(n))$ space, then M accepts a regular language.*

Theorem 1.18. $\mathsf{DSPACE}(\mathcal{O}(1)) = \mathsf{REG}$.

The relationship is non-trivial as a space-bounded TM may visit an input symbol arbitrarily often whereas a finite automaton will see it only once. We conclude the section with an example that complements Theorem 1.14: Starting from $\mathsf{DSPACE}(\text{loglog}(n))$, we obtain non-regular languages.

Example 1.19. Consider the language

$$L \ := \ \{\, \text{bin}(0) \# \, \text{bin}(1) \# \cdots \# \, \text{bin}(n) \mid n \in \mathbb{N} \,\}.$$

It can be shown that L is not regular but in $\mathsf{DSPACE}(\text{loglog}(n))$. With the Theorems 1.14 and 1.18, we conclude that we cannot do better: L does not lie in $\mathsf{DSPACE}(o(\text{loglog}(n)))$.

Chapter 2

Time and Space Complexity Classes

Goal: The goal in this chapter is to introduce the basic complexity classes. They have proven to be useful because:

- they characterize important problems like computing, searching/guessing, playing against an opponent and
- they are robust under reasonable changes to the model: P is the same class of problems no matter whether we take *polynomial-time Turing machines, polynomial-time while programs, polynomial-time RAM machines or polynomial-time C++ programs.*

2.1 Turing Machines

Goal 2.1. Define different types of Turing Machines: *deterministic TMs, non-deterministic TMs, multi-tape TMs, recognizers and deciders.*

Definition 2.2. A *deterministic 1-tape Turing machine (TM)* is a 9-tuple

$$M = (Q, \Sigma, \Gamma, \$, \llcorner\!\lrcorner, \delta, q_0, q_{acc}, q_{rej}),$$

where:

- Q is a finite set of *states* with *initial state* q_0, *accepting state* q_{acc} and *rejecting state* q_{rej},
- Σ is the finite *input alphabet* not containing $\llcorner\!\lrcorner$ nor $\$$,
- Γ is the *tape alphabet* with $\Sigma \subseteq \Gamma$. $\llcorner\!\lrcorner \in \Gamma$ is the *blank symbol*, $\$ \in \Gamma$ is the *left endmarker*,
- $\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R\}$ is the *transition function.*

We have the following requirements on M:

- The endmarker is never overwritten:

$$\forall p \in Q\ \exists q \in Q : \delta(p, \$) = (q, \$, R)$$

- Once the machine halts, it no longer writes:

$$\forall b \in \Gamma\ \exists d, d' : \delta(q_{acc}, b) = (q_{acc}, b, d) \text{ and } \delta(q_{rej}, b) = (q_{rej}, b, d')$$

For the semantics of a Turing machine we also define the notion of *configurations* and a *transition relation among configurations*:

Definition 2.3. Let $M = (Q, \Sigma, \Gamma, \$, \sqcup, \delta, q_0, q_{acc}, q_{rej})$ be a Turing machine.

a) A *configuration of* M is a triple $u\,q\,v \in \Gamma^* \times Q \times \Gamma^*$. The idea behind this notation is that M's head is on the first symbol of v, the machine is in state q and the tape content is uv.

b) The *transition relation* $\rightarrow \subseteq (\Gamma^* \times Q \times \Gamma^*) \times (\Gamma^* \times Q \times \Gamma^*)$ is defined by:

$$\begin{aligned} u.a\,q\,b.v &\rightarrow u\,q'\,a.c.v, \text{ if } \delta(q, b) = (q', c, L) \\ u.a\,q\,b.v &\rightarrow u.a.c\,q'\,v, \text{ if } \delta(q, b) = (q', c, R). \end{aligned}$$

A configuration $u\,q$ is understood to be equivalent to $u\,q\,\sqcup$.

c) The *initial configuration* of M on input $w \in \Sigma^*$ is $q_0\,\$.w$.

d) A configuration of the form $u\,q_{acc}\,v$ is called *accepting.* Similarly, a configuration of the form $u\,q_{rej}\,v$ is called *rejecting.*
Accepting and rejecting configurations are also called *halting configurations.*

e) The Turing machine M *accepts input* w, if there is a sequence of configurations:

$$c_1 \rightarrow c_2 \rightarrow \cdots \rightarrow c_n,$$

where c_1 is the initial configuration of M on input w and c_n is an accepting configuration.

f) The *language of* M is defined by:

$$L(M) := \{\, w \in \Sigma^* \mid M \text{ accepts } w \,\}.$$

Definition 2.4. A language $L \subseteq \Sigma^*$ is called *(Turing) recognizable* or *recursively enumerable*, if $L = L(M)$ for some Turing machine M.

Remark 2.5. When a TM is started on an input, there are three possible outcomes:

1. M may accept, so it reaches an accepting configuration,
2. M may reject, so it reaches a rejecting configuration or
3. M may *loop*, which means that M does not halt.

Hence, M may fail to accept by rejecting or by looping. Distinguishing looping from taking a long time to accept or reject is rather difficult. In Complexity Theory, we are only interested in machines that *halt on all inputs*, i.e., that never loop. Such machines are called *deciders*. They are also said to be *total*.

Definition 2.6. A language $L \subseteq \Sigma^*$ is called *(Turing) decidable* or *recursive*, if $L = L(M)$ for a decider M.

Remark 2.7. A *multitape Turing machine* is defined like an ordinary Turing machine but with several tapes. Initially, the input is on the first tape and the remaining tapes are empty. The transition function allows reading, writing and moving the head on some or even on all tapes simultaneously. Formally, δ is a function:

$$\delta : Q \times \Gamma^k \longrightarrow Q \times \Gamma^k \times \{L, R, S\}^k$$

Theorem 2.8. *For every multitape Turing machine M there is a 1-tape Turing machine M' with $L(M) = L(M')$. Hence, a language is recognizable if and only if some multitape Turing machine recognizes it.*

Remark 2.9. A *nondeterministic Turing machine (NTM)* may, at any point in a computation proceed according to several possibilities. Formally, the transition function takes the form:

$$\delta : Q \times \Gamma \longrightarrow \mathbb{P}(Q \times \Gamma \times \{L, R\})$$

The computation of a nondeterministic Turing machine is a tree where the branches correspond to different possibilities of the machine. If *some branch* of the tree leads to an accepting configuration, the machine *accepts the input.* The *language* of a NTM M is defined to be:

$$L(M) := \{\, w \in \Sigma^* \mid \text{ some branch of } M \text{ started on } w \text{ accepts} \,\}$$

M is called *total* or a *decider* if for every input $w \in \Sigma^*$ *all branches* halt.

Theorem 2.10. *For every (total) NTM M there is a (total) DTM M' with $L(M) = L(M')$. Hence, a language is recognizable if and only if some NTM recognizes it. Moreover, a language is decidable if and only if some NTM decides it.*

Recall 2.11. Recapitulation of the notions: *Decidable* and *Semidecidable*:

- A property $P : \Sigma^* \longrightarrow \mathbb{B}$ is called *decidable*, if $\{\, x \in \Sigma^* \mid P(x) = \text{true} \,\}$ is a recursive set. This means there is a total TM that accepts the input strings having property P, and rejects the strings that violate P.
- A property P is called *semidecidable*, if $\{\, x \in \Sigma^* \mid P(x) = \text{true} \,\}$ is a recursively enumerable set.

In short, the notions *recursive* and *recursively enumerable* apply to sets, while the notions *decidable* and *semidecidable* apply to properties. But they are interchangeable:

P is decidable	$\Leftrightarrow$	$\{\, x \in \Sigma^* \mid P(x) = \text{true} \,\}$ is recursive.
A is recursive	$\Leftrightarrow$	"$x \in A$" is decidable.
P is semidecidable	$\Leftrightarrow$	$\{\, x \in \Sigma^* \mid P(x) = \text{true} \,\}$ is recursively enumerable.
A is recursively enumerable	$\Leftrightarrow$	"$x \in A$" is semidecidable.

2.2 Time Complexity

Goal 2.12. Define the two basic *time complexity classes* $\mathsf{DTIME}_k(t(n))$ and $\mathsf{NTIME}_k(t(n))$.

Definition 2.13. Let M be a Turing machine which is potentially nondeterministic and which may have several tapes. Let $x \in \Sigma^*$ be an input of M.

a) We define the *computation time of M on x* to be:

$$\text{Time}_M(x) := \max \left\{ \text{number of transitions on } p \;\middle|\; \begin{array}{c} p \text{ is a computation} \\ \text{path of } M \text{ on } x \end{array} \right\}$$

If M does not halt on some path, we set $\text{Time}_M(x) := \infty$. Note that for a deterministic Turing machine, there is precisely one computation path.

b) For $n \in \mathbb{N}$, we define the *time complexity of M* as :

$$\text{Time}_M(n) := \max\{\, \text{Time}_M(x) \mid |x| = n \,\}$$

This measures the *worst case* behavior of M on inputs of length n.

c) Let $t : \mathbb{N} \to \mathbb{N}$ be some function. We say that M is *t-time-bounded* (also written *$t(n)$-time-bounded*), if $\text{Time}_M(n) \leq t(n)$ for all $n \in \mathbb{N}$.

Definition 2.14. Let $t : \mathbb{N} \to \mathbb{N}$ be a function. Then we set:

$$\mathsf{DTIME}_k(t(n)) := \left\{ L(M) \;\middle|\; \begin{array}{c} M \text{ is a } k\text{-tape DTM that is a decider} \\ \text{and } t(n)\text{-time-bounded} \end{array} \right\}$$

$$\mathsf{NTIME}_k(t(n)) := \left\{ L(M) \;\middle|\; \begin{array}{c} M \text{ is a } k\text{-tape NTM that is a decider} \\ \text{and } t(n)\text{-time-bounded} \end{array} \right\}$$

We also write $\mathsf{DTIME}(t(n))$ and $\mathsf{NTIME}(t(n))$ if we assume the Turing machine to have one tape.

Note 2.15. *Sublinear time* is not meaningful for Turing machines that do not have random access to the input. The problem is that the machine cannot read the whole input on the tape:
Let M be a deterministic Turing machine and assume there is an $n \in \mathbb{N}$ so that M reads at most $n-1$ symbols of the input x, for every x with $|x| = n$. Then there are words $a_1, \ldots, a_m$ with $|a_i| < n$ for $1 \leq i \leq m$ so that:

$$L(M) = \bigcup_{i=1}^{m} a_i.\Sigma^*.$$

2.3 Space Complexity

Goal 2.16. Define the two basic *space complexity classes* $\mathsf{DSPACE}_k(s(n))$ and $\mathsf{NSPACE}_k(s(n))$.

Remark 2.17. Different from the case of time complexity, it is interesting to study computations that run in *sublinear space*. Therefore, we will assume that a Turing machine has an *extra input tape*. This tape is *read only* and not counted towards the consumption of space. An illustration of such a machine is given in Figure 2.1.
Technically, *read-only* amounts to requiring that wherever the Turing machine reads a symbol, it has to write the same symbol.

Definition 2.18. Let M be a Turing machine which is potentially nondeterministic, which has an additional input tape and which may have several work tapes.

a) Let $x \in \Sigma^*$ be an input of M and let c be a configuration of M on x. Then the *space consumption of* c is defined by:

$$\text{Space}(c) := \max\{\, |w| \mid w \text{ is a work tape content of c} \,\}.$$

b) The *space consumption of* x is defined to be:

$$\text{Space}_M(x) := \max\left\{ \text{Space}(c) \;\middle|\; \begin{array}{c} c \text{ is a configuration that occurs} \\ \text{in a computation of } M \text{ on } x \end{array} \right\}$$

If the space grows unboundedly, we set $\text{Space}_M(x) := \infty$.

c) For $n \in \mathbb{N}$, we define the *space complexity* of M as:

$$\text{Space}_M(n) := \max\{\, \text{Space}_M(x) \mid |x| = n \,\}.$$

Like for the time complexity, this measures the worst case space consumption of M on inputs of length n.

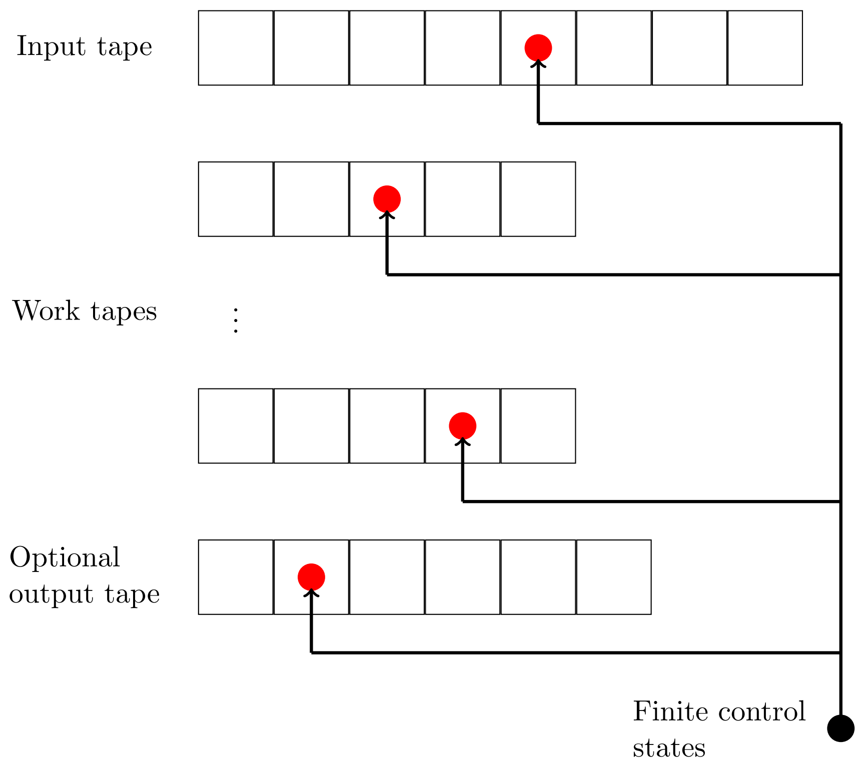


Figure 2.1: A Turing machine with additional input tape which is *read only*, several work tapes and an optional output tape. Those machines are used to define the notion of *space complexity* in Definition 2.18. The red dots in the tapes represent the positions of the machine's heads. These are controlled by a finite number of states.

d) Let $s : \mathbb{N} \to \mathbb{N}$ be some function. We say that M is *s-space-bounded* (also written *$s(n)$-space-bounded*), if $\text{Space}_M(n) \leq s(n)$ for all $n \in \mathbb{N}$.

Definition 2.19. Let $s : \mathbb{N} \to \mathbb{N}$ be a function. Then we define:

$$\mathsf{DSPACE}_k(s(n)) := \left\{ L(M) \;\middle|\; \begin{array}{l} M \text{ is a } k\text{-tape DTM with extra input} \\ \qquad \text{tape that is a decider} \\ \qquad \text{and } s(n)\text{-space-bounded} \end{array} \right\}$$

$$\mathsf{NSPACE}_k(t(n)) := \left\{ L(M) \;\middle|\; \begin{array}{l} M \text{ is a } k\text{-tape NTM with extra input} \\ \qquad \text{tape that is a decider} \\ \qquad \text{and } s(n)\text{-space-bounded} \end{array} \right\}$$

Example 2.20. Consider the following language:

$$L = \{\, x \in \{a, b\}^* \mid \text{the number of } a\text{s in } x \text{ equals the number of } b\text{s in } x \,\}.$$

Then L is in $\mathsf{DSPACE}(\mathcal{O}(\log n))$.

