Karlsruhe Institute of Technology

Theoretische Grundlagen der Informatik

Wintersemester 2013/2014

Institut für Theoretische Informatik
Fakultät für Informatik
Karlsruher Institut für Technologie

Autoren: Prof. Dr. Jörn Müller-Quade
Dirk Achenbach
Jeremias Mechler
Tobias Nilges

Version: Beth
30. Januar 2014

Institut für Theoretische Informatik
Fakultät für Informatik
Karlsruher Institut für Technologie
Am Fasanengarten 5
76128 Karlsruhe

Inhaltsverzeichnis

Vorwort

Liebe Leserin, lieber Leser,

Sie halten das Vorlesungsskript „Theoretische Grundlagen der Informatik“ in der Hand, das begleitend zu meiner gleichnamigen Vorlesung angeboten wird. Das Skript soll Ihnen das Mitschreiben in der Vorlesung ersparen und es Ihnen somit leichter machen, dem dort Gesagten zu folgen.

Das Skript ist aus Aufzeichnungen hervorgegangen, die in meinen Vorlesungen „Informatik 3“ und „TGI“ unter der Mithilfe von Nico Döttling entstanden waren. Die Arbeit des Zusammentragens geht auf Simon Stroh, Jeremias Mechler und Fabian Tinsz zurück. Die ursprüngliche Konzeption meiner „Info 3“-Vorlesung war inspiriert durch die Vorlesungen Professor Prautzschs und meines Vorgängers Professor Beth. Neben dem vorliegenden Skript empfehle ich Ihnen als weiterführende Literatur das Buch *Introduction to the Theory of Computation* von Michael Sipser. Wir folgen mit vielen Konventionen und Methoden diesem Lehrbuch. Für die Informationstheorie empfehlen wir *Elements of Information Theory* von Cover und Thomas.

Leider können wir Fehler in diesem Skript nicht ganz ausschließen. Sollten Sie während seiner Lektüre auf eine Ungereimtheit oder auf Fehler stoßen, melden Sie diese bitte via E-Mail an Dirk Achenbach <dirk.achenbach@kit.edu>.

Das Ziel der Vorlesung „Theoretische Grundlagen der Informatik“ ist, Ihnen ein Verständnis des theoretischen Fundaments der Informatik näherzubringen. Sie lernen in der Vorlesung die der Informatik grundlegenden Konzepte und Modelle kennen. Auch zeigen wir Ihnen, wie man in der Informatik richtig argumentiert, also Beweise führt. Der Aufbau des Skripts folgt dem der zugehörigen Vorlesung. Im Kapitel „Sprachen und Berechnungsmodelle“ wenden wir uns den grundlegenden Berechnungsmodellen, den endlichen Automaten und den Kellerautomaten, zu. Das Kapitel „Berechenbarkeitstheorie“ widmet sich dem bedeutendsten Berechnungsmodell der Informatik – der Turingmaschine – und ihren Grenzen. Im darauffolgenden Kapitel „Komplexitätstheorie“ wird der Frage

nachgegangen, wann ein Problem als schwer gilt. Das letzte Kapitel „Informationstheorie“ widmet sich schließlich dem Konzept der Information.

Bestimmte Beweise haben wir in diesem Skript nicht ausgeführt, sondern Ihnen zur „Übung“ überlassen. Sie sind eingeladen, sich selbst an diesen Beweisen zu versuchen. Manche der ausgelassenen Beweise und Probleme werden auch in der großen Saalübung besprochen werden.

Zuletzt wünsche ich Ihnen viel Freude bei der Lektüre des Skripts, dem Besuch der Vorlesung und der Teilnahme am Übungsbetrieb. Ich hoffe, ich kann Sie ein wenig für die theoretische Informatik begeistern.

Jörn Müller-Quade

Versionshistorie

Version	Datum	Autor(en)	Änderungen
Beth	30.1.2014		Verbindliche Version für die Klausur.

1. Sprachen und Berechnungsmodelle

Wir wenden uns in diesem Kapitel den grundlegenden Berechnungsmodellen in der Informatik zu: Semi-Thue-Systemen, endlichen Automaten und Kellerautomaten. Wir charakterisieren die Funktion eines bestimmten Automaten mittels einer sogenannten (formalen) Sprache. Sie wird durch eine Grammatik erzeugt.

1.1 Grundlagen

Definition 1.1 (Alphabete, Wörter und Sprachen (Wiederholung))**.**

- *Ein Alphabet* Σ *ist eine nicht leere, endliche Zeichenmenge.*
- *Ein Wort* $w = a_1 \dots a_n$ *über ein Alphabet* Σ *ist eine endliche Zeichenkette mit* $a_i \in \Sigma$ $(i = 1, \dots, n)$.
- *Das leere Wort wird mit* ε *(teilweise auch* λ*) bezeichnet.*
- $|w|$ *bezeichnet die Länge des Wortes* w*. Insbesondere gilt* $|\varepsilon| = 0$.
- $\overline{w}$ *bezeichnet das Spiegelbild des Wortes* w*, für* $w = a_1 \dots a_n$ *ist also* $\overline{w} = a_n \dots a_1$.
- *Für Wörter* a, b, c *ist* $w = abc$ *ihre Konkatenation und* a, b, c *sind Teilwörter von* w*.* ε *ist Teilwort jedes Wortes.*
- *Bezüglich der Konkatenation ist* ε *das neutrale Element:* $\varepsilon w = w = w\varepsilon$.
- *Eine Menge* L *von Wörtern wird Sprache genannt.*
- *Die Menge* Σ^+ *ist die Menge nicht-leerer Wörter über* Σ.
- $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$ *heißt kleenescher Abschluss von* Σ.

Information.

Die leere Sprache $L = \emptyset$ enthält nicht das leere Wort ε!
Zu $a \in \Sigma$ sagen wir auch Symbol oder Buchstabe, zu $w = a_1 \dots a_n$ auch String oder Zeichenkette.

Beispiel 1.
Sei $\Sigma = \{0,1\}, \alpha = 01, \beta = 0110$. Dann sind $\Sigma^+ = \{0,1,00,01,10,11,000,\dots\}$ und $\{0\}^* = \{\varepsilon,0,00,000,\dots\}$ Sprachen über Σ und $\alpha\beta = 010110 \in \Sigma^+$ sowie $\beta\alpha = 011001 \in \Sigma^+$.

Nachdem diese Definitionen klären, was Wörter und Sprachen sind, soll in den folgenden Kapiteln behandelt werden, wie sich Sprachen formalisieren und darstellen lassen. Dazu werden bereits aus anderen Vorlesungen bekannte Konzepte wie Automaten und Grammatiken noch einmal wiederholt, aber wie beispielsweise bei den nichtdeterministischen endlichen Automaten auch erweitert.

Typische Fragen, die sich im Zusammenhang mit Sprachen stellen, sind die äquivalente Repräsentation durch verschiedene Maschinen, Ausdrücke und Grammatiken oder das Wortproblem (siehe Definition 1.2).

Problem 1.2 (Wortproblem)**.** *Für eine gegebene Sprache L über einem Alphabet Σ nennen wir die Frage, ob ein Wort $w \in \Sigma^*$ in der Sprache enthalten ist, also ob $w \in L$ ist, das Wortproblem für L.*

1.1.1 Semi-Thue-Systeme

Bevor wir formale Grammatiken einführen, stellen wir einen Vorläufer vor, ein sogenanntes Termersetzungssystem. Dieses ist nicht so mächtig wie die folgenden Grammatiken, sondern spiegelt nur die Grundidee wider.

Definition 1.3 (Semi-Thue-System)**.** *Eine Produktion ist ein Wortpaar $(\alpha,\beta) \in \Sigma^* \times \Sigma^*$, das auch mit $\alpha \to \beta$ bezeichnet wird. Ein Semi-Thue-System besteht aus einem Alphabet Σ und einer endlichen Menge von Produktionen $P \subseteq \Sigma^* \times \Sigma^*$. Wir schreiben (Σ, P).*

Ist ein Semi-Thue-System (Σ, P) gegeben, so sagen wir für $\alpha \to \beta \in P$, dass $\mu\beta\upsilon$ aus $\mu\alpha\upsilon$ erzeugt wird. Wir schreiben $\mu\alpha\upsilon \Rightarrow \mu\beta\upsilon$ und sagen: $\mu\beta\upsilon$ ist aus $\mu\alpha\upsilon$ *direkt ableitbar*. Ist $\mu\beta\upsilon$ in mehreren Schritten aus $\mu\alpha\upsilon$ ableitbar, schreiben wir: $\mu\alpha\upsilon \Rightarrow^* \mu\beta\upsilon$.

Beispiel 2.
$\Sigma = \{\alpha,\beta,\gamma\}, P = \{\alpha\beta \to \gamma, \beta\alpha \to \gamma, \alpha\gamma \to \beta, \alpha\alpha \to \varepsilon, \beta\beta \to \varepsilon, \gamma\gamma \to \varepsilon\}$. Wir leiten ε aus $\alpha\beta\beta\alpha$ ab: $\alpha\beta\beta\alpha \Rightarrow \alpha\alpha \Rightarrow \varepsilon$ oder $\alpha\beta\beta\alpha \Rightarrow \gamma\gamma \Rightarrow \varepsilon$.

1.1.2 Grammatiken

Während Semi-Thue-Systeme mit ihren Produktionen grammatik-ähnliche Eigenschaften vorweisen, fehlt es ihnen an zwei wesentlichen Eigenschaften: Da es nur ein einzelnes Alphabet Σ gibt, lässt sich nicht notwendigerweise zwischen Zwischenergebnissen und fertig abgeleiteten Wörtern unterscheiden; ebenso gibt es keine dediziertes Startsymbol. Erweitert man sie um diese Forderungen, ergibt sich Folgendes:

Definition 1.4 (Grammatik)**.** *Eine Grammatik* G *ist ein 4-Tupel* $G = (\Sigma, V, S, P)$ *mit*

1. *dem endlichen Alphabet* Σ *der Terminalsymbole,*
2. *dem endlichen Alphabet* V *der Variablen mit* $V \cap \Sigma = \emptyset$*,*
3. *dem Startsymbol* $S \in V$ *und*
4. *der Menge von Produktionen* $P \subseteq (V \cup \Sigma)^* V (V \cup \Sigma)^* \times (V \cup \Sigma)^*$ *der Form* $(V \cup \Sigma)^* V (V \cup \Sigma)^* \to (V \cup \Sigma)$.

Die Sprache $L(G)$*, die von* G *erzeugt wird, ist* $\{z \mid z \in \Sigma^*, S \Rightarrow^* z\}$.

Beispiel 3.
$G = (\Sigma, V, S, P)$ mit $\Sigma = \{0, 1\}$, $V = \{S, A, B\}$ und

$$\begin{aligned} P = \{&S \to 0A, \\ &A \to 1A \mid 1B, \\ &B \to \varepsilon\}. \end{aligned}$$

1.2 Endliche Automaten und reguläre Sprachen

Während die allgemeine Grammatik-Definition (Definition 1.4) keinerlei Einschränkungen bezüglich der erlaubten Produktionen macht, gibt es interessante Klassen von Grammatiken, die besondere Eigenschaften aufweisen.

Je nach Einschränkungen ergeben sich verschiedene Klassen von Grammatiken, die eine Hierarchie bilden. Neben den Grammatiken vom Chomsky-Typ 3 aus Definition 1.5 gibt es noch Grammatiken der Chomsky-Typen 0, 1 und 2. Sie sind in der Chomsky-Hierarchie (siehe Definition 1.49) geordnet:
Chomsky-3 $\subset$ Chomsky-2 $\subset$ Chomsky-1 $\subset$ Chomsky-0.

1.2.1 Reguläre Sprachen

Die stärksten Einschränkungen der Produktionen sind bei regulären Sprachen gegeben. Dies führt auch dazu, dass sie am einfachsten zu analysieren sind.

Definition 1.5 (Reguläre Grammatik)**.** *Eine reguläre Grammatik* G *ist ein 4-Tupel* $G = (\Sigma, V, S, P)$ *mit*

1. *dem endlichen Alphabet* Σ *der Terminalsymbole,*
2. *dem endlichen Alphabet* V *der Variablen mit* $V \cap \Sigma = \emptyset$*,*
3. *dem Startsymbol* $S \in V$ *und*
4. *der Menge von Produktionen* P *der Form* $A \to v$ *mit* $v = \varepsilon$*,* $v = a$ *oder* $v = aB$ *mit* $a \in \Sigma$ *und* $A, B \in V$*.*

Eine solche Grammatik wird auch rechtslineare Grammatik *genannt. Besitzt eine Grammatik statt Produktionen* $A \to aB$ *nur Produktionen der Form* $A \to Ba$*, nennt man sie* linkslinear*. Links- oder rechtslineare Grammatiken werden als Grammatiken vom Chomsky-Typ 3 bezeichnet.*

Die regulären Sprachen sind genau die Sprachen, die von regulären Grammatiken erzeugt werden.

Information.

Eine Grammatik darf entweder nur linkslineare oder nur rechtslineare Produktionen enthalten, damit sie regulär ist. Eine Grammatik, die beide Produktionsformen enthält, erzeugt eine Sprache, die zwar noch kontextfrei (Definition 1.26), aber unter Umständen nicht mehr regulär ist.

Information.

Terminierende Produktionen können vollständig durch ε- und rechts- oder linkslineare Produktionen ersetzt werden.

Satz 1.6. *Jede endliche Sprache ist regulär.*

Beweis. In der Übung. □

1.2.2 Einfache Maschinenmodelle

Ein wesentliches Konzept der theoretischen Informatik im Kontext der Berechenbarkeit und Entscheidbarkeit sind Maschinen. Diese bestehen (anschaulich) aus einer Steuereinheit mit einer konstanten, endlichen Menge an Zuständen und einem Speicher mit zugehörigem Speicherzugriffsmechanismus. Bekannte Beispiele, die in den folgenden Kapiteln behandelt werden, sind Zustandsmaschinen (endliche Automaten), Kellerautomaten (Pushdown Automata) und Bandmaschinen (Turing-Maschinen).

Je nach Maschinentyp variieren:

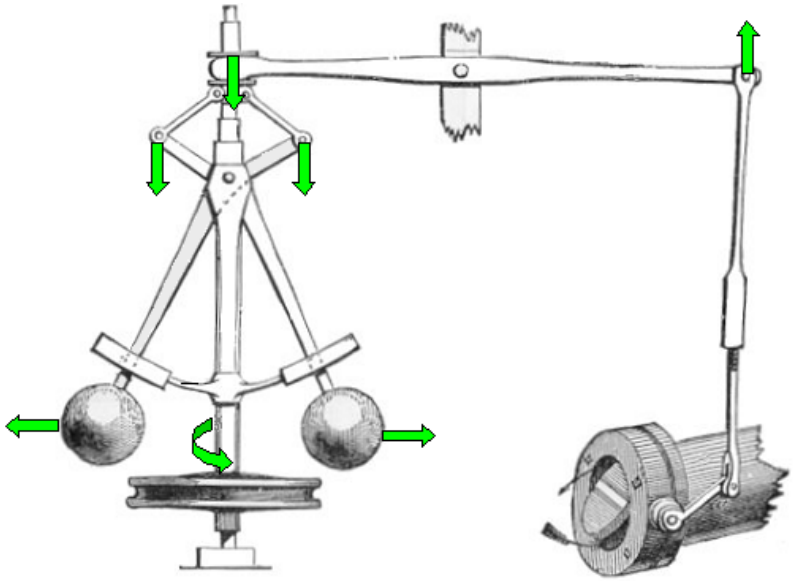
- die Speichergröße und -mechanismus:
  - fest,
  - variabel in der Eingabelänge (z. B. linear oder logarithmisch),
  - unbeschränkt,
  - nur ein Zähler,
  - nur ein Speicher für den aktuellen Zustand und die Zustandsübergangsfunktion.
- das Speicherzugriffsmodell:
  - Last In First Out (LIFO),
  - Random Access Memory (RAM).

Anwendungsbeispiele:

- Aufzugsteuerung,
- Analogrechner,
- Quantenrechner,
- Realzeitanforderungen,
- Parallelrechner,
- kommunizierende Maschinen.

Fliehkraftregler:

- Früher: Spezialhardware,
- heute: frei programmierbares eingebettetes System.



Grafik: Wikimedia.

Abbildung 1.1: Ein Fliehkraftregler.

Übrigens ...

Ein einfaches Beispiel für einen technischen Regelkreis ist der Fliehkraftregler:

„Er wurde 1788 von James Watt in den allgemeinen Maschinenbau eingeführt. James Watt benutzte den Fliehkraftregler, um die Arbeitsgeschwindigkeit der von ihm verbesserten Dampfmaschine konstant zu halten. Die Dimensionierung (Dynamik des Einschwingverhaltens, Schwingungsneigung und Regelcharakteristik) der Fliehkraftregler für diese Aufgabe war die Geburtsstunde der modernen, mathematischen Regelungstechnik.

[...]

Die frühe Bauform zur Regelung an Dampfmaschinen wurde über einen Riemen angetrieben, sie nutzte die Gewichtskraft als rückstellende Kraft. In Ruhe ist die Drosselklappe der Dampfleitung zur Dampfmaschine vollständig geöffnet. Mit Bewegung des Kolbens der Dampfmaschine beginnt der Fliehkraftregler sich zu drehen. Durch diese Drehung werden die zwei Gewichte (z. B. aus Gusseisen) durch die Fliehkraft immer weiter gegen die Schwerkraft nach oben und außen gezogen. Über einen Gelenk- und Hebelmechanismus (siehe Kniehebel) wird in der Dampfleitung der Maschine eine Drosselklappe betätigt, die die Zufuhr des Dampfes zur Maschine verringert. Die Maschine läuft daraufhin langsamer, bis sich ein stabiler Zustand bzw. eine konstante Drehzahl einstellt. Diese Anordnung ist ein Beispiel für einen Regelkreis mit negativer Rückkopplung (Je schneller die Maschine läuft, desto weniger Dampf wird ihr zur Verfügung gestellt).“ Quelle: Wikipedia.

1.2.2.1 Deterministische endliche Automaten

Der einfachste Automatentyp sind deterministische endliche Automaten: Ausgehend vom aktuellen Zustand vollziehen sie für jedes gelesene Zeichen einen Zustandsübergang. Eine formale Definition folgt:

Definition 1.7 (Deterministischer endlicher Automat)**.** *Ein deterministischer endlicher Automat (DEA) ist ein 5-Tupel* $A = (Q, \Sigma, \delta, q_0, F)$ *mit*

- *der endlichen Zustandsmenge* Q,
- *dem endlichen Eingabealphabet* Σ,
- *einer Zustandsübergangsfunktion* $\delta : Q \times \Sigma \to Q$,
- *einem Startzustand* q_0 *und*
- *der Menge der akzeptierenden Zustände (Finalzustände)* $F \subseteq Q$.

DEAs werden auch Akzeptoren genannt.

Information.

Auf Englisch wird ein DEA *deterministic finite automaton (DFA)* genannt.

Übrigens ...

In dieser Vorlesung werden nur Akzeptoren behandelt, also Automaten, die nur 0 oder 1 ausgeben. Automaten mit weiterer Ausgabe, also beispielsweise Mealy- und Moore-Automaten, werden in anderen Vorlesungen behandelt.

Ein DEA $A = (Q, \Sigma, \delta, q_0, F)$ akzeptiert das Wort $w = a_1 \dots a_n$ $(a_i \in \Sigma)$, wenn es eine Abfolge von Zuständen $r_0, \dots r_n$ gibt, sodass

1. $r_0 = q_0$,
2. $\delta(r_i, a_{i+1}) = r_{i+1}$ für $i = 0, \dots, n-1$ und
3. $r_n \in F$.

Man kann sich das „Einlesen“ eines Wortes durch einen endlichen Automaten folgendermaßen vorstellen: Ausgehend vom aktuellen Zustand wird das nächste Zeichen der Eingabe betrachtet. Die Übergangsfunktion gibt dann an, welcher Folgezustand angenommen wird. Daraufhin wird das nächste Zeichen eingelesen, das Eingabeword also sukzessive abgearbeitet. Befindet sich der Automat nach dem Einlesen der Eingabe in einem akzeptierenden Zustand, gilt die Eingabe als akzeptiert; andernfalls gilt sie als abgelehnt.

Die Funktionsweise von endlichen Automaten ist intuitiv einfacher zu erfassen, wenn man sie graphisch repräsentiert. Sie werden dann üblicherweise als gerichtete Graphen gezeichnet. Knoten entsprechen dabei Zuständen. Der Startzustand wird durch einen kleinen Pfeil markiert, akzeptierende Zustände zweigestrichen gezeichnet. Kanten repräsentieren Übergänge; die Kantenbeschriftung gibt an, bei welcher Eingabe der Übergang erfolgt. Abbildung 1.2 stellt einen nichtdeterministischen Automaten dar.

Um auch Wörter als Eingabe zuzulassen, definieren wir die Funktion

$$\overline{\delta} : Q \times \Sigma^* \to Q,\ (q, vw) \mapsto \begin{cases} q & \text{falls } vw = \varepsilon \\ \overline{\delta}(\delta(q, v), w) & \text{falls } v \in \Sigma \text{ und } w \in \Sigma^* \end{cases}$$

Oftmals schreibt man der Einfachheit halber statt $\overline{\delta}$ nur δ.

Übrigens ...

DEAs lassen sich auf verschiedene Weisen implementieren. Beispielhaft seien zwei Möglichkeiten genannt:

- Als Zustandsübergangs-Tabelle: Die Zustands-Übergangsfunktion wird als zweidimensionales Array umgesetzt. Eine Dimension repräsentiert den Zustand, die zweite die Eingabe, in der so indizierten Speicherstelle steht der Folgezustand. Diese Variante ist sehr cache-effizient, vermeidet Sprünge und ist insbesondere für vollständige Zustandsübergangsfunktionen geeignet; ansonsten muss ein besonderer „ablehnender Zustand“ hinzugefügt werden.
- Als Switch-Case-Struktur: Für jeden Zustand sind die möglichen Übergänge bei entsprechender Eingabe ausprogrammiert, was bei unvollständigen Zustandsübergangsfunktionen vorteilhaft sein kann. Diese Variante ist insbesondere für objekt-orientierte Sprachen geeignet.

1.2.2.2 Nichtdeterministische endliche Automaten

Während es bei DEAs zu jeder Eingabe (höchstens) einen Folgezustand gibt, erlauben wir nun für dieselbe Eingabe mehrere Folgezustände sowie Zustandsübergänge ohne Eingabe. Das Verhalten eines solchen Automaten ist also nicht mehr deterministisch. Formal erweitern wir dazu die Zustandsübergangsfunktion δ dahingehend, dass es statt einzelnen Folgezuständen nun Mengen von Folgezuständen gibt und auch das leere Wort ε eine gültige Eingabe darstellt. Ein Beispiel ist in Abbildung 1.2 gezeigt.

Definition 1.8 (Nichtdeterministischer endlicher Automat)**.** *Ein nichtdeterministischer endlicher Automat (NEA) ist ein 5-Tupel* $A = (Q, \Sigma, \delta, q_0, F)$ *mit*

- *der endlichen Zustandsmenge* Q,
- *dem endlichen Eingabealphabet* Σ,
- *einer Zustandsübergangsfunktion* $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$,
- *einem Startzustand* q_0 *und*
- *der Menge der akzeptierenden Zustände* $F \subseteq Q$.

Information.

Auf Englisch wird ein NEA *nondeterministic finite automaton (NFA)* genannt.

Definition 1.9 (ε-Abschluss)**.** *Sei* $A = (Q, \Sigma, \delta, q_0, F)$ *ein NEA. Die Funktion*

$$E : Q \to 2^Q,\ q \mapsto \{q_i \in Q \mid q_i \text{ ist von } q \text{ durch eine Reihe von } \varepsilon\text{-Übergängen erreichbar}\}$$

heißt ε*-Abschluss von* q*. Insbesondere ist* $q \in E(q)$.

Wie schon bei DEAs definieren wir die Funktion $\bar{\delta}$, die die Funktionalität von δ „erweitert“:

1. $\overline{\delta} : Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$

$$\overline{\delta}(q_i, a) = \begin{cases} E(q_i) & \text{falls } a = \varepsilon \\ \bigcup\limits_{q_j \in E(q_i)} \left(\bigcup\limits_{q_k \in \delta(q_j, a)} E(q_k) \right) & \text{für } a \in \Sigma \end{cases}$$

   $\overline{\delta}$ berücksichtigt nun den ε-Abschluss für einzelne Zeichen.

2. $\overline{\delta} : 2^Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$

$$\overline{\delta}(R, a) = \begin{cases} \bigcup\limits_{q_i \in R} E(q_i) & \text{falls } a = \varepsilon \\ \bigcup\limits_{q_i \in R} \overline{\delta}(q_i, a) & \text{für } a \in \Sigma \end{cases}$$

   $\overline{\delta}$ berücksichtigt nun Zustandsmengen.

3. $\overline{\delta} : Q \times \Sigma^* \to 2^Q$

$$\overline{\delta}(q_i, w) = \begin{cases} E(q_i) & \text{falls } w = \varepsilon \\ \overline{\delta}(q_i, w) & \text{falls } w \in \Sigma \\ \bigcup\limits_{q_j \in \overline{\delta}(q_i, v)} \overline{\delta}(q_j, a) & \text{falls } w = va, a \in \Sigma, |v| > 0 \end{cases}$$

   $\overline{\delta}$ unterstützt nun Wörter als Eingabe.

4. $\overline{\delta} : 2^Q \times \Sigma^* \to 2^Q$

$$\overline{\delta}(R, w) = \bigcup_{q_i \in R} \overline{\delta}(q_i, w)$$

   $\overline{\delta}$ unterstützt nun Zustandsmengen bei Wörtern als Eingabe.

Ein NEA $A = (Q, \Sigma, \delta, q_0, F)$ akzeptiert das Wort $w = a_1 \dots a_n$, wenn es $a'_1 \dots a'_m$ mit $a'_i \in \Sigma \cup \{\varepsilon\}$ und eine Abfolge von Zuständen $r_0, \dots, r_m$ gibt, sodass

1. $r_0 = q_0$,
2. $a'_1 \dots a'_m = w$,
3. $r_{i+1} \in \delta(r_i, a'_{i+1})$ für $i = 0, \dots, m-1$ und
4. $r_m \in F$.

Definition 1.10 (Sprache eines Automaten). *Zu einem gegebenen Automaten* $A = (Q, \Sigma, \delta, q_0, F)$ *bezeichnet* $L(A)$ *die von ihm akzeptierte Sprache:*

$$L(A) = \{w \in \Sigma^* \mid \overline{\delta}(q_0, w) \cap F \neq \emptyset\}.$$

Beispiel 4.

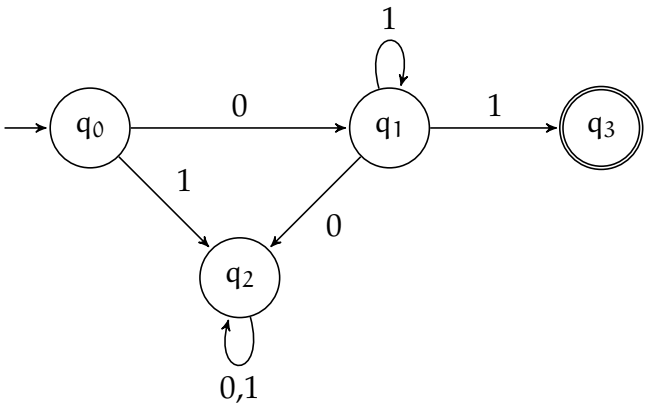


Abbildung 1.2: Nichtdeterministischer Automat (NEA), der 01^+ erkennt.

