Theoretische Informatik

Janis Hutz
https://janishutz.com

3. Dezember 2025

TITLE PAGE COMING SOON

"*Wenn Sie die Turing-Maschine als Touring-Maschine in der Prüfung schreiben, dann macht mich das sehr traurig. Ich seh das jeweils. Teilweise sind das sehr elaborierte Trolle, manchmal Leute die nie in die Vorlesungen kommen* (2025-10-14T08:51Z+02:00")

"*Sie können also alle C Programme in Kanonischer Ordnung aufzählen. Sollten Sie dies tun. Wahrscheinlich nicht. Was aber zählt ist, sie* ***können*** *es tun*"

- Prof. Dr. Dennis Komm, 2025

HS2025, ETHZ

Summary of the book Theoretische Informatik

by Prof. Dr. Juraj Hromkovic

Inhaltsverzeichnis

- *Note: Definitions, Lemmas, etc are often 1:1 copies from the book or paraphrased (as I did not find an easier way of stating them)*
- *Note: In case I forgot to add the PDF page numbers, you can take the PDF page number is given by* $P_{PDF} = P_{Book} + 15$

1 Combinatorics

1.1 Introduction

Combinatorics was developed from the willingness of humans to gamble and the fact that everybody wanted to win as much money as possible.

1.2 Simple counting operations

The easiest way to find the best chance of winning is to write down all possible outcomes. This can be very tedious though when the list gets longer.

We can note this all down as a list or as a tree diagram. So-called Venn Diagrams might also help represent the relationship between two sets or events. Essentially a Venn Diagram is a graphical representation of set operations such as $A \cup B$.

1.3 Basic rules of counting

1.3.1 Multiplication rule

If one has n possibilities for a first choice and m possibilities for a second choice, then there are a total of $n \cdot m$ possible combinations.

When we think about a task, and we have an **and** in between e.g. properties, we need to multiply all the options.

1.3.2 Addition rule

If two events are mutually exclusive, the first has n possibilities and the second one has m possibilities, then both events together have $n + m$ possibilities.

When we think about a task, and we have an **or** in between e.g. properties, then we need to add all the options.

1.4 Factorial

Factorial **Definition 1.1**

The factorial stands for the product of the first n natural numbers where $n \geq 1$. Notation: !

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 3 \cdot 2 \cdot 1$$

Additionally, $0! = 1$. We read $n!$ as "*n factorial*"

1.4.1 Operations

We can rewrite $n!$ as $n \cdot (n-1)!$ or $n \cdot (n-1) \cdot (n-2)!$ and so on.

It is also possible to write $7 \cdot 6 \cdot 5$ with factorial notation: $\frac{7!}{4!}$, or in other words, for any excerpt of a factorial sequence:

$$n \cdot (n-1) \cdot \ldots \cdot m = \frac{n!}{(m-1)!}$$

1.5 Permutations

Permutations **Definition 1.2**

A permutation of a group is any possible arrangement of the group's elements in a particular order

Permutation rule without repetition: The number of n ***distinguishable*** elements is defined as: $n!$

1.5.1 Permutation with repetition

For n elements $n_1, n_2, \ldots, n_k$ of which some are identical, the number of permutations can be calculated as follows:

$$p = \frac{n!}{n_1! \cdot n_2! \cdot \ldots \cdot n_k!}$$

where n_k is the number of times a certain element occurs. As a matter of fact, this rule also applies to permutations without repetition, as each element occurs only once, which means the denominator is 1, hence $\frac{n!}{(1!)^n} = n!$

Beispiel 1.1: CANADA has 6 letters, of which 3 letters are the same. So the word consists of 3 A's, which can be arranged in 3! different ways, a C, N and D, which can be arranged in 1! ways each. Therefore, we have:

$$\frac{6!}{3! \cdot 1! \cdot 1! \cdot 1!} = \frac{6!}{3!} = 6 \cdot 5 \cdot 4 = 120$$

Since 1! equals 1, we can always ignore all elements that occur only once, as they won't influence the final result.

1.6 Variations

Variations **Definition 1.3**

A ***variation*** is a selection of k elements from a universal set that consists of n *distinguishable* elements.

Variation rule without repetition: The ${}_n\mathrm{P}_k$ function is used to ***place*** n elements on k places. In a more mathematical definition: The number of different variations consisting of k different elements selected from n distinguishable elements can be calculated as follows:

$$\frac{n!}{(n-k)!} = {}_n\,\mathrm{P}_k$$

1.6.1 Variations with repetition

If an element can be selected more than once and the order matters, the number of different variations consisting of k elements selected from n distinguishable elements can be calculated using n^k

1.7 Combinations

Combination **Definition 1.4**

A combination is a selection of k elements from n elements in total without any regard to order or arrangement.
Combination rule without repetition:

$${}_n\mathrm{C}_k = \binom{n}{k} = \frac{{}_n\mathrm{P}_k}{k!} = \frac{n!}{(n-k)! \cdot k!}$$

1.7.1 Combination with repetition

In general the question to ask for combinations is, in how many ways can I distribute k objects among n elements?

$${}_{n+k-1}\mathrm{C}_k = \binom{n+k-1}{k} = \frac{(n+k-1)!}{k!(n-1)!}$$

1.8 Binomial Expansion

Binomial expansion is usually quite hard, but it can be much easier than it first seems. The first term of the expression of $(a+b)^n$ is always $1a^nb^0$. Using the formula for combination without repetition, we can find the coefficients of each element:

$$\text{6th row} \begin{cases} {}_6C_0 & {}_6C_1 & {}_6C_2 & {}_6C_3 & {}_6C_4 & {}_6C_5 & {}_6C_6 \\ 1 & \frac{6}{1} & \frac{6\cdot5}{1\cdot2} & \frac{6\cdot5\cdot4}{1\cdot2\cdot3} & \frac{6\cdot5\cdot4\cdot3}{1\cdot2\cdot3\cdot4} & \frac{6\cdot5\cdot4\cdot3\cdot2}{1\cdot2\cdot3\cdot4\cdot5} & \frac{6!}{6!} \\ 1 & 6 & 15 & 20 & 15 & 6 & 1 \end{cases}$$

This theory is based on the Pascal's Triangle and the numbers of row n correspond to the coefficients of each element of the expanded term.

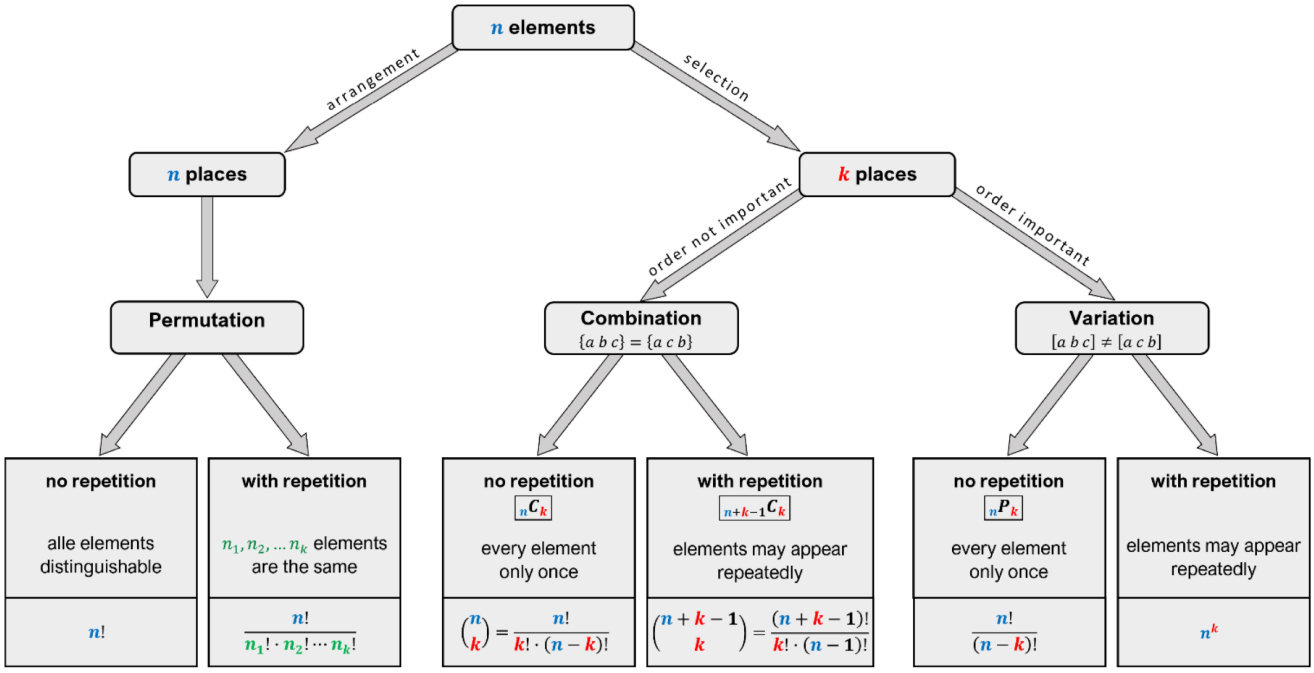
We can calculate the coefficient of each part of the expanded term k with combinatorics as follows: $\binom{n}{k}$

Binomial Expansion **Formel 1.1**

In general:

$$(a+b)^n = 1a^nb^0 + \binom{n}{1}a^{n-1}b^1 + \binom{n}{2}a^{n-2}b^2 + \ldots + \binom{n}{n-1}a^1b^{n-1} + \binom{n}{n}a^0b^n$$

1.9 Overview


n elements
arrangement
selection
n places
k places
order not important
order important
Permutation
Combination
{a b c} = {a c b}
Variation
[a b c] ≠ [a c b]
no repetition
alle elements distinguishable
n!
with repetition
n1, n2, ... nk elements are the same
n! / (n1! · n2! ··· nk!)
no repetition
nCk
every element only once
(n k) = n! / (k! · (n − k)!)
with repetition
n+k−1Ck
elements may appear repeatedly
(n+k−1 k) = (n+k−1)! / (k! · (n−1)!)
no repetition
nPk
every element only once
n! / (n − k)!
with repetition
elements may appear repeatedly
n^k

2 Alphabete, Wörter, Sprachen und Darstellung von Problemen

2.2 Alphabete, Wörter, Sprachen

Alphabet **Definition 2.1**

Eine endliche, nicht leere Menge Σ. Elemente sind Buchstaben (Zeichen & Symbole).
Beispiele: Σ_{bool}, Σ_{lat} latin characters, Σ_{Tastatur}, Σ_m m-adische Zahlen (m-ary numbers, zero index)

Wort **Definition 2.2**

Über Σ eine (möglicherweise leere) Folge von Buchstaben aus Σ. Leeres Wort λ (ab und zu ε) hat keine Buchstaben.
$|w|$ ist die Länge des Wortes (Anzahl Buchstaben im Wort), während Σ^* die Menge aller Wörter über Σ ist und $\Sigma^+ = \Sigma^* - \{\lambda\}$
In diesem Kurs werden Wörter ohne Komma geschrieben, also $x_1x_2\dots x_n$ statt $x_1, x_2, \dots, x_n$. Für das Leersymbol gilt $|\text{␣}|$, also ist es nicht dasselbe wie λ

Für viele der Berechnungen in Verbindung mit der Länge der Wörter kann Kombinatorik nützlich werden. In Kapitel 1 findet sich eine Zusammenfassung über jenes Thema (in English)

Ein mögliches Alphabet beispielsweise um einen Graphen darzustellen ist folgendes:

Angenommen, wir speichern den Graphen als Adjezenzmatrix ab, dann können wir beispielsweise mit dem Alphabet $\Sigma = \{0, 1, \#\}$ diese Matrix darstellen, in dem wir jede neue Linie mit einem # abgrenzen. Das Problem hierbei ist jedoch, dass dies nicht so effizient ist, besonders nicht, wenn der Graph sparse ist, da wir dann viele # im Vergleich zu nützlicher Information haben.