Proof. We give a construction of a Turing machine that is $\mathcal{O}(\log n)$-space-bounded and that decides L:
We read the input from left to right. On the work tape, we keep a binary counter. There are two cases:

1. If we read a, we increment $(+1)$ the binary counter.
2. If we read b, we decrement (-1) the binary counter.

We will accept the input if the counter value reached in the end is 0. This clearly decides the language L. For the space consumption, note the following:
In every step, we store a number $\leq |x|$ in binary on the work tape. This needs $\log |x|$ bits. The construction also requires us to increment and decrement in binary. But this does not cause any space overhead. Hence, the constructed Turing machine needs at most $\mathcal{O}(\log n)$ space. □

2.4 Common Complexity Classes

Definition 2.21. We now define the common robust complexity classes:

$$\begin{aligned}
\mathsf{L} &:= \mathsf{DSPACE}(\mathcal{O}(\log n)) && \text{(aka } \mathsf{LOGSPACE}\text{)}\\
\mathsf{NL} &:= \mathsf{NSPACE}(\mathcal{O}(\log n)) && \text{(aka } \mathsf{NLOGSPACE}\text{)}\\
\mathsf{P} &:= \bigcup_{k\in\mathbb{N}} \mathsf{DTIME}(\mathcal{O}(n^k)) && \text{(aka } \mathsf{PTIME}\text{)}\\
\mathsf{NP} &:= \bigcup_{k\in\mathbb{N}} \mathsf{NTIME}(\mathcal{O}(n^k))\\
\mathsf{PSPACE} &:= \bigcup_{k\in\mathbb{N}} \mathsf{DSPACE}(\mathcal{O}(n^k))\\
\mathsf{NPSPACE} &:= \bigcup_{k\in\mathbb{N}} \mathsf{NSPACE}(\mathcal{O}(n^k))\\
\mathsf{EXP} &:= \bigcup_{k\in\mathbb{N}} \mathsf{DTIME}\left(2^{\mathcal{O}(n^k)}\right) && \text{(aka } \mathsf{EXPTIME}\text{)}\\
\mathsf{NEXP} &:= \bigcup_{k\in\mathbb{N}} \mathsf{NTIME}\left(2^{\mathcal{O}(n^k)}\right) && \text{(aka } \mathsf{NEXPTIME}\text{)}\\
\mathsf{EXPSPACE} &:= \bigcup_{k\in\mathbb{N}} \mathsf{DSPACE}\left(2^{\mathcal{O}(n^k)}\right)\\
\mathsf{NEXPSPACE} &:= \bigcup_{k\in\mathbb{N}} \mathsf{NSPACE}\left(2^{\mathcal{O}(n^k)}\right).
\end{aligned}$$

We will also consider complement complexity classes:

Definition 2.22. Let $C \subseteq \mathbb{P}(\{0,1\}^*)$ be a complexity class. Then we define the *complement complexity class of* C to be:

$$\mathsf{co}\text{-}C := \{\, L \subseteq \{0,1\}^* \mid \overline{L} \in C \text{ where } \overline{L} = \{0,1\}^* \setminus L \,\}.$$

Note that co-C is not the complement of C, but it contains the complements of the sets in C. Intuitively, a problem in co-C contains the "no"-instances of a problem in C.

Example 2.23. Consider the following problem:

$$\mathsf{UNSAT} := \{\, \varphi \text{ a formula in CNF} \mid \varphi \text{ is not satisfiable} \,\}$$

Then $\overline{\mathsf{UNSAT}} = \mathsf{SAT}$ and since SAT is a problem in NP, we get that UNSAT is in co-NP.

Remark 2.24. Goals of Complexity Theory are:

- to understand the aforementioned common complexity classes in more detail. What are the problems they capture ? What do their algorithms look like ?
- to understand the relationship among the classes.

The next theorem is a simple observation that focuses on the second goal:

Theorem 2.25. *If C is a deterministic time or space complexity class, then: C = co-C. In particular, we have:* L = co-L, P = co-P *and* PSPACE = co-PSPACE.

The proof is left as an exercise.

Definition 2.26. A complexity class C is said to be *closed under complement*, if for all $L \in C$ we have $\overline{L} \in C$.

Remark 2.27. As a direct consequence of the definition, we get the following equivalences:

$$\begin{aligned} C = \mathsf{co}\text{-}C &\Leftrightarrow C \text{ is closed under complement} \\ &\Leftrightarrow \mathsf{co}\text{-}C \text{ is closed under complement} \end{aligned}$$

Further basic inclusions that focus on the second goal of Remark 2.24 are given in the following lemma:

Lemma 2.28. *Let $t, s : \mathbb{N} \to \mathbb{N}$ be two functions. Then these inclusions hold:*

$$\begin{aligned} \mathsf{DTIME}(t(n)) &\subseteq \mathsf{NTIME}(t(n)) \\ \mathsf{DSPACE}(s(n)) &\subseteq \mathsf{NSPACE}(s(n)) \\ \mathsf{DTIME}(t(n)) &\subseteq \mathsf{DSPACE}(t(n)) \\ \mathsf{NTIME}(t(n)) &\subseteq \mathsf{NSPACE}(t(n)). \end{aligned}$$

Proof. Since any deterministic Turing machine is also nondeterministic one, the first two inclusions are clear. To prove the latter two inclusions, note that a machine can only scan/write one cell per step. So the tape usage is bounded by the time. $\square$

Chapter 3

Alphabet reduction, Tape reduction, Compression and Speed-up

Goal: We show that the definition of the basic complexity classes is *robust* in the sense that it *does not depend* on the *details* of the Turing machine definition. These details are the tape alphabet, the number of tapes and constant factors. As a consequence, we do not have to be too accurate about these details. This will simplify proofs a lot.
Technically, we show that a one tape Turing machine with tape alphabet $\{\$, \llcorner\!\lrcorner, 0, 1\}$ can simulate the other features efficiently.

3.1 Alphabet Reduction

We will use the following notion of *simulation*:

Definition 3.1. Let M and M' be two Turing machines over the input alphabet Σ. Then M' is said to *simulate* M, if $\forall x \in \Sigma^*$ we have: $x \in L(M)$ if and only if $x \in L(M')$.

Lemma 3.2 (Alphabet Reduction:)**.** *Let $M = (Q, \Sigma, \Gamma, \$, \llcorner\!\lrcorner, \delta, q_0, q_{acc}, q_{rej})$ be a decider that is $t(n)$-time bounded. Then there is a decider*

$$M' = (Q, \{0,1\}, \{\$, \llcorner\!\lrcorner, 0, 1\}, \$, \llcorner\!\lrcorner, \delta, q_0, q_{acc}, q_{rej})$$

that is $C \cdot \log|\Gamma| \cdot t(n)$-time bounded and for all $x \in \Sigma^*$ we have:

$$x \in L(M) \textit{ if and only if } \mathrm{bin}(x) \in L(M').$$

Here, $\mathrm{bin}(-)$ *is a fixed binary encoding for the letters in* Γ.
Moreover we have that M' is deterministic if and only if M is deterministic and that M' has k tapes if and only if M has k tapes.

Proof. The Turing machine M' will mimic the operations of M on the binary encoding of the alphabet. To this end, it will:

- use $\log|\Gamma|$ steps to read from each tape the $\log|\Gamma|$ bits encoding a symbol of Γ.
- use its local state to store symbols it has read.
- use M's transition function/relation to compute the symbols that M writes and the state that M enters.
- store this information in the (control) state of M'.
- use $\log|\Gamma|$ steps to write the encoding into the tapes.

The number of states of M' is bounded by the number $C \cdot |Q| \cdot |\Gamma|^k \log|\Gamma|$. We get the factor $|Q|$ since we store the states of M, the factor $|\Gamma|^k$ since M uses k tapes and M' stores the symbols it has read and the factor $\log|\Gamma|$ for counting from 1 up to $\log|\Gamma|$. □

3.2 Tape Reduction

For the tape reduction, we first show a general construction that we then analyze with respect to its time and space usage.

Theorem 3.3 (Tape Reduction). *For every k-tape Turing machine M, there is a one tape Turing machine M' that simulates M. Moreover, M' is deterministic if and only if M is deterministic and if M uses an additional input tape, M' will also use an additional input tape.*

Proof. M' simulates one step of M by a sequence of steps. The idea is to store k tapes into one single tape. This tape will be understood as divided into $2k$ tracks. Technically, the tape alphabet is:

$$\Gamma' := (\Gamma \times \{*, -\})^k \cup \Sigma \cup \{\$, \textvisiblespace\}$$

The $(2\ell-1)$-st component of a letter in Γ' stores the content of the ℓ-th tape. The (2ℓ)-th component, which is in $\{*, -\}$, marks the position of the head on the ℓ-th tape by $*$. There is precisely one $*$ on the 2ℓ-th track, the remaining symbols are $-$. For an illustration of the construction, consider Figure 3.1.

A step of M is simulated by M' as follows: M' always starts on the left endmarker \$. It moves to the right until it finds the first pure blank symbol $\textvisiblespace$. Note that $\textvisiblespace$ is neither a vector containing $\textvisiblespace$ nor a vector containing $-$. On the way, M' stores the k symbols where the heads of M point to. Once M' has collected all k symbols, it can simulate a transition of M. It moves back to the start and, on its way, makes the changes to the tape that M would make. When arriving at \$, M' changes the control state. □

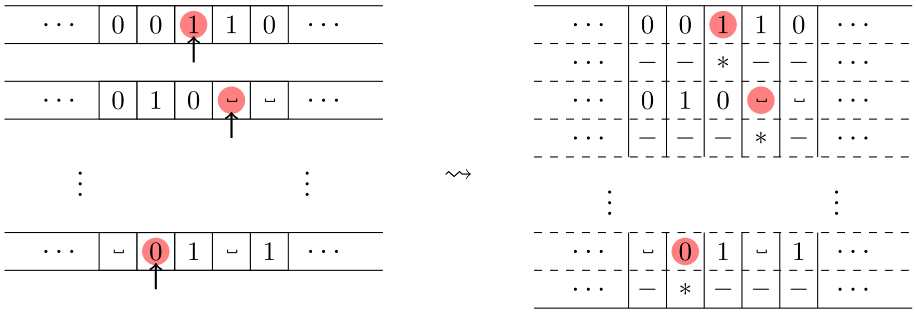


Figure 3.1: The different tapes of the Turing machine are combined in one tape with different tracks. The $2\ell-1$-st track keeps the content of tape ℓ, the 2ℓ-th track stores the position of the tape's head. One symbol in the new tape alphabet is represented by a column in the big tape.

Definition 3.4. Let $t, s : \mathbb{N} \to \mathbb{N}$ be two functions. Define:

$$\mathsf{DTIMESPACE}_k(t,s) := \left\{ L(M) \;\middle|\; \begin{array}{c} M \text{ is a } k\text{-tape DTM with extra input} \\ \text{tape that is a decider and that is} \\ t(n)\text{-time-bounded and } s(n)\text{-space-bounded} \end{array} \right\}$$

$$\mathsf{NTIMESPACE}_k(t,s) := \left\{ L(M) \;\middle|\; \begin{array}{c} M \text{ is a } k\text{-tape NTM with extra input} \\ \text{tape that is a decider and that is} \\ t(n)\text{-time-bounded and } s(n)\text{-space-bounded} \end{array} \right\}$$

We again drop the index if there is only one work tape.

Lemma 3.5. *For all functions $t, s : \mathbb{N} \to \mathbb{N}$, we have:*

$$\mathsf{DTIMESPACE}_k(t,s) \subseteq \mathsf{DTIMESPACE}(\mathcal{O}(t \cdot s), s) \text{ and}$$
$$\mathsf{NTIMESPACE}_k(t,s) \subseteq \mathsf{NTIMESPACE}(\mathcal{O}(t \cdot s), s)$$

Proof. Consider $L(M) \in \mathsf{DTIMESPACE}_k(t,s)$. The Turing machine M' from Theorem 3.3 simulates each step of M by $\mathcal{O}(s(n))$ steps. Since M makes at most $t(n)$ steps, M' makes at most $\mathcal{O}(t(n) \cdot s(n))$ steps. Hence, M' is $\mathcal{O}(t(n) \cdot s(n))$-time-bounded. Note that M' does not use more space than M does. This completes the first inclusion.
For the second inclusion, we may use the same arguments again since Theorem 3.3 also works in the nondeterministic case. □

Corollary 3.6. *For all function $t : \mathbb{N} \to \mathbb{N}$, we have:*

$$\mathsf{DTIME}_k(t) \subseteq \mathsf{DTIME}(\mathcal{O}(t^2)) \text{ and}$$
$$\mathsf{NTIME}_k(t) \subseteq \mathsf{NTIME}(\mathcal{O}(t^2))$$

Proof. Let M be a $t(n)$-time bounded Turing machine. We observed in Lemma 2.28, that in $t(n)$-steps, M can visit only $t(n)$-cells. Thus, we can deduce: $\mathrm{Space}_M(n) \leq t(n)$ and $L(M) \in \mathsf{DTIMESPACE}_k(t,t)$. Using Lemma 3.5 we get that

$$\mathsf{DTIMESPACE}_k(t,t) \subseteq \mathsf{DTIMESPACE}(\mathcal{O}(t^2),t) \subseteq \mathsf{DTIME}(\mathcal{O}(t^2)).$$

Hence, $L(M) \in \mathsf{DTIME}(\mathcal{O}(t^2))$. □

Remark 3.7 (Oblivious Turing-Machines:)**.** The construction in Theorem 3.3 can be modified to ensure that M' is *oblivious*: This means that the head movement of M does not depend on the input, but only on the input length. Formally, for every $x \in \Sigma^*$ and $i \in \mathbb{N}$, the location of each of M's heads at the i-th step of execution on input x is only a function in $|x|$ and i. The fact that every Turing machine can be simulated by an oblivious machine will simplify proofs.

3.3 Compression and linear Speed-up

The following tape compression result shows that we do not need to care about constant factors: Every Turing machine M can be simulated by a machine M' that use only a constant fraction of the space used by M.

Lemma 3.8 (Tape Compression)**.** *For all* $0 < \varepsilon \leq 1$ *and for all functions* $s : \mathbb{N} \to \mathbb{N}$*, we have:*

$$\mathsf{DSPACE}(s(n)) \subseteq \mathsf{DSPACE}(\lceil \varepsilon \cdot s(n) \rceil) \text{ and}$$
$$\mathsf{NSPACE}(s(n)) \subseteq \mathsf{NSPACE}(\lceil \varepsilon \cdot s(n) \rceil)$$

The statement can be understood as being the converse of the alphabet reduction, see Lemma 3.2. Rather than distributing one symbol to several cells, we enlarge the tape alphabet to store several symbols in one symbol. The idea can be compared to having a 64 bit architecture that is able to store more information per cell than an 8 bit architecture.

Proof. let $c := \lceil \frac{1}{\varepsilon} \rceil$ and let M be a single-tape deterministic Turing machine. We simulate M by a another DTM M' with tape alphabet:

$$\Gamma' := \Gamma^c \cup \Sigma \cup \{\$, \text{␣}\}.$$

A block of c cells is encoded into one cell of M'. So instead of $s(n)$ cells, M' only uses

$$\left\lceil \frac{s(n)}{c} \right\rceil \leq \lceil \varepsilon \cdot s(n) \rceil$$

cells. For the inequality, note that $c \geq \frac{1}{\varepsilon} \geq 1 \Rightarrow \frac{s}{c} \leq \varepsilon \cdot s \Rightarrow \lceil \frac{s}{c} \rceil \leq \lceil \varepsilon \cdot s \rceil$.

M' can simulate M step by step. To this end, M' stores the position of M's head inside a block of c cells in its control states. The head of M' always points onto the block where M's head is currently in.

- If M moves its head and does not leave a block, M' does not move the head but only changes the symbol and the control state.
- If M moves its head and leaves a block, M' changes the symbol, moves his head and adjusts the control state.

$\square$

A similar trick also works for the time consumption of a Turing machine. This is called *Linear Speed-Up*:

Lemma 3.9 (Linear Speed-Up)**.** *For all $k \geq 2$, all $t : \mathbb{N} \to \mathbb{N}$ and all $0 < \varepsilon \leq 1$, we have:*

$$\mathsf{DTIME}_k(t(n)) \subseteq \mathsf{DTIME}_k(n + \varepsilon(n + t(n))) \textit{ and}$$
$$\mathsf{NTIME}_k(t(n)) \subseteq \mathsf{NTIME}_k(n + \varepsilon(n + t(n)))$$

As before, the idea is to store $c \in \mathbb{N}$ cells into one new cells. Formally, we will copy the input and compress it. This costs $n + \varepsilon n$ steps since we read from left to right and go back to the beginning. To get the speed-up, M' now has to simulate c-steps of M with just a single step.

- If M stays within the c cells of one block, this is no problem: we can precompute the outcome of the c-steps.
- This gets harder if M moves back and forth between two cells that belong the neighboring blocks. In this case, M' stores three blocks in its finite control, the current block B, the block to the left of B, and the block to the right of B. With those three blocks, M' can simulate any c steps of M in its finite control. After this simulation, M' updates the tape content and if M has left block B, then M' has to update the blocks in its finite control.