Nachdem wir jetzt zwei Maschinenmodelle kennengelernt haben, stellt sich die Frage nach der Mächtigkeit dieser Modelle. Dazu zeigen wir zuerst, dass diese beiden Modelle äquivalent sind.

Satz 1.11. *Für jeden nichtdeterministischen endlichen Automaten existiert ein deterministischer endlicher Automat, der dieselbe Sprache akzeptiert.*

Beweis. Zum NEA $A = (Q, \Sigma, \delta, q_0, F)$ konstruieren wir einen DEA $A' = (Q', \Sigma', \delta', q_0', F')$:

Wir fassen dazu für jeden möglichen Übergang die Folgezustände in Mengenschreibweise in einen neuen Zustand zusammen. Ein einzelner Folgezustand q_j wird also zum Zustand $\{q_j\}$, die Folgezustände $q_k, \ldots, q_l$ werden zum Zustand $\{q_k, \ldots, q_l\}$. Wir haben also im Vergleich zum NEA keine Menge von Zuständen mehr, sondern einzelne Zustände, die Mengen enthalten.

1. Setze $Q' = 2^Q$ und $q_0' = \{q_0\}$. Wir legen also eine neue Zustandsmenge als Potenzmenge der alten Zustandsmenge fest. Die Zustände des neuen Zustands sind also Mengen. Unser neuer Startzustand ist die Menge, die ausschließlich den alten Startzustand enthält.

2. Für $R \in Q'$ und $a \in \Sigma$ sei $\delta'(R, a) = \{q_i \in Q \mid q_i \in \delta(q_j, a)$ für ein $q_j \in R\}$. Die neue Übergangsfunktion fasst also alle möglichen Zustände $q_i \in Q$ zusammen, die man aus allen Zuständen der Zustandsmenge durch Eingabe von a erreichen kann.

3. $F' = \{R \in Q' \mid \exists q \in R : q \in F\}$. Akzeptierende Zustände sind Zustände, die einen Finalzustand enthalten.

Diese Konstruktion ist ausreichend, solange $(Q, \Sigma, \delta, q_0, F)$ keine ε-Übergänge hat. Befinden sich ε-Übergänge in δ, erweitern wir die Definition von δ' und q_0' um den ε-Abschluss (vgl. Definition 1.9):

1. $q_0' = E(\{q_0\})$
2. $\delta'(R, a) = \{q_i \mid q_i \in E(\delta(q_j, a))$ für $q_j \in R\}$

□

Durch die Äquivalenz der beiden Modelle können wir elegant beweisen, dass endliche Automaten durch Grammatiken vom Chomsky-Typ 3 beschrieben werden können und umgekehrt.

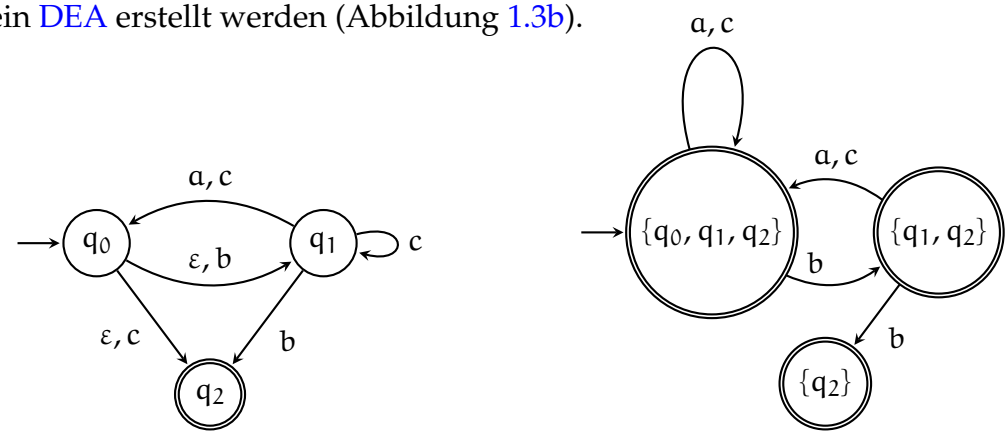
Satz 1.12. *Die Klasse der von endlichen Automaten akzeptierten Sprachen und die Klasse der von Grammatiken vom Chomsky-Typ 3 erzeugten Sprachen stimmen überein.*

Beweis. „⇒“
Gegeben sei ein DEA $A = (Q, \Sigma, \delta, q_0, F)$ für die Sprache $L = L(A)$. Wir konstruieren eine Grammatik $G = (\Sigma, V, S, P)$ für L, indem wir eine Berechnung eines Automaten simulieren. Dazu sei $V = Q, \Sigma = \Sigma'$ und $S = q_0$, wobei Σ' das Alphabet von L ist. Für $\delta(q, a) = q'$ nehmen wir $q \to aq'$ in P auf, für alle $q \in F$ zusätzlich $q \to \varepsilon$. Falls $w = a_1 \dots a_n \in L$, dann existiert eine Abfolge von Zuständen $q_0, q_1, \dots, q_n$ mit $q_n \in F$ und $\delta(q_i, a_{i+1}) = q_{i+1}$. Entsprechend gilt in der Grammatik $q_0 \Rightarrow a_1 q_1 \Rightarrow a_1 a_2 q_2 \Rightarrow \cdots \Rightarrow a q_n \Rightarrow w$. Da alle Ableitungen diese Form haben, erzeugt G genau L.

„⇐“
Umgekehrt sei eine Grammatik $G = (\Sigma, V, S, P)$ vom Chomsky-Typ 3 gegeben, deren Produktionen o. B. d. A. der Form $A \to aB$ oder $A \to \varepsilon$ sind (man sieht einfach, dass sich $A \to a$ durch Produktionen $A \to aB$ und $B \to \varepsilon$ ausdrücken lässt) und $L(G)$ die von G erzeugte Sprache. Wir konstruieren einen NEA $A = (Q, \Sigma, \delta, q_0, F)$, der $L(G)$ akzeptiert. Dazu sei $Q = V$, $q_0 = S$ und F die Menge aller $A \in V$ für die $A \to \varepsilon \in P$. Schließlich sei $\delta(A, a) = \{B \mid (A \to aB) \in P\}$. Aus der Folge von Ableitungen der Grammatik lässt sich also eine Folge von Zustandsübergängen für A konstruieren.

□

Beispiel 5.
Durch die Potenzmengenkonstruktion kann aus dem NEA aus Abbildung 1.3a ein DEA erstellt werden (Abbildung 1.3b).



(a) NEA (b) DEA

1.2.3 Reguläre Ausdrücke

Neben Grammatiken und Automaten bieten reguläre Ausdrücke eine weitere Repräsentationsmöglichkeit von regulären Sprachen. Sie finden in der Praxis zum Filtern und Parsen von Text häufig Anwendung und werden sowohl von Stringbibliotheken gängiger Programmiersprachen als auch eigenständigen Tools unterstützt.

Definition 1.13 (Reguläre Ausdrücke). *Sei Σ ein endliches Alphabet. Dann ist*

- *$\emptyset$ der reguläre Ausdruck, der die leere Sprache,*
- *ε der reguläre Ausdruck, der die Sprache des leeren Worts und*
- *$a \in \Sigma$ der reguläre Ausdruck, der die Sprache mit dem Wort $w = a$*

bezeichnet. Sind A_1 und A_2 reguläre Ausdrücke, dann sind auch

- *die Vereinigung $A_1 + A_2$,*
- *die Konkatenation $A_1 \cdot A_2$ und*
- *der kleenesche Abschluss $(A_1)^*$*

reguläre Ausdrücke.

> **Information.**
>
> Statt $+$ ist auch $|$ („oder“) für die Vereinigung von Sprachen üblich.

Häufig wird die Schreibweise vereinfacht, d. h. Klammern und Multiplikationszeichen weggelassen. Dabei gilt folgende Operatoren-Präzedenz:

1. Kleenescher Abschluss: *
2. Konkatenation: $\cdot$
3. Vereinigung: $+$

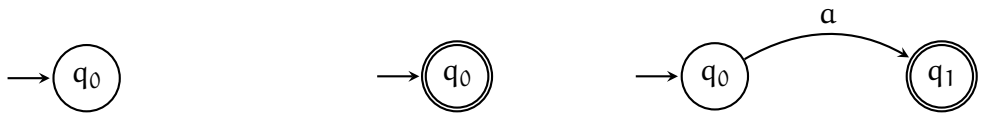
Beispiel 6.
Sei $\Sigma = \{a, b\}$. Es gilt: $a(a + b)^* \equiv (a) \cdot (((a) + (b))^*)$

Satz 1.14. *Genau die regulären Sprachen lassen sich durch reguläre Ausdrücke beschreiben.*

Beweis. „⇒“:
Übung.

„⇐“:
Wir skizzieren eine strukturelle Induktion.

$\emptyset$, ε und a für $a \in \Sigma$ bezeichnen reguläre Sprachen. Diese werden durch die Automaten in Abbildung 1.4 erkannt.



(a) Ein Automat, der $\emptyset$ erkennt.

(b) Ein Automat, der die Sprache $L = \{\varepsilon\}$ erkennt.

(c) Ein Automat, der die Sprache $L = \{a\}$ erkennt.

Abbildung 1.4: Endliche Automaten für einfache reguläre Ausdrücke.

Für A_1 und A_2 gebe es schon endliche Automaten, die die zugehörigen Sprachen akzeptieren. Seien diese mit A_{A_1} und A_{A_2} bezeichnet.

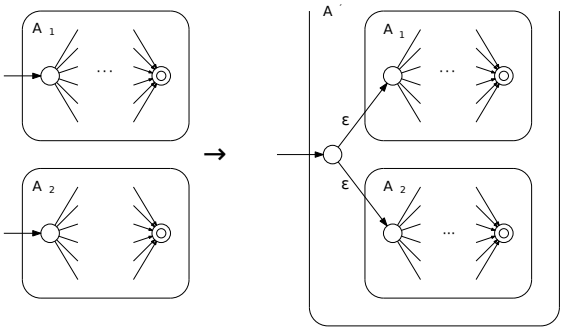
Wir müssen nun zeigen, dass je ein Automat existiert, der $A_1 + A_2$, $A_1 \cdot A_2$ und A_1^* akzeptiert.

- $A_1 + A_2$: Wir führen einen neuen Startzustand ein. Aus diesem führen zwei ε-Übergänge in die Anfangszustände von A_{A_1} und A_{A_2}.
  Siehe Abbildung 1.5a.

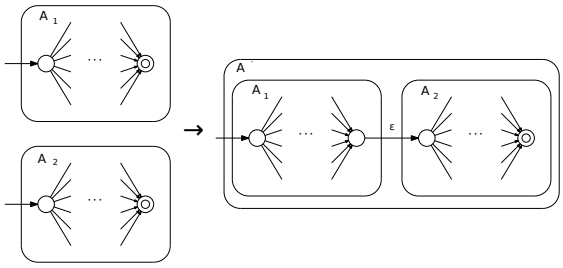
- $A_1 \cdot A_2$: Wir „konkatenieren“ A_{A_1} und A_{A_2}: Aus jedem akzeptierenden Zustand von A_{A_1} führen wir einen ε-Übergang in den Startzustand von A_{A_2}.
  Siehe Abbildung 1.5b.

- A_1^*: Wir führen einen neuen Startzustand ein. Dieser ist gleichzeitig ein akzeptierender Zustand. Aus dem neuen Startzustand führt ein ε-Übergang in den Startzustand von A_{A_1}. Alle akzeptierenden Zustände von A_{A_1} führen mittels ε-Übergängen zusätzlich in den neuen Startzustand. Siehe Abbildung 1.5c.

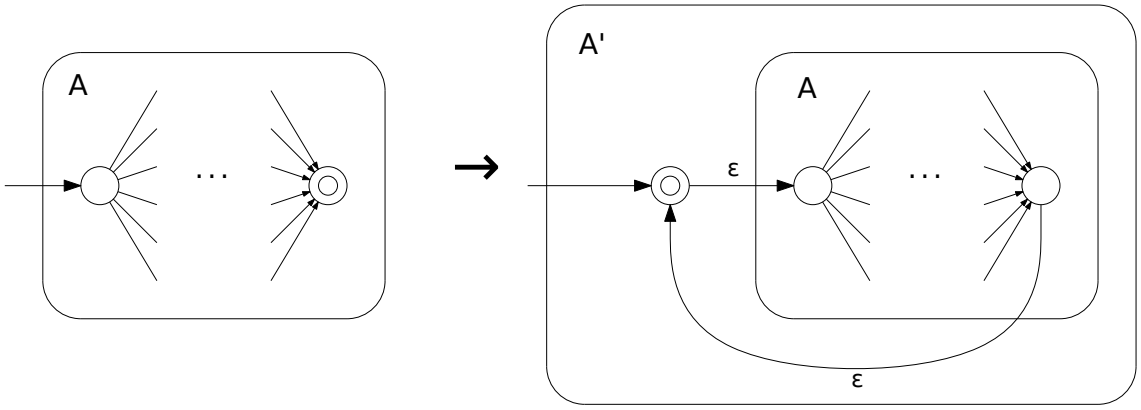
□



(a) Konstruktion, um aus Automaten für A_1 und A_2 einen Automaten zu machen, der $A_1 + A_2$ akzeptiert.



(b) Konstruktion, um aus Automaten für A_1 und A_2 einen Automaten zu machen, der $A_1 \cdot A_2$ akzeptiert.



(c) Konstruktion, um aus einem Automaten für A_1 einen Automaten zu machen, der A_1^* akzeptiert.

Abbildung 1.5: Automaten für reguläre Ausdrücke.

Übrigens ...

Die regulären Ausdrücke von Werkzeugen wie `awk`, `Perl` oder `vim` sind mächtiger als das, was wir als reguläre Ausdrücke definiert haben. Diese „erweiterten regulären Ausdrücke“ beschreiben wesentlich mehr, als von endlichen Automaten erkannt werden kann.

Literaturempfehlung: „Reguläre Ausdrücke“ von Jeffrey E. F. Friedl bei O'Reilly.

1.2.4 Minimierung von Automaten

Gegeben sei eine Sprache L und ein DEA $A = (Q, \Sigma, \delta, q_0, F)$ mit $L(A) = L$.

Satz 1.15. *Zu A gibt es einen Automaten $A' = (Q', \Sigma', \delta', q_0', F')$ mit $L(A') = L(A) = L$ und minimalem $|Q'|$.*

Die Idee ist, einen Automaten zu konstruieren, bei dem alle Zustände, die für beliebige Eingaben von q_0 aus nie erreicht werden können, entfernt werden. Diese

Zustände nennen wir auch *überflüssige Zustände*. Weiterhin sollen alle äquivalenten Zustände, also solche, die für alle Wörter zu gleichem Akzeptanzverhalten führen, zusammengefasst werden. Der so enstehende Automat heißt Äquivalenzklassenautomat.

Definition 1.16 (Äquivalente Zustände)**.** *Zwei Zustände q_i, q_j eines endlichen Automaten heißen äquivalent ($q_i \equiv q_j$), wenn für alle Wörter $w \in \Sigma^*$ gilt:*

$$\overline{\delta}(q_i, w) \cap F \neq \emptyset \iff \overline{\delta}(q_j, w) \cap F \neq \emptyset,$$

also genau dann, wenn aus ihnen heraus genau die gleichen Wörter in einen akzeptierenden Zustand führen.

$[q]$ bezeichnet die Äquivalenzklasse von q, also alle $q_i \in Q$, für die $q_i \equiv q$ gilt.

Damit können wir nun einen Äquivalenzklassenautomat definieren.

Definition 1.17 (Äquivalenzklassenautomat)**.** *Der Äquivalenzklassenautomat $A' = (Q', \Sigma', \delta', q_0', F')$ zu einem deterministischen endlichen Automaten $A = (Q, \Sigma, \delta, q_0, F)$ ist definiert durch:*

- $Q' = \{[q] \mid q \in Q\}$
- $\Sigma' = \Sigma$
- $q_0' = [q_0]$
- $F' = \{[q] \mid q \in F\}$
- $\delta'([q], a) = [\delta(q, a)]$

Satz 1.18. *Der Äquivalenzklassenautomat A' ist wohldefiniert und akzeptiert die gleiche Sprache wie A.*

Beweis. Dazu gilt es zu zeigen, dass ein Finalzustand von A' nur zu einem Finalzustand von A äquivalent ist und δ bei gleicher Eingabe stets in einen äquivalenten Zustand überführt.

1. Wohldefiniertheit von F':
   Für $q_i \in [q]$ ist zu zeigen: $q_i \in F \iff q \in F$. Wir betrachten nun die Äquivalenz für das leere Wort ε. Nach Definition gilt:

   $$q \in F \iff \delta(q, \varepsilon) \in F \iff \delta(q_i, \varepsilon) \in F \iff q_i \in F$$

   Es gilt also entweder $q_i \in F \wedge q \in F$ oder $q_i \notin F \wedge q \notin F$. Damit ist F' wohldefiniert.

2. Wohldefiniertheit von δ':
   Es ist zu zeigen: $\forall a \in \Sigma : q_i \equiv q \Rightarrow \delta(q_i, a) \equiv \delta(q, a)$. Nach Definition gilt für $q_i \equiv q$: $\forall w \in \Sigma^* : q_i \equiv q \Rightarrow \overline{\delta}(q_i, w) \equiv \overline{\delta}(q, w)$.

$$\begin{aligned} q_i \equiv q \Rightarrow \forall w \in \Sigma^* : \overline{\delta}(q_i, w) \in F &\iff \overline{\delta}(q, w) \in F \\ &\Rightarrow \forall a \in \Sigma, w \in \Sigma^* : \overline{\delta}(\delta(q_i, a), w) = \overline{\delta}(q_i, aw) \in F \\ &\iff \forall a \in \Sigma, w \in \Sigma^* : \overline{\delta}(\delta(q, a), w) = \overline{\delta}(q, aw) \in F \\ &\Rightarrow \delta(q_i, a) \equiv \delta(q, a) \end{aligned}$$

3. Es bleibt zu zeigen, dass A' dieselbe Sprache akzeptiert:
   Dazu sei $w \in \Sigma^*$ und $q_0, q_1, \ldots, q_n$ die von A beim Lesen durchlaufende Zustandsfolge. Nach Konstruktion durchläuft A' $[q_0], [q_1], \ldots, [q_n]$. A akzeptiert genau dann, wenn $q_n \in F$. Dies gilt genau dann, wenn $[q_n] \in F'$ und dies genau dann, wenn A' akzeptiert.

□

1.2.4.1 Der Satz von Nerode

In diesem Abschnitt wird die Nerode-Relation eingeführt, welche sowohl einen Ansatz zur Minimierung von Automaten als auch ein Kriterium für die Regularität von Sprachen liefert.

Definition 1.19 (Rechtslineare Äquivalenzrelation)**.** *Eine Äquivalenzrelation* R *auf* Σ^* *heißt rechtslinear, wenn gilt:*

$$\forall x, y, z \in \Sigma^* : x \, R \, y \Rightarrow xz \, R \, yz$$

Mit dem Index von R, $\mathrm{ind}(R)$, *bezeichnet man die Anzahl der Äquivalenzklassen von* R.

Beispiel 7.
Es sei A ein endlicher Automat und $x, y \in \Sigma^*$. Dann ist

$$R = \{(x, y) \mid \overline{\delta}(q_0, x) = \overline{\delta}(q_0, y)\}$$

eine rechtslineare Relation. Der Index von R ist gleich der Anzahl nicht überflüssiger Zustände von A.

Definition 1.20 (Nerode-Relation)**.** *Für eine Sprache* $L \subseteq \Sigma^*$ *ist die Nerode-Relation* R_L *definiert durch:*

$$\forall x, y \in \Sigma^* : x \, R_L \, y \iff (\forall z \in \Sigma^* : xz \in L \iff yz \in L)$$

R_L *ist eine Äquivalenzrelation und rechtslinear:*

$$\begin{aligned} x \, R_L \, y \Rightarrow \forall w \in \Sigma^* : xw \in L &\iff yw \in L \\ &\Rightarrow \forall w, z \in \Sigma^* : xzw \in L \iff yzw \in L \\ &\Rightarrow \forall z \in \Sigma^* : xz \, R_L \, yz \end{aligned}$$

Satz 1.21 (Satz von Myhill-Nerode). *Für* $L \subseteq \Sigma^*$ *sind die folgenden Aussagen äquivalent:*

1. L *wird von einem DEA akzeptiert.*
2. L *ist die Vereinigung von Äquivalenzklassen einer rechtslinearen Äquivalenzrelation mit endlichem Index.*
3. *Die Nerode-Relation* R_L *hat endlichen Index.*

Beweis.

- $(1) \Rightarrow (2)$
  Sei $A = (Q, \Sigma, \delta, q_0, F)$ der DEA, der L akzeptiert. Weiterhin sei die Äquivalenzrelation R definiert durch

  $$\forall x, y \in \Sigma^* : x \, R \, y \iff \overline{\delta}(q_0, x) = \overline{\delta}(q_0, y).$$

  R ist offensichtlich rechtslinear und besitzt maximal $|Q| \neq \infty$ Äquivalenzklassen, hat also endlichen Index. Die Vereinigung der Äquivalenzklassen, für die $\overline{\delta}(q_0, w) \in F$ gilt, ist gerade L.

- $(2) \Rightarrow (3)$
  Sei R nach Voraussetzung wie oben. Dann gilt:

  $$\forall x, y, z \in \Sigma^* : x \, R \, y \Rightarrow xz \, R \, yz$$

  Daraus folgt:

  $$xz \in L \iff yz \in L$$

  Insgesamt gilt also:

  $$x \, R \, y \Rightarrow x \, R_L \, y$$

  Da R nach Voraussetzung endlichen Index hat, folgt daraus, dass $\mathrm{ind}(R_L) \leqslant \mathrm{ind}(R) < \infty$.

- $(3) \Rightarrow (1)$
  Nach Voraussetzung hat R_L endlich viele Äquivalenzklassen. Wir konstruieren den Nerode-Automaten $A' = (Q', \Sigma', \delta', q_0', F')$:

  1. $Q' = \{[w] \mid w \in \Sigma^*\}$
  2. $\Sigma' = \Sigma$
  3. $q_0' = [\varepsilon]$
  4. $F' = \{[w] \mid w \in L\}$
  5. $\overline{\delta}'([w], a) = [wa]$

$\overline{\delta}'$ (und damit δ') ist wohldefiniert, denn ist $[w] = [w']$, gilt $w \, R_L \, w'$. Aus der Rechtslinearität von R_L folgt $wa \, R_L \, w'a$. Daher ist auch $[wa] = [w'a]$ und damit $\overline{\delta}'([w], a) = \overline{\delta}'([w'], a)$.

□

Korollar 1.22. *Der Nerode-Automat ist minimal.*

Beweis. Es sei $A = (Q, \Sigma, \delta, q_0, F)$ der Nerode-Automat und $A' = (Q', \Sigma', \delta', q_0', F')$ ein DEA, der L akzeptiert. Aus (1) $\Rightarrow$ (2) des letzten Beweises folgt, dass eine rechtslineare Äquivalenzrelation R mit Index $\mathrm{ind}(R) \leqslant |Q'|$ existiert. Wegen (2) $\Rightarrow$ (3) ist $\mathrm{ind}(R_L) \leqslant \mathrm{ind}(R)$. Insgesamt hat A also $|Q| = \mathrm{ind}(R_L) \leqslant \mathrm{ind}(R) = |Q'|$ Zustände und ist damit minimal.

□

Es bleibt zu zeigen, dass der Äquivalenzklassenautomat, der durch die Relation $\equiv$ induziert wird, minimal ist.

Satz 1.23. *Es gilt* $\mathrm{ind}(\equiv) = \mathrm{ind}(R_L)$. *Der Äquivalenzklassenautomat ist also minimal.*

Beweis. Wir müssen zeigen, dass $\mathrm{ind}(\equiv) \leqslant \mathrm{ind}(R_L)$ gilt, also für alle $x, y \in \Sigma^*$

$$x \, R_L \, y \Rightarrow \overline{\delta}(q_0, x) \equiv \overline{\delta}(q_0, y)$$

gilt.

$$\begin{aligned} x \, R_L \, y &\Rightarrow \forall z \in \Sigma^* : xz \in L \iff yz \in L \\ &\Rightarrow \forall z \in \Sigma^* : \overline{\delta}(q_0, xz) \in F \iff \overline{\delta}(q_0, yz) \in F \\ &\Rightarrow \forall z \in \Sigma^* : \overline{\delta}(\delta(q_0, x), z) \in F \iff \overline{\delta}(\delta(q_0, y), z) \in F \\ &\Rightarrow \overline{\delta}(q_0, x) \equiv \overline{\delta}(q_0, y). \end{aligned}$$

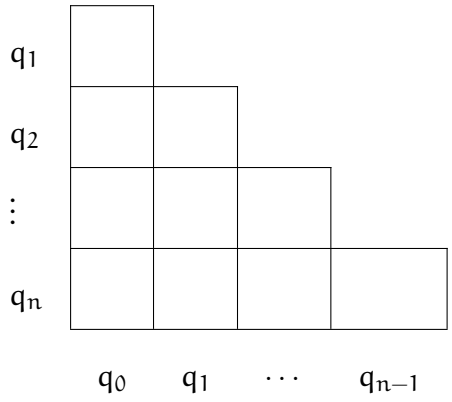
□

Der Satz von Myhill-Nerode liefert also eine Möglichkeit, die minimale Anzahl von Zuständen zu bestimmen und einen minimalen Automaten zu konstruieren. Weiterhin gibt er ein Kriterium an, um reguläre Sprachen zu erkennen.

1.2.4.2 Konstruktion des Äquivalenzklassenautomaten

Gegeben sei ein DEA $A = (Q, \Sigma, \delta, q_0, F)$ ohne überflüssige Zustände. Indem wir systematisch alle Zustände, die nicht zueinander äquivalent sind, bestimmen, konstruieren wir den dazu gehörigen Äquivalenzklassenautomaten $A' = (Q', \Sigma', \delta', q_0', F')$.

1. Entferne alle überflüssigen Zustände. Es bleiben die Zustände $q_0, \ldots, q_n$ übrig. Erstelle eine Tabelle wie folgt:



2. Die Zustandspaare $(q_i, q_j) \in (F \times \overline{F}) \cup (\overline{F} \times F)$ (also die, bei denen q_i und q_j unterschiedliches Akzeptanzverhalten haben), sind nicht verschmelzbar und werden markiert.

3. Markiere alle Paare (q_i, q_j), für die es ein $a \in \Sigma$ gibt, für das $(\delta(q_i, a), \delta(q_j, a))$ markiert ist.

4. Wiederhole Schritt 3, bis keine Markierungen hinzu kommen.

5. Verschmelze die unmarkierten Paare.

Wir erhalten A', indem wir für alle i den Zustand

$$[q_i] = \{q_i\} \cup \{q_j \mid (q_i, q_j) \text{ ist nicht markiert.}\}$$

setzen und den Rest entsprechend Definition 1.17 anpassen.

1.2.4.3 Korrektheit des Algorithmus

Beweis. Beweis mit vollständiger Induktion über die Wortlänge $n = |w|$.