Konkatenation **Definition 2.3**

$\Sigma^* \times \Sigma^* \to \Sigma^*$, so dass $\text{Kon}(x, y) = x \cdot y = xy \ \ \forall x, y \in \Sigma^*$.
Intuitiv ist dies genau das was man denkt: Wörter zusammenhängen (wie in Programmiersprachen). Die Operation ist assoziativ und hat das Neutralelement λ, was heisst, dass (Σ^*, Kon) ein Monoid ist.
Offensichtlich ist die Konkatenation nur für ein-elementige Alphabete kommutativ.
Die Notation $(abc)^n$ wird für die n-fache Konkatenation von abc verwendet

Umkehrung **Definition 2.4**

Sei $a = a_1a_2\dots a_n$, wobei $a_i \in \Sigma$ für $i \in \{1, 2, \dots, n\}$, dann ist die Umkehrung von a, $a^{\text{R}} = a_na_{n-1}\dots a_1$

Iteration **Definition 2.5**

Die i-te Iteration x^i von $x \in \Sigma^*$ für alle $i \in \mathbb{N}$ ist definiert als $x^0 = \lambda$, $x^1 = x$ und $x^i = xx^{i-1}$

Teilwort, Präfix, Suffix **Definition 2.6**

Seien $v, w \in \Sigma^*$
- v heisst ***Präfix*** von $w \iff \exists y \in \Sigma^* : w = vy$
- v heisst ***Suffix*** von $w \iff \exists x \in \Sigma^* : w = xv$
- v heisst ***Teilwort*** von $w \iff \exists x, y \in \Sigma^* : w = xvy$
- $v \neq \lambda$ heisst ***echtes*** Teilwort (gilt auch für Präfix, Suffix) von w genau dann, wenn $v \neq w$ und v ein Teilwort (oder eben Präfix oder Suffix) von w ist

Kardinalität, Vorkommen und Potenzmenge — Definition 2.7

Für Wort $x \in \Sigma^*$ und Buchstabe $a \in \Sigma$ ist $|x|_a$ definiert als die Anzahl Male, die a in x vorkommt.
Für jede Menge A ist $|A|$ die Kardinalität und $\mathcal{P}(A) = \{S | S \subseteq A\}$ die Potenzmenge von A

Kanonische Ordnung — Definition 2.8

Wir definieren eine Ordnung $s_1 < \ldots < s_m$ auf Σ. Die ***kanonische Ordnung*** auf Σ^* für $u, v \in \Sigma^*$ ist definiert als:

$$u < v \iff |u| < |v| \vee (|u| = |v| \wedge u = x \cdot s_i \cdot u' \wedge v = x \cdot s_j \cdot v') \text{ für beliebige } x, u', v' \in \Sigma^* \text{ und } i < j$$

Oder in Worten, geordnet nach Länge und dann danach für den ersten nicht gemeinsamen Buchstaben, nach dessen Ordnung.

Sprache — Definition 2.9

$L \subseteq \Sigma^*$ ist eine Sprache, deren Komplement $L^C = \Sigma^* - L$ ist. Dabei ist $L_\emptyset$ die ***leere Sprache*** und L_λ die einelementige Sprache die nur aus dem leeren Wort besteht.
Die ***Konkatenation*** von L_1 und L_2 ist $L_1 \cdot L_2 = L_1 L_2 = \{vw \mid v \in L_1 \wedge w \in L_2\}$ und $L^0 := L_\lambda$ und $L^{i+1} = L^i \cdot L \;\; \forall i \in \mathbb{N}$ und $L^* = \bigcup_{i \in \mathbb{N}} L^i$ ist der ***Kleene'sche Stern*** von L, wobei $L^+ = \bigcup_{i \in \mathbb{N} - \{0\}} L^i = L \cdot L^*$

Für jede Sprache L gilt $L^2 \subseteq L \implies L = \emptyset \vee L = \{\lambda\} \vee L$ ist undendlich. Diese Aussage muss jedoch an der Prüfung bewiesen werden (nicht im Buch vorhanden)

Da Sprachen Mengen sind, gelten auch die Üblichen Operationen, wie Vereinigung ($\cup$) und Schnitt ($\cap$). Die Gleichheit von zwei Sprachen bestimmen wir weiter mit $A \subseteq B \wedge B \subseteq A \Rightarrow A = B$. Um $A \subseteq B$ zu zeigen reicht es hier zu zeigen dass für jedes $x \in A$, $x \in B$ hält. Wir betrachten nun, wie die üblichen Operationen mit der neu hinzugefügten Konkatenation interagieren.

Distributivität von Kon und $\cup$ — Lemma 2.1

Für Sprachen L_1, L_2 und L_3 über Σ gilt: $L_1 L_2 \cup L_1 L_3 = L_1 (L_2 \cup L_3)$

Der Beweis hierfür läuft über die oben erwähnte "Regel" zur Gleichheit. Um das Ganze einfacher zu machen, teilen wir auf: Wir zeigen also erst $L_1 L_2 \subseteq L_1 (L_2 \cup L_3)$ und dann equivalent für $L_1 L_3$.

Distributivität von Kon und $\cap$ — Lemma 2.2

Für Sprachen L_1, L_2 und L_3 über Σ gilt: $L_1 (L_2 \cap L_3) \subseteq L_1 L_2 \cap L_1 L_3$

L 2.3: Es existieren $U_1, U_2, U_3 \in (\Sigma_{\text{bool}})^*$, so dass $U_1 (U_2 \cap U_3) \subsetneq U_1 U_2 \cap U_1 U_3$

Homomorphismus — Definition 2.10

Σ_1, Σ_2 beliebige Alphabete. Ein ***Homomorphismus*** von Σ_1^* nach Σ_2^* ist jede Funktion $h : \Sigma_1^* \to \Sigma_2^*$ mit:
(i) $h(\lambda) = \lambda$
(ii) $h(uv) = h(u) \cdot h(v) \;\; \forall u, v \in \Sigma_1^*$

Erneut gilt hier, dass im Vergleich zu allgemeinen Homomorphismen, es zur Definition von einem Homomorphismus ausreichtt, $h(a)$ für alle Buchstaben $a \in \Sigma_1$ festzulegen.

2.3 Algorithmische Probleme

Ein Algorithmus $A : \Sigma_1^* \to \Sigma_2^*$ ist eine Teilmenge aller Programme, wobei ein Program ein Algorithmus ist, sofern es für jede zulässige Eingabe eine Ausgabe liefert, es darf also nicht eine endlosschleife enthalten.

Entscheidungsproblem **Definition 2.11**

Das ***Entscheidungsproblem*** (Σ, L) ist für jedes $x \in \Sigma^*$ zu entscheiden, ob $x \in L$ oder $x \notin L$. Ein Algorithmus A löst (Σ, L) (erkennt L) falls für alle $x \in \Sigma^*$: $A(x) = \begin{cases} 1, & \text{falls } x \in L \\ 0, & \text{falls } x \notin L \end{cases}$.

Funktion **Definition 2.12**

Algorithmus A berechnet (realisiert) eine ***Funktion (Transformation)*** $f : \Sigma^* \to \Gamma^*$ falls $A(x) = f(x)$ $\forall x \in \Sigma^*$ für Alphabete Σ und Γ

Berechnung **Definition 2.13**

Sei $R \subseteq \Sigma^* \times \Gamma^*$ eine Relation in den Alphabeten Σ und Γ. Ein Algorithmus A ***berechnet*** R (***löst das Relationsproblem*** R) falls für jedes $x \in \Sigma^*$, für das ein $y \in \Gamma^*$ mit $(x, y) \in R$ existiert gilt: $(x, A(x)) \in R$

2.4 Kolmogorov-Komplexität

Falls ein Wort x eine kürzere Darstellung hat, wird es ***komprimierbar genannt*** und wir nennen die Erzeugung dieser Darstellung eine ***Komprimierung*** von x.

Eine mögliche Idee, um den Informationsgehalt eines Wortes zu bestimmen, wäre einem komprimierbaren Wort einen kleinen Informationsgehalt zuzuordnen und einem unkomprimierbaren Wort einen grossen Informationsgehalt zuzuordnen.

Wenn wir also das Wort 011011011011011011011011 haben, so kann man es auch als $(011)^8$ darstellen und hat so also einen kleineren Informationsgehalt als bspw. 010110100010110100111011001O.

Die Idee mit der Komprimierung den Informationsgehalt zu bestimmen ist jedoch nicht ideal, da für jede Komprimierung bei unendlich langen Wörtern immer eine weitere Komprimierung existiert, die für unendlich viele Wörter besser geeignet ist.

Hier kommt die Kolmogorov-Komplexit zum Zuge: Sie bietet eine breit Gültige Definition des Komplexitätsmasses.

Kolmogorov-Komplexität **Definition 2.17**

Für jedes Wort $x \in (\Sigma_{\text{bool}})^*$ ist die ***Kolmogorov-Komplexität*** $K(x)$ ***des Wortes*** x das Minimum der binären Längen der Pascal-Programme, die x generieren.

Hierbei ist mit der binären Länge die Anzahl Bits gemeint, die beim Übersetzen des Programms in einen vordefinierten Maschinencode entsteht.

Ein Pascal-Programm in diesem Kurs ist zudem nicht zwingend ein Programm in der effektiven Programmiersprache Pascal, sondern eine Abwandlung davon, worin es auch erlaubt ist, gewisse Prozesse zu beschreiben und nicht als Code auszuformulieren, da das nicht das Ziel dieses Kurses ist.

Kolmogorov-Komplexität **Lemma 2.4**

Für jedes Wort $x \in (\Sigma_{\text{bool}})^*$ existiert eine Konstante d so dass $K(x) \leq |x| + d$

Beweis: Für jedes $x \in (\Sigma_{\text{bool}})^*$ kann folgendes Programm A_x verwendet werden:

```
1 $A_x$: begin
2 write(x);
3 end
```

Alle Teile, ausser x sind dabei von konstanter Länge, also ist die Länge der Bit-repräsentation des Programms ausschliesslich von der binären Länge des Wortes x abhängig.

□

Für regelmässige Wörter gibt es natürlich Programme, bei denen das Wort nicht als komplette Variable vorkommt. Deshalb haben diese Wörter auch (meist) eine kleinere Kolmogorov-Komplexität.

Definition 2.18: *($K(n)$ für $n \in \mathbb{N}$)* Die ***Kolmogorov-Komplexität einer natürlichen Zahl*** n ist $K(n) = K(\text{Bin}(n))$, wobei $|\text{Bin}(x)| = \lceil \log_2(x+1) \rceil$

Lemma 2.5: Für jede Zahl $n \in \mathbb{N} - \{0\}$ existiert ein Wort $w_n \in (\Sigma_{\text{bool}})^n$ so dass $K(w_n) \geq |w_n| = n$, oder in Worten, es existiert für jedes n ein nicht komprimierbares Wort.

Eine wichtige Eigenschaft der Kolmogorov-Komplexität ist, dass sie nicht wirklich von der gewählten Programmiersprache abhängt. Man kann also beliebig auch `C++`, `Swift`, `Python`, `Java` oder welche auch immer, ohne dass die Kolmogorov-Komplexität um mehr als eine Konstante wächst (auch wenn diese bei `Java` sehr gross ist):

Unterschiedliche Programmiersprachen — **Satz 2.1**

Für jede Programmiersprachen A und B existiert eine Konstante $c_{A,B}$, die nur von A und B abhängig ist, so dass für alle $x \in (\Sigma_{\text{bool}})^*$ gilt:

$$|K_A(x) - K_B(x)| \leq c_{A,B}$$

Anwendungen der Kolmogorov-Komplexität

Zufall Der Zufall ist ein intuitiver, aber nicht sehr formeller Begriff, der mit der Kolmogorov-Komplexität formalisiert werden kann:

Zufall — **Definition 2.19**

Ein Wort $x \in (\Sigma_{\text{bool}})^*$ (eine Zahl n) heisst ***zufällig***, falls $K(x) \geq |x|$ ($K(n) = K(\text{Bin}(n)) \geq \lceil \log_2(n+1) \rceil - 1$)

Existenz eines Programms vs Kolmogorov-Komplexität

Programm vs Komplexität — **Satz 2.2**

Sei L eine Sprache über Σ_{bool} und für jedes $n \in \mathbb{N} - \{0\}$ sei z_n das n-te Wort in L bezüglich der kanonischen Ordnung. Falls ein Programm A_L existiert, das das Entscheidungsproblem $(\Sigma_{\text{bool}}, L)$ löst, so gilt für alle $n \in \mathbb{N} - \{0\}$ dass

$$K(z_n) \leq \lceil \log_2(n+1) \rceil + c \qquad (c \text{ ist eine von } n \text{ unabhängige Konstante})$$

Primality testing

Primzahlensatz — **Satz 2.3**

$$\lim_{n \to \infty} \frac{\text{Prim}(n)}{\frac{n}{\ln(n)}} = 1$$

Die Annäherung von $\text{Prim}(n)$ and $\frac{n}{\ln(n)}$ wird durch folgende Ungleichung gezeigt:

$$\ln(n) - \frac{3}{2} < \frac{n}{\text{Prim}(n)} < \ln(n) - \frac{1}{2} \quad \forall n \geq 67 \in \mathbb{N}$$

Anzahl Primzahlen mit Eigenschaften — **Lemma 2.6**

Sei $n_1, n_2, \ldots$ eine stetig steigende unendliche Folge natürlicher Zahlen mit $K(n_i) \geq \frac{\lceil \log_2(n_i) \rceil}{2}$. Für jedes $i \in \mathbb{N} - \{0\}$ sei q_i die grösste Primzahl, die n_i teilt. Dann ist die Menge $Q = \{q_i \mid i \in \mathbb{N} - \{0\}\}$ unendlich.

Lemma 2.6 zeigt nicht nur, dass es unendlich viele Primzahlen geben muss, sondern sogar, dass die Menge der grössten Primzahlfaktoren einer beliebigen unendlichen Folge natürlicher Zahlen mit nichttrivialer Kolmogorov-Komplexität unendlich ist.