Chapter 4

Space vs. Time and Non-determinism vs. Determinism

Goal: We want to prove more relations between space and time classes like in Lemma 2.28. In particular, we are interested in the inclusions: $\mathsf{NTIME}(t(n) \subseteq \mathsf{DSPACE}(t(n))$ and $\mathsf{NSPACE}(s(n) \subseteq \mathsf{DTIME}(2^{\mathcal{O}(s(n))})$. Before proving this, we need to define the notion of a constructible function.

4.1 Constructible Functions and Configuration Graphs

Goal 4.1. We will first define a notion of *constructible functions* that can be computed in a time/space economic way. After this, we will treat *configuration graphs* of Turing machines.

Definition 4.2. Let $s, t : \mathbb{N} \to \mathbb{N}$ be two functions with $t(n) \geq n$.

a) The function $t(n)$ is called *time constructible*, if there is an $\mathcal{O}(t(n))$-time-bounded deterministic Turing machine that computes the function $1^n \mapsto \text{bin}(t(n))$.
   Computing the function means that the result value is supposed to appear on a designated *output tape*, when the machine enters its accepting state. This output tape is a *write-only* tape.

b) The function $s(n)$ is called *space constructible*, if there is an $\mathcal{O}(s(n))$-space-bounded DTM that computes the function $1^n \mapsto \text{bin}(s(n))$.

Example 4.3. Most of the elementary functions are time and space constructible. For example: $n, n \log n, n^2$ and 2^n.

Definition 4.4. Let M be a Turing machine.

a) The *set of all configurations* of M is denoted by $\mathrm{Conf}(M)$. The *transition relation among configurations* is denoted by $\rightarrow_M$.

b) The *configuration graph of* M is a graph, defined by

$$\mathrm{CG}(M) := (\mathrm{Conf}(M), \rightarrow_M).$$

Remark 4.5. The configuration graph is typically infinite. However, to find out whether M accepts an input $x \in \Sigma^*$, all we have to do is find out whether we can reach an accepting configuration from the initial configuration $c_0(x)$. The task is undecidable in general, but it becomes feasible when the Turing machine is time or space bounded.

Lemma 4.6. *Let $s : \mathbb{N} \to \mathbb{N}$ be a function so that $s(n) \geq \log n$ and let M be an $s(n)$-space-bounded Turing machine. Then there is a constant c, depending only on M, so that M on input $x \in \Sigma^*$ can reach at most $c^{s(|x|)}$ configurations from $c_0(x)$.*

Proof. Let $M = (Q, \Sigma, \Gamma, \$, \sqcup, \delta, q_0, q_{acc}, q_{rej})$ be a k-tape Turing machine. A configuration of M is described by the current state, the content of the work tapes, the position of the heads on the work tapes and the position of the head on the input tape. Therefore, the number of configurations is bounded by:

$$|Q| \cdot \left(|\Gamma|^{s(|x|)}\right)^k \cdot s(|x|)^k \cdot (|x| + 2).$$

Since $s(n) \geq \log n$, there is a c, depending on $|Q|, |\Gamma|$ and k, so that

$$|Q| \cdot \left(|\Gamma|^{s(|x|)}\right)^k \cdot s(|x|)^k \cdot (|x| + 2) \leq c^{s(|x|)}.$$

Note that we used the assumption $s(n) \geq \log n$ to bound $|x| + 2$. □

Since we require deciders to always halt, an immediate consequence of this estimation is given in the following lemma.

Lemma 4.7. *Let $s : \mathbb{N} \to \mathbb{N}$ be a function so that $s(n) \geq \log n$. Then we have:*

$$\mathsf{DSPACE}(s(n)) \subseteq \mathsf{DTIME}(2^{\mathcal{O}(s(n))}) \text{ and}$$
$$\mathsf{NSPACE}(s(n)) \subseteq \mathsf{NTIME}(2^{\mathcal{O}(s(n))}).$$

Proof. Let $L \in \mathsf{NSPACE}(s(n))$. Then $L = L(M)$ for an NTM M that is a decider and $s(n)$-space bounded. Assume, M would repeat a configuration. Then, in a computation path, there is a loop. Since the computation tree contains all possible computation paths, there are also paths which go through the loop an unbounded number of times. So we get an infinite path. But this contradicts the fact that M is a decider. Hence, the running time of M is bounded by the number of reachable configurations. By Lemma 4.6, this is: $c^{s(n)} \in 2^{\mathcal{O}(s(n))}$. □

Space constructible functions are particularly interesting because they can be used to enforce termination. Intuitively, under the assumption of space constructibility, the Turing machine knows the space bound it is operating under.

Definition 4.8. Let $s : \mathbb{N} \to \mathbb{N}$ be a space constructible function so that $s(n) \geq \log n$ and let $L \subseteq \Sigma^*$. We say that a non-deterministic Turing machine M is an *$s(n)$-weak recognizer of L*, if

- $L = L(M)$ and
- for every $w \in L$ there is an accepting path $c_0(x) \to \cdots \to c_m$ with $\text{Space}(c) \leq s(n)$ for all configurations c on the path.

Proposition 4.9 (Self-Timing Technique)**.** *Let $s : \mathbb{N} \to \mathbb{N}$ be a space constructible function so that $s(n) \geq \log n$ and let $L \subseteq \Sigma^*$. If there is an $s(n)$-weak recognizer M of L, then there is also a non-deterministic Turing machine M' that decides L is space $\mathcal{O}(s(n))$.*

Proof. Since M is an $s(n)$-weak recognizer, every $x \in L$ has an accepting path $c_0(x) \to \cdots \to c_m$ with $\text{Space}(c) \leq s(n)$ for all configurations c on the path. Now assume there are duplicate configurations on the path: there are $i < j$ so that $c_i = c_j$. Then the path is of the form:

$$c_0(x) \to \cdots \to c_i \to \cdots \to c_j \to \cdots \to c_m.$$

But then we can cut out the loop and get the path:

$$c_0(x) \to \cdots \to c_i \to c_{j+1} \to \cdots \to c_m,$$

which is still accepting and space-bounded by $s(n)$. If we eliminate all duplicate configurations, the result is an accepting path of length bounded by $c^{s(n)}$, since there are at most $c^{s(n)}$ configurations under the space bound $s(n)$, see Lemma 4.6.

Hence, we know that M also has a short accepting path. Now construct M' to simulate M for $c^{s(n)}$ steps. If M accepts/rejects before this bound is reached, M' will accept/reject. If the bound is reached, M' will reject.
To enforce termination after the time bound, the idea is to add to M a counter that keeps track of how many steps have been taken so far. The space usage for this is $\mathcal{O}(\log c^{s(n)}) = \mathcal{O}(s(n))$. If a computation of M' reaches a length longer than $c^{s(n)}$, the machine rejects. This turns M' into a decider.

Besides checking the length of the computation, M' will make sure that the computation obeys the space bound. To this end, it initially marks $s(n)$ cells on the work tape, which can be done because $s(n)$ is space-constructible.

If the computation reaches a configuration that leaves this $s(n)$ cells, M' rejects.

Altogether, M' will still accept L as it will find all short and $s(n)$-space bounded accepting computations of M. The space used by M' on input $x \in \Sigma^*$ is bounded by:

$$\underbrace{s(n)}_{\text{configurations of } M} + \underbrace{\mathcal{O}(s(n))}_{\text{step counter}} = \mathcal{O}(s(n)).$$

□

Remark 4.10. A consequence of the self-timing technique in Proposition 4.9 is that we could have defined the complexity classes in a more liberal way: via weak recognizers.

4.2 Stronger Results

Goal 4.11. We want to improve the inclusions $\mathsf{NSPACE}(s(n)) \subseteq \mathsf{NTIME}(2^{\mathcal{O}(s(n))})$ and $\mathsf{NTIME}(t(n)) \subseteq \mathsf{NSPACE}(t(n))$ from Lemma 4.7 and Lemma 2.28 to:

$$\mathsf{NSPACE}(s(n)) \subseteq \mathsf{DTIME}(2^{\mathcal{O}(s(n))}) \text{ and}$$
$$\mathsf{NTIME}(t(n)) \subseteq \mathsf{DSPACE}(t(n)).$$

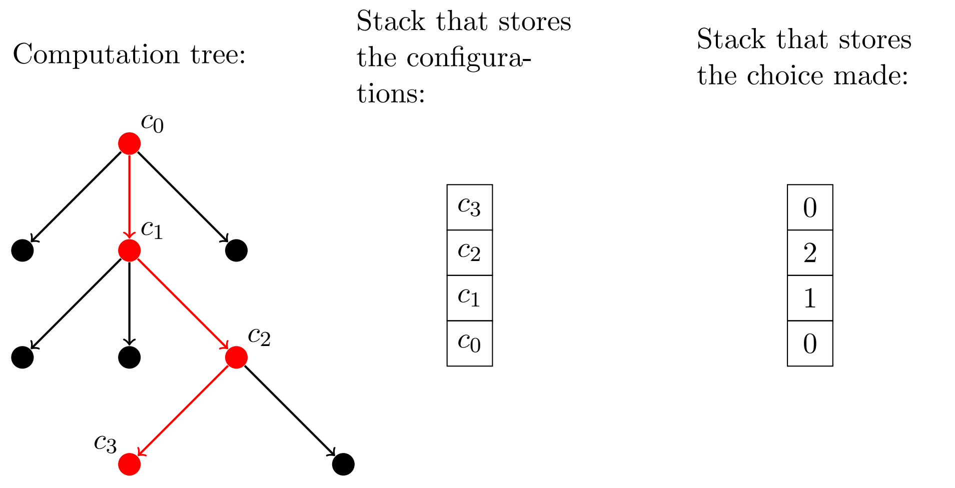
We directly start with the second inclusion. The result makes use of a space-economic way to store a stack for a depth-first search in a tree. An illustration of the idea used in the proof can be seen in Figure 4.1.

Theorem 4.12. *Let $t : \mathbb{N} \to \mathbb{N}$ be a function, then we have:*

$$\mathsf{NTIME}(t(n)) \subseteq \mathsf{DSPACE}(t(n)).$$

Proof. Let M be a non-deterministic Turing machine that is $t(n)$-time-bounded. To discover an accepting configuration of M deterministically, we do a depth-first search on the computation tree of M. We construct the tree on-the-fly and accept if an accepting configuration is encountered.
The depth of the tree is $t(n)$ and each configuration needs $t(n)$ space. Hence, a simple solution that stores a stack of configurations needs $\mathcal{O}(t(n)^2)$ space. But there is a more efficient way:

There exists a k that only depends on M such that every non-deterministic choice in the computation tree of M is at most k-ary. Since we only have to reconstruct the path from the initial configuration to the configuration currently visited, we only need to store the choices we made on this path.



Computation tree:
Stack that stores the configurations:
Stack that stores the choice made:
c_0
c_1
c_2
c_3

c_3
c_2
c_1
c_0

0
2
1
0

Figure 4.1: To perform a depth-first search in the computation tree of the Turing machine, we need to store a stack. Instead of saving a configuration in each stack frame, we only store the choices that we made on a path through the tree. This is enough to reconstruct the path and it is a significant saving of space.

There are at most k choices, so that in each stack frame we only need constant space. Hence, we can reconstruct the current configuration in $\mathcal{O}(t(n))$ space: Start with the initial configuration $c_0(x)$ and then simulate M using the stack to resolve choices. □

Theorem 4.13. *Let $s : \mathbb{N} \to \mathbb{N}$ be a function so that $s(n) \geq \log n$. Then we have the following inclusion:*

$$\mathsf{NSPACE}(s(n)) \subseteq \mathsf{DTIME}(2^{\mathcal{O}(s(n))}).$$

Proof. Let us first assume that $s(n)$ is space constructible. We use this assumption to determine the set of all configurations that use space at most $s(n)$. To enumerate the configurations, we encode them as strings:

- The states are given numbers and we write them down in unary.
- To separate the components of a configuration we use a fresh symbol (i.e. #).

The strings encoding each configuration have length at most $d \cdot s(n)$, for some constant d.

For the actual enumeration, we first mark $d \cdot s(n)$ cells. This can be done since $s(n)$ is space constructible. We use these cells as reference to enumerate all strings of length $d \cdot s(n)$ in lexicographic order. For each of the enumerated strings, we check whether it is a configuration. Altogether, the enumeration can be done in $2^{\mathcal{O}(s(n))}$ time.

On the resulting set of configurations, we do a reachability check. We may do a least fixed point and mark the reachable configurations (An alternative would be to enumerate the edges and to do a graph traversal). To this end, we repeatedly scan all configurations and mark them if they are reachable via the transition function δ. One scan needs $2^{\mathcal{O}(s(n))}$ time and we have to do at most $2^{\mathcal{O}(s(n))}$ scans. Hence, the reachability check needs $2^{\mathcal{O}(s(n))} \cdot 2^{\mathcal{O}(s(n))} = 2^{\mathcal{O}(s(n))}$ time.

To get rid of the assumption of space constructibility, we do the above procedure for a fixed space bound $s = 0, 1, 2, \ldots$. If we encounter a configuration that needs more space than s, we set $s := s + 1$. We eventually hit $s(n)$ in which case no configuration needs more space. The time requirement is then:

$$\sum_{s=0}^{s(n)} d^s = \frac{d^{s(n)+1} - 1}{d - 1} \in 2^{\mathcal{O}(s(n))}.$$

□

Chapter 5

Savitch's Theorem

Goal: The most important question in Theoretical Computer Science is the question whether $\mathsf{P} = \mathsf{NP}$. A similar question arises for the classes $\mathsf{DPSPACE}$ and $\mathsf{NPSPACE}$. The relation between them was solved rather early by Walter Savitch. In this chapter, we will prove Savitch's Theorem:

Theorem 5.1 (Savitch, 1970)**.** *Let $s : \mathbb{N} \to \mathbb{N}$ be a function so that $s(n) \geq \log n$. Then we have:*

$$\mathsf{NSPACE}(s(n)) \subseteq \mathsf{DSPACE}(s(n)^2).$$

In particular, $\mathsf{NPSPACE} = \mathsf{DPSPACE}$ *and typically referred to as* PSPACE*.*

For proving the theorem, we generalize acceptance of a Turing machine to the problem PATH in a directed graph, the configuration graph. A proper definition is given below:

Definition 5.2. The following problem is called PATH:
Given: Configurations c_1, c_2, and a time bound t.
Question: Can we get from c_1 to c_2 in at most t steps.

Our goal is to solve PATH deterministically in $s(n)^2$ space. Since the configuration graph has size $2^{\mathcal{O}(s(n))}$, we want to solve reachability in a directed graph with n nodes deterministically in $(\log n)^2$ space.
The main idea to achieve this space bound is to search for an intermediate configuration c and *recursively* check

- whether c_1 can get to c in $\frac{t}{2}$ steps and
- whether c can get to c_2 in $\frac{t}{2}$ steps.

If we *reuse the space* for each of the checks, we obtain a significant saving of space. Figure 5.1 provides an illustration of the procedure.

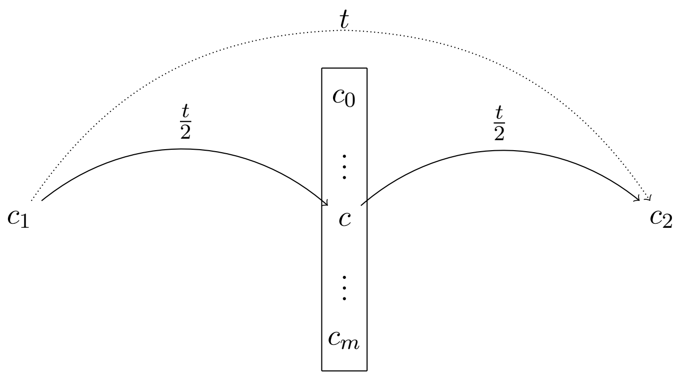


Figure 5.1: In order to check whether c_1 can reach c_2 in at most t steps, we look for an intermediate configuration c so that c_1 can reach c in at most $\frac{t}{2}$ steps and c can reach c_2 in at most $\frac{t}{2}$ steps. This is applied recursively.