1. Induktionsanfang: Für $w = \varepsilon$, also $n = |w| = 0$ werden die nichtäquivalenten Zustände in Schritt 2 korrekt und vollständig markiert.

2. Induktionsvoraussetzung: Für alle Wörter w mit $|w| = n$ seien alle nichtäquivalenten Zustände korrekt markiert. Schritt 3 wurde dabei $n-1$-mal ausgeführt.

3. Induktionsschritt: Wir betrachten nun die Wörter $w' = aw$ mit $|w'| = n+1$ und $a \in \Sigma$. Für ein nichtäquivalentes Zustandspaar (q_i, q_j) wurden die Folgezustände $(\delta(q_i, a), \delta(q_j, a))$ bereits in der n-ten Iteration von Schritt 3 betrachtet und nach Induktionsvoraussetzung korrekt markiert. In der aktuellen Iteration wird also auch (q_i, q_j) markiert.

□

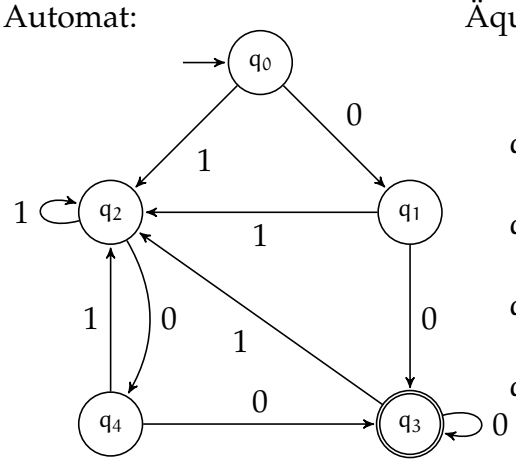
Information.

Neben dem hier vorgestellten Markierungsverfahren gibt es noch weitere, beispielsweise das in der Vorlesung von Prof. Dr. Wagner behandelte Verfahren:

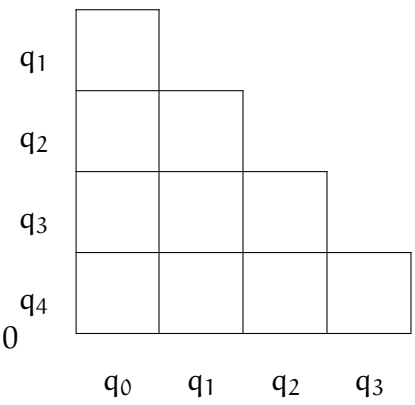
1. Alle Zustände werden zu einer Zustandsmenge zusammengefasst.
2. Die Zustände, die für die Eingabe $w = \varepsilon$ unterschiedliches Akzeptanzverhalten haben, werden in zwei unterschiedliche Zustandsmengen getrennt.
3. Iterativ für alle $w \in \Sigma^+$ mit aufsteigender Länge werden die so entstandenen Zustandsmengen bei unterschiedlichem Akzeptanzverhalten weiter aufgeteilt.
4. Ändern sich die Zustandsmengen für alle Wörter der Länge n nicht weiter, endet das Verfahren. Die einzelnen Mengen enthalten die zueinander äquivalenten Zustände.

Beispiel 8.

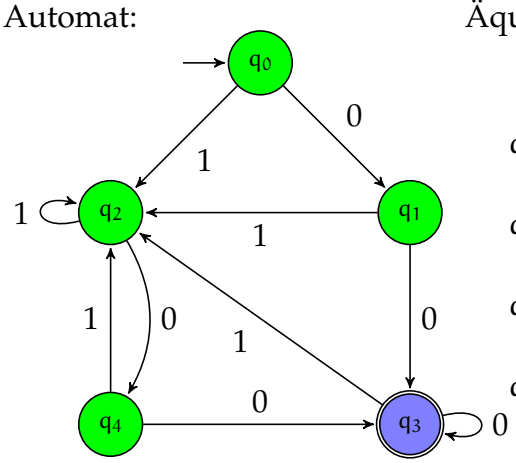
0. Schritt. Automat:



Äquivalenzklassen:



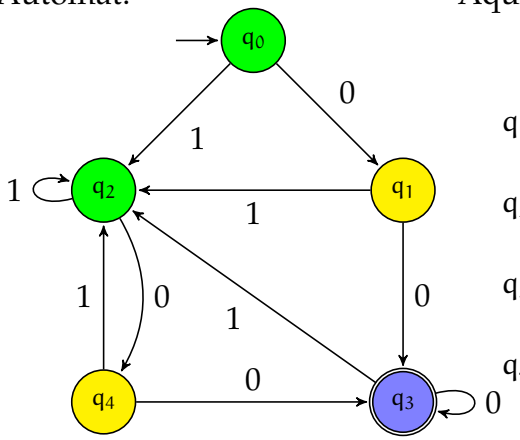
1. Schritt. Automat: Äquivalenzklassen:



	q_0	q_1	q_2	q_3
q_1				
q_2				
q_3	0	0	0	
q_4				0

Akzeptierende Zustände werden von nicht akzeptierenden Zuständen getrennt.

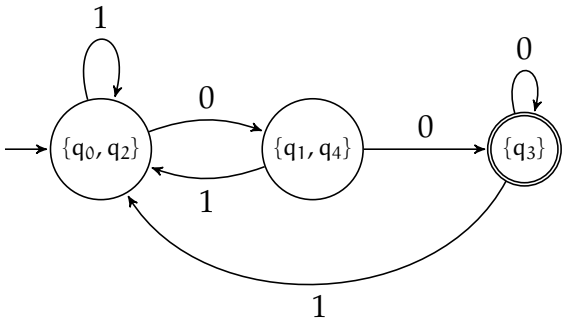
2. Schritt. Automat: Äquivalenzklassen:



	q_0	q_1	q_2	q_3
q_1	1			
q_2		1		
q_3	0	0	0	
q_4	1		1	0

Alle Zustandspaare (q_i, q_j), für die es ein $a \in \Sigma$ gibt, für das $(\delta(q_i, a), \delta(q_j, a))$ im letzten Schritt markiert wurde, werden markiert. Es sind keine weiteren Markierungen mehr möglich, die unmarkierten Zustandspaare werden zu den Zuständen $\{q_0, q_2\}$ und $\{q_1, q_4\}$ zusammengefasst.

3. Resultierender Äquivalenzklassenautomat:



Mysterium.

Die Sternhöhe eines regulären Ausdrucks bezeichnet die Tiefe, in der der Kleenesche Stern in einem regulären Ausdruck geschachtelt ist. Beispielsweise hat der Ausdruck a^* die Sternhöhe 1 und $(a^*b)^*$ die Sternhöhe 2. Die (minimale) Sternhöhe einer *Sprache* wiederum ist die geringste Schachtelungstiefe, mit der eine Sprache durch einen regulären Ausdruck beschrieben werden kann. Da in unserem Beispiel $(a^*b)^* = (a + b)^*$, hat die zugehörige Sprache die Sternhöhe 1. Es existiert ein Algorithmus, der die Sternhöhe einer Sprache bestimmen kann.

Führt man das Komplement als zusätzlichen Operator ein, gewinnt man eine Möglichkeit, in bestimmten Situationen auf die Verwendung des Kleeneschen Sterns zu verzichten. Zu einem gegebenen regulären Ausdruck R bezeichnet sein Komplement $\overline{R}$ die Menge aller Wörter über dem Alphabet, die durch R *nicht* beschrieben werden, also $\Sigma^* \setminus R$.

Unter Verwendung des Komplementoperators gelingt nun folgender Trick: $(a + b)^* = \overline{b} + b$. Wir konnten in diesem Fall also auf die Verwendung des Kleeneschen Sterns verzichten. Die Sternhöhe einer Sprache zu bestimmen, wenn man reguläre Ausdrücke mit Komplement zur Verfügung hat, ist bislang ungelöst. Auch ist unklar, ob überhaupt Sprachen mit einer Sternhöhe größer 1 existieren.

1.2.5 Das Pumping-Lemma für reguläre Sprachen

Mit dem Satz von Myhill-Nerode (vgl. Satz 1.21) haben wir schon ein notwendiges und hinreichendes Kriterium für die Regularität von Sprachen kennengelernt. Die Überprüfung dieses Kriteriums ist allerdings meist aufwendig. Das Pumping-Lemma für reguläre Sprachen liefert zwar nur ein notwendiges Kriterium, dieses ist jedoch in unserem Fall ausreichend und einfach überprüfbar.

Satz 1.24 (Pumping-Lemma für reguläre Sprachen)**.** *Ist L eine reguläre Sprache, dann existiert eine Zahl p, sodass für jedes Wort $w \in L$ mit $|w| \geqslant p$ Folgendes gilt:*
w kann in drei Teilwörter x, y, z zerlegt werden, also $w = xyz$, sodass

1. $|y| > 0$ $(y \neq \varepsilon)$,
2. $|xy| \leqslant p$ *und*
3. *für jedes* $i \geqslant 0$ $xy^iz \in L$ *ist.*

Übrigens ...

Die Regularität von endlichen Sprachen widerspricht dem Pumping-Lemma nicht.

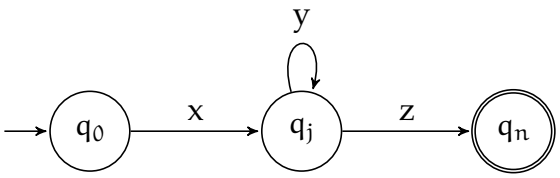


Abbildung 1.6: Prinzip des Pumping-Lemmas: Der endlicher Automat hat eine Schleife.

Idee. Wenn die Sprache regulär ist, gibt es einen endlichen Automaten, der sie akzeptiert. Sei dieser o. B. d. A. deterministisch (Satz 1.11, Seite 10) und bezeichne p die Anzahl seiner Zustände. Wir wählen nun ein Wort $s \in \Sigma^*$ mit $n = |s| \geqslant p$, das von ihm akzeptiert wird. Betrachten wir nun die Zustandsfolge $(q_i)(i \in \{0, \ldots, p-1\})$ der Länge n, die der Automat beim Akzeptieren durchläuft. Ein Zustand muss hier mehrfach durchlaufen werden (Schubfachprinzip). Ein Zustand q_i muss sich

$$\begin{array}{lccccccccc} s & = & & s_1 & & s_2 & & \cdots & & s_n \\ & & \uparrow & & \uparrow & & \uparrow & & \uparrow & & \uparrow \\ & & q_0 & & q_1 & & q_2 & & q_{p-2} & & q_{p-1} \in F \end{array} \qquad \begin{array}{l} (s_i \in \Sigma, n \geqslant p) \\ \\ \leftarrow \text{Schubfachprinzip} \end{array}$$

in dieser Folge wiederholen. Also enthält der Automat eine Schleife. Diese Schleife kann beliebig oft durchlaufen werden. Das Teilwort von s, das mit der Schleife im Automaten korrespondiert, kann an dieser Stelle also „aufgepumpt“ werden. □

Übrigens ...

Das *Schubfach-* oder *Taubenschlagprinzip* (*pigeonhole principle*) besagt: Verteilt man n Tauben auf m Fächer und ist $n > m$, dann gibt es mindestens ein Fach, das mehr als eine Taube enthält. Man kann sich leicht davon überzeugen, dass dies zutrifft.

Für die Regularität von Sprachen ist das Pumping-Lemma lediglich ein notwendiges, aber kein hinreichendes Kriterium. Es gibt also nicht-reguläre Sprachen, die es erfüllen. Entsprechend kann man es lediglich verwenden, um zu zeigen, dass eine Sprache *nicht* regulär ist. Dazu reicht es aber nicht, einen Widerspruch für ein bestimmtes, festes p zu finden, sondern für unendlich viele. Es genügt auch nicht, den Widerspruch nur für eine Zerlegung herzuleiten, er muss für alle möglichen Zerlegungen gefunden werden.

Beispiel 9.
$L = \{w \in \{0,1\}^* \mid w = 0^n 1^n, n \in \mathbb{N}\}$ ist nicht regulär. Beweis durch Widerspruch:
Annnahme: L sei regulär. Sei p die Pumping-Zahl. Dann gibt es für $w = 0^p 1^p$ eine Zerlegung $w = xyz$ mit $|y| > 0, |xy| \leqslant p$, sodass gilt: $xy^i z \in L$ für alle $i \geqslant 0$.
Aus $|xy| \leqslant p$ folgt notwendigerweise $y = 0^l$ und $x = 0^r$ mit $l > 0, r + l \leqslant p$.

Betrachte nun das Wort $w' = xy^2z$.
Es gilt: $w' = xy^2z = 0^r0^{2l}0^{p-(r+l)}1^p = 0^{r+2l+p-(r+l)}1^p = 0^{p+l}1^p \notin L$. Dies ist ein Widerspruch zur Annahme, L ist also nicht regulär.

Beispiel 10.
$L = \{ww \mid w \in \{0,1\}^*\}$
Wähle $w = 0^p10^p1 \in L$ und die Zerlegung $w = xyz$ wie im Pumping-Lemma, $|x| = r, |y| = l, l > 0, r + l \leqslant p$. Betrachte xy^2z:
$xy^2z = 0^r0^{2l}0^{p-(r+l)}10^p1 = 0^{r+2l+p-r-l}10^p1 = 0^{l+p}10^p1$. Da $l + p \neq p$ ist $xy^2z \notin L$.

Beispiel 11.
Widersprüche mithilfe des Pumping-Lemmas lassen sich nicht nur durch „Aufpumpen“ finden: Sei $L = \{0^i1^j \mid i,j \in \mathbb{N}, i > j\}$, p die Pumping-Zahl, $w = 0^{p+1}1^p$ und xyz eine Zerlegung von w wie im Pumping-Lemma. Dann ist $xy^0z = 0^{p+1-|y|}1^p \notin L$, da $|y| \geqslant 1$ nach Voraussetzung.

1.2.6 Abschlusseigenschaften

Satz 1.25 (Abschlusseigenschaften von regulären Sprachen)**.** *Es seien* L_1, L_2 *reguläre Sprachen. Dann sind auch*

1. *die Vereinigung* $L = L_1 \cup L_2$,
2. *der Durchschnitt* $L = L_1 \cap L_2$,
3. *das Komplement* $L = \overline{L_1}$,
4. *die Konkatenation* $L = L_1 \cdot L_2$ *und*
5. *der kleenesche Abschluss* $L = L_1^*$

regulär.

Beweis. Zur Übung. □

1.3 Kellerautomaten und kontextfreie Sprachen

Im letzten Kapitel wurde gezeigt, dass reguläre Sprachen und endliche Automaten gleich mächtig sind. Sie haben jedoch gewisse Beschränkungen: Die Sprache

$$L = \{w \mid w = 0^n1^n, n \in \mathbb{N}\}$$

ist nicht regulär (vgl. Beispiel 9) und lässt sich nicht durch einen endlichen Automaten akzeptieren. Umgangssprachlich könnte man diese Begrenzung folgendermaßen ausdrücken: Endliche Automaten können nicht zählen. Mit den kontextfreien Sprachen und den Kellerautomaten stehen uns mächtigere Konzepte zur Verfügung, die nicht dieser Begrenzung unterliegen.

1.3.1 Kontextfreie Sprachen

Im Vergleich zu regulären Grammatiken können bei kontextfreien Grammatiken unter anderem links- und rechtlineare Produktionen gleichzeitig vorkommen.

Definition 1.26 (Kontextfreie Grammatik)**.** *Eine kontextfreie Grammatik* G *ist ein 4-Tupel* $G = (\Sigma, V, S, P)$ *mit*

1. *dem endlichen Alphabet* Σ *der Terminalsymbole,*
2. *dem endlichen Alphabet* V *der Variablen mit* $V \cap \Sigma = \emptyset$*,*
3. *dem Startsymbol* $S \in V$ *und*
4. *der Menge von Produktionen* P *der Form* $u \to v$ *mit* $u \in V$ *und* $v \in (\Sigma \cup V)^*$*.*

Eine solche Grammatik wird auch Grammatik vom Chomsky-Typ 2 genannt.

Beispiel 12.
Eine kontextfreie Grammatik ist zum Beispiel $G = (\Sigma, V, \langle \text{Expr} \rangle, P)$ mit
$\Sigma = \{a, \times, +, (,)\}$, $V = \{\langle \text{Expr} \rangle, \langle \text{Term} \rangle, \langle \text{Factor} \rangle\}$ und

$$\begin{aligned} P = \{&\langle \text{Expr} \rangle \to \langle \text{Expr} \rangle + \langle \text{Term} \rangle \mid (\langle \text{Term} \rangle) \mid \langle \text{Term} \rangle, \\ &\langle \text{Term} \rangle \to \langle \text{Term} \rangle \times \langle \text{Factor} \rangle \mid \langle \text{Factor} \rangle, \\ &\langle \text{Factor} \rangle \to \langle \text{Expr} \rangle \mid a\} \end{aligned}$$

Eine Möglichkeit, die Anwendung der verschiedenen Regeln einer Grammatik zu einem bestimmten Wort darzustellen, ist der *Ableitungsbaum*. Seine Wurzel ist das Startsymbol. Kindknoten entstehen sukzessive als Produkte der Produktionen. Die Blätter des Baums sind Terminalsymbole. Abbildung 1.7 zeigt einen solchen Baum beispielhaft. Der Ableitungsbaum zu einem Wort muss nicht eindeutig sein.

1.3.2 Kellerautomaten / Pushdown Automata

Die kontextfreie Sprache $L = \{w \mid w = 0^n 1^n, n \in \mathbb{N}\}$ lässt sich, wie wir bereits wissen, nicht mit endlichen Automaten erkennen. Ein mächtigeres Maschinenmodell stellen die Kellerautomaten dar, die lesend auf das Eingabewort zugreifen können und weiterhin einen Kellerspeicher (Stack) besitzen, auf den lesend und schreibend zugegreiffen werden kann. Deterministische Kellerautomaten können L erkennen, die Sprache der Palindrome ($L' = \{wx\overline{w} \mid w \in \Sigma^*, x \in (\Sigma \cup \{\varepsilon\})\}$) kann nur nicht-deterministisch erkannt werden. Dies liegt daran, dass der Kellerautomat sonst erkennen müsste, wenn er das Wort w gelesen hat. Nichtdeterministische Kellerautomaten sind also echt mächtiger als deterministische.

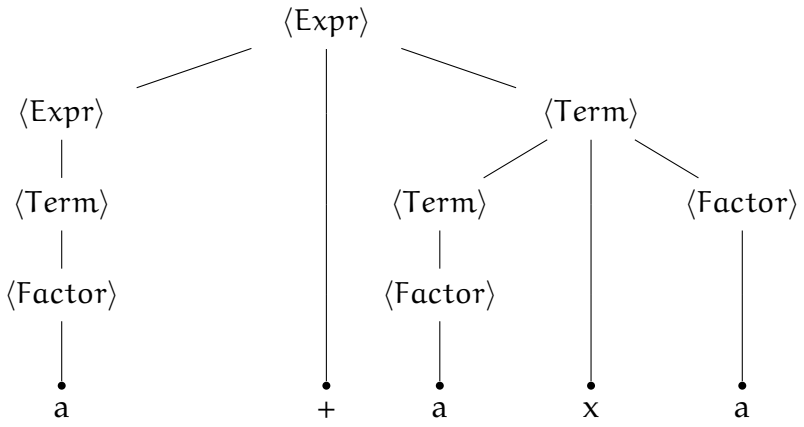


Abbildung 1.7: Ableitungsbaum für den Ausdruck $a + a \times a$. Wir legen für die Ableitung die Produktionsregeln aus Beispiel 12 zugrunde.

Information.

Ein *Stack*, auch *Stapel(-)* oder *Kellerspeicher* genannt, ist eine einfache Datenstruktur mit Zugriff nach dem *Last-in-first-out-Prinzip* (LIFO). Elemente können oben auf den Stack (*top of stack*) gelegt (*push*) oder herunter genommen (*pop*) werden, woraufhin sie gelesen werden können. Manche Implementierungen bieten auch eine Funktion, die das Lesen des obersten Elements (*peek*) erlaubt, ohne dass es vom Stack gepoppt und wieder gepusht werden muss.

Definition 1.27 (Kellerautomat)**.** *Ein (nichtdeterministischer) Kellerautomat (PDA, (N)PDA) ist ein 6-Tupel* $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$ *mit*

- *der endlichen Zustandsmenge* Q,
- *dem endlichen Eingabealphabet* Σ,
- *dem endlichen Stack-Alphabet* Γ,
- *der Zustandsüberganfunktion* $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) \to 2^{(Q \times \Gamma \cup \{\varepsilon\})}$,
- *dem Anfangszustand* $q_0 \in Q$ *und*
- *der Menge der akzeptierenden Zustände* $F \subseteq Q$.

Ein PDA ist genau dann deterministisch, wenn $|\delta(q, a, b)| + |\delta(q, \varepsilon, b)| \leqslant 1$ für alle $q \in Q$, $a \in \Sigma$ und $b \in \Gamma$ gilt.

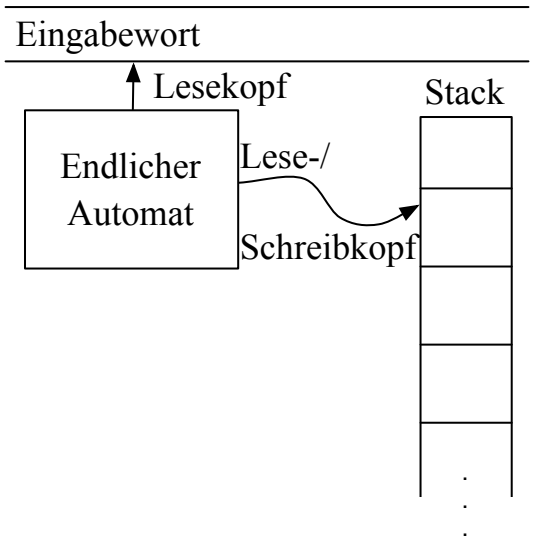


Abbildung 1.8: Ein Kellerautomat besteht aus einem Eingabewort w, das von links nach rechts abgearbeitet wird, einem Kellerspeicher, auf den lesend und schreibend zugegriffen werden kann, sowie einer Steuerung.

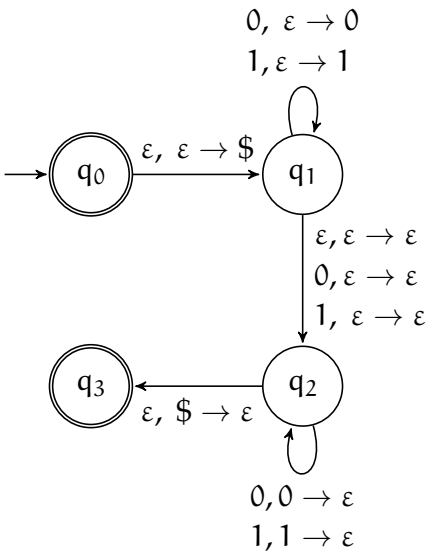


Abbildung 1.9: Ein Kellerautomat, der $L = wx\overline{w}, w \in \Sigma^*, x \in \Sigma \cup \{\varepsilon\}$ erkennt. Ein Zustandsübergang $a, b \to c$ bedeutet, dass bei Eingabe a das Zeichen b auf dem Stack durch c ersetzt wird. Um ein neues Zeichen c auf den Stack zu legen verwenden wir $a, \varepsilon \to c$, um das oberste Zeichen b zu entfernen $a, b \to \varepsilon$.

Abhängig vom Eingabe-Zeichen a, dem Symbol oben auf dem Stack γ und dem aktuellen Zustand q geht $\delta(q, a, \gamma) = (q', \gamma')$ in den Zustand q' über und ersetzt γ durch γ', wobei $a, \gamma \to \gamma' = \varepsilon$ erlaubt ist. Somit können Symbole auf den Stack gelegt oder von ihm heruntergenommen und Spontanübergänge realisiert werden.

Information.

Neben der Festlegung, dass ein Kellerautomat dann akzeptiert, wenn er in einen akzeptierenden Zustand gerät, gibt es noch zwei weitere mögliche Festlegungen:

- Ein PDA akzeptiert, sobald sein Stack leer ist.
- Ein PDA akzeptiert nur, wenn er in einen Haltezustand gerät *und* sein Stack leer ist.

Diese Akzeptanzverhalten sind aber äquivalent (Übung).

Der PDA akzeptiert $w = a_1 a_2 \dots a_m$ ($a_i \in (\Sigma \cup \{\varepsilon\})$), wenn es eine Abfolge von Zuständen $(r_0, r_1, \dots, r_m)$ und eine Abfolge von Wörtern $s_0, s_1, \dots, s_m \in \Gamma^*$ mit den folgenden Bedingungen gibt:

1. $r_0 = q_0$ und $s_0 = \varepsilon$.
2. Für $i = 0, \dots, m-1$ ist $(r_{i+1}, l) \in \delta(r_i, a_{i+1}, k)$ mit $s_i = kt$ und $s_{i+1} = lt$ für beliebige $k, l \in (\Gamma \cup \{\varepsilon\})$ und $t \in \Gamma^*$.
3. $r_m \in F$.

Abbildung 1.9 zeigt ein Beispiel für einen Kellerautomaten.

Satz 1.28. *Die kontextfreien Sprachen sind genau die von NPDAs erkannten Sprachen.*

Bevor wir diesen Satz beweisen, stellen wir fest, dass wir zu jedem PDA P einen PDA P' angeben können, der dieselbe Sprache erkennt und

1. der nur einen einzigen akzeptierenden Zustand q_{accept} hat,
2. dessen Stack leer ist, wenn er im Zustand q_{accept} ist und
3. dessen Zustandsübergänge alle entweder ein Zeichen pushen oder poppen.

Dies lässt sich erreichen, indem man

1. ein neues Zeichen „$\square$“ zum Stack-Alphabet hinzufügt,
2. einen neuen Finalzustand q_{accept} einführt und
3. jeden Übergang $\delta(q_i, a, S) \ni (q_j, T)$ mit $S, T \neq \varepsilon$ durch zwei Übergänge $\delta(q_i, a, S) \ni (q_{i,j,S,T}, \varepsilon)$ und $\delta(q_{i,j,a,S,T}, \varepsilon, \varepsilon) \ni (q_j, T)$ ersetzt. Dabei ist $q_{i,j,a,S,T}$ ein neuer Zustand ist. Jeder Zustandsübergang $\delta(q_i, a, \varepsilon) \ni (q_j, \varepsilon)$ wird durch zwei Übergänge $\delta(q_i, a, \varepsilon) \ni (q_{i,j,a}, R)$ und $\delta(q_{i,j,a}, \varepsilon, R) \ni (q_j, \varepsilon)$ ersetzt, wobei $q_{i,j,a}$ ein neuer Zustand und R ein neues Zeichen des Eingabealphabets Σ ist.

Beweis. „⇐“
Sei $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$ ein so konstruierter Automat. Wir definieren die kontextfreie Grammatik $G' = (\Sigma', V', S', P')$ mit $\Sigma' = \Sigma$, $V' = \{A_{pq} \mid p, q \in Q\}$, $S = A_{q_0 q_{accept}}$ und den Produktionen P', wobei P'

- für alle $p, q, r, s \in Q$, $t \in \Gamma$ und $a, b \in \Sigma \cup \{\varepsilon\}$ die Regel $A_{pq} \to aA_{rs}b$ genau dann enthält, wenn $\delta(p, a, \varepsilon) \ni (r, t)$ und $\delta(s, b, t) \ni (q, \varepsilon)$,
- für alle $p, q, r \in Q$ die Regel $A_{pq} \to A_{pr}A_{rq}$ und
- für alle $p \in Q$ die Regel $A_{pp} \to \varepsilon$ enthält.