Untere Schranke für Anzahl Primzahlen — **Satz 2.4**

Für unendlich viele $k \in \mathbb{N}$ gilt

$$\text{Prim}(k) \geq \frac{k}{2^{17} \log_2(k) \cdot (\log_2(\log_2(k)))^2}$$

Der Beweis hierfür ist sehr ausführlich ab Seite 42 (= 57 im PDF) im Buch erklärt

3 Endliche Automaten

3.2 Darstellung

Folgende Fragen müssen zur Definition eines Berechnungsmodells beantwortet werden:

1. Welche elementaren Operationen stehen zur Verfügung (um das Programm zusammenzustellen)?
2. Wie funktioniert der Speicher?
3. Wie funktioniert die Eingabe (und welches Alphabet verwendet sie)?
4. Wie funktioniert die Ausgabe (und welches Alphabet verwendet sie)?

Endliche Automaten haben keinen Speicher, mit Ausnahme des Zeigers (can be understood similarly to a program counter)

Ein endlicher Automat mit dem Eingabealphabet $\Sigma = \{a_1, \dots, a_k\}$ darf nur den Operationstyp `select` verwenden.

$$\begin{aligned} &\texttt{select } \textit{input} = a_1 \texttt{ goto } i_1 \\ &\vdots \\ &\textit{input} = a_k \texttt{ goto } i_k \end{aligned}$$

Alternativ, falls $|\Sigma| = 2$ (typischerweise für Σ_{bool}), kann man statt `select` auch `if...then...else` nutzen. Typischerweise werden solche Programme für Entscheidungsprobleme genutzt und die Checks sind dann:

$$\texttt{if } \textit{input} = 1 \texttt{ then goto } i \texttt{ else goto } j$$

Wir wählen eine Teilmenge $F \subseteq \{0, \dots, m-1\}$, wobei m die Anzahl Zeilen des Programms ist. Ist die Zeile auf der das Programm endet ein Element von F, so akzeptiert das Programm die Eingabe. Die Menge F wird auch die ***vom Programm akzeptierte Sprache*** genannt Ein Programm A arbeitet dann Buchstabe für Buchstabe das Eingabewort ab und springt so also kontinuierlich durch das Programm bis die Eingabe endet. Mit formaleren Begriffen ist das Eingabewort als ***Band*** dargestellt, welches von einem ***Lesekopf***, der sich nur nach links oder rechts bewegen kann gelesen wird und die gelesene Eingabe dann dem ***Programm*** weitergibt.

Diese Notation wird jedoch heute kaum mehr verwendet (because `goto` bad, Prof. Roscoe would approve). Heute verwendet man meist einen gerichteten Graphen $G(A)$:

- Hat so viele Knoten (= ***Zustände***) wie das Programm A Zeilen hat
- Wenn das Programm beim Lesen von Symbol b von Zeile i auf j sprint, so gibt es in $G(A)$ eine gerichtete Kante (i, j) von Knoten i nach Knoten j mit Markierung b. Sie wird als ***Übergangsfunktion*** bezeichnet
- Jeder Knoten hat den Ausgangsgrad $|\Sigma|$ (wir müssen alle Fälle abdecken)

Endlicher Automat **Definition 3.1**

Ist eine Quitupel $M = (Q, \Sigma, \delta, q_0, F)$:
(i) Q ist eine endliche Menge von ***Zuständen***
(ii) Σ ist das ***Eingabealphabet***
(iii) $\delta : Q \times \Sigma \to Q$ ist die ***Übergangsfunktion***. $\delta(q, a) = p$ bedeutet Übergang von Zustand q nach p falls in q a gelesen wurde
(iv) $q_0 \in Q$ ist der ***Anfangszustand***
(v) $F \subseteq Q$ ist die ***Menge der akzeptierenden Zustände***

- ***Konfiguration***: Element aus $Q \times \Sigma^*$
- ***Endkonfiguration***: Jede aus $Q \times \{\lambda\}$
- ***Startkonfiguration*** auf x: (q_0, x)
- ***Schritt***: Relation auf Konfigurationen $\vdash_M \subseteq (Q \times \Sigma^*) \times (Q \times \Sigma^*$ definiert durch $(q, w) \vdash_M (p, x) \Leftrightarrow w = ax, a \in \Sigma$ und $\delta(q, a) = p$. Einfacher: Anwendung von δ auf die aktuelle Konfiguration
- ***Berechnung*** C: Endliche Folge von Konfigurationen, $C_i \vdash_M C_{i+1}$. Auf Eingabe $x \in \Sigma^*$, C_0 Startkonfiguration und C_n Endkonfiguration. Falls $C_n \in F \times \{\lambda\}$, C ***akzeptierende Berechnung***, M ***akzeptiert Wort*** x. Anderenfalls ist C eine ***verwerfende Berechnung*** und M ***verwirft (akzeptiert nicht) das Wort*** x
- ***Akzeptierte Sprache*** $L(M) = \{w \in \Sigma^* \mid M$ akzeptiert das Wort w und M endet in Endkonfig.$\}$
- $\mathcal{L}_{EA} = \{L(M) | M$ ist ein EA$\}$ ist die Klasse aller Sprachen die von endlichen Automaten akzeptiert werden, auch genannt ***Klasse der regulären Sprachen*** und für jede Sprache $L \in \mathcal{L}_{EA}$ gilt: L regulär

Die Übergangsfunktion kann auch gut graphisch oder tabellarisch (wie eine Truth-Table) dargestellt werden.

M ist in der Konfiguration $(q, w) \in Q \times \Sigma^*$, wenn M in Zustand q ist und noch das Suffix w zu lesen hat (also auf dem Eingabeband hinter dem Zeiger noch w steht)

Reflexive und transitive Hülle — **Definition 3.2**

Sei $M = (Q, \Sigma, \delta, q_0, F)$ ein endlicher Automat. Die reflexive und transitive Hülle $\vdash_M^*$ der Schrittrelation $\vdash_M$ von M als $(q, w) \vdash_M^* (p, u) \Leftrightarrow (q = p \wedge w = u) \vee \exists k \in \mathbb{N} - \{0\}$ so dass

(i) $w = a_1 \dots a_k u, a_i \in \Sigma$ für $i = 1, \dots, k$

(ii) $\exists r_1, \dots, r_{k-1} \in Q$, so dass $(q, w) \vdash_M (r_1, a_2 \dots a_k u) \vdash_M (r_2, a_3 \dots a_k u) \vdash_M \dots (r_{k-1}, a_k u) \vdash_M (p, u)$

Wir definieren $\widehat{\delta} : Q \times \Sigma^* \to Q$ durch

(i) $\widehat{\delta}(q, \lambda) = q \quad \forall q \in Q$ *(ii)* $\widehat{\delta}(q, wa) = \delta(\widehat{\delta}(q, w), a) \forall a \in \Sigma, w \in \Sigma^*, q \in Q$

$\vdash_M^*$ und $\widehat{\delta}(q, w)$ — **Intuition**

$(q, w) \vdash_M^* (p, u)$ bedeutet, dass es eine Berechnung von M gibt, die von der Konfiguration (q, w) zu (p, u) führt. Eine wichtiger Aspekt ist die Transitivität, was ja dann bedeutet, dass es (beliebig viele) Zwischenschritte gibt, so dass die Relation erfüllt ist. Oder noch viel einfacher: Es gibt irgendwieviele Zwischenschritte zwischen dem linken und rechten Zustand

$\widehat{\delta}(q, w) = p$ repräsentiert den letzen Zustand der Berechnung ausgehend von (q, w). Etwas formaler bedeutet dies $(q, w) \vdash_M^* (p, \lambda)$, also falls M im Zustand q das Wort w zu lesen beginnt, M im Zustand p endet.

Also gilt $L(M) = \{w \in \Sigma^* \mid (q_0, w) \vdash_M^* (p, \lambda) \quad \forall p \in F\} = \{w \in \Sigma^* \mid \widehat{\delta}(q_0, w) \in F\}$.

Das folgende Lemma bezieht sich auf den Automaten M, den wir in der Tabelle weiter unten definieren. Der Automat entscheidet, ob die beiden Zahlen gerade oder ungerade sind. Dies kann man aber auch folgendermassen in formaler Ausdrucksweise ausdrücken:

Lemma 3.1: $L(M) = \{w \in \{0, 1\}^* \mid |w|_0 + |w|_1 \equiv 0 \bmod 2\}$

Jeder EA teilt die Menge Σ^* in $|Q|$ Klassen $\text{Kl}[p] = \{w \in \Sigma^* \mid \widehat{\delta}(q_0, w) = p\} = \{w \in \Sigma^* \mid (q_0, w) \vdash_M^* (p, \lambda)\}$ auf und entsprechend gilt:

$$\bigcup_{p \in Q} \text{Kl}[p] = \Sigma^* \text{ und } \text{Kl}[p] \cup \text{Kl}[q] = \emptyset \quad \forall p \neq q \in Q$$

In dieser Terminologie gilt dann $L(M) = \bigcup_{p \in F} \text{Kl}[p]$. Die Notation $|w|_i$ bedeutet die Länge der Buchstaben i in w.

Wir können $L(M)$ mit Klassen bestimmen und haben eine Äquivalenzrelation $xR_\delta y \Leftrightarrow \widehat{\delta}(q_0, x) = \widehat{\delta}(q_0, y)$ auf Σ^*. Man beweist die Korrektheit der gewählten Klassen oft mithilfe von Induktion über die Länge der Wörter. Wir beginnen mit der Länge an Wörtern der Länge kleiner gleich zwei und erhöhen dies dann während unseres Induktionsschrittes.

Intuition: Die Klassen sind Mengen, die hier Wörter mit gewissen Eigenschaften, die der EA bestimmt hat, wenn er in Zustand q_i endet, enthalten. Diese Eigenschaften sind beispielsweise, dass alle Wörter, für die der EA in Zustand q_i endet mit einer gewissen Sequenz enden, sie einen gewissen Zahlenwert haben, etc.

Die Klassen bestimmen wir vor dem Beginn der Induktion auf und jede Klasse repräsentiert einen der Zustände.

Zustand	0	1
q_0	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

Haben wir einen EA M mit nebenstehender Tabelle, so sind die Klassen $\text{Kl}[q_0], \dots, \text{Kl}[q_3]$, definiert durch:

$$\text{Kl}[q_0] = \{w \in (\Sigma_{\text{bool}})^* \mid |w|_0 \text{ und } |w|_1 \text{ sind gerade}\}$$
$$\text{Kl}[q_1] = \{w \in (\Sigma_{\text{bool}})^* \mid |w|_0 \text{ ist gerade, } |w|_1 \text{ ist ungerade}\}$$
$$\text{Kl}[q_2] = \{w \in (\Sigma_{\text{bool}})^* \mid |w|_0 \text{ ist ungerade, } |w|_1 \text{ ist gerade}\}$$
$$\text{Kl}[q_3] = \{w \in (\Sigma_{\text{bool}})^* \mid |w|_0 \text{ und } |w|_1 \text{ sind ungerade}\}$$

Falls ein EA A genügend anschaulich und strukturiert dargestellt ist, kann man die Sprache $L(A)$ auch ohne Beweis bestimmen.

Idealerweise konstruieren wir einen EA so, dass wir die Menge aller Wörter aus Σ^* so in Klassen aufteilen, sodass Wörter mit denselben Eigenschaften in derselben Klasse liegen und wir dann Übergangsfunktionen zu anderen Klassen finden, die nur einen Buchstaben aus Σ zum Wort hinzufügen

Beispiel 3.1: Das Buch enthält einige zwei gute Beispiele (Beispiel 3.1 und 3.2) mit ausführlichen Erklärungen ab Seite 58 (= Seite 73 im PDF).

3.3 Simulationen

Der Begriff der Simulation ist nicht ein formalisiert, da er je nach Fachgebiet, eine etwas andere Definition hat. Die engste Definition fordert, dass jeder elementare Schritt der zu Berechnung, welche simuliert wird, durch eine Berechnung in der Simulation nachgemacht wird. Eine etwas schwächere Forderung legt fest, dass in der Simulation auch mehrere Schritte verwendet werden dürfen.

Es gibt auch eine allgemeinere Definition, die besagt, dass nur das gleiche Eingabe-Ausgabe-Verhalten gilt und der Weg, oder die Berechnungen, welche die Simulation geht, respektive durchführt, wird ignoriert, respektive wird nicht durch die Definition beschränkt.

Hier werden wir aber die enge Definition verwenden

Lemma 3.2: *(Produktautomaten)* Wir haben zwei EA $M_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$ und $M_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$, die auf dem Alphabet Σ operieren. Für jede Mengenoperation $\odot \in \{\cup, \cap, -\}$ existiert ein EA M, so dass $L(M) = L(M_1) \odot L(M_2)$

Was dieses Lemma nun aussagt ist folgendes: Man kann einen endlichen Automaten bauen, so dass das Verhalten von zwei anderen EA im Bezug auf die Mengenoperation simuliert wird. Ein guter, ausführlicher Beweis dieses Lemmas findet sich im Buch auf Seite 64 (= Seite 79 im PDF)

Dieses Lemma hat weitreichende Nutzen. Besonders ist es also möglich einen modularen EA zu bauen, in dem Teile davon in kleinere und einfachere EA auszulagern, die dann wiederverwendet werden können.