The algorithm needs space for storing a stack. Each stack frame has to hold c_1, c_2, the current intermediate configuration c and the counter t, stored in binary. Since $t \in 2^{\mathcal{O}(s(n))}$, by Lemma 4.7, such a frame can be stored in $\mathcal{O}(s(n))$. The depth of the recursion is $\log t$. Hence, the stack needs the following space:

$$\underbrace{\mathcal{O}(s(n))}_{\text{stack height}} \cdot \underbrace{\mathcal{O}(s(n))}_{\text{stack frame}} = \mathcal{O}(s(n)^2)$$

Proof. We may assume that $s(n)$ is space constructible. Otherwise we can apply the enumeration trick from Theorem 4.13. Since the given NTM M is $s(n)$-space bounded, we also know that it is $c^{s(n)}$-time bounded by Lemma 4.7. In Theorem 4.13, we also discussed how configurations of M can be encoded as strings of length $d \cdot s(n)$ over an alphabet Γ.

Let $\alpha, \beta \in \Gamma^{d \cdot s(n)}$. We write

$$\alpha \xrightarrow{\leq k} \beta$$

if α and β represent configurations of M and α can go to β in at most k steps without exceeding the space bound $s(n)$. The deterministic algorithm will now check if

$$c_{init} \xrightarrow{\leq k} c_{acc},$$

where c_{init} and c_{acc} denote the initial and the accepting configurations of M. We can assume c_{acc} to be unique: by deleting the tape content, moving left, and only then accept.

The function to check $\alpha \xrightarrow{\leq k} \beta$ is described by the deterministic algorithm, Algorithm 1. It is clear that we have:

$$sav(\alpha, \beta, k) = \text{true iff } \alpha \xrightarrow{\leq k} \beta$$

Algorithm 1 $sav(\alpha, \beta, k)$

```
if $k = 0$ then
    return $(\alpha = \beta)$
end if
if $k = 1$ then
    return $(\alpha \to \beta)$
end if
if $k > 1$ then
    for all $\gamma \in \Gamma^{d \cdot s(n)}$ enumerated in lexicographical order do
        if $\gamma$ is a configuration then
            bool $a_{left} := sav(\alpha, \gamma, \lceil \frac{k}{2} \rceil)$
            bool $a_{right} := sav(\gamma, \beta, \lfloor \frac{k}{2} \rfloor)$
            if $a_{left} \wedge a_{right}$ then
                return $true$
            end if
        end if
    end for
    return $false$
end if
```

For the space requirement, consider the following: the depth of the recursion is $\log(c^{s(n)}) = \mathcal{O}(s(n))$. Each stack frame contains α, β, γ and the value k stored in binary. This needs $3 \cdot d \cdot s(n) + \log(c^{s(n)}) = \mathcal{O}(s(n))$ space. Together, we obtain the space bound: $\mathcal{O}(s(n)^2)$. We can get rid of the constant using tape compression, see Lemma 3.8. $\square$

Chapter 6

Space and Time Hierarchies

Goal: In this chapter, we show that higher space and time bounds lead to more powerful Turing machines. We will use *Universal Turing machines* to separate space and time classes by strict inclusions.

Recall 6.1 (The proof technique diagonalization)**.** Show the existence of a language L with certain properties, i.e. space requirements, that *cannot* be decided by a Turing machine taken from a given set $\{M_1, M_2, \ldots\}$. To this end, start with some language L_1, decided by M_1 and having the property. For $i > 1$, change the language L_{i-1} to L_i so that L_i still has the property but none of $M_1, \ldots, M_{i-1}$ decides L_i. The *limit* of this construction is the language L, we are looking for.

For the following Lemma, we provide two different proofs. The first one makes use of an uncountable set, the second proof uses the diagonalization technique described above.

Lemma 6.2. *There are undecidable languages.*

First proof. The set of languages over $\{0,1\}$ is $\mathbb{P}(\{0,1\}^*)$ and therefore uncountable. The set of Turing machines is countable. Hence, there are languages which cannot be decided by Turing machines. $\square$

Second proof. Let $x_1, x_2, \ldots$ be an enumeration of all binary strings and $M_1, M_2, \ldots$ an enumeration of all Turing machines over $\{0,1\}$. Define the language:

$$L = \{\, x_i \in \{0,1\}^* \mid M_i(x_i) = 0 \,\}$$

An illustration of the language L can be found in Figure 6.1.

Now assume that there exists a Turing machine M that decides L. Then there is an $i \in \mathbb{N}$ such that $M = M_i$. Now consider the string $x_i \in \{0,1\}$. then there are two cases:

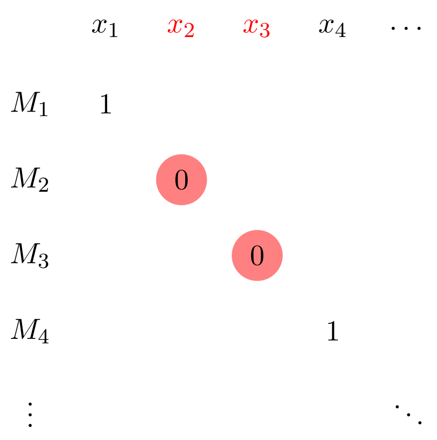


Figure 6.1: The strings x_2 and x_3 are not accepted by the Turing machines M_2, respectively M_3. Hence, these two elements are in the language L. Since x_1 and x_4 are accepted by M_1, respectively M_4, these two strings are not in L.

- If $M(x_i) = 1$ then $M_i(x_i) = 1$. So, $x_i \notin L$ but $M(x_i) = 1$, which is a contradiction.
- If $M(x_i) = 0$, then $M_i(x_i) = 0$. So, $x_i \in L$, but $M(x_i) = 0$, which is again a contradiction.

Hence, L cannot be decided by a Turing machine. This finishes the proof. □

6.1 Universal Turing Machine

Goal 6.3. To make use of diagonalization, we have to encode and simulate Turing machines. For the encoding, we will use a *Gödel numbering* of Turing machines, for the simulation we will make use of a *universal Turing machine*. The size of the encoding that we use will be a constant since we fix our Turing machine. So there is no need to be space efficient and indeed, we will see that we encode TMs unary. In contrast to this, the universal Turing machine has to be *efficient*. It has to obey space and time bounds.

Let us start with the encoding of deterministic Turing machines into $\{0,1\}^*$.

Definition 6.4. Let $M = (Q, \Sigma, \Gamma, \$, \sqcup, \delta, q_{init}, q_{acc}, q_{rej})$ be a k-tape deterministic Turing machine. Moreover, let

$$Q = \{\underbrace{q_{init}}_{=q_1}, \underbrace{q_{acc}}_{=q_2}, \underbrace{q_{rej}}_{=q_3}, \ldots, q_{|Q|}\}$$

be the set of states and

$$\Gamma = \{\underbrace{\sigma_1}_{=\gamma_1}, \dots, \underbrace{\sigma_{|\Sigma|}}_{=\gamma_{|\Sigma|}}, \underbrace{\$}_{=\gamma_{|\Sigma|+1}}, \underbrace{\textvisiblespace}_{=\gamma_{|\Sigma|+2}}, \dots, \gamma_{|\Gamma|}\}$$

be the tape alphabet. We encode a transition

$$\delta(q_i, \gamma_{i_1}, \dots, \gamma_{i_k}) = (q_j, \gamma_{j_1}, \dots, \gamma_{j_k}, d_1, \dots, d_k),$$

where a direction d_i is given by

$$d_i = \begin{cases} -1 \text{ (left)} \\ \ \ 0 \text{ (stay)} \\ \ \ 1 \text{ (right)} \end{cases}$$

as the following string in $\{0,1\}^*$:

$$\underbrace{0^i}_{\text{state}}\ 1\ \underbrace{0^{i_1}}_{\substack{\text{letter on}\\ \text{tape 1}}}\ 1 \dots 1\ \underbrace{0^{i_k}}_{\substack{\text{letter on}\\ \text{tape k}}}\ 1 \dots$$

$$\dots \underbrace{0^j}_{\substack{\text{new}\\ \text{state}}}\ 1\ \underbrace{0^{j_1}}_{\substack{\text{new letter}\\ \text{tape 1}}}\ 1 \dots 1\ \underbrace{0^{j_k}}_{\substack{\text{new letter}\\ \text{tape k}}}\ 1\ \underbrace{0^{2+d_1}}_{\substack{\text{direction}\\ \text{tape 1}}}\ 1\ \underbrace{0^{2+d_k}}_{\substack{\text{direction}\\ \text{tape k}}}$$

If we concatenate all these strings, separated by 11, we get the *encoding* of the transition function δ, denoted by $\mathrm{enc}(\delta)$.
The *encoding* of M is then defined to be the string:

$$\mathrm{enc}(M) = \underbrace{0^k}_{\text{tapes}}\ 11\ \underbrace{0^{|Q|}}_{\text{states}}\ 11\ \underbrace{0^{|\Gamma|}}_{\substack{\text{tape}\\ \text{alphabet}}}\ 11\ \underbrace{0^{|\Sigma|}}_{\substack{\text{first } |\Sigma| \\ \text{elements of } \Gamma \\ \text{are letters from} \\ \text{the input alphabet}}}\ 11\ \underbrace{\mathrm{enc}(\delta)}_{\text{transitions}}$$

If $x = x_1 \dots x_n$ is an input string from Σ^* then we denote by $\langle M, x\rangle$ the string:

$$\langle M, x\rangle = \mathrm{enc}(M)\, 1111\, 0^{x_1}\, 1\, 0^{x_2}\, 1\, \dots\, 1\, 0^{x_n}$$

Theorem 6.5. *There is a 1-tape deterministic Turing machine U so that U on input $\langle e, x\rangle$ computes $E(x)$, where E is a deterministic Turing machine, $e = \mathrm{enc}(E)$ and $x \in \Sigma_E^*$.*
If E uses $s(n)$ space, then U uses $\mathcal{O}(|e| \cdot s(n))$ space for the simulation and for each step of E, U does $\mathcal{O}(|e|^2 \cdot s(n))$ steps.

The machine U from Theorem 6.5 is called *universal* for the class of deterministic Turing machines. U can also be modified to a nondeterministic machine that is universal for the class of NTMs. As an intuition, universal machines can be understood as assembly interpreters written in assembly.

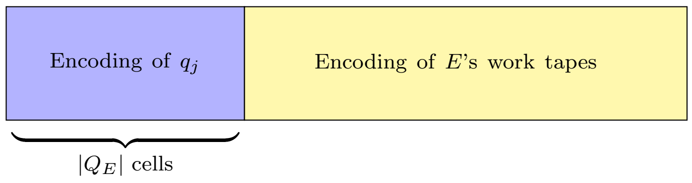


Figure 6.2: U's tape is separated into two parts. First, the encoding of E's current state is stored. This needs $|Q_E|$ cells. The remaining tape is used to store the encoding of E's work tapes.

Proof idea. Let E have k tapes. The machine U will store them on one tape using the tape reduction trick from Theorem 3.3. Hence, the letters of U will have $2k$ tracks.
But there is a problem since there is no bound on the size of the work alphabet Γ_E. Nevertheless, we have to fix Γ_U. The solution to this is to store a symbol

$$\begin{pmatrix} \gamma_1 \\ m_1 \\ \vdots \\ \gamma_k \\ m_k \end{pmatrix} \in (\Gamma_E \times \{*, -\})^k$$

as a string of length $|\Gamma_E|$ of the form:

$$\begin{pmatrix} f(\gamma_1) \\ g(m_1) \\ \vdots \\ f(\gamma_k) \\ g(m_k) \end{pmatrix} \in (\{0,1\}^{2k})^{|\Gamma_E|},$$

where $f(\gamma_j) := 0^j\,1^{|\Gamma_E|-j}$, $g(*) := 1^{|\Gamma_E|}$ and $g(-) := 0^{|\Gamma_E|}$.

The current state q_j of E is stored as the string $0^j\,1^{|Q_E|-j}$ of length $|Q_E|$. We store this string at the beginning of U's work tape, followed by the above encoding of the tapes of E. An illustration can be found in Figure 6.2

The simulation of one transition of E works as follows: U goes through the encoding of E's transition function to find the first entry, where the state matches the current state of E. To this end, U will compare the 0s after the 11 for enc(δ) symbol by symbol to see whether they match the current state of E, stored at the beginning of the work tape.
If the state does not match, U goes to the next entry of the transition function. If a transition for the state is found, U will go over the work tape

to check whether the current symbols match what the transition expects. If the symbols do not match, U will move to the next transition. If the symbols do match, U moves back over the tape and performs the required changes.

If E obeys the space bound $s(n)$ then the tape of U uses $\mathcal{O}(|e| \cdot s(n))$ space since a symbol of Γ_E^k is stored as a string of length $|\Gamma_E| \leq |e|$.
To simulate one step of E, U has to find the right transition. To this end, it will scan, for any transition of E, the whole tape 2 times: first, it compares the symbols to find the transition. If U did not find the matching transition, it will move back. If the right transition is found, it has to move back to perform the changes. Hence, the time requirement to simulate a single step of E is

$$\mathcal{O}(|e| \cdot 2 \cdot |e|\, s(n)) = \mathcal{O}(|e|^2\, s(n)).$$

□

6.2 Deterministic Space Hierarchy

Goal 6.6. We want to make use of the universal Turing machine to separate deterministic space classes. The corresponding result will be called *deterministic space hierarchy* and as a consequence, we will get our first strict inclusion involving the robust complexity classes: $\mathsf{L} \subsetneq \mathsf{PSPACE}$.

Theorem 6.7 (Deterministic Space Hierarchy). *Let $s_2(n) \geq \log n$ be space constructible and let $s_1 = o(s_2)$. Then we have:*

$$\mathsf{DSPACE}(s_1) \subsetneq \mathsf{DSPACE}(s_2).$$

And in particular: $\mathsf{L} \subsetneq \mathsf{PSPACE}$.

Proof. Let U be the universal Turing machine from Theorem 6.5. We construct a deterministic machine M that is s_2-space-bounded and so that $L(M) \notin \mathsf{DPSPACE}(s_1)$. This means that $L(M)$ cannot be decided by a s_1-space-bounded DTM.

As an input, M gets a string $y \in \{0,1\}^*$, interpreted as $y = \langle e, x\rangle$, where $e = \text{enc}(E)$ and E is a Turing machine. Then M outputs the following:

$$M(\langle e, x\rangle) = \begin{cases} 1, & \text{if } E \text{ does not accept } y \text{ in space } s_2(|y|) \\ 0 & \end{cases}$$

To this end, M works the following way:

1. Mark $s_2(|y|)$ cells on the tape. Note that we need the space constructibility of s_2 here.

2. Let $y = \langle e, x\rangle$. Check whether e is a valid encoding of a DTM E. This can be done in $\log |y|$ space.
3. Now M simulates E on input y (not x). To this end, M behaves like the universal Turing machine U.
4. On an extra tape, M counts the steps of U, using a ternary counter with $s_2(|y|)$ digits. Hence, the machine can count up to $3^{s_2(|y|)}$.
5. If during the simulation, U leaves the marked space, M rejects.
6. If the simulation makes more than $3^{s_2(|y|)}$ steps, M accepts.
7. If E halts, M also halts and:
   - → if E accepts, M will reject.
   - → if E rejects, M will accept.

Then M is s_2-space-bounded and for the language of M, we have:

$$L(M) = \{\, \langle e, x\rangle \mid E \text{ does not accept } \langle e, x\rangle \text{ in space } s_2(|\langle e, x\rangle|)\,\}$$

To show that $L(M) \notin \mathsf{DSPACE}(s_1)$, we proceed by contradiction and assume there is a deterministic Turing machine N that is s_1-space-bounded and total so that $L(N) = L(M)$. We can assume that N has one work tape and an extra input tape. Let e be the encoding of N and let $y = \langle e, x\rangle$ for an sufficiently long input string x. We distinguish two cases:

If y is in $L(M)$ then M accepts y. By definition of M, either the simulation of N terminated or N makes more than $3^{s_2(|y|)}$ steps.

- → If the simulation of N terminated then N rejected y. But then we have that $y \notin L(N) = L(M)$ which is a contradiction.
- → For the latter case, note that N cannot make more than

  $$c^{s_1(|y|)}(s_1(|y|) + 2)(|y| + 2)$$

  steps since N is s_1-space-bounded and a decider. If N would make more steps then it would enter an infinite loop.
  Since x was chosen to be rather long and $s_1 = o(s_2)$, the following inequality holds:

  $$\log 3 \cdot s_2(|y|) > \log c \cdot s_1(|y|) + \log(s_1(|y|) + 2) + \log(|y| + 2)$$

  But this is equivalent to:

  $$3^{s_2(|y|)} > c^{s_1(|y|)}(s_1(|y|) + 2)(|y| + 2)$$

  Hence, N does not make more than $3^{s_2(|y|)}$ steps and the second case does not occur at all.