Lemma 1.29. $A_{pq} \Rightarrow^* x$ $(x \in \Sigma^*)$ *gilt genau dann, wenn* P *bei Lesen von* x*, ausgehend von Zustand* p *und leerem Stack, den Zustand* q *erreichen kann, wobei bei Erreichen von* q *der Stack wieder leer ist.*

Beweis. Beide Richtungen der Behauptung lassen sich per Induktion zeigen:
„⇒“ Induktion über die Anzahl der Ableitungsschritte.

1. Induktionsanfang: Ist $k = 1$, so ist $x = \varepsilon$. Es findet also kein Zustandswechsel von P statt, weshalb sich der Stack nicht verändert. Dies entspricht der Produktion $A_{pp} \to \varepsilon$.
2. Induktionsannahme: Für alle Ableitungen $A_{pq} \Rightarrow^* x$ mit k Ableitungsschritten gilt Lemma 1.29.
3. Induktionsschritt: Ist $k > 1$, so ist die erste Regelanwendung der Ableitung entweder von der Form $A_{pq} \to aA_{rs}b$ oder $A_{pq} \to A_{pr}A_{rq}$.
   - Ist die erste Produktion $A_{pq} \to aA_{rs}b$, so ist $x = ayb$, wobei $A_{rs} \Rightarrow^* y$ eine Ableitung der Länge k ist. Nach Induktionsannahme kann P also bei Lesen von y mit leerem Stack von r nach s überführt werden. Nach Konstruktion der Produktion $A_{pq} \to aA_{rs}b$ gilt aber, dass es ein $t \in \Gamma$ gibt, sodass $\delta(p, a, \varepsilon) \ni (r, t)$ und $\delta(s, b, t) \ni (q, \varepsilon)$. Also erhält die Ableitung $A_{pq} \Rightarrow^* x$ den leeren Stack.
   - Ist die erste Produktion $A_{pq} \to A_{pr}A_{rq}$, so ist $x = yz$ mit $A_{pr} \Rightarrow^* y$ und $A_{rq} \Rightarrow^* z$. Die beiden letzten Ableitungen haben jeweils Länge k, also trifft für sie die Induktionsannahme zu und es gilt, dass P bei Lesen von y mit leerem Stack von p nach r und bei Lesen von z mit leerem Stack von r nach q überführt werden kann. Insgesamt also kann P bei Lesen von $x = yz$ mit leerem Stack von p nach q überführt werden.

„⇐“
Induktion über die Anzahl der Zustandsübergänge. Weiterhin ist nach Konstruktion $k = 2n$ für ein $n \in \mathbb{N}_0$.

1. Induktionsanfang: Ist $k = 0$, dann bleibt P im Zustand p und es gilt notwendigerweise $x = \varepsilon$. Dazu passt die Produktion $A_{pp} \to \varepsilon \in P'$, also $A_{pp} \Rightarrow^* x$.

2. Induktionsannahme: Für das Erreichen von q von p in k Schritten beim Lesen von x gilt Lemma 1.29.

3. Induktionsschritt: Ist $k > 1$, so bleibt der Stack während der Berechnung von P entweder immer gefüllt oder er wird irgendwann zwischendurch leer.

   - Im ersten Fall (Stack wird zwischendurch nie leer) muss das beim ersten Zustandsübergang auf den Stack gepushte Zeichen t im letzten Schritt wieder entfernt werden. Zerlegen wir also $x = ayb$. Der Nachfolgezustand von p sei r, der vorletzte Zustand sei s. Es gibt also ein $t \in \Gamma$, sodass $\delta(p, a, \varepsilon) \ni (r, t)$ und $\delta(s, b, t) \ni (q, \varepsilon)$. Also gibt es auch eine Produktion $A_{pq} \to aA_{rs}b$ in P'. Da P bei Lesen von y mit leerem Stack in maximal k Schritten von r nach s gelangt, gilt nach Induktionsannahme $A_{rs} \Rightarrow^* y$, insgesamt also $A_{pq} \Rightarrow^* x$.
   - Im zweiten Fall (Stack wird zwischendurch leer) erreicht P irgendwann einen Zustand r mit leerem Stack. Liest P zwischen p und r das Wort y, so können wir $x = yz$ zerlegen. P erreicht also jeweils von p bei Lesen von y den Zustand r mit leerem Stack und von Zustand r aus bei Lesen von z den Zustand q mit leerem Stack. Es gilt also $A_{pr} \Rightarrow^* y$ und $A_{rq} \Rightarrow^* z$, wobei die beiden Teilableitungen mit jeweils k Berechnungsschritten von P durchgeführt werden und somit die Induktionsannahme gilt. Da es in P' die Produktion $A_{pq} \to A_{pr}A_{rq}$ gibt, gilt insgesamt $A_{pq} \Rightarrow^* x$.

□

„⇒“
Für eine kontextfreie Grammatik $G = (\Sigma, V, S, P)$ mit $L = L(G)$ konstruieren wir einen NPDA, der genau L erkennt, indem für $w \in L$ überprüft wird, ob es eine Folge von Ableitungen gibt, für die $S \Rightarrow^* w$ gilt. Dazu wird in jedem Schritt nichtdeterministisch die „richtige“ Produktion gewählt. Zur Einfachheit erlauben wir, beliebig lange Wörter in einem Schritt auf den Stack zu pushen. Dies erweitert nicht die Mächtigkeit des PDA.

Information.

Beim Angeben von Zustandsübergängen mit Wörtern existiert folgende Konvention: Der Übergang $\delta(q_i, a, xyz) \ni (q_j, xyza)$ bedeutet, dass das Zeichen a durch das Wort $xyza$ ersetzt wird, wobei oben auf dem Stapel das Zeichen x liegt.

Der so entstehende PDA arbeitet wie folgt:

1. Lege das Trennzeichen \$ und das Startsymbol S auf den Stack.
2. Wiederhole
   (a) Liegt oben auf dem Stack ein Variablensymbol $A \in V$, wähle nichtdeterministisch eine Produktion $A \to v \in P$ und ersetze A auf dem Stack durch v.
   (b) Liegt oben auf dem Stack ein Terminalsymbol $a \in \Sigma$, lies das nächste Eingabesymbol a' und vergleiche es mit a. Wiederhole diesen Schritt, falls $a = a'$.
   (c) Liegt oben auf dem Stack das Trennzeichen \$, akzeptiere.

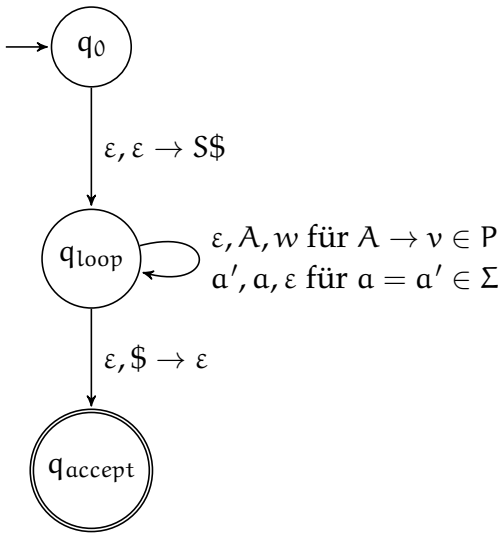


Abbildung 1.10: Schema des PDA aus dem Beweis für Satz 1.28.

□

1.3.3 Die Chomsky-Normalform

Um leichter algorithmisch mit kontextfreien Grammatiken arbeiten zu können, kann die Chomsky-Normalform benutzt werden.

Definition 1.30 (Chomsky-Normalform). *Eine Grammatik* $G = (\Sigma, V, S, P)$ *ist in Chomsky-Normalform, wenn jede Produktion der Form*

- $A \to BC$
- $A \to a$
- $S \to \varepsilon$, *falls* $\varepsilon \in L(G)$

ist, mit $A \in V$, $B, C \in V \setminus \{S\}$, $a \in \Sigma$.

Satz 1.31. *Jede kontextfreie Grammatik lässt sich in Chomsky-Normalform bringen.*

Beweis. Es sei $G = (\Sigma, V, S, P)$ eine Grammatik und $L = L(G)$ die von ihr erzeugte Sprache. Weiterhin seien $u_i \in (V \cup \Sigma)$ $(i \in \mathbb{N})$, $v \in (V \cup \Sigma)^+$ und $a \in \Sigma$.

1. Wir führen ein neues Startsymbol S_0 und die Produktion $S_0 \to S$ ein. Falls $\varepsilon \in L$, fügen wir die Produktion $S_0 \to \varepsilon$ hinzu.

2. Es sei V' die Menge aller Variablen, die auf ε abbilden. Für alle Produktionen der Form $A \to \varepsilon, A \neq S_0$ (ε-Produktionen) wird A zu V' hinzugenommen. Für alle Variablen in V' wird geprüft, ob durch ihr Ersetzen auf der rechten Seite von Produktionen neue Produktionen der Form $X \to \varepsilon$ entstehen. Diese Variablen X werden zu V' hinzugefügt, und das ganze solange wiederholt, bis keine neuen ε-Produktionen entstehen. Für jede Produktion, in der $A \in V'$ auf der rechten Seite vorkommt, fügen wir eine neue Produktion hinzu, bei der A gestrichen wurde. Jetzt können alle Produktionen $A \to \varepsilon, A \in V'$ entfernt werden.

   Beispiel: $R \to u_1Au_2Au_3$ liefert die neuen Produktionen $R \to u_1u_2Au_3, R \to u_1Au_2u_3, R \to u_1u_2u_3$.

3. Nun werden alle Produktionen der Form $A \to B$ entfernt. Für eine Regel $B \to v$ nehmen wir dazu eine Produktion $A \to v$ hinzu, es sei denn $A \to v$ wurde schon einmal entfernt.

4. Zuletzt werden alle verbliebenen Regeln in die Form $A \to BC$ und $A \to a$ überführt.
   $A \to u_1u_2 \dots u_j (j \geqslant 3)$ wird durch die Produktionen $A \to u_1A_1, A_1 \to u_2A_2, \dots, A_{j-2} \to u_{k-1}u_k$ ersetzt, wobei A_i neu eingeführte Variablen sind. Alle $u_k \in \Sigma$ aus den eben erstellten Regeln A_l werden durch U_k ersetzt und falls notwendig eine Produktion $U_k \to u_k$ hinzugefügt.

□

Information.

Dieser konstruktive Beweis lässt sich direkt als Umformungsalgorithmus nutzen.

Beispiel 13.
Sei $G = (\Sigma, V, S, P)$ eine kontextfreie Grammatik mit $\Sigma = \{a, b, c\}$, $V = \{S, A, B\}$ und

$$\begin{aligned} P = \{&S \to AB \mid AaA \mid A \mid BBB, \\ &A \to c \mid \varepsilon, \\ &B \to b\} \end{aligned}$$

1. Schritt: Einführen eines neuen Startsymbols. Wir stellen fest: $\varepsilon \in L(G)$.

$$\begin{aligned} P = \{&S_0 \to S \mid \varepsilon, \\ &S \to AB \mid AaA \mid A \mid BBB, \\ &A \to c \mid \varepsilon, \\ &B \to b\} \end{aligned}$$

2. Schritt: Entfernen der ε-Produktionen:

$$\begin{aligned} P = \{&S_0 \to S \mid \varepsilon, \\ &S \to AB \mid AaA \mid B \mid aA \mid Aa \mid a \mid A \mid BBB, \\ &A \to c, \\ &B \to b\} \end{aligned}$$

3. Schritt: Entfernen der Kettenregeln:

$$\begin{aligned} P = \{&S_0 \to AB \mid AaA \mid b \mid aA \mid Aa \mid a \mid c \mid BBB \mid \varepsilon, \\ &S \to AB \mid AaA \mid b \mid aA \mid Aa \mid a \mid c \mid BBB, \\ &A \to c, \\ &B \to b\} \end{aligned}$$

4. Schritt: Überführen in Chomsky-Normalform:

$$\begin{aligned} P = \{&S_0 \to AB \mid AS_1 \mid b \mid V_aA \mid AV_a \mid a \mid c \mid BS_2 \mid \varepsilon, \\ &S \to AB \mid AS_1 \mid b \mid V_aA \mid AV_a \mid a \mid c \mid BS_2, \\ &A \to c, \\ &B \to b, \\ &S_1 \to V_aA, \\ &S_2 \to BB, \\ &V_a \to a\} \end{aligned}$$

1.3.4 Der Cocke-Younger-Kasami-Algorithmus

Während das Wortproblem für reguläre Sprachen sehr einfach entschieden werden kann, ist dies für kontextfreie Sprachen nicht so einfach. Insbesondere wäre es denkbar, dass dies nicht effizient entscheidbar ist. Der Cocke-Younger-Kasami-Algorithmus gibt eine effiziente Möglichkeit an, das Wortproblem zu lösen.

Satz 1.32. *Für eine kontextfreie Grammatik $G = (\Sigma, V, S, P)$ in Chomsky-Normalform und ein Wort $w \in \Sigma^+$ mit $|w| = n$ entscheidet der Cocke-Younger-Kasami-Algorithmus (CYK-Algorithmus) in der Zeit $O(|P| \cdot n^3)$, ob $w \in L(G)$.*

Korollar 1.33. *Das Wortproblem für kontextfreie Sprachen lässt sich in Polynomialzeit lösen.*

Wir entscheiden das Wortproblem, indem wir berechnen, ob es für ein Wort $w = a_1 \dots a_n$, $a_i \in \Sigma$ eine Folge von Ableitungen ausgehend von S gibt, die w erzeugt. Dazu berechnen wir Mengen von Variablen aus $V_{ij} \subseteq V$ $(j \geqslant i)$, wobei V_{ij} alle Variablen enthält, aus denen sich das Teilwort $a_i \dots a_j$ (in mehreren Schritten) ableiten lässt, also $V_{ij} = \{A \in V \mid A \Rightarrow^* a_i \dots a_j\}$. Wir erstellen eine Tabelle (siehe Tabelle 1.1), in der wir mit aufsteigendem $l = j - i$ die V_{ij} eintragen.

V_{ij}	1	2	…	n
1	V_{11}	V_{12}	…	V_{1n}
2		V_{22}	…	V_{2n}
⋮			⋱	⋮
n				V_{nn}

Tabelle 1.1: Schema für den CYK-Algorithmus mit Zeilenindex i und Spaltenindex j.

Beweis. Ausgehend von $l = 0$ (also $i = j$) tragen wir in die Diagonaleinträge V_{kk} alle Produktionen ein, aus denen man a_k ableiten kann.
Für $1 \leqslant l < n$ gilt $A \Rightarrow^* a_i \dots a_j$ genau dann, wenn es ein k $(i \leqslant k < j)$ und eine Produktion $A \to BC$ mit $B \in V_{ik}, C \in V_{k+1,j}$ gibt (anschaulich bedeutet das, dass wir ausgehend von V_{ij} alle bereits ausgefüllten Tabelleneinträge links und unterhalb betrachten, die zusammen $l + 1$ Zellen von V_{ij} entfernt sind). Diesen Schritt führen wir für aufsteigendes l insgesamt $n - 1$-mal durch.
Die Berechnung betrachtet also für jedes V_{ij} $|P|$ viele Produktionen und höchstens $n - 1$ Werte für k. Es werden also $O(n^2)$ Mengen V_{ij} mit einer Laufzeitkomplexität von $O(|P| \cdot n)$ berechnet. Insgesamt beträgt die Laufzeitkomplexität also $O(|P| \cdot n^3)$.
Ist $S \in V_{1n}$, gibt es eine Folge von Ableitungen, die w erzeugt, und es ist $w \in L$. □

Beispiel 14.

$P = \{S \to AB \mid BC,$
$A \to BA \mid a,$
$B \to CC \mid b,$
$C \to AB \mid a\}$

V_{ij}	1	2	3	4	5	6
1	B	∅	A	S, C	B	A, S
2		B	A, S	S, C	B	A, S
3			A, C	S, C	B	A, S
4				B	A, S	∅
5					A, C	B
6						A, C

Abbildung 1.11: Beispiel: $w = bbabaa$.

1.3.5 Das Pumping-Lemma für kontextfreie Sprachen

Ebenso wie mit dem Pumping-Lemma für reguläre Sprachen kann mit dem Pumping-Lemma für kontextfreie Sprachen überprüft werden, ob es sich bei einer gegebenen Sprache um eine kontextfreie handelt. Analog zu regulären Sprachen wird nur ein notwendiges Kriterium geliefert, kein hinreichendes.

Satz 1.34 (Pumping-Lemma für kontextfreie Sprachen)**.** *Ist* L *eine kontextfreie Sprache, dann gibt es eine Zahl* p*, so dass für jedes Wort* $w \in L$ *mit* $|w| \geqslant p$ *mit der Zerlegung* $w = uvxyz$ *folgende Bedingungen gelten:*

1. *Für jedes* $i \geqslant 0$ *ist* $uv^ixy^iz \in L$*,*
2. $|vy| > 0$ *und*
3. $|vxy| \leqslant p$*.*

Beweis. Es sei $G = (\Sigma, V, S, P)$ eine kontextfreie Grammatik für die Sprache L und b die größte Anzahl von Symbolen auf der rechten Seite einer Produktion (bei Chomsky-Normalform ist $b \leqslant 2$). Also hat jeder Knoten im Ableitungsbaum höchstens b Kinder. Mit Höhe h hat der Baum höchstens b^h Blätter. Wir setzen $p = b^{|V|+1}$.

Sei nun $s \in L$ mit $|s| \geqslant p$ und τ ein Ableitungsbaum mit der kleinsten Anzahl von Knoten (wenn es mehrere gibt, wähle einen mit der geringsten Anzahl von Knoten). Wegen $|s| \geqslant p = b^{|V|+1}$ hat τ mindestens die Höhe $|V|+1$ und der längste Pfad in τ hat Länge $\geqslant |V|+1$. Wir betrachten diesen Pfad. Er enthält $\geqslant |V|+1$ Variablen (da höchstens ein Terminalsymbol auf dem Pfad ist). Wegen des Schubfachprinzips wiederholt sich mindestens eine Variable. Wenn es mehrere Wiederholungen gibt, betrachten wir im Folgenden die Wiederholung, die sich am weitesten unten im Baum befindet, also in den untersten $|V|+1$ Variablen.

Wir zerlegen s in $uvxyz$ gemäß folgender Abbildung:

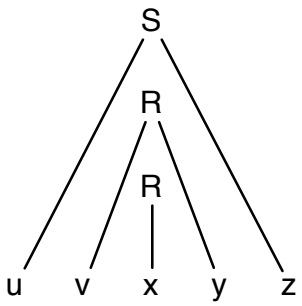


Abbildung 1.12: Ableitungsbaum für $s = uvxyz$. Ist $uvxyz \in L$, dann auch $uxz \in L$.

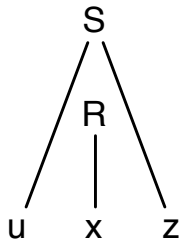


Abbildung 1.13: Statt aus dem oberen R das Teilwort vxy abzuleiten, könnten wir direkt x ableiten (wie es beim unteren R geschieht). Also ist auch $uxz \in L$.

Ebenso können wir aus dem unteren R statt x auch vxy ableiten, wie wir es bei dem oberen R tun, und es gilt $uv^2xy^2z \in L$. Durch Wiederholung gilt auch $uv^ixy^iz \in L\, \forall i \in \mathbb{N}$:

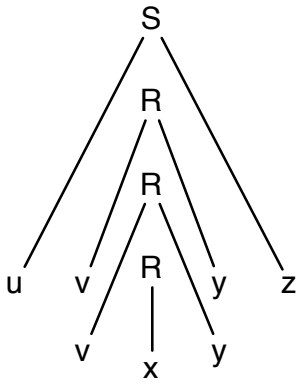


Abbildung 1.14: Ist $uvxyz \in L$, dann auch $uv^ixy^iz \in L$.

Es sind jedoch Bedingungen einzuhalten, damit unsere Argumentation richtig ist.

Wären v und y leer, würde im Ableitungsbaum $R \to R$ abgeleitet und es gäbe einen kleineren Ableitungsbaum, der s erzeugt. Das steht im Widerspruch zu der Wahl des Ableitungsbaums. Deshalb fordern wir $|vy| > 0$.

Betrachten wir weiterhin das Vorkommen von R in dem Ableitungsbaum. So wie wir R gewählt haben, sind beide Vorkommen in den unteren $|V| + 1$ Variablen auf dem Pfad. (Wir betrachten den längsten Pfad im Baum.) Der Teilbaum, in dem R das Teilwort vxy erzeugt, hat höchstens die Höhe $|V| + 1$. Ein Baum dieser Höhe erzeugt ein Wort der Länge $\leqslant b^{|V|+1} = p$. Deshalb fordern wir, dass $|vxy| \leqslant p$. □

Beispiel 15.
Annahme: $L = \{a^nb^nc^n \mid n \in \mathbb{N}\}$ ist kontextfrei. Dann gibt es ein $n \in \mathbb{N}$, sodass $|a^nb^nc^n| \geqslant p$. Für die Zerlegung $w = uvxyz, w \in L$ beobachtet man, dass $|v|$ und $|y|$ höchstens 1 sind, da sonst nicht einmal die Abschwächung $uv^nxy^nz \in \{a^*b^*c^*\}$ gelten würde. Insgesamt werden also höchstens zwei Symbole gepumpt, sodass sich für alle $n > 1$ ein Widerspruch ergibt.

1.3.6 Abschlusseigenschaften

Korollar 1.35. *Seien $i, j \in \mathbb{N}$. $L_1 = \{a^jb^ic^i\}$ und $L_2 = \{a^ib^ic^j\}$ sind kontextfrei (leichte Übung), der Schnitt $L_1 \cap L_2$ aber nicht (vgl. Beispiel 15). Die kontextfreien Sprachen sind also nicht bezüglich Durchschnitt abgeschlossen.*

Satz 1.36 (Abschlusseigenschaften von kontextfreien Sprachen)**.** *Es seien L_1, L_2 kontextfreie Sprachen. Dann sind auch*

1. *die Vereinigung $L = L_1 \cup L_2$,*
2. *die Konkatenation $L = L_1 \cdot L_2$ und*
3. *der kleenesche Abschluss $L = L_1^*$*

kontextfrei.

Beweis. Zur Übung. □

1.4 Turing-Maschinen und weitere Sprachen

1.4.1 Kontextsensitive Sprachen

Definition 1.37 (Kontextsensitive Grammatik). *Eine kontextsensitive Grammatik* G *ist ein 4-Tupel* $G = (\Sigma, V, S, P)$ *mit*

1. *dem endlichen Alphabet* Σ *der Terminalsymbole,*
2. *dem endlichen Alphabet* V *der Variablen mit* $V \cap \Sigma = \emptyset$*,*
3. *dem Startsymbol* $S \in V$ *und*
4. *der Menge von Produktionen* P *der Form* $u \to v$ *mit* $u \in V^+$, $v \in ((V \setminus \{S\}) \cup \Sigma)^+$ *und* $|u| \leqslant |v|$ *oder* $S \to \varepsilon$*.*

Eine solche Grammatik wird auch Grammatik vom Chomsky-Typ 1 genannt.

Information.

Für die Einschränkungen an die Produktionen der From $u \to v$ ist unter anderem folgende Variante möglich:

- $u \to v$ ist von der Form $\alpha A \beta \to \alpha \gamma \beta$ oder $S \to \varepsilon$, wobei $\alpha, \beta, \gamma \in ((V \setminus \{S\})^* \cup \Sigma)$ und $\gamma \neq \varepsilon$.

Man überzeugt sich leicht, dass diese Definition äquivalent zu der in Definition 1.37 ist.

Um später aus eine kontextsensitiven Grammatik G leichter einen Automaten zu konstruieren, der dieselbe Sprache erkennt, die G beschreibt, kann man sie wie folgt umformen.

Satz 1.38. *Zu jeder kontextsensitiven Grammatik* $G = (\Sigma, V, S, P)$ *gibt es eine sprachäquivalente Grammatik, deren Produktion der Form* $S \to a, A \to a$ *mit* $a \in \Sigma$ *(oder* $S \to \varepsilon$*) oder* $\alpha A \beta \to \alpha y \beta$ *mit* $\alpha, \beta, y \in V^*, y \neq \varepsilon, A \in V$*.*

$\alpha, \beta, y \in V^*$ *ist keine Einschränkung, da sonst in Produktionen alle* $\alpha \in \Sigma$ *durch eine neue Variable* A_a *ersetzt werden können und man die Produktion um* $A_a \to a$ *erweitern kann.*

Beweis. Ersetze alle Produktionen $A_1 \dots A_m \to B_1 \dots B_n$ aus $V^+ \times V^+$ (beachte, dass $m \leqslant n$) durch:

$$\begin{aligned}
A_1 \dots A_m &\to C_1 A_2 \dots A_m \\
C_1 A_2 \dots A_m &\to C_1 C_2 A_3 \dots A_m \\
&\vdots \\
C_1 \dots C_{m-1} A_m &\to C_1 \dots C_{m-1} B_m \dots B_n \\
C_1 \dots C_{m-1} B_m \dots B_n &\to C_1 \dots C_{m-2} B_{m-1} \dots B_n \\
&\vdots \\
C_1 B_2 \dots B_n &\to B_1 B_2 \dots B_n
\end{aligned}$$

Die Grammatik ist dann von der gewünschten Form und erzeugt dieselbe Sprache. □

1.4.2 Sprachen vom Chomsky-Typ 0

Falls keine Anforderungen an die Form der Produktionen gestellt werden, erhalten wir die allgemeinste Klasse von Grammatiken, die den Chomsky-Typ 0 charakterisieren.

Definition 1.39 (Grammatik vom Chomsky-Typ 0)**.** *Eine Grammatik* G *vom Chomsky-Typ 0 ist ein 4-Tupel* $G = (\Sigma, V, S, P)$ *mit*

1. *dem endlichen Alphabet* Σ *der Terminalsymbole,*
2. *dem endlichen Alphabet* V *der Variablen mit* $V \cap \Sigma = \emptyset$*,*
3. *dem Startsymbol* $S \in V$ *und*
4. *der Menge von Produktionen* $P \subseteq (V \cup \Sigma)^* V (V \cup \Sigma)^* \times (V \cup \Sigma)^*$.

Eine solche Grammatik wird auch semi-entscheidbar *oder* rekursiv aufzählbar *genannt.*

1.4.3 Turing-Maschinen

Das Maschinenmodell für Grammatiken vom Chomsky-Typ 0 und Chomsky-Typ 1 sind Turing-Maschinen, wobei diese, ähnlich den Produktionen, eingeschränkt sind (oder eben nicht).

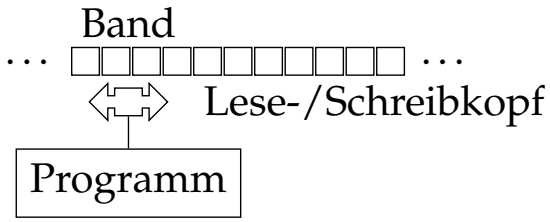


Abbildung 1.15: Skizze einer Turing-Maschine. Das Programm steuert einen Lese-/Schreibkopf auf einem unendlich langen Band.