Produktautomaten — **Intuition**

Produktautomaten erstellt man, in dem man die (meist zwei) Automaten als einen Gridgraph aufschreibt und eine Art Graph-Layering betreibt, so dass der eine Graph horizontal und der andere Graph vertikal orientiert ist. Dann werden die Übergänge folgendermassen definiert: Für jeden Eingang liefert der Graph, der horizontal ausgerichtet ist, ob wir nach links oder rechts gehen (oder bleiben), während der vertikal ausgerichtete Graph entscheidet, ob wir nach oben oder unten gehen (oder bleiben).

Beispiel 3.3: Dieses Beispiel im Buch ist sehr gut erklärt und findet sich auf Seiten 65, 66 & 67 (= Seite 80, 81 & 82 im PDF)

3.4 Beweise der Nichtexistenz

Im Gegensatz zum Beweis, dass eine bestimmte Klasse von Programmen (Algorithmen) ein Problem lösen kann (was ein einfacher Existenzbeweis ist, bei welchem man eine korrekte Implementation liefern kann), ist der Beweis, dass diese Klasse von Programmen (Algorithmen) dies nicht tun kann viel schwieriger, da man (logischerweise) nicht für alle (unendlich vielen) Programme zeigen kann, dass sie das Problem nicht lösen.

In diesem Kurs werden wir aber vorerst nur die Klasse der endlichen Automaten behandlen, welche sehr stark eingeschränkt sind, was diese Beweise verhältnismässig einfach macht. Falls also ein EA A für zwei unterschiedliche Wörter x und y im gleichen Zustand endet (also $\widehat{\delta}(q_0, x) = \widehat{\delta}(q_0, y)$)), so heisst das für uns von jetzt an, dass A nicht zwischen x und y unterscheiden kann:

Unterscheidung von Wörtern — **Lemma 3.3**

Sei A ein EA über Σ und $x \neq y \in \Sigma^*$ so dass

$$(q_0, x) \overset{*}{\underset{A}{\vdash}} (p, \lambda) \text{ und } (q_0, y) \overset{*}{\underset{A}{\vdash}} (p, \lambda)$$

für ein $p \in Q$ (also $\widehat{\delta}_A(q_0, x) = \widehat{\delta}(q_0, y) = p(x, y \in \text{Kl}[p])$). Dann existiert für jedes $z \in \Sigma^*$ ein $r \in Q$, so dass $xz, yz \in \text{Kl}[r]$, also gilt insbesondere

$$xz \in L(A) \iff yz \in L(A)$$

Das obenstehende Lemma 3.3 ist ein Spezialfall einer Eigenschaft, die für jedes (deterministische) Rechnermodell gilt. Es besagt eigentlich nichts anderes, als dass wenn das Wort xz akzeptiert wird, so wird auch das Wort yz

Mithilfe von Lemma 3.3 kann man für einen grossteil Sprachen deren Nichtregularität beweisen.

Beispiel: Sei $L = \{0^n 1^n \mid n \in \mathbb{N}\}$. Intuitiv ist diese Sprache Nichtregulär, da n unendlich gross sein kann, aber ein EA logischerweise endlich ist. Wir müssen hier nur formal ausdrücken, dass das Zählen benötigt wird, dass L akzeptiert wird:

Dazu benutzen wir einen Widerspruchsbeweis. Sei A ein EA über Σ_{bool} und $L(A) = L$. Wir nehmen an, dass L regulär ist und betrachten die Wörter $0^1, 0^2, \ldots, 0^{|Q|+1}$. Weil wir $|Q| + 1$ Wörter haben, existiert per Pigeonhole-Principle o.B.d.A $i < j \in \{1, 2, \ldots, |Q| + 1\}$ (die Ungleichheit kann in komplexeren Beweisen sehr nützlich werden, da wir dann besser mit Längen argumentieren können), so dass $\widehat{\delta}_A(q_0, 0^i) = \widehat{\delta}_A(q_0, 0^j)$, also gilt nach Lemma $0^i z \in L \Leftrightarrow 0^j z \in L \ \forall z \in (\Sigma_{\text{bool}})^*$. Dies gilt jedoch nicht, weil für jedes $z = 1^i$ zwar jedes $0^i 1^i \in L$ gilt, aber $0^i 1^j \notin Lh$

Um die Nichtregularität konkreter Sprachen zu beweisen, sucht man nach einfach verifizierbaren Eigenschaften, denn wenn eine Sprache eine dieser Eigenschaften *nicht* erfüllt, so ist sie nicht regulär.

Pumping

Eine weitere Methode zum Beweis von Aussagen $L \notin \mathcal{L}_{\text{EA}}$ nennt sich ***Pumping*** und basiert auf folgender Idee: Wenn für ein Wort x und einen Zustand p gilt, dass $(p, x) \overset{*}{\underset{A}{\vdash}} (p, \lambda)$, so gilt auch für alle $i \in \mathbb{N}$, dass $(p, x^i) \overset{*}{\underset{A}{\vdash}} (p, \lambda)$. Also kann A nicht zwischen x und x^i unterscheiden, oder in anderen Worten, wie viele x er gelesen hat, also akzeptiert A entweder alle Wörter der Form $yx^i z$ (für $i \in \mathbb{N}$) oder keines davon

Pumping-Lemma für reguläre Sprachen — **Lemma 3.4**

Sei L regulär. Dann existiert eine Konstante $n_0 \in \mathbb{N}$, so dass sich jedes Wort $w \in \Sigma^*$ mit $|w| \geq n_0$ in $w = yxz$ zerlegen lässt, wobei

(i) $|yx| \leq n_0$
(ii) $|x| \geq 1$
(iii) Für $X = \{yx^k z \mid k \in \mathbb{N}\}$ *entweder* $X \subseteq L$ oder $X \cap L = \emptyset$ gilt

Bei der Wahl von den Teilen von w sollte man idealerweise einen Teil (der dann $= y$ in $w = yxz$ ist) bereits gross genug zu wählen, so dass (i) zutrifft, was es nachher einfacher macht.

Beispiel: Wir verwenden wieder die Sprache $L = \{0^n 1^n \mid n \in \mathbb{N}\}$ und wieder einen Widerspruchsbeweis:

Wir nehmen an, dass L regulär ist, also gilt Lemma 3.4 und es existiert eine Konstante n_0 so dass $|w| \geq n_0$. Um zu zeigen, dass eine Sprache nicht regulär ist, reicht es aus, zu zeigen, dass es ein (hinreichend langes) Wort gibt, für das eine der Eigenschaften in Lemma 3.4 nicht zutrifft.

Wir wählen $w = 0^{n_0}1^{n_0}$, also ist $|w| = 2n_0 \geq n_0$. Zudem müssen wir eine sinnvolle Zerlegung wählen – denn eine solche existiert für jedes Wort w mit $|w| \geq n_0$ laut Lemma 3.4 – wir wählen $yx = 0^{n_0}$, also ist $y = 0^l$ und $x = 0^m$ für irgendwelche $l, m \in \mathbb{N}$, so dass $l + m \leq n_0$.

Nach Lemma 3.4 (ii) ist $m \neq 0$ ($|x| \geq 1$). Nun, da $w = 0^{n_0}1^{n_0} \in L$, ist $\{yx^k z \mid k \in \mathbb{N}\} = \{0^{n_0-m+km}1^{n_0} \mid k \in \mathbb{N}\} \subseteq L$, was aber ein Widerspruch ist, da $yx^0z = yz = 0^{n_0-m}1^{n_0} \notin L$ ($0^{n_0}1^{n_0}$ ist sogar das einzige Wort aus der Menge, das in L liegt)

Intuition: Woher kommt 0^{n_0-m+km}? Das Ganze wird mit Klammern bedeutend offensichtlicher: $0^{(n_0-m)+(km)}$. Also ist der Ursprung der Koeffizienten auch klar, und sie kommen von $|y| = n_0 - m$ und $|x^k| = km$. Die Addition im Exponent kommt dann deshalb zustande, da dies ja nicht ein Exponent ist, sondern die Anzahl der Repetitionen.

Kolmogorov-Komplexität basiert

Kolmogorov-Komplexität regulärer Sprachen — Satz 3.1

Sei $L \subseteq (\Sigma_{\text{bool}})^*$ eine reguläre Sprache. Sei $L_x = \{y \in (\Sigma_{\text{bool}})^* \mid xy \in L\}$ für jedes $x \in (\Sigma_{\text{bool}})^*$. Dann existiert eine Konstante c, so dass für alle $x, y \in (\Sigma_{\text{bool}})^*$ gilt, dass

$$K(y) \leq \lceil \log_2(n+1) \rceil + c$$

falls y das n-te Wort in der Sprache L_x ist

Beispiel: Wir verwenden wieder die Sprache $L = \{0^n 1^n \mid n \in \mathbb{N}\}$ und wieder einen Widerspruchsbeweis:

Dazu nehmen wir wieder an, dass L regulär ist. Für jedes $m \in \mathbb{N}$ ist 1^m das erste Wort in der Sprache $L_{0^m} = \{y \mid 0^m y \in L\} = \{0^j 1^{m+j} \mid j \in \mathbb{N}\}$. Die zweite Menge beinhaltet also alle möglichen Wörter y, die noch immer in L sind, wenn man sie mit 0^m als $0^m 0^j 1^{m+j}$ konkateniert und ist deshalb eine konkrete Beschreibung von L_{0^m}.

Also gibt es laut Satz 3.1 eine Konstante c, die unabhängig von $x = 0^m$ und $y = 1^m$ und somit von m ist, so dass $K(1^m) \leq \lceil \log_2(1+1) \rceil + c = 1 + c$ ($n = 1$ hier, da 1^m das erste Wort in L_{0^m} ist und wir dieses Wort betrachten wollen), also gilt für eine Konstante $d = 1 + c$, dass $K(1^m) \leq d$. Dies ist aber unmöglich, da:

(i) die Anzahl aller Programme, deren Länge $\leq d$ ist, ist höchstens 2^d und entsprechend endlich

(ii) die Menge $\{1^m \mid m \in \mathbb{N}\}$ unendlich ist

Für komplexere Sprachen ist es oft einfach, L_x so zu wählen, dass $x = a^{\alpha+1}$ ist, wobei α der Exponent (nach Variabelnwechsel) aus der Sprache ist. Also beispielsweise für $L = \{0^{n^2 \cdot 2n} \mid n \in \mathbb{N}\}$ ist $\alpha = m^2 \cdot 2m$, also ist $x = 0^{m^2 \cdot 2m+1}$. y_1 (das erste Wort der Sprache L_x) ist dann $y_1 = 0^{(m+1)^2 \cdot 2(m+1) - m^2 \cdot 2m + 1}$.

Wir können dann mit der Länge des Wortes $|y_1|$ und dem Theorem 3.1 argumentieren, dass wir einen Widerspruch erreichen und so also die Sprache nichtregulär ist.

Dazu sagen wir, dass für jedes $m \in \mathbb{N}$ eine Konstante c existiert, so dass $K(y_1) \leq \lceil \log_2(1+1) \rceil + c = 1 + c$. Da unser Wort y_1 unendlich lang werden kann, gibt es unendlich viele solcher Wörter. Dies widerspricht jedoch dem Fakt, dass es nur endlich viele Programme mit Kolmogorov-Komplexität $\leq 1 + c$ gibt.

3.5 Nichtdeterminismus

Einfach gesagt werden hier Automaten behandelt, die zufällige (genannt ***nichtdeterministische***) Entscheidungen treffen. Beispielsweise für ein Entscheidungsproblem (Σ, L) bedeutet dies, dass ein nichtdeterministischer EA A eine Sprache L akzeptiert, falls für jedes $x \in L$ mindestens eine akzeptierende Berechnung von A auf x existiert und für $y \in \Sigma^* - L$ keine solve existiert.

Wir notieren das Ganze in graphischer Darstellung so, dass wir aus einem Zustand mehrere Übergänge mit dem gleichen Eingabesymbol erlauben.

nichtdeterministischer Endlicher Automat (NEA) **Definition 3.3**

Ein NEA ist eine Quitupel $M = (Q, \Sigma, \delta, q_0, F)$:
(i) ***Zustandsmenge:*** Q ist eine endliche Menge
(ii) ***Eingabealphabet:*** Σ ist ein Alphabet
(iii) ***Übergangsfunktion:*** $\delta : Q \times \Sigma \to \mathcal{P}(Q)$. $\mathcal{P}(Q)$ ist das Powerset hierbei
(iv) ***Anfangszustand:*** $q_0 \in Q$
(v) ***Akzeptierende Zustände:*** $F \subseteq Q$
Ein ***Schritt*** in der δ-Notation ist im Vergleich zum deterministischen EA nicht $\delta(q,a) = p$, sondern $p \in \delta(q,a)$ ist, da die Übergangsfunktion ja jetzt ins Powerset von Q, anstelle von nach Q direkt mapped. Die komplette Definition des Schritts ist also:

$$(q,w) \vdash_M (p,x) \iff w = ax \text{ für ein } a \in \Sigma \text{ und } p \in \delta(q,a)$$

Eine ***Berechnung von*** M ist eine endliche Folge $D_1, D_2, \dots, D_k$ von Konfigurationen, wobei $D_i \vdash_M D_{i+1}$ für $i = 1, \dots, k-1$
Eine ***Berechnung von*** M ***auf*** x hingegen ist eine Berechnung $C_0, C_1, \dots, C_m$ von M, wobei $C_0 = (q_0, x)$ und entweder $C_m \in Q \times \{\lambda\}$ oder $C_M = (q, ay)$ für ein $a \in \Sigma, y \in \Sigma^*$ und $q \in Q$, so dass $\delta(q,a) = \emptyset$.
$C_0, \dots, C_m$ ist ***akzeptierend*** falls $C_m = (p, \lambda)$ für ein $p \in F$
Die Sprache $L(M) = \{w \in \Sigma^* \mid (q_0, w) \vdash_M^* (p, \lambda) \text{ für ein } p \in F\}$
Für die $\widehat{\delta}$-Funktion, gilt nun $\widehat{\delta}(q, \lambda) = \{q\}$ für jedes $q \in Q$ und wir definieren:

$$\begin{aligned}\widehat{\delta}(q, wa) &= \{p \in Q \mid \text{es existiert ein } r \in \widehat{\delta}(q,w), \text{ so dass } p \in \delta(r,a)\} \\ &= \bigcup_{r \in \widehat{\delta}(q,w)} \delta(r,a) \quad \forall q \in Q, a \in \Sigma, w \in \Sigma^*\end{aligned}$$

Ein Wort ist in $L(M)$, falls M mindestens eine akzeptierende Berechnung auf x hat.