If y is not in $L(M)$ then M ran out of space or N terminated.

→ If the simulation of N terminated, we have that N accepted y. Hence, $y \in L(N) = L(M)$. But this is a contradiction.

→ Like above we show that this case cannot happen. Since N is s_1-space-bounded, the simulation via U needs $|e| \cdot s_1(|y|)$ space by Theorem 6.5. The length of the input string x and the fact that $s_1 = o(s_2)$ imply the following inequality:

$$|e| \cdot s_1(|y|) \leq s_2(|y|).$$

Hence, M cannot run out of space and this case does not occur.

Finally, $L(M)$ cannot be in the class $\mathsf{DSPACE}(s_1)$. This completes the proof. □

6.3 Further Hierarchy Results

Goal 6.8. The universal Turing machine can be used to derive further hierarchy results. We state a separation theorem for time complexity classes, similar to Theorem 6.7 and we use Savitch's Theorem to prove a hierarchy result for nondeterministic space complexity classes.

In Theorem 6.5 we observed that the universal Turing machine is slower than the given Turing machine by a quadratic factor. This influences the next separation theorem - it is not as effective as for space complexity classes.

Lemma 6.9 (Deterministic Time Hierarchy). *Let t_2 be time constructible and let $t_1^2 = \sigma(t_2)$. Then we have:*

$$\mathsf{DTIME}(t_1) \subsetneq \mathsf{DTIME}(t_2).$$

And in particular: $\mathsf{P} \subsetneq \mathsf{EXP}$.

There exists a more efficient construction of a universal Turing machine due to Hennie and Stearns. This allows us to strengthen Lemma 6.9.

Theorem 6.10 (Hennie and Stearns, 1966). *Let t_2 be time constructible and let $t_1 \cdot \log(t_1) = \sigma(t_2)$. Then we have:*

$$\mathsf{DTIME}(t_1) \subsetneq \mathsf{DTIME}(t_2).$$

For nondeterministic space, we can combine the Deterministic Space Hierarchy with Savitch's theorem.

Theorem 6.11 (Nondeterministic Space Hierarchy)**.** *Let $s_2(n) \geq \log n$ be space constructible and let $s_1 = o(s_2)$. Then we have:*

$$\mathsf{NSPACE}(s_1) \subsetneq \mathsf{NSPACE}(s_2^2).$$

And in particular: $\mathsf{NL} \subsetneq \mathsf{PSPACE}$.

Proof. From Savitch's Theorem, 5.1, we know that $\mathsf{NSPACE}(s_1) \subseteq \mathsf{DSPACE}(s_1^2)$. Since $s_1 = o(s_2)$ implies $s_1^2 = o(s_2^2)$, we may apply the DSH, Theorem 6.7, and obtain: $\mathsf{DSPACE}(s_1^2) \subsetneq \mathsf{DSPACE}(s_2^2)$. But this is certainly contained in $\mathsf{NSPACE}(s_2^2)$ which proves the claim. $\square$

Chapter 7

Translation

Goal: Assume we have a certain inclusion about two complexity classes. We want to introduce a technique to get out of this more inclusions, usually among larger complexity classes, without putting more effort into it.

To achieve this, we need two things. First, we have to artificially extend the input by a new symbol - this is called *padding*. A padded language is not more complicated than the original language but it has reduced computational effort in the sense that the input words are now larger.
Then we have to prove *translation theorems* that allow us to switch between the original language and the padded language.

7.1 Padding and the Translation Theorems

Definition 7.1. Let $L \subseteq \Sigma^*$ be a language and $f : \mathbb{N} \to \mathbb{N}$ a function with $f(n) \geq n$. Moreover, let $\#$ be a symbol not in Σ. Then we set:

$$\mathrm{Pad}_f(L) := \{\, x\#^{f(|x|)-|x|} \,|\, x \in L \,\}.$$

This is a language in $\Sigma \cup \{\#\}$ and we call it *padded language* of L.

Note 7.2. Padding turns a word in L of length n into a word from $L\#^*$ of length $f(n)$. Hence, the content of the word does not change, only its length.

Now we state the translation theorems. They show that it is possible to switch between a language and its padded language. The theorems also show that the computational effort decreases when switching to $\mathrm{Pad}_f(L)$:

Theorem 7.3 (Translation for time)**.** *Let f, g be functions with $f(n), g(n) \geq n$ and let g be monotone and time constructible. Given 1^n, let $1^{f(n)}$ be computable in time $g(f(n))$. Then we have for $L \subseteq \Sigma^*$:*

a) $\mathrm{Pad}_f(L) \in \mathsf{DTIME}(\mathcal{O}(g))$ *if and only if* $L \in \mathsf{DTIME}(\mathcal{O}(g \circ f))$ *and* *b)* $\mathrm{Pad}_f(L) \in \mathsf{NTIME}(\mathcal{O}(g))$ *if and only if* $L \in \mathsf{NTIME}(\mathcal{O}(g \circ f))$.

Proof. We proof Part a, the proof for Part b is similar.

First, let $\mathrm{Pad}_f(L) \in \mathsf{DTIME}(\mathcal{O}(g))$ and let $x \in \Sigma^*$ be an input string. We check $x \in L$ in $\mathsf{DTIME}(\mathcal{O}(g(f(|x|))))$ as follows. We compute

$$y = x\#^{f(|x|)-|x|}$$

in time $\mathcal{O}(g(f(|x|)))$. Since $|y| = f(|x|)$, we can check $y \in \mathrm{Pad}_f(L)$ in time $\mathcal{O}(g(|y|)) = \mathcal{O}(g(f(|x|)))$ and by definition of $\mathrm{Pad}_f(L)$, we have:

$$y \in \mathrm{Pad}_f(L) \text{ if and only if } x \in L.$$

For the other direction, let $L \in \mathsf{DTIME}(\mathcal{O}(g \circ f))$ and let $x \in (\Sigma \cup \{\#\})^*$ be an input. We check in $\mathsf{DTIME}(\mathcal{O}(g(|x|)))$ time whether $x \in Pad_f(L)$ as follows. First, we check in time $|x| \leq g(|x|)$ whether x has the form $w\#^*$ for some $w \in \Sigma^*$. If this is not the case, we can reject x.
Let $x = w\#^{|x|-|w|}$. Now we compute $1^{g(|x|)}$ in time $\mathcal{O}(g(|x|))$. This works as g is time constructible and the binary representation can be converted to unary in $\mathcal{O}(g(|x|))$ steps. Then we check in time $g(|x|)$ whether

$$|x| = f(|w|)$$

holds. To this end, we compute $1^{f(|w|)}$ in time $g(f(|w|))$. If the machine wants to compute more than $g(|x|)$ steps, we reject. But why should we do this: since g is monotonic, we have:

$$g(f(|w|)) > g(|x|) \Rightarrow f(|w|) > |x|\,.$$

If we managed to compute $1^{f(|w|)}$, we can compare it to $1^{|x|}$. If $|x| \neq f(|w|)$, reject. Otherwise, we have: $x = w\#^{f(|w|)-|w|}$. Finally, we check in time $\mathcal{O}(g(f(|w|))) = \mathcal{O}(g(|x|))$ whether $w \in L$. □

Theorem 7.4 (Translation for space)**.** *Let $g(n) \geq \log n$ be space constructible. Let $f(n) \geq n$ so that given an input 1^n we can compute* $\mathrm{bin}\, f(n)$ *in space $g(f(n))$. Then we have for $L \subseteq \Sigma^*$:*

a) $\mathrm{Pad}_f(L) \in \mathsf{DSPACE}(g)$ *if and only if* $L \in \mathsf{DSPACE}(g \circ f)$ *and*

b) $\mathrm{Pad}_f(L) \in \mathsf{NSPACE}(g)$ *if and only if* $L \in \mathsf{NSPACE}(g \circ f)$.

7.2 Applications of the Translation Theorems

Remark 7.5. As a consequence of the translation results, it is more likely that larger complexity classes will collapse. Phrased differently, to show a separation among complexity classes one should consider the lower end of the hierarchy.

A first application of Theorem 7.4 is given by the following lemma:

Lemma 7.6. *The following implication holds true:*

$$\mathsf{DSPACE}(n) \neq \mathsf{NSPACE}(n) \Rightarrow \mathsf{L} \neq \mathsf{NL}.$$

Proof. We proceed by contraposition and assume $\mathsf{NL} \subseteq \mathsf{L}$. From this we derive the contradiction $\mathsf{NSPACE}(n) \subseteq \mathsf{DSPACE}(n)$. To this end, let L be in $\mathsf{NSPACE}(n)$. Since n can be written as $n = \log \circ \exp$, we may apply Theorem 7.4 and get:

$$\mathrm{Pad}_{\exp}(L) \in \mathsf{NSPACE}(\mathcal{O}(\log n)) = \mathsf{NL}.$$

By our assumption we have that $\mathsf{NL} \subseteq \mathsf{L} = \mathsf{DSPACE}(\mathcal{O}(\log n))$. Thus, we can apply the translation for space again and get

$$L \in \mathsf{DSPACE}(\mathcal{O}(\log \circ \exp)) = \mathsf{DSPACE}(n).$$

Note that the last equality uses tape compression. □

The next theorem shows that P and the class of deterministic, context-sensitive languages are different classes. We benefit from the translation theorems and from the deterministic space hierarchy, Theorem 6.7.

Theorem 7.7. *We have that* $\mathsf{P} \neq \mathsf{DSPACE}(n)$. *Hence,* P *is not the class of deterministic, context-sensitive languages.*

Proof. Consider a language $L \in \mathsf{DSPACE}(n^2) \setminus \mathsf{DSPACE}(n)$. Such a language exists by the deterministic space hierarchy, Theorem 6.7. Set $f(n) = n^2$. We get $\mathrm{Pad}_f(L) \in \mathsf{DSPACE}(n)$ by Theorem 7.4. If now $\mathsf{DSPACE}(n) = \mathsf{P}$, we would get:

$$\mathrm{Pad}_f(L) \in \mathsf{DTIME}(\mathcal{O}(n^k))$$

for some $k \in \mathbb{N}$. With Theorem 7.3, we get that

$$L \in \mathsf{DTIME}(\mathcal{O}((n^2)^k)) = \mathsf{DTIME}(\mathcal{O}(n^{2k})).$$

But then $L \in P = \mathsf{DSPACE}(n)$. But this contradicts the choice of L. □

Remark 7.8. Today, it is know that $\mathsf{P} \neq \mathsf{DSPACE}(n)$, but it is not known whether $\mathsf{P} \subsetneq \mathsf{DSPACE}(n)$ or $\mathsf{DSPACE}(n) \subsetneq \mathsf{P}$. It is also not known whether $\mathsf{DSPACE}(\log n) = \mathsf{P}$.

Chapter 8

Immerman and Szelepcsényi's Theorem

Goal: In this chapter we want to prove that the class NL is closed under complement. To this end, we need the theorem of Immerman and Szelepcényi. It shows that for $s(n) \geq \log n$, the class $\mathsf{NSPACE}(s(n))$ is closed under complement.

The theorem is of particular importance. It is not only used to show $\mathsf{NL} = \mathsf{co\text{-}NL}$, the space analogue to NP vs $\mathsf{co\text{-}NP}$, it also solves the *second LBA problem* posed by Kuroda in 1964:
Kuroda showed that nondeterministic, linear bounded automata (nondeterministic Turing machines with linear space bound) accept *precisely* the context-sensitive languages. But there were two more open problems:

(1) Are the languages accepted by nondeterministic, linear bounded automata those languages that are accepted by *deterministic*, linear bounded automata ? Phrased in our terms, this is the question whether the equality
$$\mathsf{NSPACE}(\mathcal{O}(n)) = \mathsf{DSPACE}(\mathcal{O}(n))$$
holds.

(2) Are the languages accepted by nondeterministic, linear bounded automata closed under complement ? So, do we have the equality
$$\mathsf{NSPACE}(\mathcal{O}(n)) = \mathsf{co\text{-}NSPACE}(\mathcal{O}(n)) \ ?$$

Kuroda also showed that $\neg(2) \Rightarrow \neg(1)$. But this implication did not help, as Immerman and Szelepcsényi proved (2) to hold.
Nowadays, Problem (1) is still open.

The theorem was proven independently in 1987 and 1988 by

- *Neil Immerman* - University of Massachusetts Amhorst - and
- *Rbert Szelepcsényi* - student in Bratislava, Slovakia.

Both of them received the Gödel-Prize in 1995. With their theorem they brought the method of *inductive counting* to complexity theory.

Theorem 8.1 (Immerman and Szelepcsényi, 1988 and 1987)**.** *For a function $s(n) \geq \log n$, we have:*

$$\mathsf{NSPACE}(s(n)) = \mathsf{co\text{-}NSPACE}(s(n)).$$

In particular, we get: $\mathsf{NL} = \mathsf{co\text{-}NL}$.

8.1 Non-reachability

The key of the proof is to show that *non-reachability* in a graph can be solved in nondeterministic logarithmic space. To this end, we consider the problem $\overline{\mathsf{PATH}}$:
Given: A directed graph G with nodes s and t.
Problem: Show that there is no path from s to t.

Note that if we can show that $\overline{\mathsf{PATH}} \in \mathsf{NL}$, the theorem of Immerman and Szelepcsényi follows.

Theorem 8.2. *We have that $\overline{\mathsf{PATH}} \in \mathsf{NL}$.*

To check that t is not reachable from s, the idea is to enumerate all nodes that *are reachable* from s and to check that t is not among them. But it is not clear how to enumerate all these nodes in logarithmic space. Instead, we use a different approach: we *ensure* that all nodes reachable from s were enumerated via *counting.*

Assume we are given N, the number of nodes reachable from s. In Section 8.2, we show how to compute N in NL. The nondeterministic counting of the reachable nodes works as follows: we introduce a counter *count*, initially set to 0. For any node v, we guess if v is reachable from s and in that case, we guess a path from s to v. If we guess a wrong path, we reject. If we guess a right path and it turns out that $v = t$, we can reject since then, t is reachable from s. Otherwise, we increase *count* by 1 since we have found a reachable node different from t.
At the end, we compare *count* and N. We have guessed reachability of any v correctly if and only if *count* $= N$. Hence, we accept. Otherwise, we reject since our guess was wrong for a node v.

Algorithm 2, also called *unreach*(G, s, t), is based on the above idea and due to Lemma 8.4, it checks in NL whether t is *not* reachable from s in the graph G. Note that this depends heavily on nondeterminism! There are many executions of *unreach*(G, s, t) that guess wrong at some point and reject at the end. But due to the nondeterministic acceptance criterion, it is enough to have one execution of *unreach*(G, s, t) that accepts. This execution always guesses correctly and increases *count* up to N.

Algorithm 2 *unreach*(G, s, t)

```
count := 0
for every node $v$ do
    guess whether $v$ is reachable from $s$
    if the guess is true then
        guess a path from $s$ to $v$ of length $\leq n$
        if the guessed path does not lead to $v$ then
            return false        // Wrong path or wrong guess
        else
            if $v = t$ then
                return false    // $t$ is reachable from $s$
            else
                count ++        // Another reachable node $v \neq t$ found
            end if
        end if
    end if
end for
if count $< N$ then
    return false        // Guessed incorrectly about reachability for a $v$
else
    return true         // $t$ is unreachable from $s$
end if
```

Remark 8.3. Since we can compute N in NL, see Section 8.2, and Algorithm 2 runs in NL, see Lemma 8.4, we can solve $\overline{\text{PATH}}$ in NL and we proved the theorem of Immerman and Szelepcsényi.