Definition 1.40 (Turing-Maschine). *Eine Turing-Maschine ist ein 7-Tupel* $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ *mit*

1. *endlicher Zustandsmenge* Q,
2. *endlichem Eingabealphabet* Σ *ohne das Blank-Symbol* $\sqcup$,
3. *einem endlichen Bandalphabet* Γ *mit* $(\Sigma \cup \{\sqcup\}) \subseteq \Gamma$,
4. *einer Übergangsfunktion*
   (a) $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R, N\}$ *bei deterministischen Turing-Maschinen*,
   (b) $\delta : Q \times \Gamma \to 2^{(Q \times \Gamma \times \{L,R,N\})}$ *bei nichtdeterministischen Turing-Maschinen*,
5. *einem Startzustand* $q_0 \in Q$,
6. *einem akzeptierenden Zustand* $q_{accept} \in Q$ *und*
7. *einem ablehnenden Zustand* $q_{reject} \in Q$.

Information.

Eine Turing-Maschine kann auch mit einer akzeptierenden Zustands*menge* definiert werden, allerdings ist dies äquivalent zur obigen Definition. Alle akzeptierenden Zustände können per ε-Übergang auf einen akzeptierenden Zustand geleitet werden.

Beispiel 16.
Das Programm der Turing-Maschine in Abbildung 1.16 erkennt die Sprache $L = \{w \in \{a, b, c\}^* \mid |w|_a - |w|_c = |w|_b\}$.

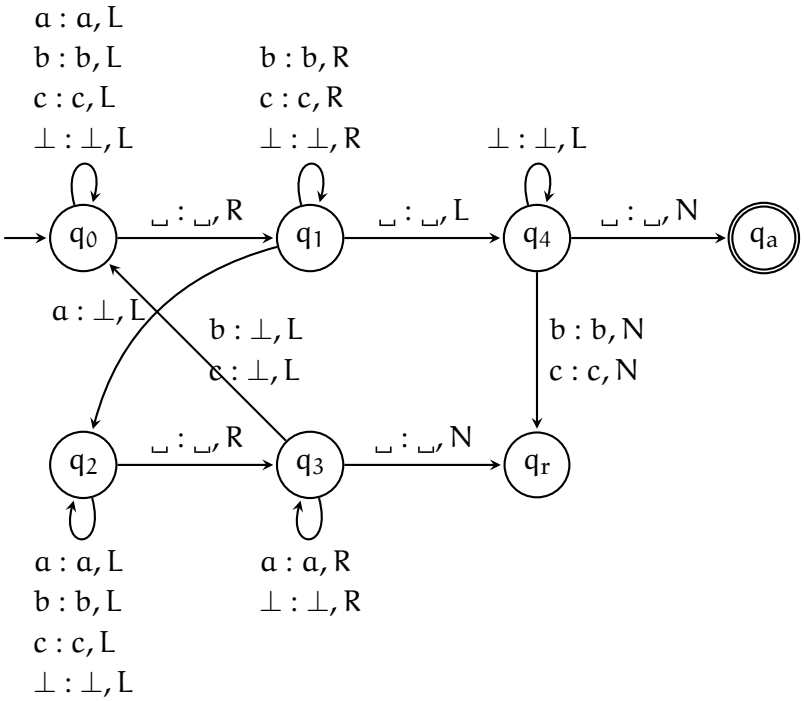


Abbildung 1.16: Automat einer Turing-Maschine. Bei einem Übergang $x : y, Z$ wird das Zeichen $x \in \Gamma$ durch $y \in \Gamma$ ersetzt und der Kopf in Richtung $Z \in \{L, R, N\}$ bewegt.

Satz 1.41. *Zu jeder nichtdeterministischen Turing-Maschine existiert eine äquivalente deterministische Turing-Maschine.*

Beweis. Zur Übung. □

Zu Beginn der Ausführung einer Turing-Maschine wird ihre Eingabe w auf das Band geschrieben und der Kopf auf dem ersten Zeichen positioniert. Die Berechnung einer Turing-Maschine lässt eine Folge von Konfigurationen (also aktueller Zustand, Position des Kopfs und Inhalt des Bandes) entstehen. Die Startkonfiguration einer Turing-Maschine ist $q_0 w$ mit $w \in \Sigma^*$, der Kopf der Turing-Maschine befindet sich also über dem ersten Symbol von w. Für eine Konfiguration $u a\, q_i\, b v$ führt $\delta(q_i, b) = (q_j, c, R)$ die Turing-Maschine also in den Zustand $u a c\, q_j\, v$ über.

Eine Berechnung ist also eine Folge von Konfigurationen. Eine Eingabe wird akzeptiert (abgelehnt), wenn der Zustand der letzten Konfiguration q_{accept} (q_{reject}) ist. In diesem Fall hält die Turing-Maschine.

Definition 1.42 (Ausgabe einer Turing-Maschine)**.** *Wir nennen den Bandinhalt der Turing-Maschine, nachdem diese gehalten hat, ihre* Ausgabe*. Hält eine Turing-Maschine M nach der Eingabe von w, bezeichnet $M(w)$ ihre Ausgabe.*

Definition 1.43 (Sprache einer Turing-Maschine)**.** *Die Sprache der von M akzeptierten Wörter nennen wir $L(M)$.*

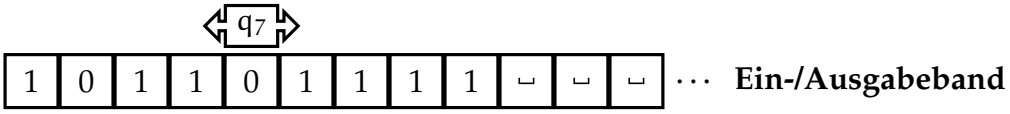


Abbildung 1.17: Eine Turing-Maschine in der Konfiguration $1011\, q_7\, 01111$.

Definition 1.44 (Linear beschränkte Turing-Maschine)**.** *Sei $k \in \mathbb{N}$ beliebig, aber fest. Eine linear beschränkte Turing-Maschine (LBA,* linear bounded automaton*) ist eine Turing-Maschine, die für eine Eingabe der Länge n nur $k \cdot n$ Zellen des Bands (inklusive der Eingabe) benutzen darf.*

Information.

In der Literatur sind auch folgende Definitionen geläufig:

- Ein LBA darf nur den Teil des Bandes benutzen, der durch die Eingabe beschrieben ist (oder höchstens noch das erste Blank-Symbol rechts neben der Eingabe).
- Ein LBA darf nur den Teil des Bandes benutzen, der durch die Eingabe beschrieben ist, sowie höchstens ein Zeichen rechts oder links davon.

Man überzeugt sich leicht davon, dass alle drei Definitionen zueinander äquivalent sind und durch eine Änderung des Bandalphabets ineinander überführt werden können.

Mysterium.

DSPACE(n) bezeichnet die Menge der Probleme, die von einem deterministischen LBA entschieden werden können. Analog ist NSPACE(n) die Menge der Probleme, die von einer nichtdeterministischen LBA entschieden werden können. Es ist nicht bekannt, ob DSPACE(n) = NSPACE(n).

Anders ausgedrückt: Es ist unbekannt, ob nichtdeterministische LBA mächtiger sind als deterministische.

Satz 1.45. *Die Sprachen, die von Grammatiken vom Chomsky-Typ 0 erzeugt werden, werden von Turing-Maschinen erkannt.*

Beweis. Übung. □

Satz 1.46. *Die von LBAs entschiedenen Sprachen sind genau die Sprachen vom Chomsky-Typ 1.*

Beweis. „⇐“
Konstruiere zu einer Grammatik $G = (\Sigma, V, S, P)$ vom Chomsky-Typ 1 einen LBA, die wie folgt arbeitet:

- M wählt nichtdeterministisch ein $\alpha \to \beta \in P$ mit $\beta \neq \varepsilon$.
- M sucht ein beliebiges (nichtdeterministisch das „richtige“) β auf dem Band und ersetzt β durch α.
- Falls $|\alpha| < |\beta|$ wird die Eingabe „zusammengeschoben“.
- $\vdots$
- Wenn das Band nur noch S enthält, geht M in den akzeptierenden Zustand.
- Ist das band leer und $\varepsilon \in L(G)$, geht M ebenfalls in den akzeptierenden Zustand. Offenbar ist M ein LBA und $L(M) = L(G)$.

„$\Rightarrow$“
Sei M ein LBA. Konstruiere eine kontextsensitive Grammatik G, die $L(M) \setminus N$ erzeugt, wobei $N = \{\alpha \in L(M) \mid |\alpha| \leqslant 1\}$

Konvention: $(\dots)$ stehe für ein Nichtterminalsymbol aus V, weiterhin seien $a, b, c, x, y \in \Sigma$.

1. Die Produktionen

$$\begin{aligned} S &\to A(a a_{\sqcup}) \\ A &\to A(aa) \mid (q_0 a a), (a \in \Sigma) \end{aligned}$$

   erzeugen $(q_0 a_1 a_1)(a_2 a_2)(a_3 a_3)\dots(a_n a_{n\sqcup})$ mit $n \geqslant 2$

   Dies entspricht allen möglichen Startkonfigurationen (die Symbolverdopplung ist eine Merkhilfe.)

2. Die Übergänge

   (a) $\delta(q_i, x) = (q_j, y, R)$

   (b) $\delta(q_i, x) = (q_j, y, L)$

   (c) $\delta(q_i, x) = (q_j, y, N)$

   werden durch folgende Produktionen simuliert:

   (a)

$$\begin{aligned} (q_i\, x a)(bc) &\to (y a)(q_j\, bc) \\ (q_i\, x a)(bc_{\sqcup}) &\to (y a)(q_j\, bc_{\sqcup}) \\ (q_i\, x a_{\sqcup}) &\to (y a\, q_j\, {\sqcup}) \end{aligned}$$

   (b) und c) analog.

3. Die Produktionen

$$\begin{aligned}(ab) &\to b\\ (ab\textvisiblespace) &\to b\\ (q_i\, ab) &\to b \text{ für } q_i \text{ akzeptierend}\\ (q_i\, ab\textvisiblespace) &\to b \text{ für } q_i \text{ akzeptierend}\\ (ab\, q_i\, \textvisiblespace) &\to b \text{ für } q_i \text{ akzeptierend}\end{aligned}$$

erzeugen α wenn $\alpha \in L(M)$

4. N ist endlich und daher insbesondere kontextsensitiv. Damit ist $(L(M) \setminus N) \cup N$ genau dann kontextsensitiv, wenn $L(M)$ kontextsensitiv ist.

□

1.4.4 Abschlusseigenschaften

Satz 1.47 (Abschlusseigenschaften von kontextsensitiven Sprachen)**.** *Es seien* L_1 *und* L_2 *kontextsensitive Sprachen. Dann sind auch*

1. *die Vereinigung* $L = L_1 \cup L_2$,
2. *der Durchschnitt* $L = L_1 \cap L_2$,
3. *das Komplement* $L = \overline{L_1}$,
4. *die Konkatenation* $L = L_1 \cdot L_2$ *und*
5. *der kleenesche Abschluss* $L = L_1^*$

kontextsensitiv.

Beweis. Zur Übung. □

Satz 1.48 (Abschlusseigenschaften von rekursiv aufzählbaren Sprachen)**.** *Es seien* L_1 *und* L_2 *Sprachen vom Chomsky-Typ 0. Dann sind auch*

1. *die Vereinigung* $L = L_1 \cup L_2$,
2. *der Durchschnitt* $L = L_1 \cap L_2$,
3. *die Konkatenation* $L = L_1 \cdot L_2$ *und*
4. *der kleenesche Abschluss* $L = L_1^*$

vom Chomsky-Typ 0.

Beweis. Zur Übung. □

1.5 Zusammenfassung

1.5.1 Die Chomsky-Hierarchie

Zusammenfassend kann aus den vorgestellten Sprachen eine Hierarchie erstellt werden, wobei wie in Abschnitt 1.2 beschrieben eine echte Teilmengenbeziehung besteht.

Definition 1.49 (Chomsky-Hierarchie)**.**

1. *Grammatiken ohne Einschränkungen heißen Grammatiken vom Chomsky-Typ 0 oder* rekursiv aufzählbar *oder* semi-entscheidbar*.*
2. *Grammatiken mit Produktionen* $u \to v$ *mit* $u \in V^+, v \in ((V \setminus \{S\}) \cup \Sigma)^+$ *und* $|u| \leqslant |v|$ *(oder* $S \to \varepsilon$*) heißen kontextsensitiv oder Grammatiken vom Chomsky-Typ 1.*
3. *Grammatiken mit Produktionen* $A \to v$ *mit* $A \in V$ *und* $v \in \{\Sigma \cup V\}^*$ *heißen kontextfrei oder Grammatiken vom Chomsky-Typ 2.*
4. *Grammatiken mit Produktionen* $A \to v$ *mit* $A \in V$ *und* $v = \varepsilon$ *oder* $v = a$ *oder* $v = aB$ *mit* $a \in \Sigma$ *und* $B \in V$ *heißen regulär oder Grammatiken vom Chomsky-Typ 3.*

Es gilt also: Chomsky-3 $\subset$ *Chomsky-2* $\subset$ *Chomsky-1* $\subset$ *Chomsky-0 (vgl. Abbildung 1.18).*

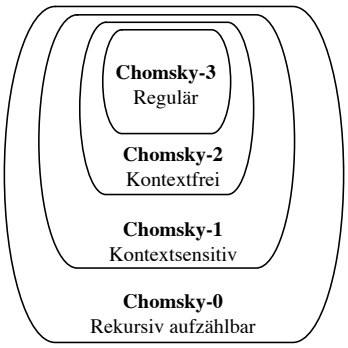


Abbildung 1.18: Die Chomsky-Hierarchie.

1.5.2 Überblick über Eigenschaften von Grammatiken

Im Folgenden sind alle Eigenschaften von Grammatiken, die in den vorangegangenen Kapiteln vorgestellt wurden, in einer Tabelle zusammengefasst:

	Chomsky-Typ 3	Chomsky-Typ 2	Chomsky-Typ 1	Chomsky-Typ 0
Name	regulär	kontextfrei	kontextsensitiv	rekursiv aufzählbar
Automat	DEA	PDA	LBA	Turing-Maschine
Entscheidbar	✓	✓	✓	-
Form der Produktionen	$A \to \nu$ mit $\nu = \varepsilon$, $\nu = a$ oder $\nu = aB$	$A \to (\Sigma \cup V)^*$	$u \to \nu$ mit $u \in V^+$, $\lvert u\rvert \leqslant \lvert\nu\rvert$, $\nu \in ((V \setminus \{S\}) \cup \Sigma)^+$ oder $S \to \varepsilon$	beliebig
Abgeschlossen unter $\cdot$	✓	✓	✓	✓
Abgeschlossen unter $^-$	✓	-	✓	-
Abgeschlossen unter $\cup$	✓	✓	✓	✓
Abgeschlossen unter $\cap$	✓	-	✓	✓
Abgeschlossen unter $*$	✓	✓	✓	✓

2. Berechenbarkeitstheorie

Nachdem wir mehrere Maschinenmodelle kennengelernt haben, stellt sich die Frage nach der Mächtigkeit dieser Modelle. Darüber macht die Church-Turing-These eine Aussage, von der es zwei Varianten gibt. Die *einfache Church-Turing-These* lautet:

Satz 2.1 (Einfache Church-Turing-These)**.** *Jedes intuitiv realisierbare Rechnermodell ist von einer Turing-Maschine simulierbar.*

Es wird allgemein angenommen, dass diese These gilt und die Gültigkeit dieser These rechtfertigt es, dass wir unsere Betrachtungen auf Turing-Maschinen beschränken. Es ist allerdings noch nichts über die Effizienz dieser Simulation gesagt. Das heißt vor allem, dass ein anderes Rechnermodell eventuell effizienter Probleme lösen könnte als die Turing-Maschine. Obwohl dies im Allgemeinen gut ist, würde es für die Kryptographie eine starke Einschränkung bedeuten. Das führt uns zur *erweiterten Church-Turing-These*, die besagt:

Satz 2.2 (Erweiterte Church-Turing-These)**.** *Jedes intuitiv realisierbare Rechnermodell ist in polynomieller Zeit von einer Turing-Maschine simulierbar.*

Seit dem Aufkommen der Idee von Quantencomputern gibt es allerdings Kritik an dieser These. Peter Shor konnte einen Algorithmus angeben, mit dem ein Quantencomputer effizient Zahlen faktorisieren kann. Dies ist mit heutigen Computern vermutlich nicht möglich (vgl. Kapitel 3.3.3). Daher wird allgemein angenommen, dass die erweiterte Church-Turing-These nicht gilt.

Mysterium.

Es ist nicht klar, ob Quantencomputer echt mächtiger als Turing-Maschinen sind. Ein Beispiel für diese Möglichkeit ist der bereits erwähnte Algorithmus zur Faktorisierung von Shor.

Hinweis.

Nachdem wir Automaten und Maschinen bisher immer ausführlich Zustand für Zustand definiert haben, werden wir in diesem Kapitel dazu übergehen, Maschinen und Programme informell über Beschreibungen ihrer Funktionsweisen anzugeben.

Definition 2.3 (Akzeptor). *Eine Turing-Maschine* $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, *die für alle* $w \in L(M)$ *aus der Startkonfiguration* $q_0 w$ *akzeptiert, heißt* Akzeptor. $L(M)$ *heißt* rekursiv aufzählbar *oder* semi-entscheidbar.

Die Klasse aller semi-entscheidbaren Sprachen bezeichnen wir mit RE *(für* recursively enumerable*).*

Analog setzen wir $co-RE = \{\overline{L} \mid L \in RE\}$. Bei einem Akzeptor ist das Verhalten für $w \notin L(M)$ nicht definiert: M kann entweder halten oder in einer Endlosschleife weiterrechnen.

Definition 2.4 (Entscheider).
Eine Turing-Maschine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, *die für alle* $w \in L(M)$ *aus der Startkonfiguration* $q_0 w$ *akzeptiert und für alle* $w \notin L(M)$ *hält, heißt* Entscheider. $L(M)$ *heißt* rekursiv *oder* entscheidbar.

Die Klasse aller entscheidbaren Sprachen bezeichnen wir mit R *(für* recursive*).*

2.1 Die Universelle Turing-Maschine

In diesem Abschnitt formalisieren wir eine eindeutige Beschreibung für das „Programm“ einer Turing-Maschine und definieren eine Turing-Maschine, die solche „Programme“ ausführen kann.

Es sei $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ eine Turing-Maschine und ohne Beschränkung der Allgemeinheit seien $\Sigma = \{X_0, \dots X_s\}$, $\Gamma = \Sigma \cup \{X_{s+1}, \dots, X_g\}$, $Q = \{q_0, \dots, q_t\}$ und o. B. d. A. seien q_0 der Anfangszustand und q_1 der einzige akzeptierende Zustand. Dann wird M durch δ vollständig beschrieben.

Sei $\delta(q_i, X_j) = (q_k, X_l, D_m)$ mit $D_1 = L, D_2 = N, D_3 = R$. Wir geben nun eine Codierung für $\delta(q_i, X_j) = (q_k, X_l, D_m)$ an: $Code_p = 0^i 1 0^j 1 0^k 1 0^l 1 0^m$.

Die gesamte Zustandsübergangstabelle wird nun wie folgt codiert:

$$111\, Code_1\, 11\, Code_2\, 11 \dots 11\, Code_t\, 111$$

Wir vereinbaren zusätzlich, dass jede nicht wohlgeformte Codierung für eine Turing-Maschine steht, die $\emptyset$ akzeptiert. Wir haben somit für jede Turing-Maschine eine natürliche Zahl (in Binärcodierung) angegeben. Jede Gödelnummer beschreibt genau eine Turing-Maschine.

Die Turing-Maschine M wird also binär nach $\mathbb{N}_0$ abgebildet.

Definition 2.5 (Gödelnummer). *Eine eindeutige Codierung einer Turing-Maschine nennen wir eine Gödelcodierung oder Gödelisierung. Die Codierung einer Turing-Maschine M heißt Gödelnummer und wird mit $\langle M \rangle$ bezeichnet.*

Information.

Manchmal wird die Turing-Maschine mit der Gödelnummer w auch mit M_w bezeichnet. Es gilt dann $\langle M_w \rangle = w$.

Wir konstruieren nun eine 3-Band-Turing-Maschine U, die für eine gegebene Gödelnummer $\langle M \rangle$ und eine Eingabe w die Turing-Maschine der Nummer $\langle M \rangle = n$ auf w simuliert.

Zunächst überprüft die Maschine die syntaktische Korrektheit der Codierung. Die Eingabe w bleibt auf Band 1, die Gödelnummer $\langle M \rangle$ wird auf Band 2 kopiert. Der Zustand der simulierten Turing-Maschine wird auf Band 3 gehalten. U liest auf Band 1, was auch M lesen würde (das Symbol X_j), auf Band 3 steht q_i codiert als 0^i. Nun sucht U auf Band 2 den zu q_i gehörigen Block $110^i10^j1 \dots 1$. Gibt es diesen Block nicht, so hält U (genauso wie M).

Ist $i = 1$, wird die Eingabe akzeptiert. Eine Ausgabe wird auf Band 1 gegeben. Wird der Block gefunden, so ist auf Band 2 die Zustandsübergangsfunktion für (q_i, X_j) mit $\dots 0^k10^l10^m11$ codiert, die angibt, was M tun würde. Wir ersetzen den alten Zustand, codiert duch 0^i auf Band 3, durch den neuen Zustand, codiert durch 0^k, schreiben das Symbol X_l auf Band 1 und bewegen den Kopf von Band 1 in Richtung D_m.

Definition 2.6 (Universelle Turing-Maschine). *Die Turing-Maschine, die bei Eingabe der Gödelnummer $\langle M \rangle$ einer Turing-Maschine M und einer Eingabe w die Ausführung von M auf w simuliert, heißt* universelle Turing-Maschine. *Es gilt $U(\langle M \rangle, w) = M(w)$.*

2.2 Eigenschaften von Sprachen

2.2.1 Diagonalisierung

Die Diagonalisierung ist eine Beweismethode aus der Mathematik, die auch in der theoretischen Informatik Anwendung findet. Sie geht auf eine Methode von Cantor zurück:

Satz 2.7. *Die reellen Zahlen $\mathbb{R}$ sind echt mächtiger als $\mathbb{N}$.*

Beweis. Wir zeigen die Behauptung, das offene Intervall $]0, 1[$ sei echt mächtiger als $\mathbb{N}$. Da $]0, 1[\subset \mathbb{R}$, ist damit auch die Behauptung gezeigt. Dazu nehmen wir zum Widerspruch an, $]0, 1[$ sei abzählbar. Dann existierte eine Bijektion $\mathbb{N} \to]0, 1[$. Wir zeigen, dass keine surjektive Abbildung $\mathbb{N} \to \mathbb{R}$ existiert: Eine Abbildung $\mathbb{N} \to]0, 1[$ induziert eine Folge auf $]0, 1[$.

Betrachte also eine beliebige Folge (z_i) auf $]0,1[$ und deren Glieder (in g-adischer Entwicklung):

$$\begin{aligned} z_1 &= 0{,}a_{11}a_{12}a_{13}\dots \\ z_2 &= 0{,}a_{21}a_{22}a_{23}\dots \\ z_3 &= 0{,}a_{31}a_{32}a_{33}\dots \\ &\vdots \end{aligned}$$

Wir konstruieren eine neue Zahl $x = 0{,}x_1x_2x_3\dots \in]0,1[$ nach folgender Vorschrift:

$$x_i = \begin{cases} 4, & \text{wenn } a_{ii} = 5 \\ 5, & \text{sonst} \end{cases}$$

Damit ist $x_i \neq a_{ii}\,\forall i \in \mathbb{N}$. Also ist (z_i) nicht surjektiv auf $]0,1[$. Es gibt also keine surjektive Abbildung $\mathbb{N} \to \mathbb{R}$. Damit ist die Behauptung gezeigt. □

Satz 2.8 (Satz von Cantor)**.** *Für jede Menge M ist $P(M)$ echt mächtiger als M selbst.*

Wir führen den Beweis nach Halmos:

Beweis. Nehmen wir zum Widerspruch an, $P(M)$ sei gleich mächtig wie M. Dann gibt es eine surjektive Abbildung

$$\sigma : M \to P(M).$$

Damit definieren wir die Menge

$$H = \{x \in M \mid x \notin \sigma(x)\}.$$

Offensichtlich gilt $H \subseteq M$, also auch $H \in P(M)$. Aus der Surjektivität von σ folgt, dass es ein $h \in M$ mit $\sigma(h) = H$ gibt. Damit gilt:

$$\begin{aligned} h \in H &\iff h \notin \sigma(h) \\ &\iff h \notin H \lightning \end{aligned}$$

□

Korollar 2.9. *Es gibt Sprachen, die nicht entscheidbar sind.*

Beweis. Für jedes endliche Alphabet Σ ist Σ^* gleich mächtig wie $\mathbb{N}$: Sei o. B. d. A. $\Sigma = \{0,1,\dots,n\}$. Jedes $w \in \Sigma^*$ ist dann eine natürliche Zahl zur Basis $n+1$. Also hat $P(\Sigma^*)$ die gleiche Mächtigkeit wie $P(\mathbb{N})$. Da $P(\Sigma^*)$ die Menge aller Sprachen über Σ^* ist, folgt damit unmittelbar, dass die Menge aller Sprachen über Σ echt mächtiger ist als $\mathbb{N}$. Die Gödelnummer ist eine eindeutige natürliche Zahl. Es gibt also nur abzählbar viele Turing-Maschinen. Jede Turing-Maschine akzeptiert genau eine Sprache. Es gibt also Sprachen, die nicht entscheidbar sind. □

2.2.2 Nichtentscheidbarkeit

Der obige Beweis war nicht konstruktiv. Wir lernen nun eine Sprache kennen, die nicht entscheidbar ist.

Definition 2.10 (Diagonalsprache)**.** *Die Sprache*

$$L_D = \{w \in \Sigma^* \mid \textit{Die Turing-Maschine } M \textit{ mit } \langle M \rangle = w \textit{ akzeptiert } w \textit{ nicht.}\}$$

heißt Diagonalsprache.

Satz 2.11. *Die Diagonalsprache L_D ist nicht entscheidbar.*

Beweis. Sei L_D entscheidbar. Dann gibt es eine Turing-Maschine T, die L_D entscheidet. Sei $t = \langle T \rangle$ die Gödelnummer von T. Es gibt nun zwei Möglichkeiten:

Falls $t \in L_D$, würde t von T akzeptiert. Dies ist ein Widerspruch zur Definition von L_D.

Falls aber $t \notin L_D$, so würde es nicht von T akzeptiert. Dann würde aber nach Definition gerade $t \in L_D$ gelten. $\lightning$ □

Korollar 2.12. $\overline{L_D} = \Sigma^* \setminus L_D$ *ist semi-entscheidbar.*

Beweis. $\overline{L_D} = \{\langle M \rangle \in \Sigma^* \mid \langle M \rangle \in L(M)\}$. Definiere einen Akzeptor T, der für $\langle M \rangle$ folgendermaßen arbeitet:

1. Simuliere M auf der Eingabe $\langle M \rangle$.
2. Akzeptiere, wenn M akzeptiert.

□

Die Begriffe der Diagonalisierung und der Diagonalsprache gehen auf Cantors Beweismethode zurück. Wir ordnen die Wörter in Σ^* lexikographisch in eine Folge $w_0, w_1, \ldots$ und erstellen eine Tabelle. Nach rechts tragen wir $w_j = w_0, w_1, \ldots$ auf, nach unten M_i mit $\langle M_i \rangle = w_i$. In die Zellen der Tabelle schreiben wir eine 1, wenn $w_j \in L(M_i)$:

	w_0	w_1	w_2	w_3	$\ldots$
M_0	0	1	1	0	$\ldots$
M_1	1	1	0	0	$\ldots$
M_2	0	0	0	1	$\ldots$
M_0	1	1	0	1	$\ldots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Die Diagonalsprache besteht nun aus genau den w_i, für die in der Zelle (M_i, w_i), also auf der Diagonalen, eine 0 steht. Aus diesen erzeugen wir einen Widerspruch.