Bei einer akzeptierenden Berechnung *auf* x wird wie beim EA gefordert, dass das ganze Wort x gelesen worden ist und M nach dem Lesen in einem akzeptierenden Zustand ist.

Bei NEA kann eine nicht akzeptierende Berechnung auch vor Beendung des Lesevorgangs enden, da wir hier nicht vorschreiben, dass es für jedes Symbol des Eingabealphabets eine definierte Übergangsfunktion gibt, es ist also erlaubt, dass bspw. $\delta(q,a) = \emptyset$.

Zudem haben wir aus der Definition von $\widehat{\delta}$ eine alternative Definition der von M akzeptierten Sprache: $L(M) = \{w \in \Sigma^* \mid \widehat{\delta}(q_0, w) \cap F \neq \emptyset\}$

Für NEA kann man einen ***Berechnungsbaum*** $\mathcal{B}_M(x)$ ***von*** M ***auf*** x erstellen, der dann anschaulich alle möglichen Enden der Berechnung darstellt. Wir beginnen den Baum mit Konfiguration (q_0, x) und führen dann mit den Kanten alle möglichen Berechnungen aus, die mit dem ersten Symbol des Wortes möglich sind.

Wir erreichen so also zum Beispiel die Konfiguration (q_1, x_1), wobei x_1 x ohne das erste Zeichen ist.

Lemma 3.5: *(NEA aus Abbildung 3.15 im Buch)* Sei M der NEA aus Abbildung 3.15 im Buch (auf Seite 77 (= 92 im PDF) zu finden). Dann ist $L(M) = \{x11y \mid x, y \in (\Sigma_{\text{bool}})^*\}$

Der Beweis für eine solche Aussage läuft oft über Teilmengen (also mit $X \subseteq Y \land Y \subseteq X \Leftrightarrow X = Y$).

Eine zentrale Frage dieses Kapitels ist es, ob $\mathcal{L}_{\text{NEA}} = \mathcal{L}_{\text{EA}}$, wobei $\mathcal{L}_{\text{NEA}} = \{L(M) \mid M \text{ ist ein NEA}\}$. In anderen Worten: Können EA die Arbeit von NEA simulieren?

Ja, es ist möglich und gilt allgemein, dass die Simulation von Nichtdeterminismus durch Determinismus nur dann realisierbar ist, wenn es möglich ist, alle nichtdeterministischen Berechnungen durch deterministische Berechnungen nachzuahmen.

Bei EA (nennen einen A im Folgenden) basiert diese Idee auf BFS der Berechnungsbäume von M. Die Idee ist dann, dass alle Knoten mit Entfernung i von der Wurzel die ersten i Symbole von x gelesen haben. Da NEA endlich viele Konfigurationen bei Entfernung i haben ist es möglich, die Transformation durchzuführen. Wenn es zwei Knoten $u \neq v$ identisch sind, so müssen wir nur in einem der Teilbäume nach einer akzeptierenden Berechnung suchen.

Potenzmengenkonstruktion: Ein Zustand $\langle P \rangle$ von A für $P \subseteq Q$ erhält die Bedeutung, dass nach der gegebenen Anzahl an Berechnungsschritten genau die Zustände aus P in den Berechnungen von M auf der gegebenen Ebene erreichbar sind, also $P = \widehat{\delta}(q_0, z)$. Ein Berechnungsschritt in A aus einem Zustand $\langle P \rangle$ für ein gelesenes Symbol a bedeutet die Bestimmung der Menge $\bigcup_{p \in P} \delta(p, a)$, also aller Zustände, die aus irgendeinem Zustand $p \in P$ beim Lesen von a erreichbar sind.

Dabei benutzen wir $\langle P \rangle$ statt P, um zu verdeutlichen, dass wir eine Zustand von A und nicht die Menge der Zustände von M bezeichnen.

Ein EA, der die Sprache, bei welcher das k-letzte Symbol 1 ist, benötigt 2^k Zustände. Er wird dabei aus dem NEA dieser Sprache mit der Potenzmengenkonstruktion gebildet.

Satz 3.2: Zu jedem NEA M existiert ein EA A, so dass $L(M) = L(A)$

Um $L(M) = L(A)$ zu zeigen, müssen wir folgende Äquivalenz beweisen:

$$\forall x \in \Sigma^* : \widehat{\delta}_M(q_0, x) = P \iff \widehat{\delta}(q_{0A}, x) = \langle P \rangle$$

Wir können dies über einen Induktionsbeweis tun und ein vollständiger Beweis findet sich unten auf Seite 82 (= Seite 97 im PDF) im Buch.

Wir sagen, dass zwei Automaten ***äquivalent*** sind, falls $L(A) = L(B)$.

Eine Folge von Satz 3.2 ist eben, dass $\mathcal{L}_{\text{EA}} = \mathcal{L}_{\text{NEA}}$, also sind die EA genau so stark wie die NEA im Bezug auf die Sprachakzeptierung. Was hingegen ein Problem sein kann, ist dass die durch die Potenzmengenkonstruktion erzeugten Automaten (exponentiell) grösser sind als die NEA.

Es gibt gewisse NEA, bei welchen man bei der Simulation des Nichtdeterminismus durch Determinismus unausweichlich in exponentiell grösseren EA resultiert. Man kann beweisen (siehe Seiten 83 und 84 mit Abbildung 3.19 im Buch (= Seiten 98 & 99 im PDF)), dass man die Potenzmengenkonstruktion nicht allgemein verbessern kann.

Lemma 3.6: Für alle $k \in \mathbb{N} - \{0\}$ muss jeder EA, der $L_k = \{x1y \mid x \in (\Sigma_{\text{bool}})^*, y \in (\Sigma_{\text{bool}})^{k-1}\}$ akzeptiert, mindestens 2^k Zustände haben.

Worked Example Zeige, das jeder endliche Automat, der die Sprache

$$L = \{w \in \{a, b\}^* \mid w \text{ enthält Teilwort } ab \text{ gleich oft wie das Teilwort } ba \text{ enthält}\}$$

mindestens $n := 5$ Zustände haben muss.

	ab	$(ab)^2$	$(ab)^3$	$(ab)^4$	$(ab)^5$
ab	-	$(ba)^2$	$(ba)^3$	$(ba)^4$	$(ba)^5$
$(ab)^2$		-	$(ba)^3$	$(ba)^4$	$(ba)^5$
$(ab)^3$			-	$(ba)^4$	$(ba)^5$
$(ab)^4$				-	$(ba)^5$
$(ab)^5$					-

Sei $S = \{ab, (ab)^2, (ab)^3, (ab)^4, (ab)^5\}$. Laut Lemma 3.3

4 Turing-Maschinen

4.3 Das Modell der Turingmaschine

Eine Turingmaschine (oft auch Turing-Maschine geschrieben) besteht informell aus

(i) einer endlichen Kontrolle, die das Programm enthält

(ii) einem Arbeitsband unendlicher Länge (das es Erlaubt, im Vergleich zum EA, Daten zu speichern)

(iii) einem Lese-/Schreibkopf, der sich in beide Richtungen auf dem Band bewegen kann

Formaler:

Turingmaschine (TM) **Definition 4.1**

Eine ***Turingmaschine*** ist eine 7-Tupel $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, wobei:
(i) Q ist die ***Zustandsmenge***
(ii) Σ ist das ***Eingabealphabet*** mit $¢, \textvisiblespace \notin \Sigma$
(iii) Γ ist das ***Arbeitsalphabet*** mit $\Sigma \subseteq \Gamma$, $¢, \textvisiblespace \in \Gamma$ und $\Gamma \cap Q = \emptyset$ ($¢$ = Startmarker, $\textvisiblespace$ = Blanksymbol)
(iv) $\delta : (Q - \{q_{\text{accept}}, q_{\text{reject}}\}) \times \Gamma \longrightarrow Q \times \Gamma \times \{L, R, N\}$ ist die ***Übergangsfunktion von*** M, wobei $\{L, R, N\}$ die möglichen Bewegunsrichtungen des Lese-/Schreibkopfs sind und hat die Eigenschaft $\delta(q, ¢) \in Q \times \{¢\} \times \{R, N\}$ für alle $q \in Q$
(v) q_0 ist der ***Anfangszustand***
(vi) q_{accept} ist der ***akzeptierende Zustand*** (genau einer in jedem M)
(vii) q_{reject} ist der ***verwerfende Zustand*** (genau einer in jedem M)
Eine ***Konfiguration*** C von M ist ein Element aus $\text{Konf}(M) = \{¢\} \cdot \Gamma^* \cdot Q \cdot \Gamma^+ \cup Q \cdot \{¢\} \cdot \Gamma^+$ (wobei $\cdot$ die Konkatenation ist)
Eine ***Startkonfiguration*** für ein Eingabewort x ist $q_0¢x$
Ein ***Schritt von*** M ist eine Relation $\vdash_M$ auf der Menge der Konfigurationen, also $\vdash_M \subseteq \text{Konf}(M) \times \text{Konf}(M)$.
$\mathcal{L}_{RE} = \{L(M) \mid M \text{ ist eine Turingmaschine}\}$
Der Rest der Definition findet sich auf Seiten 96 - 98 (= Seiten 110 - 112 im PDF)

Turingmaschinen, die immer halten, repräsentieren Algorithmen, die immer terminieren und die richtige Ausgabe liefern. Rekursive Sprachen und entscheidbare Entscheidungsprobleme sind algorithmisch erkennbar, respektive lösbar.

Es gibt auch definitionen der TM, die ohne Startmarker $¢$ auskommen, bei denen ist das Arbeitsband in beide Richtungen unendlich.

Graphisch stellt man Turingmaschinen folgendermassen dar: Wir haben wieder einen Graphen mit gerichteten Kanten. Für $\delta(q, a) = (p, b, X)$ mit $q, p \in Q$, $a, b \in \Sigma$ und $X \in \{L, R, N\}$ werden die Kanten mit folgendem Format beschriftet: $q \to a, X$.

Mit einem TM die Sprache $\{0^n 1^n \mid n \in \mathbb{N}\}$ erkennen kann man nun, indem man jeweils das linkeste und rechteste Symbol durch ein anderes Symbol ersetzt, beispielsweise, wenn das Eingabealphabet $\Sigma = \{0, 1\}$, dann könnte man Γ das Symbol 2 hinzufügen, mit dem man jedes bearbeitete Symbol ersetzt.

Im Buch wird als Beispiel auf Seite 99ff (= Seite 114ff im PDF) ein komplexeres Wort gewählt, bei welchem ein Zeichen $a \in \Sigma_{\text{bool}}$ durch $(a, B) \in \Sigma_{\text{bool}} \times \{A, B\}$ ersetzt wird, da wir zwei Phasen haben und zwischen denen unterscheiden wollen können.

4.4 Mehrband-Turingmaschinen und Church'sche These

Die Turingmaschinen sind das Standardmodell der Berechenbarkeitstheorie, aber benötigen einige Modifikationen, um wirklich geeignet zu sein (da das Von-Neumann Modell physisch unterschiedliche CPU, Eingabemedium und Speicher für Programme und Daten fordert, aber die TM ein gemeinsames Eingabemedium und Speicher hat).

Eine k-Band-Turingmaschine (für $k \in \mathbb{N}_0$) hat folgende Komponenten:

- eine endliche Kontrolle (= Programm)
- ein endliches Band mit einem Lesekopf
- k Arbeitsbänder, jedes mit eigenem Lese-/Schreibkopf

Zu Beginn ist die MTM in folgender Situation:

- Das Eingabeband enthält ¢w\$, wobei ¢ und \$ die linke / rechte Seite der Eingabe markieren
- Der Lesekopf des Eingabebands zeigt auf ¢
- Alle Arbeitsbänder beinhalten ¢␣␣... und deren Lese-/Schreibköpfe zeigen auf ¢
- Die endliche Kontrolle ist im Anfangszustand q_0

Alle $k+1$ Köpfe dürfen sich während der Berechnung in beide Richtungen bewegen (solange das nicht out-of-bounds geht). Zudem darf der Lesekopf nicht schreiben, also beleibt der Inhalt des Eingabebands gleich.