Lemma 8.4. *Algorithm 2, describing the function unreach(G, s, t), has a computation that returns true if and only if t is not reachable from s. Moreover, the algorithm runs in nondeterministic logarithmic space.*

Proof. The algorithm makes sure it enumerates all nodes reachable from s by comparing count with N. It accepts if and only if t was *not* one of the N nodes reachable from s.
The integers N and count can be at most n, so they can be written down in binary at length $\log n$. Hence, the algorithm runs in NL. □

8.2 Inductive counting

The key idea to compute N, nowadays called *method of inductive counting*, is to inductively compute the values

$$R(i) := \#\text{nodes reachable from } s \text{ in } \leq i \text{ steps.}$$

Note that $N = R(n)$. So if we can compute $R(n)$, we can return that value. Algorithm 3 makes use of this idea.

Algorithm 3 *#reach*(G, s)

```
 1: $R(0) := 0$      // Only $s$ reachable from $s$ in 0 steps.
 2: for $i = 1, \dots, n$ do
 3:     $R(i) := 0$      // Initialize $R(i)$
 4:     for every node $v$ do
 5:         // Try all nodes $u$ reachable from $s$ in $\leq i-1$ steps.
 6:         // Check if $v$ is reachable from such a $u$ in $\leq 1$ steps.
 7:         $count := 0$
 8:         for every node $u$ do
 9:             guess whether $u$ is reachable from $s$ in $\leq i-1$ steps
10:             if the guess is true then
11:                 guess a path from $s$ to $u$ of length $\leq i-1$
12:                 if the guessed path does not lead to $u$ then
13:                     return false
14:                 else
15:                     $count{+}{+}$      // If $u$ is reachable, count it in
16:                     if $(u = v)$ or $(u \to v)$ then
17:                         $R(i){+}{+}$
18:                         goto(4)      // Go to next iteration of "for $v$" loop.
19:                     end if
20:                 end if
21:             end if
22:         end for      // Loop for $u$
23:         if $count < R(i-1)$ then
24:             return false      // Wrong guess about reachability for a $u$
25:         end if
26:     end for      // Loop for $v$
27: end for      // Loop for $i$
28: return $R(n)$
```

Remark 8.5. To count up to $R(i)$, Algorithm 3 makes use of the following fact: a node v is reachable from s in $\leq i$ steps if and only if there exists a node u, reachable from s in $\leq i-1$ steps so that $u = v$ or $u \to v$.

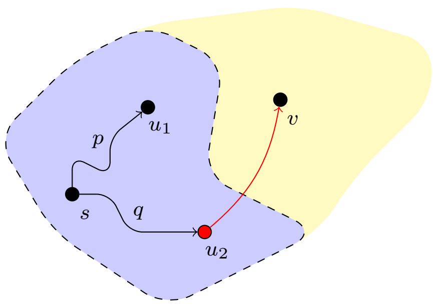


Figure 8.1: The blue marked area represents the nodes that are reachable from s in at most $i-1$ steps. Suppose the algorithm wants to check if the node v is reachable in at most i steps. Then it looks for a witness that is reachable in at most $i-1$ steps. If the algorithm arrives at u_1 and guesses the path p right, it will notice that there is no edge between u_1 and v. Hence, u_1 is no witness. The node u_2 is also reachable in at most $i-1$ steps and there is an edge to v. So, u_2 is a witness and the algorithm increases $R(i)$ by one.

Hence, provided the algorithm computed the right value of $R(i-1)$, it is possible to compute $R(i)$: for any node v, we test if there is a witness u so that the above fact is satisfied. We guess for any node u if it is reachable in $\leq i-1$ steps. If we guessed a right path then we increase *count* by one. After this, we test whether $u = v$ or $u \to v$. If this does not fail, we increase $R(i)$ since we found a witness for v.
If we do not find a witness for v, we have to check whether $count = R(i-1)$. Only in that case we have checked all nodes u that are reachable in $\leq i-1$ steps for being a witness. If $count < R(i-1)$ then we have guessed wrong about the reachability of some u and we cannot ensure that v is really unreachable.
An illustration of the algorithm can be found in Figure 8.1.

Lemma 8.6. *Algorithm 3, describing the function $\#reach(G, s)$, computes the number of nodes reachable from s.*
Moreover, the algorithm runs in nondeterministic logarithmic space.

Proof. We proceed by induction on i and show that upon termination of the iteration for i, we have:

$$R(i) = \#\text{nodes reachable from } s \text{ in } \leq i \text{ steps.}$$

Base case: $i = 0$. Then we have $R(0) = 1$ and this is correct.
Induction step: $i-1 \to i$. By induction, the equality holds for $R(i-1)$. The algorithm increments $R(i)$ on a node v if and only if v is reachable in $\leq i$ steps.

To see this, note that $R(i)$ is *not incremented* only if *all* nodes at distance $\leq i-1$ from s were tried and v is *not reachable* in ≤ 1 steps from any of them. We are sure to check all nodes at distance $\leq i-1$ by comparing count with $R(i-1)$.

At any point, Algorithm 3 only needs to remember two successive values $R(i-1)$ and $R(i)$. So it can reuse space when computing $R(1), \ldots, R(n)$, and can be made run in NL. $\square$

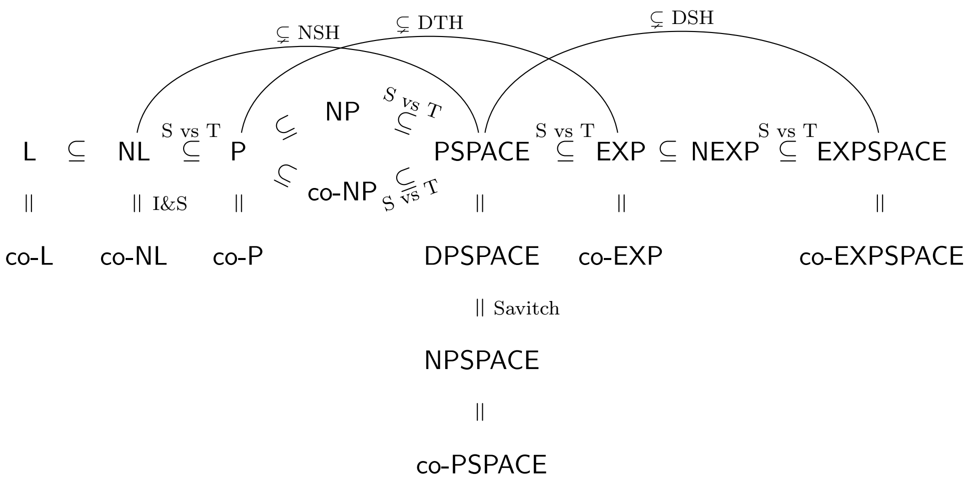
Summary: To check that t is not reachable from s, we first run Algorithm 3 to compute N. Then we run Algorithm 2 with that N. Since both (non-deterministic) algorithms run in logarithmic space, the total space required by the procedure is $\mathcal{O}(\log n)$.

Chapter 9

Summary

Goal: In the previous chapters we have proven many results about the relations among the robust complexity classes. Now our goal is to summarize our results and to create an overview.

Consider the following picture. So far, this is our understanding of the relationship among the robust complexity classes:



$\subsetneq$ NSH
$\subsetneq$ DTH
$\subsetneq$ DSH
L $\subseteq$ NL $\overset{\text{S vs T}}{\subseteq}$ P
NP
co-NP
PSPACE $\overset{\text{S vs T}}{\subseteq}$ EXP $\subseteq$ NEXP $\overset{\text{S vs T}}{\subseteq}$ EXPSPACE
S vs T
|| I&S
co-L
co-NL
co-P
DPSPACE
co-EXP
co-EXPSPACE
|| Savitch
NPSPACE
co-PSPACE

Remark 9.1. Some of the above showed relations are immediately clear: all deterministic classes are a subset of the corresponding nondeterministic class, see Lemma 2.28, and they are equivalent to their co-class, see Theorem 2.25. Other relations are more involved: all inclusions marked by the abbreviation *S vs T*, meaning *Space versus Time*, are due to the results of Chapter 4. We also get some equivalences:

- NL = co-NL is due to Immerman and Szelepcsnyi's Theorem, see Theorem 8.1.

- $\mathsf{DPSPACE} = \mathsf{NPSPACE}$ and the other equivalences involving PSPACE follow from Savitch's Theorem, see Theorem 5.1.

We also know about three strict inclusions:

- $\mathsf{NL} \subsetneq \mathsf{PSPACE}$ follows from the *nondeterministic space hierarchy* (NSH), see Theorem 6.11.
- $\mathsf{P} \subsetneq \mathsf{EXP}$ is a consequence of the *deterministic time hierarchy* (DTH), Theorem 6.9.
- $\mathsf{PSPACE} \subsetneq \mathsf{EXPSPACE}$ is due to the *deterministic space hierarchy* (DSH), see Theorem 6.7.

So far, the aforementioned strict inclusions are the only known strict inclusions in

$$\mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE} \subseteq \mathsf{EXP}$$

but it is believed that all these inclusions are actually strict.

Chapter 10

L and NL

Goal: We defined the robust complexity classes to capture interesting computational phenomena. Our goal in this and the following chapters is to study each of these classes in more detail.
Before we start looking at particular problems, we define the notion of a *complete* problem for a class. These characterize the computational phenomena captured by a complexity class:

- complete problems lie *in* the class. This means that they can be solved with the given resources.
- complete problems are *hard* for the class. This means that every problem in the class can be reduced to them.

After having defined what a complete problem exactly is, we start looking at the classes L and NL. We do not only want to understand the computational problems that can be solved using logarithmic space but we also want to study alternative models of computations that characterize L and NL. Recall that a requirement on the robust complexity classes was that they should be insensitive to the choice of the model of computation.

We will see that NL is all about paths: existence and absence. A hint on this was already given by Immerman & Szelepcsnyi. The class L mainly focuses on basic arithmetic operations.

10.1 Reductions and Completeness in Logarithmic Space

Goal 10.1. In order to explain the notion of a *complete problem*, we first need to introduce the concept of *many one reductions*. We are especially interested in reductions that are computable in logspace.

Definition 10.2. Let R be a set of functions from $\Sigma_1^* \to \Sigma_2^*$. A language $A \subseteq \Sigma_1^*$ is *R-many-one reducible* to a language $B \subseteq \Sigma_2^*$, if there is a function $f \in R$ so that for all $x \in \Sigma_1^*$ we have:

$$x \in A \Leftrightarrow f(x) \in B.$$

We call f the *reduction* and we also write: $A \leq_m^R B$.

Remark 10.3. Intuitively, $A \leq_m^R B$ means that membership in A can be checked by deciding membership in B. The notion *many-one* refers to the fact that the reduction need not to be injective. Injective reductions are called *one-one.*

Definition 10.4. Let C be a complexity class, R a set of functions and B a language.

a) The language B is called *C-hard with respect to R-many-one reductions*, if for all $A \in C$ we have that $A \leq_m^R B$. Intuitively, this means that B is at least as hard as any problem in C.

b) B is called *C-complete with respect to R-many-one reductions*, if B is C-hard wrt. R-many-one reductions and $B \in C$. Intuitively, B is the hardest problem in C.

We also phrase $B \in C$ as: *C is an upper bound*, and B is C-hard as: *C is a lower bound.*

Remark 10.5. A reduction will only be of interest if it satisfies the following two properties:

a) It should not have too much computational power: it should be weaker than the presumably harder one of the two complexity classes we are comparing. Otherwise, the considered *source*-problem may already be computed by the reduction itself and not by the *target*-problem.
In particular, if A is R-many-one reducible to B and $B \in C$, then it should also follow that $A \in C$. In short: C should be *closed under R-many-one reductions.*

b) Reducibility should be *transitive.* In particular, if A is C-hard and we have that $A \leq_m^R B$, then B should also be C-hard.

Now that we have stated our restrictions, we need to figure out which choices of the set of functions R are suitable. In fact, we will consider two different kinds of functions:

- The *polynomial-time computable* functions ($\leq_m^{\mathsf{log}}$) and
- the *logarithmic-space computable* functions ($\leq_m^{\mathsf{poly}}$).

In the literature, many-one reductions are also called *Karp reductions*. There are also so-called *Turing reductions* that may invoke the harder problem multiple times, like an oracle. Turing reductions are useful for proving undecidability results.

Definition 10.6. A function $f : \Sigma_1^* \to \Sigma_2^*$ is called *logspace computable* if there is a deterministic Turing machine M with

- read-only *input* tape over Σ_1,
- write-only (this means: write and move right) *output* tape over Σ_2 and
- read-write *work* tape over Γ,

so that M is a decider whose work tape is $\mathcal{O}(\log n)$-space bounded and M has written $f(x) \in \Sigma_2^*$ onto the output tape when halting on input x. We call M *logspace transducer.*

A language $A \subseteq \Sigma_1^*$ is *logspace-many-one reducible* to a language $B \subseteq \Sigma_2^*$, if there exists a logspace computable function $f : \Sigma_1^* \to \Sigma_2^*$ so that for all $x \in \Sigma_1^*$ we have:

$$x \in A \Leftrightarrow f(x) \in B.$$

Now we show that the definition of logspace reductions really satisfies the transitivity requirement:

Lemma 10.7. *Let $f : \Sigma_1^* \to \Sigma_2^*$ and $g : \Sigma_2^* \to \Sigma_3^*$ be logspace computable functions, then so is $g \circ f$.*

In particular: if $A \leq_m^{\mathsf{log}} B$ and $B \leq_m^{\mathsf{log}} C$, then $A \leq_m^{\mathsf{log}} C$.

Proof. Let f and g be computed by the logspace-bounded deterministic Turing machines M and N. Note that $|f(x)|$ is polynomial in $|x|$. This is due to Lemma 4.7, a logspace-bounded Turing machine can run at most a polynomial number of steps.

To compute $g \circ f$, we want to simulate the machine N on input $f(x)$. But we cannot just compute $f(x)$ by M in advance since it would not fit onto a logspace-bounded work tape. Instead, we will make use of *lazy evaluation.*

Technically, during the simulation of N, a subroutine will provide the symbols of $f(x)$ on demand. Whenever N wishes to read the i-th symbol of $f(x)$, we compute this symbol as follows:

- The index i is given to a subroutine that simulates M on input x from scratch.

- The simulator starts to compute $f(x)$. It counts and throws away all symbols up to the i-th.
- Then it returns the i-th symbol to the caller and stops.

For this construction, we need space to simulate the work tapes of M and N and two counters that count up to $|f(x)|$. One counter is the head position of N on $f(x)$, the second one is used by the subroutine that computes the i-th symbol of $f(x)$. Altogether, we need $\mathcal{O}(\log|x|)$ cells. $\square$

The following lemma states that polynomial time reductions are as least as powerful as logspace reductions. In fact, it is even known that polynomial time reductions are strictly stronger than logspace reductions.

Lemma 10.8. *Let A and B be two languages. If $A \leq_m^{\mathsf{log}} B$ then $A \leq_m^{\mathsf{poly}} B$.*

Proof. The proof is easy, just apply Lemma 4.7. $\square$

Besides transitivity, we also wanted that our reductions do not have too much computational power, see Remark 10.5. The next lemma shows that for the class L, the logspace reductions are already too powerful:

Lemma 10.9. *For all languages $A \subseteq \Sigma^*$, we have that $A \in \mathsf{L}$ if and only if $A \leq_m^{\mathsf{log}} \{0,1\}$. Moreover, any language $A \in \mathsf{L}$ so that $A \neq \emptyset$ and $A \neq \Sigma^*$ is L-complete.*

Hence, a reduction is only meaningful in a class that is computationally stronger than the reduction itself.

Note that the basic classes are closed under logspace reductions:

Lemma 10.10. *Let $A \leq_m^{\mathsf{log}} B$. If $B \in \mathsf{L}, \mathsf{NL}$ or P, then we get that $A \in \mathsf{L}, \mathsf{NL}$ or P, respectively.*

Finally, we can state a Lemma which shows the importance of *hard* problems. If a hard problem can be shown to be solvable with less computational effort then the whole class will collapse:

Lemma 10.11. *Let A be a language.*

a) If A is NL-hard under logspace many-one reductions and $A \in \mathsf{L}$ then $\mathsf{NL} = \mathsf{L}$.

b) If A is P-hard under logspace many-one reductions and $A \in \mathsf{NL}$ then $\mathsf{P} = \mathsf{NL}$.

10.2 Problems complete for NL

Goal 10.12. In this section we want to state the first NL-complete problem: PATH. We will give a precise definition and prove the completeness. To achieve the latter, we have to show that *every* problem in NL admits a reduction to PATH since we do not have another NL-hard problem yet. Our approach will be to construct a logspace-reduction from any nondeterministic Turing machine that is $\mathcal{O}(\log n)$-space bounded to PATH.