Definition 2.13 (Universelle Sprache). *Die universelle Sprache ist definiert durch*

$$A_{TM} = \{\langle M\rangle w \in \Sigma^* \mid \langle M\rangle \text{ beschreibt eine Turing-Maschine und } M \text{ akzeptiert } w.\}.$$

Satz 2.14. A_{TM} *ist nicht entscheidbar.*

Beweis. Sei A_{TM} entscheidbar und T der dazugehörige Entscheider. Dann gibt es einen Entscheider $\overline{T}$, der genau dann akzeptiert, wenn T nicht akzeptiert (und umgekehrt). Wendet man $\overline{T}$ auf die Eingabe $\langle M\rangle\langle M\rangle$ an, so könnte er L_D entscheiden. ↯ □

Definition 2.15 (Berechenbarkeit). *Sei Σ ein endliches Alphabet. Eine Funktion $f : \Sigma^* \to \Sigma^*$ heißt berechenbar, falls es eine Turing-Maschine M gibt, die für jede Eingabe $w \in \Sigma^*$ nach endlich vielen Schritten mit $f(w)$ als einzigem Bandinhalt hält.*

Eine solche Maschine unterscheidet sich von Akzeptoren und Entscheidern dadurch, dass die Ausgabe auf das Band codiert ist. So gut wie alle einfachen Operatoren wie $+, -, \times, \div$ sind auf diskreten Strukturen wie $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$ berechenbar.

2.2.3 Many-one-Reduktion

Definition 2.16 (Many-one-Reduzierbarkeit). *Es seien $A, B \subseteq \Sigma^*$ Sprachen. A ist auf B many-one-reduzierbar, kurz $A \leqslant_m B$, falls eine berechenbare Funktion $f : \Sigma^* \to \Sigma^*$ existiert, sodass für alle $w \in \Sigma^*$: $w \in A \Leftrightarrow f(w) \in B$.*

Man sagt informell, B ist schwieriger als A. f heißt Many-one-Reduktion von A auf B.

Wir betrachten also ein Wort a, reduzieren dieses auf ein Wort $b = f(a)$, prüfen, ob $b \in B$ ist und folgern daraus, ob auch $a \in A$ gilt.

Satz 2.17. *Sei $A \leqslant_m B$. Wenn B entscheidbar ist, dann auch A.*

Beweis. Es sei $A \leqslant_m B$ und B entscheidbar. Dann gibt es einen Entscheider T für B und eine Many-one-Reduktion f von A auf B. Wir konstruieren einen Entscheider T' für A.

Für die Eingabe w:

1. Berechne $f(w)$.
2. Simuliere T für die Eingabe $f(w)$ und akzeptiere genau dann, wenn T akzeptiert.

Offensichtlich gilt: Wenn $w \in A$, dann ist $f(w) \in B$, weil T' w akzeptiert, wenn T $f(w)$ akzeptiert. □

Korollar 2.18. *Ist $A \leqslant_m B$ und A nicht entscheidbar, so ist auch B nicht entscheidbar.*

Beweis. Zur Übung. □

2.2.4 Das Halteproblem

Problem 2.19 (Halteproblem). *Das Halteproblem ist definiert durch*

$$\texttt{HALT}_{\texttt{TM}} = \{\langle M\rangle w \mid M \text{ ist eine Turing-Maschine und } M \text{ hält auf der Eingabe } w.\}.$$

Information.

Die Definition des Halteproblems umfasst *alle* Turing-Maschinen und Eingaben. Diskutiert man eine konkrete Maschine mit einer konkreten Eingabe, spricht man von einer sogenannten *Instanz*.

Übrigens ...

Wenn wir davon sprechen, ein Problem sei nicht entscheidbar, bedeutet das, dass kein Entscheider für *alle* Instanzen existiert. Es kann aber durchaus Entscheider für bestimmte Instanzen geben.

Satz 2.20. *Das Halteproblem* $\texttt{HALT}_{\texttt{TM}}$ *ist nicht entscheidbar.*

Beweis. Wir geben eine Many-one-Reduktion f von der universellen Sprache A_{TM} auf $\texttt{HALT}_{\texttt{TM}}$ an. Dazu sei $X = \langle M\rangle w$ eine Instanz von A_{TM}. Wir definieren nun $f(\langle M\rangle w) = \langle M'\rangle w$ und konstruieren die Turing-Maschine M' für die Eingabe v wie folgt:

1. Simuliere M mit Eingabe v.
2. Falls M v nicht akzeptiert, gehe in eine Endlosschleife.
3. Falls M v akzeptiert, akzeptiere.

Es ist klar, dass M' für die Eingabe w genau dann hält, wenn M die Eingabe w akzeptiert. Also ist f eine Many-one-Reduktion von A_{TM} auf $\texttt{HALT}_{\texttt{TM}}$. □

Satz 2.21. A_{TM} *und* $\texttt{HALT}_{\texttt{TM}}$ *sind semi-entscheidbar.*

Beweis. Wir zeigen die Semi-Entscheidbarkeit von A_{TM}. Dazu geben wir einen Akzeptor T für A_{TM} an. Wir konstruieren T für die Eingabe $\langle M\rangle w$:

1. Simuliere M mit Eingabe w.
2. Akzeptiere, wenn M akzeptiert.

Der Beweis für $\texttt{HALT}_{\texttt{TM}}$ ist zur Übung. □

Satz 2.22. *Es gilt* $R = RE \cap co - RE$. *Eine Sprache ist also genau dann entscheidbar, wenn sie und ihr Komplement semi-entscheidbar sind.*

Beweis. Es seien L und $\overline{L}$ Sprachen.
„$\subseteq$“: Folgt direkt aus der Definition.
„$\supseteq$“: Es sei T_1 ein Akzeptor für L und T_2 ein Akzeptor für $\overline{L}$. Wir konstruieren daraus einen Entscheider T für L: Simuliere parallel T_1 mit Eingabe w und T_2 mit Eingabe w.

1. Sobald T_1 akzeptiert, akzeptiere.
2. Sobald T_2 akzeptiert, lehne ab.

Da w entweder in L oder $\overline{L}$ liegt, wird w entweder von T_1 oder T_2 nach endlicher Zeit akzeptiert. Somit ist T ein Entscheider für L. □

2.2.5 Das Post'sche Korrespondenzproblem

Problem 2.23 (Post'sches Korrespondenzproblem)**.** *Gegeben sei eine endliche Menge von Puzzlestücken* $S = \left\{ \binom{t_1}{b_1} \dots \binom{t_n}{b_n} \right\}$ *mit* $t_1, \dots, t_n, b_1, \dots, b_n \in \Sigma^*$. *Gibt es Indizes* $i_1, \dots, i_k \in \{1, \dots, n\}$, *sodass* $t_{i_1} \dots t_{i_k} = b_{i_1} \dots b_{i_k}$ *gilt?*

Beispiel 17.
Sei $\Sigma = \{a, b, c\}$ und $S = \left\{ \binom{b}{ca}, \binom{a}{ab}, \binom{ca}{a}, \binom{abc}{c} \right\}$.
Eine Lösung für S ist $\left\{ \binom{a}{ab}, \binom{b}{ca}, \binom{ca}{a}, \binom{a}{ab}, \binom{abc}{c} \right\}$.

Satz 2.24. *Das Post'sche Korrespondenzproblem ist nicht entscheidbar.*

Beweis. Idee: Wir reduzieren das Wortproblem für Grammatiken vom Chomsky-Typ 0 auf das Post'sche Korrespondenzproblem.

Gegeben sei eine Grammatik $G = (\Sigma, V, S, P)$ vom Chomsky-Typ 0. Wir setzen $\Sigma' = \{t_1', \dots, t_n'\}$ mit $\Sigma \cap \Sigma' = \emptyset$ als das zu Σ gestrichte Terminalalphabet. Für ein Wort $w \in \Sigma^*$ sei $w' \in (\Sigma')^*$ das gestrichte Wort. Analog definieren wir die Menge der Variablen V'.

Wir setzen das Alphabet des PKP auf $\Sigma_{PKP} = \Sigma \cup \Sigma' \cup V \cup V' \cup \{*, *'\}$. Wir definieren nun die Puzzlestücke unseres PKPs: Für jede Produktion $\alpha \to \beta \in P$ führen wir folgende Puzzlestücke in S_{PKP} ein:

$$\binom{\beta'}{\alpha}, \binom{\beta}{\alpha'}.$$

Falls α oder β beispielsweise der Form aXb sind, werden sie zu $a'X'b'$. Für jedes $t \in \Sigma$ definieren wir die Puzzlestücke

$$\binom{t'}{t}, \binom{t}{t'}.$$

Nun benötigen wir nur noch Puzzlestücke für die Anfangs- und die Endbedingung, sowie für die Trennung der einzelnen Ableitungsschritte. Dabei ist S das Startsymbol der Grammatik und w das Wort, welches wir ableiten wollen:

$$\binom{S*'}{\varepsilon}, \binom{\varepsilon}{*w}, \binom{*'}{*}, \binom{*}{*'}.$$

Haben wir nun eine Ableitung

$$S \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \cdots \Rightarrow w_k \Rightarrow w,$$

so können wir diese in eine Lösung des PKPs übersetzen:

$$\binom{S*'}{\varepsilon}\binom{w_1'}{S}\binom{*}{*'}\binom{w_2}{w_1'}\binom{*'}{*}\cdots\binom{*'}{*}\binom{w_k'}{w_{k-1}}\binom{*}{*'}\binom{w}{w_k'}\binom{\varepsilon}{*w}.$$

Damit sieht man auch, dass eine solche Lösung des PKPs eine korrekte Ableitung von w impliziert. □

Beispiel 18.
Um die Technik zu veranschaulichen, betrachten wir die Grammatik $G = (\Sigma, V, S, P)$ mit $\Sigma = \{a, b, c\}$, $V = \{S, X\}$ und

$$P = \{S \to aXb,$$
$$X \to c\}.$$

Wir wollen das Wort acb ableiten. Daraus ergeben sich für das PKP folgende Puzzlestücke:

$$\binom{S*'}{\varepsilon}, \binom{\varepsilon}{*acb}, \binom{*'}{*}, \binom{*}{*'}, \binom{a'X'b'}{S}, \binom{aXb}{S'}, \binom{c'}{X}, \binom{c}{X'}, \binom{a'}{a}, \binom{a}{a'}, \binom{b'}{b},$$
$$\binom{b}{b'}, \binom{c'}{c}, \binom{c}{c'}.$$

Eine Ableitung von acb sieht dann aus wie folgt:

$$\binom{S*'}{\varepsilon}, \binom{a'X'b'}{S}, \binom{*}{*'}, \binom{a}{a'}, \binom{c}{X'}, \binom{b}{b'}, \binom{\varepsilon}{*acb}.$$

2.2.6 Der Satz von Rice

Satz 2.25 (Satz von Rice)**.** *Sei P die Menge aller Turing-berechenbaren Funktionen und S eine nichtleere Menge davon, also $\emptyset \neq S \subset P$. Dann ist die Menge*

$$C(S) = \{\langle M\rangle \mid M \text{ berechnet eine Funktion aus } S\}$$

nicht entscheidbar.

Eine Interpretation des Satzes von Rice ist: „Sei S eine nichttriviale Eigenschaft der von einer Turing-Maschine akzeptierten Sprache. Die Sprache $C(S) = \{\langle M\rangle \mid L(M)$ hat Eigenschaft $S\}$ ist nicht entscheidbar." Eine Eigenschaft ist nichttrivial, wenn sie nicht schon für alle Turing-Maschinen oder keine Turing-Maschine gilt. Beispiele sind die Programmäquivalenz ($M(x) = M'(x)$) oder die Zugehörigkeit zu bestimmten Funktionsklassen (z. B.: M berechnet eine Funktion, die die natürlichen Zahlen als Wertebereich hat).

Beweis. Wir nehmen zum Widerspruch an, $C(S)$ sei entscheidbar. Wir entscheiden die Universalsprache A_{TM}.

Da S nichttrivial ist, gibt es eine Turing-Maschine T mit $\langle T\rangle \in C(S)$.

Wir konstruieren die Turing-Maschine D für die Eingabe $\langle M\rangle w$:

1. Erzeuge eine Turing-Maschine M', die für die Eingabe x Folgendes berechnet:
   (a) Simuliere $M(w)$. Wenn sie ablehnt, lehne ab. Sonst:
   (b) Simuliere $T(x)$. Wenn sie akzeptiert, akzeptiere. Sonst lehne ab.
2. Entscheide, ob $\langle M'\rangle \in C(S)$. Wenn ja, akzeptiere. Sonst lehne ab.

M' simuliert T, wenn $M(w)$ akzeptiert. Dann ist $L(M') = L(T)$, sonst $L(M') = \emptyset$. Also ist genau dann $\langle M'\rangle \in C(S)$, wenn $M(w)$ akzeptiert. D entscheidet die universelle Sprache. $\lightning$ □

2.3 Kleene'sches Rekursionstheorem

Wagen wir einen philosophischen Ausflug: Dass Lebewesen sich reproduzieren können, akzeptiert man leicht. Beispielsweise trägt jedes Bakterium seinen „Code" in sich und kann sich durch Teilung reproduzieren. Nicht so leicht akzeptiert man wiederum die Vorstellung einer Maschine, die Maschinen erzeugen kann. Schließlich hat eine Maschine einen Bauplan und diesen Bauplan muss man kennen, um sie zu bauen. Die Komplexität einer Maschine, die eine andere Maschine bauen soll, muss also höher sein als die der zu konstruierenden Maschine, denn neben dem Bauplan müssen ja auch Routinen zum Konstruieren an sich und zum Selbsterhalt vorhanden sein. Und das bedeutet schlussendlich, dass Maschinen sich nicht selbst reproduzieren können, da sie ja ihren eigenen „Code" als Subroutine enthalten müssen. Es stellt sich heraus, dass diese Argumentation so nicht richtig ist:

- Es gibt Turing-Maschinen, die ihren eigene Beschreibung, also ihre Gödelnummer, ausgeben können.
- Es gibt Turing-Maschinen, die ihren eigenen Code verwenden können.

Lemma 2.26. *Es gibt eine berechenbare Funktion* $q : \Sigma^* \to \Sigma^*$*, die für jedes Wort* $w \in \Sigma^*$ *die Gödelnummer einer Maschine* P_w *berechnet, die* w *ausgibt und hält, also* $q(w) = \langle P_w \rangle$.

Beweis. Konstruiere eine Turing-Maschine Q, welche q berechnet. Für die Eingabe w:

1. Konstruiere die folgende Turing-Maschine P_w:

   P_w = Für jede beliebige Eingabe
   1. Lösche die Eingabe.
   2. Schreibe w auf das Band.
   3. Halte.

2. Gib $\langle P_w \rangle$ aus.

□

2.3.1 Die SELF-Maschine (ein Quine)

In diesem Abschnitt beantworten wir die Frage „Kann man ein Programm schreiben, das seinen eigenen Quelltext ausgibt?".

Die Antwort ist „ja". Solche Programme heißen *Quine*s. Im WWW findet man viele Beispiele für Quines in verschiedenen Programmiersprachen. Es gibt sogar ein „ZIP-Quine", das eine Besonderheit des LZW-Algorithmus ausnutzt. Wir erstellen hier eine Turing-Maschine, die ihre eigene Gödelnummer ausgibt. Die Konstruktion folgt dabei dieser Grundidee:

> Gib zwei Kopien des Folgenden exklusive Anführungszeichen aus, die zweite Kopie in Anführungszeichen:
> „Gib zwei Kopien des Folgenden aus, die zweite Kopie in Anführungszeichen:"

Wir konstruieren SELF in zwei Turing-Maschinen-Teilen A und B:

1. $A = P_{\langle B \rangle}$
2. B = Für die Eingabe $\langle M \rangle$
   (a) Berechne $q(\langle M \rangle)$.
   (b) Kombiniere das Ergebnis mit $\langle M \rangle$, um eine die Gödelnummer einer Turing-Maschine M' zu erstellen.
   (c) Gib $\langle M' \rangle$ aus und halte.

Zuerst muss natürlich B konstruiert werden. Die Beschreibung von B ist unabhängig von A und arbeitet mit einer beliebigen Eingabe. A wird um die Gödelnummer von B herum konstruiert. Wird nun A vor B ausgeführt, findet B seine eigene Gödelnummer als Eingabe vor.

Satz 2.27. SELF = AB *gibt* ⟨SELF⟩ *aus.*

Beweis.

1. Zuerst wird A ausgeführt und schreibt ⟨B⟩ auf das Band.
2. B startet und findet die Eingabe ⟨B⟩.
3. B berechnet $q(\langle B\rangle) = \langle A\rangle$ und kombiniert dies mit ⟨B⟩ zu $\langle AB\rangle = \langle SELF\rangle$.
4. B schreibt ⟨SELF⟩ auf das Band und hält.

Das Ergebnis ⟨AB⟩ ist die Gödelnummer einer Maschine, die sich selbst ausgibt. □

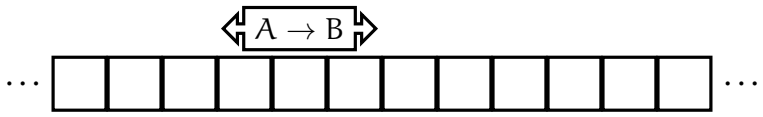


Abbildung 2.1: SELF-Maschine.

2.3.2 Das Rekursionstheorem

Das Rekursionstheorem wird es uns erlauben, Turing-Maschinen anzugeben, die ihre eigene Gödelnummer kennen. Wir sagen dann „ermittle die eigene Gödelnummer ⟨M⟩“. Auf diese Weise wäre das Programm SELF leichter zu konstruieren: „Ermittle die eigene Gödelnummer ⟨M⟩ und gib ⟨M⟩ aus“. Das Rekursionstheorem ist aber nicht nur für Spielereien nützlich, sondern allgemein ein nützliches Werkzeug für Beweise.

Information.

Die Aussage „Ermittle die eigene Gödelnummer ⟨M⟩“ ist irreführend: Die Turing-Maschine ermittelt nicht zur Laufzeit ihr eigenes Programm; dieses steht ihr vielmehr statisch zur Verfügung. Verändert man eine Turing-Maschine in einem Beweis, nachdem man das Rekursionstheorem verwendet hat, weicht die Gödelnummer, auf die die Maschine zugreift, von ihrer eigenen Gödelnummer ab. Die Aussage des Rekursionstheorems ist daher nicht mit den „Reflection“-Eigenschaften moderner Programmiersprachen zu verwechseln, mit deren Hilfe Programme bei der Laufzeitumgebung ihre eigenen Eigenschaften erfragen können.

Satz 2.28 (Rekursionstheorem). *Es sei* T *eine Turing-Maschine, welche die Funktion* $t : \Sigma^* \times \Sigma^* \to \Sigma^*$ *berechnet. Dann existiert eine Turing-Maschine* R*, welche die Funktion* $r : \Sigma^* \to \Sigma^*$ *mit*

$$r(w) = t(\langle R \rangle, w)$$

für alle $w \in \Sigma^*$ *berechnet.*

Das heißt: Für die Berechnung einer Funktion kann man eine Turing-Maschine finden, welche ihre eigene Gödelnummer mitverwenden kann.

T „weiß" nicht unbedingt, wie die eigene Codierung überhaupt aussieht. R wird so konstruiert (ähnlich zu SELF), dass diese Codierung zur Verfügung steht.

Definition 2.29 (Mehrband-Turing-Maschine). *Die Turing-Maschine* $P_x^{(i)}$ *schreibt* x *auf Band Nummer* i *einer Mehrband-Turing-Maschine.*

Beweis. T habe o. B. d. A. zwei Bänder (also zwei separate Register für die Eingabe), auf dem ersten Band stehe als Eingabe $\langle M \rangle$, auf dem zweiten w. Wir konstruieren die Turing-Maschine R aus den drei Programmteilen A, B, T (A und B seien auch 2-Band-Turing-Maschinen, T ist die gegebene Turing-Maschine), so dass $R = ABT$.

- $A = P^{(1)}_{\langle BT \rangle}$
- $B =$ Lies $\langle M \rangle$ vom ersten Band und berechne $q(\langle M \rangle) = \langle P^{(1)}_{\langle M \rangle} \rangle$. Hänge M an $P^{(1)}_{\langle M \rangle}$ an und schreibe $\langle P^{(1)}_{\langle M \rangle} M \rangle$ auf das erste Band. Übergib die Berechnung an T.

Setze $R = ABT$.

R berechnet $r(\langle R \rangle, w)$:

1. Nach Ausführung von A steht $\langle BT \rangle$ auf dem ersten Band.
2. Nach Ausführung von B steht $\langle P^{(1)}_{\langle BT \rangle} BT \rangle = \langle ABT \rangle = \langle R \rangle$ auf dem ersten Band.
3. T findet $\langle R \rangle$ auf dem ersten Band und w auf dem zweiten Band, gibt also $t(\langle R \rangle, w)$ aus.

□

Als Beispiel für den Nutzen des Rekursionstheorems beweisen wir Satz 2.14 von Seite 52 erneut:

Satz 2.30. A_{TM} *ist nicht entscheidbar.*

Beweis. Ist A_{TM} entscheidbar, existiert ein Entscheider T, der A_{TM} entscheidet. Wir konstruieren eine Turing-Maschine M für Eingabe w:

1. Ermittle die eigene Gödelnummer $\langle M \rangle$ (Rekursionstheorem).
2. Führe T auf $\langle M \rangle w$ aus.
3. Akzeptiere, wenn T ablehnt und umgekehrt.

M macht das Gegenteil von dem, was T über ihr Verhalten vorhersagt. Also ist T kein Entscheider für A_{TM}. □

Ein weiteres Beispiel für die Anwendung des Rekursionstheorems ist die „Minimalsprache“ und ihre Nichtentscheidbarkeit.

Definition 2.31 (Minimale Turing-Maschine)**.** *Sei M eine Turing-Maschine. Dann ist die Länge der Beschreibung von M die Anzahl der Zeichen von $\langle M \rangle$ (Notation: $|\langle M \rangle|$). Wir sagen, dass M minimal ist, wenn es keine funktionsäquivalente Turing-Maschine mit kürzerer Beschreibung gibt. Weiterhin sei*

$$MIN_{TM} = \{\langle M \rangle \mid M \text{ ist eine minimale Turing-Maschine.}\}$$

Satz 2.32. MIN_{TM} *ist nicht semi-entscheidbar.*

Beweis. Angenommen, MIN_{TM} wäre semi-entscheidbar. Dann gäbe es eine Turing-Maschine E, die MIN_{TM} aufzählt. Wir konstruieren eine Turing-Maschine C mit der Eingabe w:

1. Ermittle die eigene Gödelnummer $\langle C \rangle$ (Rekursionstheorem).
2. Lasse E so lange laufen, bis sie die Gödelnummer einer Turing-Maschine D mit $|\langle D \rangle| > |\langle C \rangle|$ ausgibt.
3. Simuliere D auf w.

MIN_{TM} ist unendlich (Übung), also existiert so ein D mit $|\langle D \rangle| > |\langle C \rangle|$. D ist minimal, da $D \in MIN_{TM}$. C simuliert aber die Ausführung von D. D war so gewählt, dass $|\langle C \rangle| < |\langle D \rangle|$. Also ist D nicht minimal. Das steht im Widerspruch zur Definition von MIN_{TM}. Also zählt E nicht MIN_{TM} auf. ↯ □

2.3.3 Das Rekursionstheorem (zweite Form)

Satz 2.33 (Rekursionstheorem, zweite Form). *Es sei* $t : \Sigma^* \to \Sigma^*$ *eine beliebige berechenbare Funktion. Dann gibt es eine Turing-Maschine* F*, für welche* $t(\langle F \rangle)$ *die Gödelnummer einer Turing-Maschine* G *ist, welche dieselbe Funktion wie* F *berechnet (*G *ist äquivalent zu* F*).*

Stellt ein Wort $w \in \Sigma^*$ keine gültige Turing-Maschine dar, wird im Folgenden angenommen, dass w eine Turing-Maschine, die die leere Sprache akzeptiert, repräsentiert.

Beweis (Skizze). Sei F die folgende Turing-Maschine für die Eingabe w:

1. Ermittle die eigene Gödelnummer $\langle F \rangle$ (Rekursionstheorem).
2. Berechne $\langle G \rangle = t(\langle F \rangle)$.
3. Simuliere G auf w.

Offensichtlich beschreiben $\langle F \rangle$ und $t(\langle F \rangle) = \langle G \rangle$ äquivalente Turing-Maschinen, da F G simuliert. □

Eine Interpretation der zweiten Form des Rekursionstheorems ist, dass es für jede Programmtransformation einen Fixpunkt gibt. (Ein Fixpunkt einer Funktion f ist ein Wert x, so dass $f(x) = x$.) Für jede Programmtransformation gibt es also (mindestens) ein Programm, das durch die Transformation nicht verändert wird.

2.4 Gödels Unvollständigkeitssatz

In den zwanziger Jahren des vorigen Jahrhunderts schlug der Mathematiker David Hilbert vor, die Mathematik als formales System neu zu formulieren (in dem sogenannten *Hilbertprogramm*). Ein Ziel war, außerhalb des gefundenen Kalküls seine Widerspruchsfreiheit zu zeigen. Außerdem sollte jede im Kalkül formulierbare Aussage als entweder wahr oder falsch bewiesen werden können. In der Folge stellte sich heraus, dass das unmöglich ist.

Satz 2.34 (Gödels Erster Unvollständigkeitssatz). *Jedes hinreichend mächtige formale System ist entweder widersprüchlich oder unvollständig.*

Wir werden später genauer spezifizieren, was hinreichend mächtig in diesem Fall bedeutet.

In diesem Abschnitt stellen wir Hilberts Programm „im Kleinen“ nach und zeigen einige interessante Aussagen dazu. Wir einigen uns zuerst auf eine formale Sprache. Wir möchten Aussagen wie beispielsweise $\forall a, b, c, n[(a, b, c > 0 \wedge n > 2) \to a^n + b^n \neq c^n]$ (Fermats großer Satz) beweisen oder widerlegen. Dazu müssen wir uns auf ein Alphabet festlegen:

$$\Sigma = \{\wedge, \vee, \neg, (,), \forall, \exists, x, R_1, \ldots, R_k\}$$

Σ besteht also aus *logischen Verknüpfungen*, *Klammern*, *Quantoren*, *Variablen* und *Relationssymbolen*. (Wir führen hier nur die Variable x an. Um mehrere Variablen zu erhalten, schreiben wir xx, xxx, und so weiter, welche für x_2, x_3, ... stehen.)