Gleich wie bei einer TM ist das Arbeitsalphabet der Arbeitsbänder Γ und alle Felder der Arbeitsbänder sind von links nach rechts nummeriert, wobei 0 bei ¢ liegt.

Eine Konfiguration einer k-Band-TM M ist $(q, w, i, u_1, i_1, u_2, i_2, \dots, u_k, i_k) \in Q \times \Sigma^* \times \mathbb{N} \times (\Gamma^* \times \mathbb{N})^k$, wobei q der Zustand ist, der Inhalt des Eingabebands ist ¢w\$, der Lesekopf zeigt auf das i-te Feld, für $j \in \{1, 2, \dots, k\}$ ist der Inhalt des j-ten Bandes ¢u_k␣... und $i_j \leq |u_j|$ ist die Position des Feldes.

Ein Berechnungsschritt von M kann mit

$$\delta : Q \times (\Sigma \cup \{¢, \$\}) \times \Gamma^k \to Q \times \{L, R, N\} \times (\Gamma \times \{L, R, N\})^k$$

dargestellt werden, wobei die Argumente $(q, a, b_1, \dots, b_k)$ der aktuelle Zustand q, das gelesene Eingabesymbol a und die k Symbole $b_i \in \Gamma$, auf welchen die Köpfe der Arbeitsbänder stehen.

Die Eingabe w wird von M akzeptiert, falls M den Zustand q_{accept} erreicht und falls M den Zustand q_{reject} erreicht oder nicht terminiert, wird die Eingabe verworfen.

Wir sagen, dass eine Maschine A äquivalent zu einer Maschine B ist, falls für jede Eingabe $x \in (\Sigma_{\text{bool}})^*$ gilt: A <property> $x \iff B$ <property> x mit <property> $\in$ {akzeptiert, verwrift, arbeitet unendlich lange auf}, also ist $L(A) = L(B)$

Lemma 4.1: Zu jeder TM A existiert eine zu A äquivalente 1-Band-TM B

Lemma 4.2: Zu jeder Mehrband-Turingmaschine A existiert eine zu A äquivalente TM B

Die Beweise dazu finden sich auf Seite 107, beziehungsweise Seite 109 (= 121 & 123 im PDF).

In diesem Kurs müssen wir glücklicherweise meist nicht Beweise der Äquivalenz durchführen, wie auch nicht dass die TM die gewünschte Tätigkeit realisiert.

Definition 4.1: Zwei Maschinenmodelle (Maschinenklassen) $\mathcal{A}$ und $\mathcal{B}$ sind äquivalent wenn beides zutrifft:

(i) für jede Maschine $A \in \mathcal{A}$ eine zu A äquivalente Maschine $B \in \mathcal{B}$ existiert

(ii) für jede Maschine $C \in \mathcal{B}$ eine zu C äquivalente Maschine $D \in \mathcal{A}$ existiert

Satz 4.1: Die Maschinenmodelle von Turingmaschinen und Mehrband-Turingmaschinen sind äquivalent

Beweis: Impliziert von Lemmas 4.1 und 4.2

Um zu beweisen, dass Turing-Maschinen äquivalent zu höheren Programmiersprachen sind argumentiert man über die Existenz eines Interpreters für TM.

4.4.1 Church'sche These

> *Die Turingmaschinen sind die Formalisierung des Begriffes "Algorithmus", das heisst, die Klasse der rekursiven Sprachen (der entscheidbaren Entscheidungsproblem) stimmt mit der Klasse der algorithmisch (automatisch) erkennbaren Sprache überein*

Die These ist nicht beweisbar, da dazu der Begriff des Algorithmus formalisiert werden müsste, was er bekanntlich nicht ist.

Dies führt zu einer interessanten Situation, in welcher es *theoretisch* möglich wäre, dass jemand ein stärkeres Modell findet, als die TM sind, eines nämlich, welches Entscheidungsprobleme lösen kann, die die TM nicht kann.

Wir nehmen also (wie in vielen Bereichen der Physik (die Relativitätstheorie ist ein gutes Beispiel) und Mathematik) und postulieren sie als Axiom.

Fun fact Die Church'sche These ist das Einzige informatikspezifische Axiom.

4.5 Nichtdeterministische Turingmaschinen

Die Ideen sind hier sehr ähnlich wie der Übergang zwischen deterministischen und nichtdeterministischen Endlichen Automaten.

Nichtdeterministische Turingmaschine (NTM) **Definition 4.2**

Hier werden nur die wichtigsten Unterschiede aufgezeigt. Formale Definition auf Seiten 113ff. (= Seiten 127ff im PDF) im Buch.
Die Übergangsfunktion geht wieder in die Potenzmenge, also gilt:

$$\delta : (Q - \{q_{\text{accept}}, q_{\text{reject}}\}) \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R, N\})$$

und $\delta(p, ¢) \subseteq (\{(q, ¢, X) \mid q \in Q, X \in \{R, N\}\})$

Die von der NTM M akzeptierte Sprache ist:

$$L(M) = \{w \in \Sigma^* \mid q_0 ¢ w \vdash_M^* y q_{\text{accept}} z \text{ für irgendwelche } y, z \in \Gamma^*\}$$

Ein gutes Beispiel für eine NTM findet sich auf Seiten 114ff. im Buch (= Seite 128ff. im PDF)

Berechnungsbaum **Definition 4.3**

Ein Berechnungsbaum $T_{M,x}$ von M (eine NTM) auf x (Wort aus Eingabealphabet von M) ist ein (potentiell un)gerichteter Baum mit einer Wurzel:

(i) Jeder Knoten von $T_{M,x}$ ist mit einer Konfiguration beschriftet
(ii) Die Wurzel ist der einzige Knoten mit $\deg_{\text{in}}(v) = 0$, ist die Startkonfiguration $q_0 ¢ x$
(iii) Jeder mit C beschriftete Knoten hat genauso viele Kinder wie C Nachfolgekonfigurationen hat und die Kinder sind mit diesen Nachfolgekonfigurationen markiert.

Diese Bäume können natürlich auch für nichtdeterministischen MTM verwendet werden.

Im Vergleich zu den Berechnungsbäumen von NEA sind die Bäume von NTM nicht immer endlich.

Satz 4.2: Sei M eine NTM. Dann existiert eine TM A, so dass $L(M) = L(A)$ und falls M keine unendlichen Berechnungen auf Wörtern aus $(L(M))^C$ hat, dann hält A immer.

Beweis: Auf Seite 117 im Buch (= 131 im PDF). Die Idee zur Umwandlung von M in die TM A ist, dass A Breitensuche im Berechnungsbaum von M durchführt.

5 Berechenbarkeit

5.2 Diagonalisierung

Wir definieren KodTM als die *Menge der binären Kodierungen aller Turingmaschinen.* Wir haben KodTM $\subseteq (\Sigma_{\text{bool}})^*$ und die obere Schranke der Kardinalität ist $|(\Sigma_{\text{bool}})^*|$, da es unendlich viele Turingmaschinen gibt.

Im Folgenden wird wieder Cantor's Diagonalisierungsmethode verwendet

Cantor's Diagonalization Argument **Repetition**

TODO: Finish

Definition 5.1: A und B sind Mengen. Dann ist $|A| \leq |B|$ falls eine *injektive* Funktion f von A nach B existiert; $|A| = |B|$ falls $|A| \leq |B|$ und $|B| \leq |A|$ (es existiert eine Bijektion); $|A| < |B|$ falls $|A| \leq |B|$ und keine injektive Abbildung von B nach A existiert.

Um zu zeigen, dass es nicht rekursiv aufzählbare (also von Turingmaschinen nicht erkennbare) Sprachen gibt. Also müssen wir laut Definition 5.1 nur zeigen, dass keine Injektion von $(\Sigma_{\text{bool}})^*$ nach $\mathcal{L}_{RE}$ existiert.

Definition 5.2: *(Abzählbarkeit)* A heisst abzählbar, falls A endlich ist oder $|A| = |\mathbb{N}|$

Lemma 5.1: Sei Σ ein beliebiges Alphabet. Dann ist Σ^* abzählbar

Satz 5.1: Die Menge KodTM der Turingmaschinenkodierungen ist abzählbar

Lemma 5.2: $(\mathbb{N} - \{0\}) \times (\mathbb{N} - \{0\})$ ist abzählbar. Die Idee ist dieselbe wie für $|Q^+| = |\mathbb{N}|$, nämlich, dass wir jedem Element einen Index zuordnen können.

Satz 5.2: $\mathbb{Q}^+$ ist abzählbar. Die Idee für den Beweis ist eine Bijektion nach obiger Menge zu finden.

Satz 5.3: $[0, 1] \subseteq \mathbb{R}$ ist nicht abzählbar. Dies kann mit Cantor's Diagonalization Argument bewiesen werden.

Satz 5.4: $\mathcal{P}((\Sigma_{\text{bool}})^*)$ ist nicht abzählbar

Korollar 5.1: $|\text{KodTM}| < |\mathcal{P}((\Sigma_{\text{bool}})^*)|$ und es existieren also unendlich viele nicht rekursiv aufzählbare Sprachen.

Um für eine spezifische Sprache zu beweisen, dass sie rekursiv aufzählbar ist, können wir einfach eine Turingmaschine konstruieren. Für eine Beweis dafür, dass eine Sprache nicht rekursiv aufzählbar ist können wir folgende Methode verwenden. Sei dazu mit $d_{ij} = 1 \iff M_i$ akzeptiert w_j

$$\begin{aligned} L_{\text{diag}} &= \{w \in (\Sigma_{\text{bool}})^* \mid w = w_i \text{ für ein } i \in \mathbb{N} - \{0\} \text{ und } M_i \text{ akzeptiert } w_i \text{ nicht}\} \\ &= \{w \in (\Sigma_{\text{bool}})^* \mid w = w_i \text{ für ein } i \in \mathbb{N} - \{0\} \text{ und } d_{ii} = 0\} \end{aligned}$$

Satz 5.5: $L_{\text{diag}} \notin \mathcal{L}_{RE}$ **Beweis:** Zum Widerspruch nehmen wir an, dass $L_{\text{diag}} \in \mathcal{L}_{RE}$. Dann gilt, dass $L_{\text{diag}} = L(M)$ für eine Turingmaschine M. M ist eine Turingmaschine in der kanonischen Ordnung der Turingmaschinen, also existiert ein $i \in \mathbb{N} - \{0\}$, so dass $M = M_i$.

Dies führt zu einem Widerspruch, denn L_{diag} kann nicht gleich $L(M_i)$ sein, da

$$w_i \in L_{\text{diag}} \iff d_{ii} = 0 \iff w_i \notin L(M_i)$$

also ist w_i genau dann in L_{diag} wenn w_i *nicht* in $L(M_i)$ ist. (= in genau einer der Sprachen L_{diag} oder $L(M_i)$)

5.3 Die Methode der Reduktion

Definition 5.3: *(Rekursiv reduzierbare Sprache)* Eine Sprache $L_1 \subseteq \Sigma_1^*$ ist auf $L_2 \subseteq \Sigma_2^*$ rekursiv reduzierbar, geschrieben $L_1 \leq_R L_2$, falls $L_2 \in \mathcal{L}_R \Rightarrow L_1 \in \mathcal{L}_R$.

Intuition: L_2 ist bezüglich der algorithmischen Lösbarkeit mindestens so schwer wie L_1. $\mathcal{L}_R$ ist die Menge aller rekursiv reduzierbaren Sprachen. Ist also L_2 lösbar, so muss auch L_1 lösbar sein.

Definition 5.4: *(EE-reduzierbare Sprache)* L_1 ist auf L_2 ***EE-reduzierbar***, geschrieben $L_1 \leq_{EE} L_2$, wenn eine TM M existiert, die eine Abbildung $f_M : \Sigma_1^* \to \Sigma_2^*$ mit der Eigenschaft $x \in L_1 \Leftrightarrow f_M(x) \in L_2$ für alle $x \in \Sigma_1^*$ berechnet. Anders ausgedrückt: die TM M reduziert die Sprache L_1 auf die Sprache L_2

Lemma 5.3: Falls $L_1 \leq_{EE} L_2$, dann auch $L_1 \leq_R L_2$. **Beweis:** Im Buch auf Seite 135 (= 148 im PDF)

Wir müssen also nur zeigen, dass $L_1 \leq_{EE} L_2$ um zu zeigen, dass $L_1 \leq_R L_2$

Lemma 5.4: Für jede Sprache $L \subseteq \Sigma^*$ gilt: $L \leq_R L^C$ und $L^C \leq_R L$

Korollar 5.2: $(L_{\text{diag}})^C \notin \mathcal{L}_R$ **Beweis:** Folgt davon, dass $L_{\text{diag}} \notin \mathcal{L}_{RE}$ (was heisst, dass $L_{\text{diag}} \notin \mathcal{L}_R$) und nach Lemma 5.4 $L_{\text{diag}} \leq_R (L_{\text{diag}})^C$ und das umgekehrte gelten muss.