Once we have shown the hardness of PATH, the hardness of another problem can be shown by reducing from PATH to the problem.

Definition 10.13. Let PATH denote the following problem:

Input: A directed graph $G = (V, E)$ and $s, t \in V$.
Question: Is there a path from s to t in G ?

We use the notation ACYCPATH to refer to the same problem with the restriction that G is an acyclic graph.

Theorem 10.14. *The problem* PATH *is* NL*-complete.*

Proof. We have two show two things, membership: PATH $\in$ NL and hardness: PATH is NL-hard.

The first can easily be shown: construct a Turing machine M that works as follows: initially, it stores the node $u = s$ and a counter c on the tape. The counter holds the initial value 0. At each step, M guesses a vertex v that is reachable from the current vertex u. The vertex v is compared with t. If they are equal, we accept. If not, c gets increased and M continues with the next step. If c reaches the value n and the last guessed node is not t, we reject.
This needs $\mathcal{O}(\log n)$ space since we only need to store the current node and a binary counter. We also know that it works correctly: if a path between s and t exists, there is also a path of length at most n. Hence, there is a computation of M that finds this path.

Let A be a problem in NL and let M be the nondeterministic $\mathcal{O}(\log n)$-space bounded Turing machine that solves A: $L(M) = A$. We show that there exists a logspace computable function f_M so that given an input x, the image $f_M(x) = (G, s, t)$ satisfies: G is a directed graph, s and t are nodes in G and

$$M \text{ accepts } x \Leftrightarrow G \text{ contains a path from } s \text{ to } t.$$

The graph G is the configuration graph of M on input x. Hence, the nodes are the possible configurations of M on x and for c_1, c_2 configurations of M, there is an edge (c_1, c_2) if c_2 is a possible next configuration of c_1. The node s is defined to be the initial configuration while t is defined to be the unique accepting configuration of M. We can always assume M to have exactly one accepting configuration since we can just clear the tape, walk all the way left until we reach \$ and enter a special state.

Now assume that M accepts x. Then there is a computation of M that accepts x and this corresponds to a path from s to t in the configuration graph G. Vice versa, if there is a path from s to t in G then there is also a computation of M accepting x.

Now it remains to show that f_M is logspace computable. The transducer that outputs (G, s, t) on input x describes G by listing all its nodes and edges.
The listing of the nodes is done as follows: each node in G is a configuration of M on input x and can hence be represented by a string of length $d \cdot \log |x|$. The transducer enumerates all strings of length at most $d \cdot \log |x|$ and tests each for being a valid configuration. It writes those strings onto the output tape that pass the test. Thus, we have the listing of nodes.
The listing of edges is done similarly: the transducer now enumerates pairs of strings of length at most $d \cdot \log |x|$. The two components of a pair are tested for being configurations c_1 and c_2. Now the transducer tests if the transition relation of M induces $c_1 \to c_2$. If this is the case, it writes the pair of strings onto the output tape.
With the above description, creating the listings needs an $\mathcal{O}(\log n)$-space bounded work tape. Hence, we constructed a logspace transducer that computes f_M. □

Note 10.15. The configuration graph of a terminating Turing machine is always acyclic. So the above reduction also shows that ACYCPATH is NL-complete. In fact, there is a direct reduction from PATH.

Theorem 10.16. *We have that* PATH $\leq_m^{\mathsf{log}}$ ACYCPATH *and in particular:* ACYCPATH *is* NL*-complete.*

Proof. This will be handled in the exercises. □

A famous result due to Cook, Levin and Karp shows that 3-SAT is NL-complete. The restriction of SAT to two literals per clause, denoted by 2-SAT, is easier to solve:

Theorem 10.17. *The problem* 2-SAT *is* NL*-complete.*

To show membership of 2-SAT in NL, we proceed as follows: we show that $\overline{\text{2-SAT}} \in$ NL, then we have that 2-SAT $\in$ co-NL. By the theorem of

Immerman and Szelepcsényi, Theorem 8.1, we get that 2-SAT is also in NL.

To get the hardness of 2-SAT, we will give a reduction from the problem $\overline{\text{ACYCPATH}}$. Note that, due to the next lemma, $\overline{\text{ACYCPATH}}$ is also NL-hard.

Lemma 10.18. *Let C be a complexity class so that co-$C = C$, R a set of functions and A a language that is C-hard with respect to R-many-one reductions. Then $\overline{A}$ is also C-hard with respect to R-many-one reductions.*

Proof. This will be proven in the exercises. □

Now we start showing membership of 2-SAT.

Lemma 10.19. *The problem* 2-SAT *is in* NL.

The key idea is construct, from a 2-CNF formula F, a graph G_F as follows:

- We get a vertex for each variable of F and their negations.

$$V(G_F) = \{\, x \mid x \text{ is a variable in } F\,\} \cup \{\, \neg x \mid x \text{ is a variable in } F\,\}.$$

- We get edges $\alpha \to \beta$ and $\neg\beta \to \neg\alpha$ if $\neg\alpha \vee \beta$ is a clause in F. Additionally, for clauses consisting of just one literal α, we get the edge $\neg\alpha \to \alpha$.

$$E(G_F) = \bigcup_{\substack{\neg\alpha\vee\beta \text{ is a} \\ \text{clause of } F}} (\{(\alpha,\beta)\} \cup \{(\neg\beta,\neg\alpha)\}) \cup \bigcup_{\substack{\alpha \text{ is a} \\ \text{clause of } F}} \{(\neg\alpha,\alpha)\}.$$

Note that the edges of G_F correspond to implications. The clause $\neg\alpha \vee \beta$ is equivalent to the implications $\alpha \to \beta$ and $\neg\beta \to \neg\alpha$. For clauses consisting of one literal, we have: $\alpha \Leftrightarrow \alpha \vee \alpha \Leftrightarrow \neg\neg\alpha \vee \alpha \Leftrightarrow \neg\alpha \to \alpha$.
Hence, also the paths in G_F correspond to implications since implications are transitive.
Also note that G_F has a symmetry: we have $\alpha \to \beta$ if and only if $\neg\beta \to \neg\alpha$.

Let us consider an example of the construction:

Example 10.20. Let F be the formula $(\neg x \vee y) \wedge (\neg y \vee z) \wedge (x \vee \neg z) \wedge (z \vee y)$. Constructing G_F yields a graph with vertex set $V(G_F) = \{x, y, z, \neg x, \neg y, \neg z\}$ and edges

$$E(G_F) = \{(x,y), (\neg y, \neg x), (y,z), (\neg z, \neg y), (z,x), (\neg x, \neg z), (\neg z, y), (y, z)\}.$$

An illustration is given in Figure 10.1.

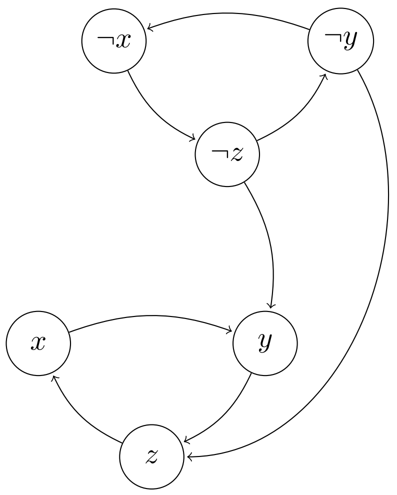


Figure 10.1: The construction explained below Lemma 10.19 applied to the formula $F = (\neg x \vee y) \wedge (\neg y \vee z) \wedge (x \vee \neg z) \wedge (z \vee y)$.

Lemma 10.21. *A 2-CNF formula F is unsatisfiable if and only if there are paths in G_F leading from x to $\neg x$ and from $\neg x$ to x for a variable x.*

Proof. Assume that the two paths exist but still assignment φ satisfies F. Without loss of generality, we may assume that $\varphi(x) = 1$. The case, where $\varphi(x) = 0$ is symmetric.
By negation, we get that $\varphi(\neg x) = 0$. Since there is a path from x to $\neg x$ in G_F, there is an edge $\alpha \to \beta$ on the path so that $\varphi(\alpha) = 1$ and $\varphi(\beta) = 0$. The edge $\alpha \to \beta$ corresponds to the clause $\neg\alpha \vee \beta$ in F. This evaluates to:

$$\varphi(\neg\alpha \vee \beta) = \neg\varphi(\alpha) \vee \varphi(\beta) = 0.$$

Thus, we have $\varphi(F) = 0$ which is a contradiction.

To prove the other direction of the lemma, we show that the absence of such paths lead to satisfiability. To this end, we construct a satisfying assignment as follows: we pick a node α so that there is no path from α to $\neg\alpha$ and assign truth values.

- To all nodes reachable from α, we assign *true*.
- To all nodes that reach $\neg\alpha$, we assign the value *false*.

We do this until every node has a truth value.

We assign a truth value to all nodes: let α be an unlabeled literal. Then also $\neg\alpha$ is unlabeled. If we cannot pick α since there is a path from α to $\neg\alpha$, we are sure that there is no path from $\neg\alpha$ to α since this would contradict the assumption. Hence, we choose $\neg\alpha$.

The resulting assignment is well-defined. Assume there is a literal β so that β and $\neg\beta$ are assigned the same truth value. Then there are two cases: there is a literal α so that either there are paths from α to β and $\neg\beta$ or β and $\neg\beta$ both reach $\neg\alpha$. We focus on the first case since the second case is symmetric. So let $\alpha \to^* \beta$ and $\alpha \to^* \neg\beta$. By the symmetry of G_F we also get paths $\neg\beta \to^* \neg\alpha$ and $\beta \to^* \neg\alpha$. But then we also have a path from α to $\neg\alpha$ which is a contradiction to the choice of α.

The resulting assignment satisfies F. If α is a literal that is assigned *true*, it cannot reach a literal that is labeled by *false* since in this case α itself would be labeled by *false*. Hence, there are no paths leading from *true* to *false* and the assignment satisfies F. □

Example 10.22. We construct the assignment from the proof of Lemma 10.21 on the graph G_F from Example 10.20. The vertices are labeled as follows: $(x, 1), (y, 1)$ and $(z, 1)$. The negated variables are labeled by 0 each. This means that the assignment that sets all variables to *true* satisfies the formula F. An illustration is given in Figure 10.2.

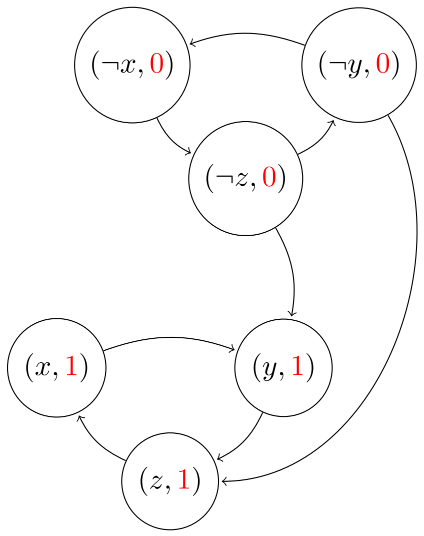


Figure 10.2: The graph from Example 10.20 with the labeling of the vertices as in the proof of Lemma 10.21. We can see that assigning *true* to all variables satisfies the formula F.

Now we can prove Lemma 10.19.

Proof. We show that $\overline{\textsf{2-SAT}}$ is in $\textsf{NL}$. By the discussion below Theorem 10.17 we then get that actually $\textsf{2-SAT}$ is in $\textsf{NL}$.

Given a 2-CNF formula F, we need to test whether F is unsatisfiable. Construct the graph G_F as we did for Lemma 10.21. This can be done in NL. Now we guess a variable x. From Theorem 10.14 we know that we can find paths from x to $\neg x$ and from $\neg x$ to x in logarithmic space. Hence, $\mathsf{2\text{-}SAT} \in \mathsf{NL}$. $\square$

Lemma 10.23. *The problem* $\mathsf{2\text{-}SAT}$ *is* NL*-hard.*

Proof. We give a logspace reduction from $\overline{\mathsf{ACYCPATH}}$ to $\mathsf{2\text{-}SAT}$. Note that $\overline{\mathsf{ACYCPATH}}$ is NL-hard by Lemma 10.18.
Let (G, s, t) be an $\overline{\mathsf{ACYCPATH}}$ instance. We construct a 2-CNF formula F as follows: the underlying set of variables of F is given by

$$\{\, x, \neg x \,|\, x \text{ is a vertex of } G \,\}.$$

For each edge $x \to y$ in G, we introduce a clause $\neg x \vee y$. Moreover, we add the clauses s and $\neg t$ for the start and the target vertex. Now it is easy to see that this $\mathsf{2\text{-}SAT}$ instance is satisfiable if and only if there is no path from s to t in G.
Note that the construction can be done in logarithmic space. $\square$

10.3 Problems in L

Goal 10.24. The board game NIM is a simple 2 player game. Computers for finding *winning strategies* for NIM were already developed in 1940. One of these computers was able to beat *Ludwig Erhard* in 1951 at the *Berliner Industrieaustellung.* In this section, we want to show that NIM can actually be decided in L.

Definition 10.25. The NIM game is defined as follows: Given is a collection of piles of sticks. The players alternately take turns and in a move, a player removes an arbitrary non-zero number of sticks from a single pile. The player who removes the very last stick wins the game.
The NIM problem is then defined as:
Input: A collection of piles $\langle s_1, \ldots, s_k \rangle$ encoded in binary.
Question: Does Player P_1 have a winning strategy from this position ?

Example 10.26. Let Player P_1 start the game from position $\langle 2, 2, 1 \rangle$. Assume the following turns:

$$\langle 2,2,1\rangle \xrightarrow{P_1} \langle 2,2,0\rangle \xrightarrow{P_2} \langle 1,2,0\rangle \xrightarrow{P_1} \langle 1,1,0\rangle \xrightarrow{P_2} \langle 0,1,0\rangle \xrightarrow{P_1} \langle 0,0,0\rangle$$

In this case, P_1 wins.

Theorem 10.27. *The problem* NIM *is in* L.

To prove the theorem, we first have to develop a *deeper understanding* of the *semantics* of NIM like we did for 2-SAT. To this end, we will need the following definition:

Definition 10.28. We assume that a position $\langle s_1, \dots, s_k \rangle$ is arranged in a matrix with rows $s_1, \dots, s_k$ and least significant bit first encoding. Such a position is called *balanced* if every column contains an even number of 1s.

Example 10.29. The position$\langle 2, 2, 1 \rangle$ is encoded by the matrix

$$\begin{bmatrix} 0 & 1 \\ 0 & 1 \\ 1 & 0 \end{bmatrix}$$

In the first column, there is an odd number of 1s. Hence, this position is unbalanced.

Finding a winning strategy relies on the following observation.

Lemma 10.30. *Let $\langle s_1, \dots, s_k \rangle = S$ be a position.*

a) If S is unbalanced, there is a move that leads to a balanced position.

b) If S is balanced, every move leads to an unbalanced position.

Proof. The proof is left to the reader. □

Example 10.31. Let us label the moves of Example 10.26 by b if a move is balanced and by u if a move is unbalanced. We obtain:

$$\underbrace{\langle 2,2,1\rangle}_{u} \xrightarrow{P_1} \underbrace{\langle 2,2,0\rangle}_{b} \xrightarrow{P_2} \underbrace{\langle 1,2,0\rangle}_{u} \xrightarrow{P_1} \underbrace{\langle 1,1,0\rangle}_{b} \xrightarrow{P_2} \underbrace{\langle 0,1,0\rangle}_{u} \xrightarrow{P_1} \underbrace{\langle 0,0,0\rangle}_{b}$$

Here, P_1 always does a move that leads to a balanced position. Hence, P_2 always generates an unbalanced position.
Note that in a balanced position that is not $\langle 0, 0, 0 \rangle$, P_2 cannot take the last stick since there are sticks on at least 2 piles. This means that P_2 cannot win when he is in a balanced position. Now the winning strategy of player P_1 is to force P_2 in balanced positions. In fact, this only depends on the initial position.