Definition 2.35 (Grundlagen Logik (1))**.**

- *Ein Term der Form* $R_i(x_1, \ldots, x_k)$ *heißt* atomare Formel*.*
- *Ein Term* ϕ *ist eine* Formel*, wenn*
  - *er eine atomare Formel ist,*
  - *er die Form* $\phi_1 \wedge \phi_2$, $\phi_1 \vee \phi_2$ *oder* $\neg\phi_1$ *hat, wobei* ϕ_1 *und* ϕ_2 *selbst Formeln sind, oder*
  - *er die Form* $\exists x_i : \phi_1$ *oder* $\forall x_i : \phi_1$ *hat, wobei* ϕ_1 *selbst eine Formel ist.*
- *Wir nehmen an, dass alle Formeln in Pränexform vorliegen, alle Quantoren also am Anfang einer Formel stehen.*
- *Eine Variable, die nicht durch einen Quantor gebunden ist, heißt* freie Variable*.*
- *Eine Formel, die keine freien Variablen enthält, heißt* Aussage*.*

Beispiel 19.
Zwei Formeln, die zweite ist eine Aussage:

1. $\forall x_1 : R_1(x_1) \wedge R_2(x_1, x_2, x_3)$
2. $\forall x_1 \exists x_2 \exists x_3 : R_1(x_1) \wedge R_2(x_1, x_2, x_3)$

Definition 2.36 (Grundlagen Logik (2))**.**

- *Variablen nehmen Werte über einem* Universum *an.*
- *Eine Zuordnung von Relationen zu Relationssymbolen zusammen mit einem Universum heißt* Struktur*. Ist* U *ein Universum und sind* $P_1, \ldots P_k$ *Relationen, schreiben wir* $\mathcal{S} = (U, P_1, \ldots, P_k)$*.*
- *Die* Sprache einer Struktur $\mathcal{S}$ *ist die Menge der Formeln, die nur die Relationen aus* $\mathcal{S}$ *(in richtiger Weise) verwendet.*
- *Die Menge der wahren Aussagen in der Sprache einer Struktur* $\mathcal{S}$ *nennen wir die* Theorie von $\mathcal{S}$ *oder* $\mathrm{Th}(\mathcal{S})$*.*

Beispiel 20 (Aussagen über $(\mathbb{N}, +, \times)$)**.**

- $\forall q \, \exists p \, \forall x, y : p > q \wedge (x, y > 1) \rightarrow x \cdot y \neq p$ (Es gibt unendlich viele Primzahlen.)
- $\forall a, b, c, n : (a, b, c > 0 \wedge n > 2) \rightarrow a^n + b^n \neq c^n$ (Großer Fermat'scher Satz)

In Gödels Erstem Unvollständigkeitssatz wurde ein hinreichend mächtiges formales System genannt. Im Folgenden werden wir sehen, dass ein formales System, welches die Zahlentheorie beschreiben kann, ausreicht. Wir beginnen mit einer abgeschwächten Form der Zahlentheorie, wo Multiplikation nicht berücksichtigt wird.

Satz 2.37. $\mathrm{Th}(\mathbb{N}, +)$ *ist entscheidbar.*

Beweis. Ohne Beweis. □

Diese Theorie ist entscheidbar, aber sobald die Multiplikation dazukommt, also alle Aussagen der Zahlentheorie abgebildet werden können, ist die Theorie nicht mehr entscheidbar.

Satz 2.38. $\mathrm{Th}(\mathbb{N}, +, \times)$ *ist unentscheidbar.*

Beweis. Idee: Wir reduzieren das Wortproblem von Turing-Maschine auf die Entscheidbarkeit von $\mathrm{Th}(\mathbb{N}, +, \times)$. Zu jeder Turing-Maschine M und jedem Wort w kann in $\mathrm{Th}(\mathbb{N}, +, \times)$ eine Formel $\phi_{M,w}$ mit freier Variable x erstellt werden, sodass die Aussage gilt

$$\exists x : \phi_{M,w} = 1 \Leftrightarrow M \text{ akzeptiert } w.$$

Da das Wortproblem für unbeschränkte Sprachen nicht entscheidbar ist, kann auch die Zahlentheorie $\mathrm{Th}(\mathbb{N}, +, \times)$ nicht entscheidbar sein. □

Um den nächsten Schritt zu Gödels Zweitem Unvollständigkeitssatz zu machen, müssen wir zuerst formal definieren, was ein Beweis ist.

Definition 2.39 (Beweis)**.** *Ein formaler Beweis π für die Aussage ϕ ist eine Folge von Aussagen $S_1, S_2, \ldots, S_l$ mit $S_l = \phi$, wobei sich jedes $S_i (i \geqslant 2)$ aus S_{i-1} unter Zuhilfenahme von grundlegenden Axiomen und logischen Regeln folgern lässt.*

Wichtige Mindesteigenschaften eines Beweises:

1. Die Korrektheit eines Beweises ist maschinell verifizierbar.
2. „Soundness“: Jede beweisbare Aussage ist wahr.

Satz 2.40. *Die Menge der beweisbaren Aussagen in* $\mathrm{Th}(\mathbb{N}, +, \times)$ *ist semi-entscheidbar.*

Beweis. Es sei P ein Algorithmus, der eine Eingabe ϕ akzeptiert, wenn es einen Beweis für ϕ gibt. P testet (unter Zuhilfenahme des „Beweisprüfers“ aus Punkt 1 der letzten Überlegung) für alle möglichen Wörter w, ob diese ein Beweis π für ϕ sind und hält, wenn es ein solches w gibt. Zähle alle Wörter auf und überprüfe, ob sie ein Beweis sind. Wenn ja, gib den Beweis aus. □

Satz 2.41 (Gödels Zweiter Unvollständigkeitssatz)**.** *Es gibt wahre Aussagen in* $\mathrm{Th}(\mathbb{N}, +, \times)$*, die nicht beweisbar sind.*

Beweis. Idee: Angenommen, jede wahre Aussage wäre beweisbar. Wir lassen also eine Maschine für eine Aussage ϕ „parallel“ testen, ob ϕ oder $\neg\phi$ einen Beweis hat. Diese Maschine kann dann aber $\mathrm{Th}(\mathbb{N}, +, \times)$ entscheiden, $\lightning$. Also gibt es wahre Aussagen $\phi \in \mathrm{Th}(\mathbb{N}, +, \times)$, die nicht beweisbar sind. □

Korollar 2.42. *Sind eine Sprache* L *und ihr Komplement semi-entscheidbar, so ist die Sprache entscheidbar.*

Beweis. Übung. □

Nachdem Satz 2.41 die Existenz von nicht beweisbaren Aussagen gezeigt hat, können wir nun ein konkretes Beispiel angeben.

Satz 2.43. *Die Aussage* $\psi = \neg\exists x : \phi_{M,0}$ *ist nicht beweisbar (Notation vgl. Beweis zu Satz 2.41).*

Beweis. Konstruiere eine Turing-Maschine M:

1. Ermittle die eigene Gödelnummer $\langle M \rangle$ *(Rekursionstheorem)*.
2. Konstruiere die Aussage $\psi = \neg\exists x : \phi_{M,0}$.
3. Führe den Algorithmus P aus Satz 2.40 auf φ aus.
4. Wenn dieser akzeptiert, akzeptiere. Wenn er hält und ablehnt, lehne ab.

□

ψ ist genau dann wahr, wenn M auf die Eingabe „0“ nicht akzeptiert. Wenn M jedoch einen Beweis für ψ findet, würde es die Eingabe „0“ akzeptieren und ψ wäre falsch. Da falsche Aussagen nicht bewiesen werden können, kann dieser Fall nicht auftreten. Deshalb kann nur der Fall eintreten, dass M keinen Beweis für ψ findet und „0“ nicht akzeptiert. Dann ist jedoch ψ wahr. $\lightning$

2.5 Turing-Reduktion

Definition 2.44 (Orakel-Turing-Maschine)**.** *Für eine Sprache* B *ist ein Orakel für* B *ein externes Gerät, das für eine Turing-Maschine* M *entscheidet, ob ein Wort* $w \in B$*. Die Turing-Maschine* M *mit Zugriff auf dieses Orakel nennen wir Orakel-Turing-Maschine und schreiben* M^O *für Orakel* O*.*

Wir interessieren uns dabei nicht für die Existenz oder Konstruktion des Orakels. Für unsere Zwecke reicht es, sich ein Orakel als „Glaskugel“ vorzustellen, die alle richtigen Antworten auf bestimmte Fragen kennt. Wir verwenden das Orakelkonzept, um Aussagen der Art „Wenn B leicht ist, dann ist es auch A“ zu machen. Wir zeigen dann, dass A leicht ist, indem wir ein Orakel für B verwenden, welches das Lösen von B tatsächlich sehr einfach macht: Wir müssen ja nur nach der Lösung fragen.

Definition 2.45 (Turing-Reduzierbarkeit)**.** *Wir schreiben* $A \leqslant_T B$*, wenn es eine Orakel-Turing-Maschine* M^O *gibt, die* A *entscheidet, wobei* O *ein Orakel für* B *ist. Wir sagen,* A *ist* relativ *zu* B *entscheidbar.*

Satz 2.46. *Ist* B *entscheidbar und* $A \leqslant_T B$*, so auch* A*.*

Beweis. Sei B entscheidbar. Dann existiert eine Turing-Maschine T_B, die B entscheidet. Da $A \leqslant_T B$, existiert eine Orakel-Turing-Maschine T_A^B, die A, unter Zuhilfenahme eines B-Orakels, entscheidet. Wir ersetzen das Orakel durch T_B. $T_A^{T_B}$ entscheidet A. Da T_B eine konkrete Berechnungsvorschrift ist, die man auch in eine andere Turing-Maschine „einsetzen“ kann, existiert eine Turing-Maschine T'_A, die A direkt entscheidet. □

3. Komplexitätstheorie

Kurt Gödel war optimistisch: Er glaubte, dass man Beweise zu Sätzen in einer Zeit finden kann, die in der Länge des Satzes „klein“ sind – also etwa quadratische Länge haben.

Mit dieser Vermutung begründete er die Frage nach P vs. $\mathcal{NP}$. Für eine deterministische Turing-Maschine M, die für alle Eingaben hält, bezeichnet die Funktion $f(n)$ die (Worst-Case-) Laufzeit, wenn f abbildet auf die maximale Anzahl von Berechnungsschritten für eine Eingabe der Länge n.

Hinweis.

Wenn nicht explizit angegeben, arbeiten Turing-Maschinen in diesem Kapitel – ohne Beschränkung der Allgemeinheit – auf dem Alphabet $\Sigma = \{0, 1\}$. Das Bandalphabet Γ enthalte weiterhin noch das Leerzeichen $\sqcup$ und das Trennzeichen #.

3.1 Der O-Kalkül

In der Informatik wird Komplexität meist *asymptotisch* betrachtet. Das heißt, die konkrete Anzahl von Berechnungsschritten zur Lösung eines konkreten Problems ist weniger interessant als das „Verhalten“ des Problems gegenüber verschiedenen Eingaben: „Wenn ich die Eingabelänge verdopple, verdoppelt sich dann auch die Laufzeit des Programms?“

Zur Vereinfachung der Darstellung zieht man Mengen von Funktionen heran:

Definition 3.1 (Die Funktionenklassen im O-Kalkül)**.**

$$
\begin{aligned}
O(f(n)) &= \{g(n) \mid \exists c > 0, n_0 \in \mathbb{N} \textit{ mit } 0 \leqslant g(n) \leqslant c \cdot f(n) \textit{ für } n \geqslant n_0\},\\
o(f(n)) &= \{g(n) \mid \forall c > 0\, \exists n_0 \in \mathbb{N} \textit{ mit } 0 \leqslant g(n) < c \cdot f(n) \textit{ für } n \geqslant n_0\},\\
\Omega(f(n)) &= \{g(n) \mid \exists c > 0, n_0 \in \mathbb{N} \textit{ mit } 0 \leqslant c \cdot f(n) \leqslant g(n) \textit{ für } n \geqslant n_0\},\\
\omega((f(n)) &= \{g(n) \mid \forall c > 0\, \exists n_0 \in \mathbb{N} \textit{ mit } 0 \leqslant c \cdot f(n) \leqslant g(n) \textit{ für } n \geqslant n_0\},\\
\Theta(f(n)) &= \{g(n) \mid g(n) \in O(f(n)) \textit{ und } g(n) \in \Omega(f(n))\}.
\end{aligned}
$$

Die Schreibweise $f(n) \in O(g(n))$ bedeutet dann, dass f höchstens so schnell wächst wie g, bis auf Konstanten. Statt der Bezeichnung „O-Kalkül“ hat sich auch die Bezeichnung *Landau-Notation* einbebürgert, da die Funktionenklassen auf den Mathematiker Edmund Landau zurückgehen.

Information.

Statt $g(n) \in O(f(n))$ ist auch die Schreibweise $g(n) = O(f(n))$ üblich. Diese ist formal eigentlich nicht korrekt, wie man schnell einsieht.

Die Landau-Notation ist aber nicht nur bei Programmlaufzeiten praktisch. Sie kann allgemein zur Beschreibung von Größenverhältnissen herangezogen werden.

Beispiel 21 (Das Verhältnis der Fläche eines Kreises zu seinem Umfang)**.**
Bezeichne r den Radius eines Kreises, U seinen Umfang und A seine Fläche. Das Verhältnis aus Umfang zu Fläche ist durch $4\pi A = U^2$ gegeben. A ist also zu U^2 proportional: $A \sim U^2$, oder $A \in \Theta(U^2)$ in Landau-Notation. (Θ ist aber schwächer als $\sim$, da $A \nsim (U^2 + 1) \in \Theta(U^2)$.)

Übrigens ...

Bezeichne z den Radius einer Pizza und a ihre Höhe, dann berechnet sich ihr Volumen durch $V = \pi \cdot z \cdot z \cdot a$.

3.2 Übersicht

Heutzutage ist es möglich, viele endliche Probleme durch Ausprobieren aller möglichen Lösungen („Brute Force“) zu lösen. Je nach Art und Größe eines Problems kann die dafür benötigte Zeit jedoch stark variieren und mitunter so groß sein, dass eine Berechnung nach menschlichen Zeitmaßstäben unmöglich ist. Die Komplexitätstheorie beschäftigt sich damit, den Zeit- und Platzbedarf von Problemen zu quantifizieren.

Auf den ersten Blick mag man verleitet sein, nur diejenigen Probleme als interessant zu betrachten, die eine geringe Komplexität besitzen und einfach lösbar sind. Jedoch finden schwierige Probleme insbesondere in der Kryptographie Anwendung: Sichere kryptographische Verfahren basieren auf Problemen, von denen

man ausgeht, dass sich eine Lösung nicht in Polynomialzeit berechnen lässt und deshalb zum korrekten Entschlüsseln der richtige Schlüssel notwendig ist.

In der Komplexitätstheorie unterscheidet man zwischen *Raum-* und *Zeitkomplexität*, also wie viel Platz (z. B. Zeichen auf dem Band einer Turing-Maschine) oder Zeit (z. B. Anzahl der Berechnungsschritte einer Turing-Maschine) benötigt wird. Diese Eigenschaften werden nicht für einzelne Probleminstanzen, sondern vielmehr asymptotisch in Abhängigkeit von der Eingabegröße des Problems angegeben.

Je größer die asymptotische Komplexität eines Problems ist, desto „einfacher“ lässt sich die Lösung durch die Erhöhung der Problemgröße erschweren. Der größte qualitative Unterschied wird im Allgemeinen zwischen polynomieller und superpolynomieller Komplexität gesehen.

Beispiel 22 (Multiplikation von natürlichen Zahlen)**.**
Ein Problem, zu dessen Lösung es Algorithmen mit unterschiedlicher Laufzeitkomplexität gibt, ist zum Beispiel das Problem, zwei ganze Zahlen zu multiplizieren. Hier ist es so, dass die Antwort auf die Frage, welcher dieser Algorithmen am wenigsten Zeit benötigt um ein solches Problem zu lösen, von der Eingabelänge abhängig ist. Für kleine Zahlen ist die Multiplikation, die man üblicherweise in der Grundschule lernt, ausreichend, ab einem bestimmten Punkt ist jedoch z. B. der Karatsuba-Algorithmus effizienter. Für wirklich sehr große Eingaben hingegen gibt es einen noch effizientere Algorithmen, beispielsweise den Schönhage-Strassen-Algorithmus. Auch wenn dieser für die meisten praktischen Anwendungen langsamer als die anderen ist, da die Zahlen nicht groß genug sind, so ist er trotzdem im Sinne der Komplexitätstheorie von diesen der beste Algorithmus.

Bei der Komplexität unterscheidet man beispielsweise zwischen folgenden Kriterien:

- *Worst Case*, also der Aufwand für die schwierigste Instanz.
- *Average Case*, der Mittelwert über alle Eingaben. Das ist interessant, wenn etwa fast alle Instanzen eines Problems effizient lösbar sind.
- *Best Case*, also „mindestens dieser Aufwand“.

Diese Vorlesung beschäftigt sich mit der Worst-Case-Komplexität und deterministischen Algorithmen. Weiterhin existieren noch probabilistische Algorithmen, bei denen manche Entscheidungen zufällig getroffen werden. Interessant sind im Wesentlichen zwei Komplexitätsklassen: P und $\mathcal{NP}$.

Definition 3.2 (Laufzeit einer deterministischen Turing-Maschine)**.** *Es sei* M *eine deterministische Turing-Maschine, die für alle Eingaben hält. Wir nennen*

$$f(n) = \max_{w \in \{0,1\}^n} \{k \mid M(w) \textit{ hält nach (höchstens) } k \textit{ Schritten.}\}$$

die Laufzeit *oder* Zeitkomplexität *von* M.

Wir betrachten die Laufzeit für eine Turing-Maschine also in der Regel nicht für eine konkrete Eingabe, sondern als das Maximum für alle Eingaben der Länge n.

Information.

Üblicherweise bezeichnet man in der theoretischen Informatik Problemgrößen oder Eingabelängen mit dem Buchstaben n.

Definition 3.3 (Klasse TIME). *Es sei* $t : \mathbb{N} \to \mathbb{N}$. *Die Komplexitätsklasse* $\mathrm{TIME}(t(n))$ *bezeichnet* $\mathrm{TIME}(t(n)) = \{L \mid L$ *ist entscheidbar durch eine Turing-Maschine, die bei Eingabelänge* n $O(t(n))$ *Schritte benötigt}.*

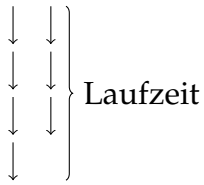
Für nichtdeterministische Turing-Maschinen definieren wir die Laufzeit wie folgt:

Definition 3.4 (Laufzeit einer nichtdeterministischen Turing-Maschine). *Es sei* M *eine nichtdeterministische Turing-Maschine, die für alle Eingaben hält. Wir nennen*

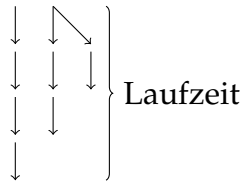
$$f(n) = \max_{w \in \{0,1\}^n} \{k \mid M(w) \textit{ hält auf jedem Berechnungspfad nach (höchstens)} \ k \textit{ Schritten.}\}$$

die Laufzeit *oder* Zeitkomplexität *von* M.

Wir betrachten also für eine feste Eingabe w mit $|w| = n$ alle möglichen Zustandsfolgen, die M abarbeiten kann. Die Folge mit der größten Anzahl Zustandsübergängen ist die Laufzeit für die Eingabe w. Das Maxiumum für alle Eingaben w' mit $|w'| = n$ ist die Laufzeit von M in Abhängigkeit von n.



(a) Laufzeit einer deterministischen Turing-Maschine: Die Länge der längsten Berechnungsfolge für alle Wörter der Länge n.



(b) Laufzeit einer nichtdeterministischen Turing-Maschine: Die Länge des längsten Pfads aller Bäume aller Berechnungsfolgen für alle Wörter der Länge n.

Information.

Man unterscheidet zwischen nichtdeterministischen und probabilistischen Turing-Maschinen (siehe Definition 3.49). Nichtdeterministische Turing-Maschinen haben eine ganze Menge von möglichen Berechnungspfaden. Sie haben aber keinen „Zufallsgenerator“. Bei probabilistischen Turing-Maschinen ist das anders: Ihr Programm kann von einer Zufallseingabe Gebrauch machen.

3.2.1 Speed-Up-Sätze

Wir betrachten die Sprache $L = \{0^k 1^k \mid k \in \mathbb{N}\}$. Eine Turing-Maschine, die für ein w mit $|w| = n$ entscheidet, ob $w \in L$, könnte folgendermaßen funktionieren:

1. Überprüfe, ob die Eingabe von der Form 0^*1^* ist.
2. Gehe bis zum ersten Blank-Symbol nach links, streiche eine Null. Gehe bis zum ersten Blank-Symbol nach rechts und streiche eine Eins.
3. Wiederhole Schritt 2, bis das Band leer ist und akzeptiere.

Dieser Algorithmus hat offensichtlich quadratische Laufzeit. Die Anzahl der Schritte lässt sich jedoch dadurch verringern, indem die Streichungen „blockweise“ vorgenommen werden.

Satz 3.5. *Zu jeder Turing-Maschine gibt es eine sprachäquivalente Turing-Maschine, die um einen konstanten Faktor schneller ist.*

Beweis. Idee: Man „verkompliziert“ die Steuerung der Turing-Maschine und vergrößert das Alphabet. So kann man (blockweise) konstant viele Schritte auf einmal machen. □

Es lässt sich folgender Algorithmus skizzieren, der eine Laufzeit von $O(n \cdot \log(n))$ besitzt:

1. Überprüfe, ob die Eingabe von der Form 0^*1^* ist.
2. Überprüfe, ob die Eingabe gerade Länge hat.
3. Solange das Band nicht leer ist:

   Streiche jede zweite Null.

   Streiche jede zweite Eins.
4. Akzeptiere bei leerem Band.

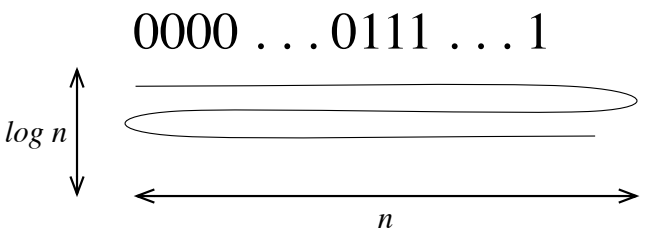


Abbildung 3.2: Arbeitsweise des Algorithmus mit Kompexlität $O(n \cdot \log(n))$.

Definition 3.6 (k-Band-Turing-Maschine). *Eine* k*-Band-Turing-Maschine verfügt über* k *verschiedene Arbeitsbänder, je mit eigenem Lese-/Schreibkopf. Ihre Übergangsfunktion ist*

$$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R, N\}^k.$$

Geht es noch besser? Wir betrachten folgende Algorithmenskizze für eine 2-Band-Turing-Maschine:

1. Kopiere die Eingabe auf das zweite Band.
2. Streiche Nullen von links auf Band 1, streiche Einsen von rechts auf Band 2.
3. Akzeptiere, wenn beide Lese-/Schreibköpfe sich in der Mitte treffen.

Diese Vorgehensweise hat eine nur lineare Laufzeitkomplexität.

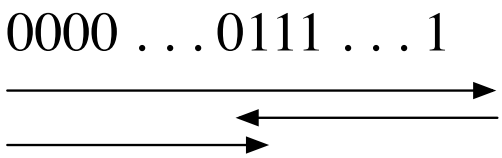


Abbildung 3.3: Laufzeitkomplexität einer Zwei-Band-Turing-Maschine: $O(n)$.

Der Aufwand einer Berechnung hängt also (auch) vom Berechnungsmodell ab.

Satz 3.7. *Es sei* $t(n) \geqslant n$ *eine Funktion. Dann lässt sich jede* $O(t(n))$-TIME-k*-Band-Turing-Maschine in eine* 1*-Band-Turing-Maschine übersetzen, die in* $O(t^2(n)) - \mathrm{TIME}$ *läuft.*

Beweisidee. Wir schreiben die Bandinhalte nebeneinander durch ein Trennzeichen getrennt auf das Band, ein Sonderzeichen markiert die i-te Kopfposition:

$$\#\#00101[\text{Kopf } 1]0011[\text{Trenn}]00 \ldots [\text{Kopf } i] \ldots \#$$

Für jeden einzelnen Schritt der k-Band-Turing-Maschine gehen wir einmal über das ganze Band. Wird dabei lediglich ein Zeichen durch ein anderes ersetzt, so lässt sich der Gesamtaufwand für die Simulation eines Schrittes durch $O(t(n))$ abschätzen. Muss ein Band verlängert oder verkürzt werden, müssen rechts davon stehende Bänder verschoben werden. Auch hier beträgt die Komplexität $O(t(n))$. Für jeden Schritt der k-Band-Turing-Maschine machen wir also $O(t(n))$ Schritte. Die Ursprungsmaschine macht $O(t(n))$ Schritte, die Simulation mit einem Band macht daher $O(t^2(n))$ Schritte. □

3.2.2 Die Klasse P

Die Komplexitätsklasse P umfasst alle Probleme, die wir als „hinreichend einfach“ ansehen.

Information.

Komplexitätsklassen sind Mengen von Sprachen. Das bedeutet, dass man, streng genommen, nur bei Entscheidungsproblemen über ihre Zugehörigkeit zu einer bestimmten Komplexitätsklasse sprechen kann. Such-, Optimierungs- und Optimalwertprobleme lassen sich jedoch auf Entscheidungsprobleme zurückführen.

Definition 3.8 (Klasse P)**.** P *ist die Menge der Sprachen, für die es eine deterministische Turing-Maschine gibt, die in höchstens* $p(|x|)$ *Schritten entscheiden kann, ob* $x \in L$*. Dabei ist* p *ein beliebiges Polynom.*

Also ist $\mathsf{P} = \bigcup_{k \in \mathbb{N}_0} \mathrm{TIME}(n^k)$.

Die Klasse P ist interessant, weil sie unter der Annahme, dass die erweiterte Church-Turing-These gilt, robust bei der Wahl des Berechnungsmodells ist. Dass ein Problem in P ist, sagt nichts darüber aus, ob es einen effizienten Algorithmus gibt, der das Problem löst. So würde ein Problem, für das es einen Algorithmus mit Laufzeitkomplexität $O(n^{17})$ gibt, zwar P angehören, für hinreichend große n aber praktisch nicht lösbar sein. In der Praxis sind Algorithmen mit einer Laufzeitkomplexität von $O(n^3)$ oftmals schon zu ineffizient.

Problem 3.9 (PRIMES)**.** $\mathtt{PRIMES} = \{n \in \mathbb{N} \mid n \textit{ ist eine Primzahl}\}$.

Satz 3.10. $\mathtt{PRIMES} \in \mathsf{P}$.

Beweis. Ohne Beweis. □

Beispiel 23 (Primzahlen)**.**
Es gilt $\mathtt{PRIMES} \in \mathsf{P}$. Es gibt also einen Algorithmus, der in polynomieller Laufzeit entscheiden kann, ob eine Zahl prim ist oder nicht. Ein solcher Algorithmus (der AKS-Primzahltest) wurde erst 2002 entwickelt. Für ein verwandtes Problem, nämlich das Faktorisieren einer Zahl (also die Darstellung als Produkt von Primzahlen) ist kein polynomieller Algorithmus bekannt. Man könnte verleitet sein, anzunehmen, dass für eine ungerade Zahl $n > 3$ die Probedivision mit allen Zahlen von 3 bis $\lfloor \sqrt{n} \rfloor$ einen Algorithmus mit Laufzeitkomplexität von $O(\sqrt{n})$ darstellt. Wir beschreiben jedoch mit n nicht eine konkrete Zahl, sondern vielmehr die *Länge* der Zahlen – von denen es $9 \cdot 10^{n-1}$ der Länge n gibt. Bei dieser Betrachtungsweise wächst die Anzahl der Zahlen, die kleiner oder gleich n sind, nicht mehr linear, sondern exponentiell.