Lemma 5.5: $(L_{\text{diag}})^C \in \mathcal{L}_{RE}$ **Beweis:** Auf Seite 137 (= 150 im PDF) wird eine Turingmaschine aufgezeigt, die $(L_{\text{diag}})^C$ akzeptiert.

Korollar 5.3: $(L_{\text{diag}})^C \in \mathcal{L}_{RE} - \mathcal{L}_R$ und daher $\mathcal{L}_R \subsetneq \mathcal{L}_{RE}$

Folgende Sprachen sind nicht rekursiv, liegen aber in $\mathcal{L}_{RE}$

Definition 5.5: *(Universelle Sprache)* $L_U = \{\text{Kod}(M)\#w \mid w \in (\Sigma_{\text{bool}})^* \text{ und TM } M \text{ akzeptiert } w\}$

Satz 5.6: *(Universelle TM)* Eine TM U, so dass $L(U) = L_U$, also gilt $L_U \in \mathcal{L}_{RE}$
Beweis: Auf Seite 138 (= 151 im PDF)

Was dies bedeutet, es existiert eine TM ohne Haltegarantie, die eine beliebige Turingmaschine auf einer gegebenen Eingabe simulieren kann. Untenstehendes Resultat bedeutet, dass man das Resultat der Berechnung einer TM M auf einer Eingabe x anders berechnen kann, als die Berechnung von M auf x zu simulieren.

Satz 5.7: $L_U \notin \mathcal{L}_R$

Wenn jetzt aber M unendlich lange auf x arbeitet, so wissen wir nicht, ob wir die Simulation beenden können. Dies führt zum Halteproblem

Halteproblem **Definition 5.6**

Das Halteproblem ist das Entscheidungsproblem $(\{0,1,\#\}, L_H)$ mit

$$L_H = \{\text{Kod}(M)\#x \mid x \in \{0,1\}^* \text{ und } M \text{ hält auf } x\}$$

Dies scheint vorerst nicht ein allzu grosses Problem zu sein, jedoch besagt das nächste Resultat, dass es keinen Algorithmus gibt, der testen kann, ob ein gegebenes Programm immer terminiert.

Satz 5.8: $L_H \notin \mathcal{L}_R$ **Beweis:** Auf Seiten 140 - 142 (153 - 155 im PDF)

Betrachten wir die Sprache $L_{\text{empty}} = \{\text{Kod}(M) \mid L(M) = \emptyset\}$, die die Kodierungen aller Turingmaschinen enthält, die die leere Menge (kein Wort) akzeptieren. Es gilt

$$(L_{\text{empty}})^C = \{x \in (\Sigma_{\text{bool}})^* \mid x \notin \text{Kod}(\overline{M}) \forall \text{ TM } \overline{M} \text{ oder } x = \text{Kod}(M) \text{ und } L(M) \neq \emptyset\}$$

Lemma 5.6: $(L_{\text{empty}})^C \in \mathcal{L}_{RE}$ **Beweis:** Auf Seiten 142 - 143 (155 - 156 im PDF)

Lemma 5.7: $(L_{\text{empty}})^C \notin \mathcal{L}_R$

Wir haben als wiederum die Nichtexistenz eines Algorithmus zur Überprüfung, ob ein gegebenes Programm die leere Menge akzeptiert. Ein Beweis dazu findet sich auf Seiten 143 und 144 im Buch (156 - 157 im PDF)

Korollar 5.4: $L_{\text{empty}} \notin \mathcal{L}_R$

Korollar 5.5: $L_{EQ} = \{\text{Kod}(M)\#\text{Kod}(\overline{M}) \mid L(M) = L(\overline{M})\}$ ist nicht entscheidbar (also $L_{EQ} \notin \mathcal{L}_R$)

5.4 Der Satz von Rice

Definition 5.7: L heisst ***semantisch nichttriviales Entscheidungsproblem über Turingmaschinen***, falls folgende Bedingungen gelten:

(i) Es gibt eine TM M_1, so dass $\text{Kod}(M_1) \in L$ (also $L \neq \emptyset$)

(ii) Es gibt eine TM M_2, so dass $\text{Kod}(M_2) \notin L$ (also sind nicht alle Kodierungen in L)

(iii) für zwei TM A und B: $L(A) = L(B) \Rightarrow \text{Kod}(A) \in L \Leftrightarrow \text{Kod}(B) \in L$

Sei $L_{H,\lambda} = \{\text{Kod}(M) \mid M \text{ hält auf } \lambda\}$ ein spezifisches Halteproblem.

Lemma 5.8: $L_{H,\lambda} \notin \mathcal{L}_R$ **Beweis:** Auf Seite 146 im Buch (= 159 im PDF)

Satz von Rice **Satz 5.9**

Jedes semantisch nichttriviale Entscheidungsproblem über Turingmaschinen ist unentscheidbar.

Beweis: Ausführlich im Buch auf Seiten 146 - 149 beschrieben (= 159 - 162 im PDF)

5.6 Die Methode der Kolmogorov-Komplexität

Satz 5.10: Das Probelem, für jedes $x \in (\Sigma_{\text{bool}})^*$ die Kolmogorov-Komplexität $K(x)$ von x zu berechnen ist algorithmisch unlösbar.

Lemma 5.9: Falls $L_H \in \mathcal{L}_R$, dann existiert ein Algorithmus zur Berechnung der Kolmogorov-Komplexität $K(x)$ für jedes $x \in (\Sigma_{\text{bool}})^*$

6 Komplexitätstheorie

6.2 Komplexitätsmasse

Zeitkomplexität **Definition 6.1**

Sei M eine Mehrband-TM oder TM, die immer hält, $x \in \Sigma^*$ und $D = C_1, C_2, \dots, C_k$ die Berechnung von M auf x, deren Zeitkomplexität definiert ist durch:

$$\text{Time}_M(x) = k - 1$$

also durch die Anzahl der Berechnungsschritte in D Die Zeitkomplexität der TM M ist dabei definiert durch:

$$\text{Time}_M(n) = \max\{\text{Time}_M(x) \mid x \in \Sigma^n\}$$

Wir können weiterhin die big-O-notation verwenden um den Worstcase anzugeben.

Speicherplatzkomplexität **Definition 6.2**

Sei $C = (q, x, i, \alpha_1, i_1, \alpha_2, i_2, \dots, \alpha_k, i_k)$ mit $0 \leq i|x|+1$ und $0 \leq i_j \leq |\alpha_j|$ für $j = 1, \dots, k$ eine Konfiguration von M, welche eine k-Band TM ist. ***Die Speicherplatzkomplexität von*** C ist

$$\text{Space}_M(C) = \max\{|\alpha_i| \mid i = 1, \dots, k\}$$

Für die Berechnung $C_1, C_2, \dots, C_l$ von M auf x haben wir:

$$\text{Space}_M(x) = \max\{\text{Space}_M(C_i) \mid i = 1, \dots, l\}$$

Und die ***Speicherplatzkomplexität von*** M ist

$$\text{Space}_M(n) = \max\{\text{Space}_M(x) \mid x \in \Sigma^n\}$$

Es ist auch möglich $\text{Space}_M(n)$ als eine Summe zu definieren, aber laut Lemma 4.2 wissen wir, dass man eine k-Band-TM mit einer 1-Band-TM simulieren kann.

Lemma 6.1: Sei $k \in \mathbb{N}$. Für jede k-Band-TM A, die immer hält existiert eine äquivalente 1-Band-TM B, so dass $\text{Space}_B(n) \leq \text{Space}_A(n)$

Lemma 6.2: Für jede k-Band-TM A, existiert eine äquivalente k-Band-TM B, so dass $L(A) = L(B)$ und $\text{Space}_B(n) \leq \frac{\text{Space}_A(n)}{2} + 2$

Definition 6.3: Wir notieren mit der big-O-notation folgendermassen: Falls $r \in \mathcal{O}(f(n))$, so wächst r asymptotisch nicht schneller als f. Äquivalent für $s \in \Omega(g(n))$ und $l \in \Theta(h(n))$ sagen wir asymptotisch mindestens so (gleich) schnell. Falls $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, dann wächst g asymptotisch schneller als f und $f(n) = o(g(n))$

Satz 6.1: Es existiert ein Entscheidungsproblem $(\Sigma_{\text{bool}}, L)$, so dass für jede MTM A, die $(\Sigma_{\text{bool}}, L)$ entscheidet, eine MTM B existiert, die es auch entscheidet und für die gilt: $\text{Time}_B(n) \leq \log_2(\text{Time}_A(n))$ für alle $n \in \mathbb{N}$

Schranken, Optimal **Definition 6.4**

$\mathcal{O}(g(n))$ ($\Omega(f(n))$) ist eine ***obere (untere) Schranke für die Zeitkomplexität von*** L, falls eine MTM A (B) existiert, die L entscheidet und $\text{Time}_A(n) \in \mathcal{O}(g(n))$ ($\text{Time}_B(n) \in \Omega(f(n))$)
Eine MTM C heisst ***optimal für*** L, falls $\text{Time}_C(n) \in \mathcal{O}(f(n))$ gilt und $\Theta(f(n))$ eine untere Schranke für die Zeitkomplexität von K ist.

6.3 Komplexitätsklassen und die Klasse P

Komplexitätsklasen — **Definition 6.5**

Für alle Funktionen $f, g : \mathbb{N} \to \mathbb{R}^+$ definieren wir:

$$\begin{aligned}
\text{TIME}(f) &= \{L(B) \mid B \text{ ist eine MTM mit } \text{Time}_B(n) \in \mathcal{O}(f(n))\} \\
\text{SPACE}(g) &= \{L(A) \mid A \text{ ist eine MTM mit } \text{Space}_A(n) \in \mathcal{O}(g(n))\} \\
\text{DLOG} &= \text{SPACE}(\log_2(n)) \\
\text{P} &= \bigcup_{c \in \mathbb{N}} \text{TIME}(n^c) \\
\text{PSPACE} &= \bigcup_{c \in \mathbb{N}} \text{SPACE}(n^c) \\
\text{EXPTIME} &= \bigcup_{d \in \mathbb{N}} \text{TIME}(2^{n^d})
\end{aligned}$$

Lemma 6.3: Für alle $t : \mathbb{N} \to \mathbb{R}^+$ gilt $\text{TIME}(t(n)) \subseteq \text{SPACE}(t(n))$ **Korollar 6.1:** $\text{P} \subseteq \text{PSPACE}$

Platz- und Zeitkonstruierbarkeit — **Definition 6.6**

Eine Funktion $s : \mathbb{N} \to \mathbb{N}$ heisst ***platzkonstruierbar***, falls eine 1-Band-TM M existiert, so dass
1. $\text{Space}_M(n) \leq s(n) \quad \forall n \in \mathbb{N}$
2. für jede Eingabe 0^n für $n \in \mathbb{N}$, generiert M das Wort $0^{s(n)}$ auf ihrem Arbeitsband und hält in q_{accept}

Eine Funktion $t : \mathbb{N} \to \mathbb{N}$ heisst ***zeitkonstruierbar***, falls eine MTM A existiert, so dass
1. $\text{Time}_A(n) \in \mathcal{O}(t(n))$
2. für jede Eingabe 0^n für $n \in \mathbb{N}$, generiert A das Wort $0^{t(n)}$ auf dem ersten Arbeitsband und hält in q_{accept}

Wichtig ist, dass wir hier nicht *zwingend* eine 1-Band-TM konstruieren müssen, eine MTM geht auch.

Lemma 6.4: Sei s platzkonstruierbar und M eine MTM mit $\text{Space}_M(x) \leq s(|x|) \ \forall x \in L(M)$. Dann existiert MTM A mit $L(A) = L(M)$ und $\text{Space}_A(n) \leq s(n)$, es gilt also $\text{Space}_A(y) \leq s(|y|) \ \forall y \in \Sigma_M$

Lemma 6.5: Sei t zeitkonstruierbar und M eine MTM mit $\text{Time}_M(x) \leq t(|x|) \ \forall x \in L(M)$. Dann existiert eine MTM A mit $L(A) = L(M)$ und $\text{Time}_A(n) \in \mathcal{O}(t(n))$

Satz 6.2: Für jede Funktion s mit $s(n) \geq \log_2(n)$ gilt $\text{SPACE}(s(n)) \subseteq \bigcup_{c \in \mathbb{N}} \text{TIME}(c^{s(n)})$

Obiger Satz trifft auch für $s(n)$-platzbeschränkten TM zu, die nicht halten, aber nur, wenn $s(n)$ platzkonstruierbar ist.