Lemma 10.32. *Player P_1 has a winning strategy if and only if the initial position is unbalanced.*

Proof. Let the initial position be unbalanced. Then P_1 chooses a move that leads to a balanced position. In a balanced position, the opponent P_2 cannot win and any move that P_2 can do, leads to an unbalanced position. Now P_1 again moves from an unbalanced position and does the same as before. Hence, P_1 has a winning strategy.
If the initial position is balanced, then the opponent P_2 has a winning strategy. He always starts from an unbalanced position and chooses the move that generates a balanced position. □

Hence, NIM reduces to checking whether the initial position $\langle s_1, \ldots, s_k \rangle$ is unbalanced. This can easily be done in deterministic logarithmic space since we only have to flip bits in any column. This proves Theorem 10.27.

Further problems in L are for example addition and multiplication:

Lemma 10.33. *Consider the two sets*

$$\begin{aligned}\mathsf{ADD} &= \{\, (x, y, z) \mid x, y, z \textit{ are encoded in binary and } x + y = z \,\}\\ \mathsf{MUL} &= \{\, (x, y, z) \mid x, y, z \textit{ are encoded in binary and } x \cdot y = z \,\}.\end{aligned}$$

Then, deciding whether a triple (x, y, z) *is in the set* ADD *or* MUL *is in* L.

Proof. The proof is left to the reader. □

Chapter 11

Models of computation for L and NL

Goal: The complexity classes L and NL were defined via Turing machines. Now a natural question arises: are there other *computation models* characterizing L or NL ? In fact, we will see in this chapter that we can characterize the classes in terms of *special automata* and *certificates.*

11.1 k-counter and k-head automata

Goal 11.1. We start with the definition of *k-counter two way automata* and *k-head two way automata.* Our goal is to prove the equivalent power of *k-counter two way automata with linearly bounded semantics*, *k-head two way automata* and logspace-bounded Turing machines.

Definition 11.2. A *k-counter two way automaton* (kCA) is a tuple $A = (\Sigma, Q, C, \rightarrow, q_0, q_f)$, where Σ is an alphabet, Q is a finite set of states, C is a set of k counters, q_0 is the initial state, q_f is the final state and

$$\rightarrow \subseteq Q \times \Sigma \times \{L, R\} \times \underbrace{\mathcal{P}(C)}_{\substack{\text{counters to be}\\ \text{tested for}\\ \text{being zero}}} \times \underbrace{\mathcal{P}(C)}_{\substack{\text{counters to be}\\ \text{increased by one}}} \times \underbrace{\mathcal{P}(C)}_{\substack{\text{counters to be}\\ \text{decreased by one}}} \times Q.$$

Intuitively, a transition rule from $\rightarrow$ works like a transition rule of a Turing machine: it reads the current state and letter and then moves the head left or right while changing the state. But in addition to that, it also acts on the set of counters C and changes the counter values.

a) The semantics of a kCA A on input x is defined in terms of *configurations*: $\text{Conf}_x^A = Q \times \mathbb{Z}^C \times [1, |x|]$. Such a configuration holds the current state, the current counter values and the current head position. The *transition relation among configurations* $\rightarrow \subseteq \text{Conf}_x^A \times \text{Conf}_x^A$ is defined as expected.

b) A kCA A is said to have *linearly bounded semantics* if the counters of A, on input x, can have value at most $|x|$ (or $-|x|$). Transitions that increment or decrement these counters are disabled.

The following famous theorem of Minsky shows that it is reasonable to bound the semantics of a k-counter automata. If we would not, the automata would be too powerful.

Theorem 11.3 (Minsky, 1967)**.** *2-counter automata are* Turing complete. *This means we can simulate any Turing machine via 2-counter automata.*

Before we show that we actually arrive at L and NL using the linearly bounded semantics, we briefly introduce another automaton model.

Definition 11.4. A *k-head two way automaton* is a finite automaton with k heads and into the input. The heads can be moved simultaneously during a transition. Note that there is no work tape and the input is read-only.

Theorem 11.5. *A language L*

(1) is decided by a logspace bounded (non-) deterministic Turing machine if and only if

(2) it is decided by a (non-) deterministic k-counter two way automaton with linearly bounded semantics if and only if

(3) it is decided by a (non-) deterministic k-head two way automaton.

Proof. We show the implications (1) $\Rightarrow$ (2) and (3) $\Rightarrow$ (1). The implication (2) $\Rightarrow$ (3) will be treat in the exercises.

First, let N be a $\mathcal{O}(\log n)$-space-bounded (non-) deterministic Turing machine with one work tape. We can assume that the tape alphabet is $\{0, 1\}$. Our goal is to simulate N via a kCA A, where the counter values are non-negative integers.

First, we show how to implement basic operations using a kCA.

- Duplicate the value of a counter c: we zero-out two other counters d and e and then we repeatedly decrement c while incrementing d and e.
- Double the value of a counter c: we zero-out a counter d. Then we repeatedly decrement c while incrementing d twice.
- Halve the value of a counter c: we zero-out a counter d. Then we repeatedly decrement c twice while incrementing d once.

- Check whether the value of a counter c is even: duplicate c and repeatedly subtract 2 from the copy. Then see whether the process leaves remainder 1.
- Add or subtract the value of a counter c to or from another counter d: for adding, increment d while decrementing c. For subtracting, decrement both until c reaches value 0.

Now we explain how to mimic configurations of N with a kCA. Given a configuration, the work tape content can be understood as a $c \cdot \log n$-bit binary number. We break this number up into c blocks of $\log n$ bits like in Figure 11.1. Each block will be represented by a counter of A.

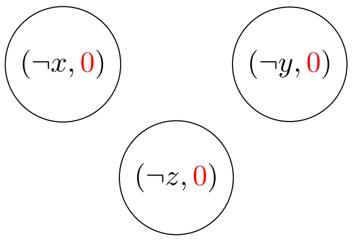


Figure 11.1: blubb

We also need to simulate the work tape head. To this end, we store the currently scanned block in A's finite control states. The position of the head in the block is stored in a counter: the i-th cell in a block is represented by the counter value 2^i.
The position of N's and A's input heads coincide. Also the control states of N are represented by the control states of A. The kCA A also needs some finite number of scratch counters so that it can perform the above mentioned operations.

□

11.2 Certificates

Chapter 12

P and NP

Text...

12.1 The Circuit Value Problem

12.2 Cook and Levin's Theorem

12.3 Context-free languages, Dynamic Programming and P

Chapter 13

PSPACE

Text...

13.1 Quantified Boolean Formula is PSPACE-complete

13.2 Winning strategies for games

13.3 Language theoretic problems

Chapter 14

Alternation

Text...

14.1 Alternating Time and Space

14.2 From Alternating Time to Deterministic Space

14.3 From Alternating Space to Deterministic Time

Chapter 15

The Polynomial Time Hierarchy

Text...

15.1 Polynomial Hierarchy defined via alternating Turing machines

15.2 A generic complete problem

Appendix A

Landau Notation

We will usually measure the complexity of an algorithm by functions $c : \mathbb{N} \to \mathbb{N}$ which take a size n and return the largest resource consumption for any input of size n. Instead of describing this function exactly, we will characterize it by its membership in classes of functions.
We are especially interested in how the resource cost scales with the size of the input. In particular, we are interested in asymptotic bounds, which characterize the behavior of a function from a certain point on. Furthermore, our notation of complexity should ignore multiplicative constants.
The Landau notation provides a formalization of those concepts.

Intuition:

$f \in \Theta(g)$	$f \in \mathcal{O}(g)$	$f \in o(g)$	$f \in \Omega(g)$	$f \in \omega(g)$
”$f = g$”	”$f \leq g$”	”$f < g$”	”$f \geq g$”	”$f > g$”

Θ-Notation: The class $\Theta(g)$ should be the class of all functions, which behave asymptotically like g.

Definition A.1. Let $g : \mathbb{N} \to \mathbb{N}$ be a function. We define $\Theta(g)$ to be the set of all functions $f : \mathbb{N} \to \mathbb{N}$ such that there is an index $n_0 \in \mathbb{N}$ and positive multiplicative constants c_1, c_2 such that from n_0 on, $f(n)$ is bounded by $c_1 \cdot g(n)$ from below and by $c_2 \cdot g(n)$ from above.

$$\Theta(g) = \{\, f : \mathbb{N} \to \mathbb{N} \mid \exists n_0 \in \mathbb{N}, c_1, c_2 \in \mathbb{R}_+ : \forall n \geq n_0 : c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \,\}$$

Remark A.2. We make use of the following notations:

- By abuse of notation, one also writes $f = \Theta(g)$ instead of $f \in \Theta(g)$.
- Sometimes, one writes $f(n) \in \Theta(g(n))$ instead of $f \in \Theta(g)$.

Example A.3. We want to prove $\frac{1}{2}n^2 - 3n \in \Theta(n^2)$. This shows that the highest occurring power of n dominates the other terms asymptotically.

Proof. We need to find c_1, c_2 such that from some n_0 on, the following inequalities are fulfilled:

$$c_1 \cdot n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 \cdot n^2.$$

We may divide all terms by n^2 to get

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2.$$

Using this inequality, $c_2 = \frac{1}{2}, c_1 = \frac{1}{14}$ seem like a reasonable choice, and one can indeed verify that from $n_0 = 7$ on, both inequalities hold. □

Example A.4. For any polynomial $f = \sum_{i=0}^{d} a_i n^i$ with a positive constant in front of the term of the highest degree (i.e. $a_d > 0$), we have $f \in \Theta(n^d)$.

$\mathcal{O}$-Notation: The class $\mathcal{O}(g)$ should be the class of all functions, which are asymptotically bounded by g from above:

Definition A.5. Let $g : \mathbb{N} \to \mathbb{N}$ be a function. We define $\mathcal{O}(g)$ to be the set of all functions $f : \mathbb{N} \to \mathbb{N}$ such that there is an index $n_0 \in \mathbb{N}$ and a positive multiplicative constant c_1 such that from n_0 on, $f(n)$ is bounded by $c \cdot g(n)$ from above.

$$\mathcal{O}(g) = \{\, f : \mathbb{N} \to \mathbb{N} \mid \exists n_0 \in \mathbb{N}, c \in \mathbb{R}_+ \colon \forall n \geq n_0 \colon f(n) \leq c \cdot g(n) \,\}$$

Example A.6. Let $f(n) = a \cdot n + b$ for positive constants a, b. Then $f \in \mathcal{O}(n)$,

Proof. Choose $c = a + b$, $n_0 = 1$. We have for all $n \in \mathbb{N}, n \geq n_0 = 1$:

$$a \cdot n + b \leq a \cdot n + b \cdot n = (a + b) \cdot n = c \cdot n.$$

□

Example A.7. By definition, $\mathcal{O}(1)$ is the set of all functions $f : \mathbb{N} \to \mathbb{N}$, which are bounded from above by a constant $c = c \cdot 1$ from a certain index n_0 on. But since each f has only finitely many values $f(0), ..., f(n_0 - 1)$ which may violate $f(n) \leq c$, we may choose $c' = \max\{c, f(0), ..., f(n-1)\}$ and get $f(n) \leq c'$ for all $n \in \mathbb{N}$. This shows that $\mathcal{O}(1)$ is actually the set of all bounded functions $f : \mathbb{N} \to \mathbb{N}$.

Remark A.8. For all $g : \mathbb{N} \to \mathbb{N}$:

$$\Theta(g) \subseteq \mathcal{O}(g).$$

o-Notation: The class $o(g)$ should be the class of all functions, which are asymptotically strictly bounded by g from above. To achieve this, the definition quantifies over arbitrary small constants.

Definition A.9. Let $g : \mathbb{N} \to \mathbb{N}$ be a function. We define $o(g)$ to be the set of all functions $f : \mathbb{N} \to \mathbb{N}$ such that for all positive multiplicative constants c, there is an index $n_c \in \mathbb{N}$ such that from n_c on, $f(n)$ is strictly bounded by $c \cdot g(n)$ from above.

$$o(g) = \{\, f : \mathbb{N} \to \mathbb{N} \mid \forall c \in \mathbb{R}_+ \colon \exists n_c \in \mathbb{N} \colon \forall n \geq n_0 : f(n) < c \cdot g(n) \,\}$$

Example A.10.

$$2n^2 \notin o(n^2).$$

Proof. For $c = 3$, we have that $2n^2 \leq 2n^3$ holds for *no* $n \leq 1$. □

Ω-Notation: The class $\Omega(g)$ should be the class of all functions, which are asymptotically bounded by g from below.

Definition A.11. Let $g : \mathbb{N} \to \mathbb{N}$ be a function. We define $\Omega(g)$ to be the set of all functions $f : \mathbb{N} \to \mathbb{N}$ such that there is an index $n_0 \in \mathbb{N}$ and a positive multiplicative constant c_1 such that from n_0 on, $f(n)$ is bounded by $c \cdot g(n)$ from below.

$$\Omega(g) = \{\, f : \mathbb{N} \to \mathbb{N} \mid \exists n_0 \in \mathbb{N}, c \in \mathbb{R}_+ \colon \forall n \geq n_0 : c \cdot g(n) \leq f(n) \,\}$$

Remark A.12.

$$\Theta(g) = \mathcal{O}(g) \cap \Omega(g)$$

Proof. Exercise! □

ω-Notation: The class $\omega(g)$ should be the class of all functions, which are asymptotically strictly bounded by g from below.

Definition A.13. Let $g : \mathbb{N} \to \mathbb{N}$ be a function. We define $\omega(g)$ to be the set of all functions $f : \mathbb{N} \to \mathbb{N}$ such that for all positive multiplicative constants c, there is an index $n_c \in \mathbb{N}$ such that from n_c on, $f(n)$ is strictly bounded by $c \cdot g(n)$ from below.

$$\omega(g) = \{\, f : \mathbb{N} \to \mathbb{N} \mid \forall c \in \mathbb{R}_+ \colon \exists n_c \in \mathbb{N} \colon \forall n \geq n_0 : c \cdot g(n) < f(n) \,\}$$

Rules:

Lemma A.14. *If $f \in \Theta(g)$ and $g \in \Theta(h)$, then $f \in \Theta(h)$.*

Proof. Choose $c = c_g \cdot c_h$ and $n_0 = max\{n_g, n_h\}$. $\square$

Remark A.15. The above lemma also holds for o, $\mathcal{O}$, ω, Ω instead of Θ. Membership in these classes is *transitive.*

Lemma A.16. *For all functions $f : \mathbb{N} \to \mathbb{N}$, we have $f \in \Theta(f)$*

Proof. Choose $c_1 = c_2 = 1$, $n_0 = 0$. $\square$

Remark A.17. The above lemma also holds for $\mathcal{O}$, Ω. Membership in these classes is *reflexive.*
It does not hold for o and ω.

Lemma A.18. *Membership in Θ is* symmetric*: $f \in \Theta(g) \iff g \in \Theta(f)$. This does not hold for the other classes.*

Some exercises:

Example A.19. Let $f, g : \mathbb{N} \to \mathbb{N}$. We define $\max(f, g)$ as the function with $(\max f, g)(n) = \max\{f(n), g(n)\}$ for all $n \in \mathbb{N}$. Claim:

$$\max(f, g) \in \Theta(f + g)$$

Proof. Choose $n_0 = 0, c_1 = \frac{1}{2}, c_2 = 1$. We have

$$\begin{aligned} c_1 \cdot (f(n) + g(n)) &\leq c \cdot 1((\max(f, g))(n) + (\max(f, g))(n)) \\ &= c \cdot 1 \cdot 2 \cdot (\max(f, g))(n) \\ &= (\max(f, g))(n) \end{aligned}$$

for all $n \in \mathbb{N}$.
We furthermore have that $(\max(f, g))(n)$ is either equal to $f(n)$ or to $g(n)$, but

$$(\max(f, g))(n) \leq 1 \cdot (f(n) + g(n))$$

holds in both cases. $\square$

Example A.20.

$$o(g) \cap \omega(g) = \emptyset$$

for any function $g : \mathbb{N} \to \mathbb{N}$.

Proof. Let g be any function. Towards a contradiction, assume that there is a function $f \in o(g) \cap \omega(g)$. This means that for any constant $c \in \mathbb{R}_+$, there are indices n_1 and n_2 such that $c \cdot g(n) < f(n)$ for all $n \geq n_1$ and $f(n) < c \cdot g(n)$ for all $n \geq n_2$. But then, for $n \geq \max\{n_1, n_2\}$, we would have $c \cdot g(n) < f(n) < c \cdot g(n)$. This inequality does not hold for any positive constant c and any $n \in \mathbb{N}$. $\square$