Korollar 6.2: $\text{DLOG} \subseteq \text{P}$ und $\text{PSPACE} \subseteq \text{EXPTIME}$

Die Korollare 6.1 und 6.2 geben zusammen $\text{DLOG} \subseteq \text{P} \subseteq \text{PSPACE} \subseteq \text{EXPTIME}$

Satz 6.3: Für $s_1, s_2 : \mathbb{N} \to \mathbb{N}$ mit folgenden Eigenschaften:
1. $s_2(n) \geq \log_2(n)$
2. s_2 ist platzkonstruierbar
3. $s_1(n) = o(s_2(n))$

Dann gilt: $\text{SPACE}(s_1) \subsetneq \text{SPACE}(s_2)$

Satz 6.4: Für $t_1, t_2 : \mathbb{N} \to \mathbb{N}$ mit folgenden Eigenschaften:
1. t_2 ist platzkonstruierbar
2. $t_1(n) \cdot \log_2(t_1(n)) = o(t_2(n))$

Dann gilt: $\text{TIME}(s_1) \subsetneq \text{TIME}(s_2)$

In den Sechzigerjahren entstand folgende "Definition" von parktisch lösbaren Problemen:

> *Ein Problem ist praktisch lösbar genau dann, wenn ein polynomialer Algorithmus zu seiner Lösung existiert. Die Klasse P ist die Klasse der praktisch entscheidbaren Probleme*

6.4 Nichtdeterministische Komplexitätsmasse

Zeit- und Speicherkomplexität — Definition 6.7

Sei M eine NMTM oder MTM und $x \in L(M) \subseteq \Sigma^*$. $\text{Time}_M(x)$ ist die länge einer kürzesten akzeptierenden Berechnung von M auf x und $\text{Time}_M(n) = \max(\{\text{Time}_M(x) \mid x \in L(M) \text{ und } |x| = n\} \cup \{0\})$.

$\text{Space}_M(C_i)$ ist die Speicherkomplexität von Konfiguration C_i und $\text{Space}_M(C) = \max\{\text{Space}_M(C_i) \mid i = 1, 2, \ldots, m\}$. Zudem ist $\text{Space}_M(x) = \min\{\text{Space}_M(C) \mid C \text{ ist akzeptierende Berechnung von } M \text{ auf } x\}$. Ausserdem ist $\text{Space}_M(n) = \max(\{\text{Space}_M(x) \mid x \in L(M) \text{ und } |x| = n\} \cup \{0\})$

Komplexitätsklassen — Definition 6.8

Für alle $f, g : \mathbb{N} \to \mathbb{R}^+$ definieren wir:

$$\begin{aligned}
\text{NTIME}(f) &= \{L(M) \mid M \text{ ist eine NMTM mit } \text{Time}_M(n) \in \mathcal{O}(f(n))\} \\
\text{NSPACE}(g) &= \{L(M) \mid M \text{ ist eine NMTM mit } \text{Space}_M(n) \in \mathcal{O}(g(n))\} \\
\text{NLOG} &= \text{NSPACE}(\log_2(n)) \\
\text{NP} &= \bigcup_{c \in \mathbb{N}} \text{NTIME}(n^c) \\
\text{NPSPACE} &= \bigcup_{c \in \mathbb{N}} \text{NSPACE}(n^c)
\end{aligned}$$

Lemma 6.6: Für alle t und s mit $s(n) \geq \log_2(n)$ gilt: $\text{NTIME}(t) \subseteq \text{NSPACE}(t)$, $\text{NSPACE}(s) \subseteq \bigcup_{c \in \mathbb{N}} \text{NTIME}(c^{s(n)})$

Satz 6.5: Für jedes $t : \mathbb{N} \to \mathbb{R}^+$ und jedes platzkonstruierbare s mit $s(n) \geq \log_2(n)$ gilt:

(i) $\text{TIME}(t) \subseteq \text{NTIME}(t)$

(ii) $\text{SPACE}(t) \subseteq \text{NSPACE}(t)$

(iii) $\text{NTIME}(s(n)) \subseteq \text{SPACE}(s(n)) \subseteq \bigcup_{c \in \mathbb{N}} \text{TIME}(c^{s(n)})$

Korollar 6.3: $\text{NP} \subseteq \text{PSPACE}$

Satz 6.6: Für jede platzkonstruierbare Funktion s mit $s(n) \geq \log_2(n)$ gilt

$$\text{NSPACE}(s(n)) \subseteq \bigcup_{c \in \mathbb{N}} \text{TIME}(c^{s(n)})$$

Korollar 6.4: $\text{NLOG} \subseteq \text{P}$ und $\text{NPSPACE} \subseteq \text{EXPTIME}$

Satz 6.7: *(Satz von Savitch)* Sei s mit $s(n) \geq \log_2(n)$ eine platzkonstruierbare Funktion. Dann gilt:

$$\text{NSPACE}(s(n)) \subseteq \text{SPACE}(s(n)^2)$$

Korollar 6.5: $\text{PSPACE} = \text{NPSPACE}$

Aus den obigen Resultaten resultiert die Komplexitätsklassenhierarchie der sequentiellen Berechnungen:

$$\text{DLOG} \subseteq \text{NLOG} \subseteq \text{P} \subseteq \text{NP} \subseteq \text{PSPACE} \subseteq \text{EXPTIME}$$

6.5 Die Klasse NP und Beweisverifikation

Da praktische Lösbarkeit eines Problems mit polynomieller Zeit verbunden wird, ist es wichtig zu wissen, welche Probleme in polynomieller Zeit lösbar sind und welche nicht.

Der Vergleich zwischen den Klassen P und NP ist äquivalent zu der Frage, ob es einfacher ist, gegebene Beweise zu verifizieren, als sie herzustellen.

Betrachten wir folgendes: Sei $L = SAT$, wobei

$$SAT = \{x \in (\Sigma_{\text{logic}})^* \mid x \text{ kodiert eine erfüllbare Formel in CNF}\}.$$

Dann ist die Aussage $\Phi \in SAT$ äquivalent zu der Behauptung "Φ ist eine erfüllbare Formel in CNF"

Für nichtdeterministische Berechnungen nennen wir $\alpha_1, \dots, \alpha_n$ ***Zertifikate*** für eine Aussage Ξ, falls für diese $\Xi(\alpha_1, \dots, \alpha_n)$ hält.

Verifizierer **Definition 6.9**

Sei $L \subseteq \Sigma^*$ und $p : \mathbb{N} \to \mathbb{N}$. Eine MTM A ist ein p-Verifizierer und $V(A) = L$, falls A mit folgenden Eigenschaften auf allen Eingaben aus $\Sigma^* \times (\Sigma_{\text{bool}})^*$ arbeitet:

(i) $\text{Time}_A(w, x) \leq p(|w|)$ für jede Eingabe $(w, x) \in \Sigma^* \times (\Sigma_{\text{bool}})^*$
(ii) Für jedes $w \in L$ existiert ein $x \in (\Sigma_{\text{bool}})^*$, so dass $|x| \leq p(|w|)$ und $(w, x) \in L(A)$. x ist ***Zeugen*** (oder ***Beweis***) der Behauptung $w \in L$
(iii) Für jedes $y \notin L$ gilt $(y, z) \notin L(A)$ für alle $z \in (\Sigma_{\text{bool}})^*$
(iv) Falls $p(n) \in \mathcal{O}\left(n^k\right)$ für ein $k \in \mathbb{N}$, so ist p ein ***Polynomialzeit-Verifizierer***. Die Klasse ist

$$VP = \{V(A) \mid A \text{ ist ein Polynomialzeit-Verifizierer }\}$$

Satz 6.8: $VP = NP$

Der Beweis für obiges Resultat ist auf Seiten 193 - 194 im Buch (= 205 - 206 im PDf) zu finden

6.6 NP-Vollständigkeit

Es sind mittlerweile über 3000 Probleme bekannt, für welche wir keinen Algorithmus kennen, der in polynomieller Zeit läuft. Es ist aber bis jetzt niemandem gelungen, eine höhere untere Schranke für alle zu beweisen, als $\Omega(n)$.

Wie bereits bei der Berechenbarkeit benutzen wir eine Reduktion. Falls jedes Problem aus NP effizient auf ein Problem $L \in NP$ reduzierbar ist, so ist L schwer.

Polynomielle Reduktion — **Definition 6.10**

$L_1 \subseteq \Sigma_1^*$ ist ***polynomiell reduzierbar auf*** $L_2 \subseteq \Sigma_2^*$, geschrieben $L_1 \leq_p L_2$, falls eine polynomielle TM A existiert, die für jedes Wort $x \in \Sigma_1^*$ ein Word $A(x) \in \Sigma_2^*$ berechnet, so dass

$$x \in L_1 \iff A(x) \in L_2$$

A wird eine polynomielle Reduktion von L_1 auf L_2 genannt.

Wieder bedeutet $L_1 \leq_p L_2$, dass L_2 mindestens so schwer ist wie L_1

NP-Schwer — **Definition 6.11**

Eine Sprache L ist NP***-Schwer***, falls für alle $L' \in NP$ gilt $L' \leq_p L$.
Eine Sprache L ist NP***-Vollständig***, falls
(i) $L \in NP$ *(ii)* L NP-Schwer ist.

Lemma 6.7: Falls $L \in P$ und L ist NP-schwer, dann gilt $P = NP$

Satz 6.9: *(Cook)* SAT ist NP-Vollständig

Der Beweis hierfür liefert eine grobe Struktur für weitere Beweise dieser Art und ist auf Seiten 199 - 205 im Buch (= Seiten 211 - 217 im PDF) zu finden. Jedoch sind diese Beweise sehr gross und deshalb nicht prüfungsrelevant.

Lemma 6.8: Falls $L_1 \leq_p L_2$ und L_1 ist NP-Schwer, so ist auch L_2 NP-Schwer

Betrachten wir folgende Sprachen:

$$SAT = \{\Phi \mid \Phi \text{ ist eine erfüllbare Formel in CNF}\}$$
$$CLIQUE = \{(G,k) \mid G \text{ ist ein ungerichteter Graph, der eine } k\text{-clique enthält}\}$$
$$VC = \{(G,k) \mid G \text{ ist ein ungerichteter Graph mit einer Kontenüberdeckung der Mächtigkeit höchstens } k\}$$

Wir erinnern uns daran, dass eine Kontenüberdeckung eines Graphen $G = (V, E)$ jede Menge von Konten $U \subseteq V$ ist, so dass jede Kante aus E mindestens einen Endpunkt in U hat.

Lemma 6.9: $SAT \leq_p CLIQUE$

Lemma 6.10: $CLIQUE \leq_p VC$

Lemma 6.11: $SAT \leq_p 3SAT$, wobei wir beim $3SAT$-Problem bestimmen wollen, ob eine Formel in $3CNF$ (CNF, aber alle Klauseln enthalten höchstens 3 Variabeln) erfüllbar ist.

NPO — **Definition 6.12**

NPO ist die Klasse der Optimierungsprobleme, mit $U = (\Sigma_I, \Sigma_O, L, \mathcal{M}, \text{cost}, \text{goal}) \in NPO$, falls folgende Bedingungen erfüllt sind:
(i) $L \in P$
(ii) Es existiert ein Polynom p_U, so dass
  (a) Für jedes $x \in L$ und jedes $y \in \mathcal{M}(x)$, $|y| \leq p_U(|x|)$
  (b) es existiert ein polynomieller Algorithmus A, der für jedes $y \in \Sigma_O^*$ und jedes $x \in L$ mit $|y| \leq p_U(|x|)$ entscheidet, ob $y \in \mathcal{M}(x)$ oder nicht
(iii) Die Funktion cost kann man in polynomieller Zeit berechnen.

Ein Optimierungsproblem U ist also in NPO, falls
1. man effizient überprüfen kann, ob ein gegebenes Wort ein Problemfall von U ist
2. die Grösse der Lösungen polynomiell in der Grösse des Problemfalls (Eingabe) und in polynomieller Zeit verifizert werden kann, ob y eine zulässige Lösung für einen gegebenen Problemfall ist
3. man die Kosten der zulässigen Lösung effizient berechnen kann

MAX-SAT liegt in NPO

PO **Definition 6.13**

PO ist die Klasse von Optimierungsproblemen $U = (\Sigma_I, \Sigma_O, L, \mathcal{M}, \text{cost}, \text{goal})$, so dass
(i) $U \in NPO$
(ii) $\exists$ polynomieller Algorithmus A, so dass $A(x)$ für jedes $x \in L$ die optimale Lösung für x ist.

Schwellenwert-Sprache **Definition 6.14**

Die Schwellenwert-Sprache für U (ein Optimierungsproblem aus NPO) ist

$$\text{Lang}_U = \{(x, a) \in L \times (\Sigma_{\text{bool}})^* \mid \text{Opt}_U(x) \leq \text{Nummer}(a)\}$$

mit $\text{Opt}_U(x)$ die optimale Lösung, falls goal = Minimum, und

$$\text{Lang}_U = \{(x, a) \in L \times (\Sigma_{\text{bool}})^* \mid \text{Opt}_U(x) \leq \text{Nummer}(a)\}$$

falls goal = Maximum
Wir sagen, dass U ***NP-schwer*** ist, falls Lang_U NP-schwer ist.

Lemma 6.12: Falls ein Optimierungsproblem $U \in PO$, dann $\text{Lang}_U \in P$

Satz 6.10: Sei $U \in NPO$. Falls U NP-schwer ist und $P \neq NP$, dann $U \notin PO$

Lemma 6.13: MAX-SAT ist NP-schwer.

Lemma 6.14: MAX-CL (Das Problem der maximalen Clique) ist NP-schwer

Um zu zeigen, dass solche Probleme U NP-schwer sind, reicht es zu zeigen, dass Lang_U NP-schwer ist, was wir mit einer P-Reduktion machen können.