Problem 3.11 (PATH)**.** $\mathtt{PATH} = \{(G, s, t) \mid G = (V, E) \textit{ ist ein gerichteter Graph und es gibt einen Pfad vom Knoten } s \textit{ zum Knoten } t\}$.

Beispiel 24.
Ein weiteres Problem aus der Kompexitätsklasse P ist PATH. Eine mögliche Beweisidee für $\mathtt{PATH} \in \mathsf{P}$ sieht folgendermaßen aus:

1. Markiere die Knoten, die von s aus in einem Schritt erreichbar sind.

2. Markiere die Knoten, die in einem Schritt von einem vorher markierten Knoten erreichbar sind, bis kein neuer Knoten markiert wurde.

Dieser Algorithmus betrachtet G $|V|$-mal.

Problem 3.12 (RELPRIME)**.** $\texttt{RELPRIME} = \{(x, y) \in \mathbb{N}^2 \mid x \textit{ und } y \textit{ sind teilerfremd}\}$.

Beispiel 25.
Ebenso ist das Problem $\texttt{RELPRIME} \in \mathrm{P}$: Berechne in polynomieller Laufzeit den größten gemeinsamen Teiler von x und y mit dem Euklidischen Algorithmus. Akzeptiere, wenn dessen Ergebnis 1 ist (x und y also keinen nichttrivialen gemeinsamen Teiler besitzen).

Information. Der Euklidische Algorithmus

Es seien $a, b \in \mathbb{N}$. Der größte gemeinsame Teiler von a und b lässt sich mit dem Euklidischen Algorithmus mit polynomieller Laufzeitkomplexität ($O(\log(\max(a, b)))$ Divisionen) folgendermaßen berechnen:

```
ggT(a,b) {
  if (b == 0) {
    return a;
  }
  else {
    return ggT(b, a mod b);
  }
}
```

Satz 3.13. *Alle kontextfreien Sprachen sind in* P.

Beweis. Siehe Beweis zu Satz 1.32. □

3.2.3 Die Klasse NP

Ebenso wie P ist NP eine Menge von Sprachen. Für NP gibt es viele Definitionen; hier sollen beispielhaft drei vorgestellt werden.

Definition 3.14 (NP 1. Variante)**.** NP *ist die Klasse aller Sprachen, die von nichtdeterministischen Turing-Maschinen in polynomieller Zeit akzeptiert werden.*

Das N von NP steht hier also für nichtdeterministische Turing-Maschinen. Wichtig ist, dass diese Definition nur Aussagen über die Laufzeit der Turing-Maschinen bei Akzeptanz macht.

3.2.3.1 Orakelmaschinen

Für die nächste Definition brauchen wir noch einen Begriff, den wir allerdings, da die Definition eher zur Veranschaulichung dient, nur informell formulieren wollen.

Definition 3.15 (Orakel-Turing-Maschine 2. Variante)**.** *Eine Orakel-Turing-Maschine ist eine Turing-Maschine, die Informationen von einem „Orakel“ beziehen kann. Das Orakel schreibt dabei zunächst auf einen separaten Bandabschnitt irgendeine Zeichenfolge endlicher Länge. Die Turing-Maschine kann nun die Ausgabe des Orakels einlesen und diese als Lösung von Problemen annehmen. Da die Ausgabe des Orakels nichtdeterministisch ist, muss die Turing-Maschine diese Lösung jedoch in der verfügbaren Rechenzeit überprüfen. Eine Orakel-Turing-Maschine akzeptiert eine Eingabe dabei, wenn es eine Antwort des Orakels gibt, mit der die Maschine akzeptiert.*

Hinweis.

Diese Definition einer Orakel-Turing-Maschine scheint zunächst widersprüchlich zu Definition 2.44. Dort leistet ein Orakel für eine Sprache B „Entscheidungshilfe“. Hier macht es einen Lösungsvorschlag, beziehungsweise liefert einen Zeugen (Definition 3.17). Der Sinn des Orakels ist aber bei beiden Definitionen der gleiche: Durch das Orakel wird ein bestimmtes Problem „einfach“. Man interessiert sich nun dafür, welche Konsequenzen das hat. Das Ziel hinter Definition 2.44 ist eine Hierarchie auf Entscheidungsproblemen. Hier interessiert uns allerdings die Frage, ob die Lösung eines Problems wenigstens einfach als solche zu erkennen ist, wenn man sie schon „fertig“ geliefert bekommt.

Damit kommen wir zur nächsten Definition:

Definition 3.16 ($\mathcal{NP}$ 2. Variante)**.** *$\mathcal{NP}$ ist die Menge der Sprachen, die in polynomieller Zeit von Orakel-Turing-Maschinen akzeptiert werden können.*

Hier ist der nichtdeterministische Anteil der ersten Definition durch das Orakel repräsentiert. Die letzte Definition schließlich, soll die sein mit der wir in Zukunft hauptsächlich arbeiten möchten.

Definition 3.17 ($\mathcal{NP}$ 3. Variante)**.** *Eine Sprache L ist genau dann in $\mathcal{NP}$, wenn es eine Relation $R_L \subseteq (L \times \{1,0\}^*)$ mit $R_L \in P$ gibt, wobei*

$$x \in L \iff \exists y : (x,y) \in R_L.$$

Weiterhin muss $|y|$ beschränkt sein durch $p(|x|)$, wobei p wieder ein beliebiges Polynom ist. Dabei heißt y Zeuge.

Das y, welches hier Zeuge genannt wird, ist dabei genau das, was das Orakel aus 3.15 errät.

Definition 3.18 (Klasse NTIME)**.** $\mathrm{NTIME}(t(n)) = \{L \mid L$ *wird in der Zeit* $O(t(n))$ *von einer nichtdeterministischen Turing-Maschine akzeptiert.)* $\}$
Damit ist $\mathcal{NP} = \bigcup_{k \in \mathbb{N}_0} \mathrm{NTIME}(n^k)$.

3.2.3.2 Beispiele

Problem 3.19 (HAMILTONKREIS). *Ein Hamilton-Kreis für einen Graphen* $G = (V, E)$ *ist eine Folge von Knoten* $v_1, \ldots, v_n$*, so dass gilt*

- *Jedes* $v \in V$ *kommt in der Folge* $v_1, \ldots, v_n$ *genau einmal vor,*
- *für* $i = 1, \ldots, n-1$ *ist* $(v_i, v_{i+1}) \in E$ *und*
- $(v_n, v_1) \in E$.

Das Hamilton-Kreis-Entscheidungsproblem HAMILTONKREIS *besteht darin, festzustellen, ob ein gegebener Graph* $G = (V, E)$ *einen Hamilton-Kreis besitzt.*

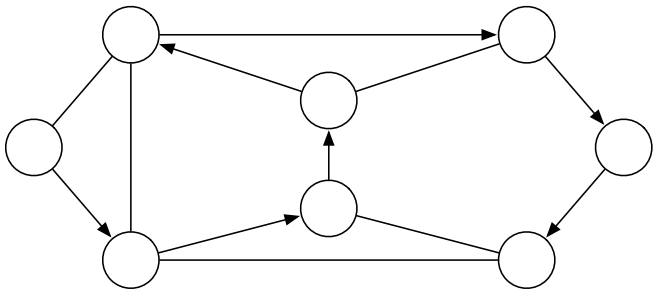


Abbildung 3.4: HAMILTONKREIS: Gibt eine eine Rundreise auf dem Graphen, die jeden Knoten genau einmal enthält?

Mysterium.

Ob HAMILTONKREIS $\in$ P, ist nicht bekannt.

Problem 3.20 (EULERKREIS). *Ein Eulerkreis für einen Graphen* $G = (V, E)$ *ist eine Folge von Knoten* $v_1, v_2, \ldots, v_m$*, sodass gilt*

- $v_1 = v_m$.
- *Für* $i = 1, \ldots, m-1$ *ist* $(v_i, v_{i+1}) \in E$*, und: jede Kante in* E *kommt genau einmal in dieser Folge von Kanten vor.*

Ein Eulerkreis ist also eine Rundreise durch einen Graphen, auf welcher jede Kante genau einmal auftaucht.

Satz 3.21. EULERKREIS $\in$ P.

Beweis. Übung. □

Problem 3.22 (COMPOSITE). COMPOSITE $= \{x \in \mathbb{N} \mid x = p \cdot q$ *für* $p, q \in \mathbb{N} \setminus \{1\}\}$.

Satz 3.23. COMPOSITE $\in$ P.

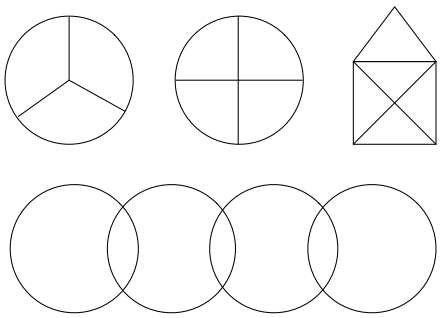


Abbildung 3.5: EULERKREIS: Gibt es eine Rundreise durch den Graphen, auf dem jede Kante genau einmal liegt? EULERKREIS liegt in P.

Beweis. Folgt direkt aus PRIMES $\in$ P. □

Problem 3.24 (SUBSET SUM)**.** *Gibt es für* $S = \{x_1, \ldots, x_n\}$ *und* $t \in \mathbb{N}_0$ *eine Menge* $\{y_1, \ldots y_l\} \subseteq S$ *mit* $\sum_{i=1}^{l} y_i = t$*?*

Hinweis.

Es gibt auch Definitionen von SUBSET SUM, die eine Gewichtsfunktion für die Elemente von S beinhalten.

3.2.3.3 coNP = NP?

Definition 3.25 (coNP)**.** *Es sei* L *eine Sprache.*

$$co\mathcal{NP} = \{\overline{L} \mid L \in \mathcal{NP}\}$$

ist die Klasse aller Sprachen, deren Komplement in NP *liegt.*

Information.

Es gilt *nicht* $\overline{\mathcal{NP}} = \text{co}\mathcal{NP}$! Vielmehr ist $\mathcal{NP} \cap \text{co}\mathcal{NP} \neq \emptyset$, da beispielsweise $P \subseteq \mathcal{NP}$ und $P \subseteq \text{co}\mathcal{NP}$. Offen ist, ob $\text{co}\mathcal{NP} = \mathcal{NP}$.

Es existiert ein Zusammenhang zu dem $P \stackrel{?}{=} \mathcal{NP}$-Problem: Sei L eine Sprache. Ist $L \in P$, so ist auch $\overline{L} \in P$. Daher gilt $P = \mathcal{NP} \Rightarrow \text{co}\mathcal{NP} = \mathcal{NP}$.

3.2.4 Ein Ausflug in andere Welten

1995 hat Russel Impagliazzo fünf verschiedene "'Welten"' vorgestellt, in denen wir leben könnten, abhängig davon, welche Antworten auf die noch ausstehenden Probleme sich als korrekt erweisen. Das hat vor allem Einfluss auf alle kryptographischen Ergebnisse. Diese Welten sind:

1. *Algorithmica*: Es gilt $P = \mathcal{NP}$.
   In dieser Welt gäbe es keine sicheren kryptographischen Verfahren, bei denen die Schlüssellänge kürzer als die Nachricht ist. Es blieben nur Verfahren wie das One-Time-Pad (siehe Definition 4.5.2.1).

2. *Heuristica*: Es gilt $P \neq \mathcal{NP}$, aber jedes $\mathcal{NP}$-vollständige Problem ist im „average case“ in polynomieller Zeit lösbar.
   Eine Folge wäre, dass das Finden schwieriger Probleme selbst ein schwieriges Problem ist, insbesondere also auch das Finden effizienter und sichere kryptographischer Verfahren.

3. *Pessiland*:

   $\mathcal{NP}$-Probleme sind „hard on average” (und damit ist $P \neq \mathcal{NP}$), es gibt aber keine Einwegfunktionen. (Einwegfunktionen sind noch zu definieren und stellen eine Voraussetzung für die meisten kryptographischen Systeme – insbesondere Secret-Key- und Public-Key-Verschlüsselung dar.) Pessiland vereint die Nachteile der anderen Welten: $\mathcal{NP}$-Probleme sind schwierig, aber nicht nutzbar für kryptographische Zwecke.

4. *Minicrypt*: Es gilt $P \neq \mathcal{NP}$, Einwegfunktionen existieren, aber keine Public-Key-Kryptographie.
   Symmetrische Kryptographie hingegen ist in dieser Welt möglich.

5. *Cryptomania*: Es gilt $P \neq \mathcal{NP}$, Einwegfunktionen und Public-Key-Kryptographie existieren.
   Zur Zeit verhalten wir uns so, als ob wir uns in dieser Welt befänden. Die heute verwendete Public-Key-Kryptographie basiert auf Problemen, von denen angenommen wird, dass sie nicht effizent lösbar sind. Nach allgemeiner Meinung ist die Wahrscheinlichkeit hoch, dass wir uns in dieser Welt befinden.

3.3 $\mathcal{NP}$-Vollständigkeit

Trivialerweise gilt $P \subseteq \mathcal{NP}$, weshalb Aussagen wie „Die Sprache L ist in $\mathcal{NP}$“ in der Regel wenig aussagekräftig sind. Interessant sind die Probleme, die unter der Annahme $P \neq \mathcal{NP}$ in $\mathcal{NP} \setminus P$ liegen würden. Im Folgenden bezeichnet „$\leqslant$“ eine Many-one-Reduktion, bei der die Transformationsfunktion f in Polynomialzeit berechenbar ist.

Definition 3.26 ($\mathcal{NP}$-Schwere)**.** *Seien* L, L' *Sprachen.* L *heißt* $\mathcal{NP}$-schwer *(englisch:* $\mathcal{NP}$-hard*), wenn*

$$\forall L' \in \mathcal{NP} : L' \leqslant L$$

gilt.

Zur Erinnerung: Many-one-Reduzierbarkeit ist nicht gleich Turing-Reduzierbarkeit: Dort gilt: $L \leqslant_T L'$ falls L mit einer Orakel-Turing-Maschine M^O mit Orakel O, das L' entscheidet, entschieden werden kann. Ob die Klasse $\mathcal{NP}$ mit polynomieller Turing-Reduzierbarkeit größer würde, ist unbekannt.

Übrigens ...

Eine $\mathcal{NP}$-schwere Sprache muss nicht selbst in $\mathcal{NP}$ sein (im Gegensatz zu $\mathcal{NPC}$, Definition 3.27). Beispielsweise ist das Halteproblem $\mathcal{NP}$-schwer, aber nicht in $\mathcal{NP}$.

Definition 3.27 ($\mathcal{NP}$-Vollständigkeit)**.** *Eine Sprache* L *heißt* $\mathcal{NP}$-vollständig *(englisch:* $\mathcal{NP}$-complete*), wenn*

1. $L \in \mathcal{NP}$ *und*
2. $\forall L' \in \mathcal{NP} : L' \leqslant L$, L *also* $\mathcal{NP}$*-schwer ist.*

Die Menge der $\mathcal{NP}$-vollständigen *Sprachen (englisch* $\mathcal{NP}$*-complete) ist*

$$\mathcal{NPC} = \{L \in \mathcal{NP} \mid \forall L' \in \mathcal{NP} : L' \leqslant L\}.$$

3.3.1 Der Satz von Cook

Ein Paradebeispiel für eine Sprache, die in $\mathcal{NP}$ liegt, ist die Sprache der erfüllbaren aussagenlogischen Formeln, also derjenigen aussagenlogischen Formeln, für die man die Variablen so belegen kann, dass die Formel `wahr` wird. Hat man einen solchen Ausdruck in konjunktiver Normalform (KNF) gegeben, sowie eine Belegung der Variablen, so dass die Formel zu `wahr` auswertet, kann man in polynomieller Zeit überprüfen, ob die Belegung korrekt ist. Hingegen ist der Beweis dafür, dass eine Formel nicht erfüllbar ist, sehr schwierig. Man muss für alle Belegungen zeigen, dass diese die Formel nicht erfüllen.

Um zu zeigen, dass für eine Sprache $L \in \mathcal{NPC}$ gilt, muss man also zeigen, dass alle Sprachen in $\mathcal{NP}$ auf dieses Problem polynomiell reduzierbar sind. Dass es überhaupt eine Sprache gibt, die dieses Kriterium erfüllt, wurde 1971 von Stephen A. Cook bewiesen. Die Definition von $\mathcal{NPC}$ offenbart das grundlegende Problem: Man muss für eine Sprache L' zeigen, dass sich alle anderen Sprachen aus $\mathcal{NP}$ auf sie reduzieren lassen. Eine polynomielle Transformation für jede Sprache anzugeben ist offensichtlich unmöglich. Cook wählte deshalb einen anderen Weg: Für jede Sprache $L \in \mathcal{NP}$ gibt es nach Definition eine Orakel-Turing-Maschine M_L, die L entscheidet. Der Beweis besteht nun darin, die Arbeitsweise von M_L durch polynomielle Transformation in einer aussagenlogischen Formel auszudrücken.

Die Einschränkung auf Ausdrücke in KNF ist zulässig, da wir die Worst-Case-Komplexität betrachten und es sich zeigen lässt, dass alle anderen Fälle einfacher sind.

Definition 3.28 (Kurzzusammenfassung: Semantik der Aussagenlogik)**.** *Sei* Σ *ein Alphabet (in der Logik: eine Signatur). Dann sind* 0, 1 *und alle* $s \in \Sigma$ Atome. 0, 1 *und alle* $s \in \Sigma$ *sind aussagenlogische* Formeln. *Sind* A *und* B *Formeln, so auch* $A \wedge B$ *(Konjunktion),* $A \vee B$ *(Disjunktion),* $\neg A$ *(Negation).* $\neg$ *bindet stärker als* $\wedge$, $\wedge$ *bindet stärker*

als $\vee$. Klammern ist nach den üblichen Regeln erlaubt. Die binären Verknüpfungen sind kommutativ. Statt $\neg x$ schreiben wir auch $\overline{x}$.

Eine Belegung *oder Interpretation ist eine Abbildung von Variablen auf Wahrheitswerte:* $I : \Sigma \to \{\texttt{wahr}, \texttt{falsch}\}$. *Die* Auswertung *einer Formel über einer Belegung ist ein Wahrheitswert* `wahr` *oder* `falsch` *gemäß folgender Regeln:*

- $0 \mapsto \texttt{falsch}$, $1 \mapsto \texttt{wahr}$,
- $\neg\texttt{wahr} = \texttt{falsch}$, $\neg\texttt{falsch} = \texttt{wahr}$,
- $\texttt{wahr} \wedge \texttt{wahr} = \texttt{wahr}$, $\texttt{wahr} \wedge \texttt{falsch} = \texttt{falsch}$, $\texttt{falsch} \wedge \texttt{falsch} = \texttt{falsch}$,
- $\texttt{wahr} \vee \texttt{wahr} = \texttt{wahr}$, $\texttt{wahr} \vee \texttt{falsch} = \texttt{wahr}$, $\texttt{falsch} \vee \texttt{falsch} = \texttt{falsch}$.

Information.

Die Implikation $\to$ kann mit $\vee$ und $\neg$ erklärt werden: $A \to B = \neg A \vee B$. Die Äquivalenz $\leftrightarrow$ ergibt sich durch $A \leftrightarrow B = A \wedge B \vee \overline{A} \wedge \overline{B} = (A \to B) \wedge (B \to A)$.

Definition 3.29 (Erfüllbare Formel)**.** *Eine Formel heißt* erfüllbar*, wenn eine erfüllende Belegung existiert, so dass sie zu* `wahr` *auswertet.*

Definition 3.30 (Konjunktive Normalform)**.** *Ein Literal ist ein Atom oder ein negiertes Atom. Eine Formel ist in konjunktiver Normalform (KNF), wenn sie eine Konjunktion von Disjunktionen von Literalen ist.*

Problem 3.31 (SAT)**.** *Sei ϕ eine aussagenlogische Formel. Das Entscheidungsproblem „Ist ϕ erfüllbar?“ heißt* SAT *(von englisch* satisfiability*).*

Satz 3.32 (Satz von Cook)**.** $\texttt{SAT} \in \mathcal{NP}C$.

Beweis. Zuerst ist zu zeigen, dass $\texttt{SAT} \in \mathcal{NP}$. Sei o. B. d. A. ϕ eine aussagenlogische Formel in KNF mit a Klauseln und b Literalen. Dann lässt sich für eine gegebene Interpretation in $O(a \cdot b)$ prüfen, ob ϕ zu `wahr` auswertet. Damit ist gezeigt, dass $\texttt{SAT} \in \mathcal{NP}$. Nun müssen wir für ein beliebiges Problem aus $\mathcal{NP}$ zeigen, dass dieses in polynomieller Zeit auf SAT reduzierbar ist. Sei also L eine beliebige Sprache mit $L \in \mathcal{NP}$, sowie M_L die Orakel-Turing-Maschine (Definition 3.15), die L entscheidet.

Wir konstruieren nun eine aussagenlogische Formel, die genau dann erfüllbar ist, wenn M_L durch eine zulässige Folge von Zustandsübergängen die Eingabe x entscheidet.

Als Vorüberlegung machen wir uns klar, dass M_L bei Eingabe x mit $|x| = n$ $p(n)$ Berechnungsschritte machen kann, wobei p ein Polynom ist. Entsprechend ist auch die Anzahl der benutzten Bandfelder (zu denen noch die konstant lange Ausgabe des Orakels kommt) durch $p(n)$ begrenzt. Da wir eine polynomielle Transformation angeben, ist auch die Anzahl der Klauseln polynomiell in n beschränkt.

O. B. d. A. habe M_L die Zustände $q_0, \ldots, q_g$ und es sei γ_l das l-te Zeichen aus Γ mit $|\Gamma| = d$. Wir definieren folgende Literale:

- $q_{i,t_j} = 1$, wenn sich M_L zum Zeitpunkt t_j ($0 \leqslant j \leqslant e$) im Zustand q_i befindet, sonst 0.
- $s_{k,t_j,l} = 1$, wenn sich zum Zeitpunkt t_j an der Stelle $0 \leqslant k \leqslant f$ das Zeichen γ_l befindet, sonst 0.

Offensichtlich sind f und e nach den Vorüberlegungen durch $p(n)$ beschränkt und $l \in \{0, \ldots, d-1\}$. Wir können eine akzeptierende Zustandsfolge unter Zuhilfenahme der eben definierten Literale folgendermaßen darstellen:

$$q_{0,t_0}, q_{1,t_0}, \ldots, q_{g,t_0}, s_{0,t_0,0}, \ldots, s_{0,t_0,d-1}, \ldots, s_{f,t_0,0}, \ldots, s_{f,t_0,d-1}$$
$$\vdots$$
$$q_{0,t_e}, q_{1,t_e}, \ldots, q_{g,t_e}, s_{0,t_e,0}, \ldots, s_{0,t_e,d-1}, \ldots, s_{f,t_e,0}, \ldots, s_{f,t_e,d-1}$$

Jede Zeile entspricht einen Berechnungsschritt, in den Spalten sind der aktuelle Zustand und der momentane Bandinhalt angegeben. Wir setzen t_0 als Start-, t_e als Endzeitpunkt, $q_{0,t_0} = 1$ und $q_{i,t_0} = 0$ für alle $i \neq 0$, da M_L als deterministische Turing-Maschine per Konvention genau einen Startzustand, nämlich q_0, hat.

Analog muss in jeder Bandzelle zu jedem Zeitpunkt genau ein Zeichen aus Γ stehen:

$$\bigwedge_{j\in\{0,\ldots,e\},k\in\{0,\ldots,f\}} \left(\bigvee_{l\in\{0,\ldots,d-1\}} s_{k,t_j,l} \wedge \left(\bigwedge_{a,b\in\{0,\ldots,d-1\},a\neq b} (\overline{s_{k,t_j,a}} \vee \overline{s_{k,t_j,b}}) \right) \right)$$

Auf den Bandfeldern mit den Indizes 0 bis $n-1$ steht zum Startzeitpunkt t_0 die Eingabe x, an den Stellen n bis f die Ausgabe des Orakels. Wir legen weiterhin o. B. d. A. fest, dass nicht der Kopf der Turing-Maschine, sondern stattdessen das Band bewegt wird. Für eine Bewegung des Kopfes nach links wird also das Band nach rechts verschoben (und umgekehrt). Wir definieren wir die Literale (c_{0,t_j}, c_{1,t_j}), die für die Richtungsänderung des Lese-/Schreibkopfs stehen:

$$(c_{0,t_j}, c_{1,t_j}) = \begin{cases} (0,0) \text{ für } t_j \to t_{j+1} \text{ keine Bewegung} \\ (1,0) \text{ für } t_j \to t_{j+1} \text{ Bewegung nach links} \\ (0,1) \text{ für } t_j \to t_{j+1} \text{ Bewegung nach rechts.} \end{cases}$$

Im Weiteren möchten wir nun für jede der definierten Literale eine Formel erstellen, der überprüft, ob ein Schritt der Turing-Maschine gültig ist.

Das Band

Die Gültigkeit der Bandbewegung lässt sich folgendermaßen mit einer aussagenlogischen Formel pro Alphabetsymbol $\gamma_l \in \Gamma$ ausdrücken:

$(\overline{c_{0,t_j}} \vee \overline{c_{1,t_j}}) \wedge$	Entweder c_{0,t_j} oder c_{1,t_j} muss falsch sein
$((\overline{c_{0,t_j}} \wedge \overline{c_{1,t_j}}) \to (s_{k,t_{j+1},l} \leftrightarrow s_{k,t_j,l})) \wedge$	Kopf bewegt sich nicht
$((c_{0,t_j} \wedge \overline{c_{1,t_j}}) \to (s_{k-1,t_{j+1},l} \leftrightarrow s_{k,t_j,l})) \wedge$	Kopf bewegt sich nach links
$((\overline{c_{0,t_j}} \wedge c_{1,t_j}) \to (s_{k+1,t_{j+1},l} \leftrightarrow s_{k,t_j,l}))$	Kopf bewegt sich nach rechts

Wir brauchen also pro Feld auf dem Band eine Formel mit konstanter Anzahl an Klauseln, um zu überprüfen, ob ein Schritt gültig ist.

Zustandsübergänge

Als nächstes überlegen wir uns aus der Zustandsübergangsfunktion Terme, die überprüfen, ob ein Schritt gültig ist. Allgemein wird dabei eine Regel der Form

$$\delta(q_i, v, x) = (q_p, w) \text{ mit } v, w \in \Gamma, x \in \{N, L, R\}$$

umgewandelt zu einem Term der Form:

$$\begin{aligned}
&(q_{i,t_j} \wedge (s_{k,t_j,l} \leftrightarrow v)) \rightarrow (s_{k,t_{j+1},m} \leftrightarrow w) \wedge \overline{c_{0,t_j}} \wedge \overline{c_{1,t_j}} \wedge q_{p,t_{j+1}} \text{ für } x = N\\
&(q_{i,t_j} \wedge (s_{k,t_j,l} \leftrightarrow v)) \rightarrow (s_{k,t_{j+1},m} \leftrightarrow w) \wedge c_{0,t_j} \wedge \overline{c_{1,t_j}} \wedge q_{p,t_{j+1}} \text{ für } x = L\\
&(q_{i,t_j} \wedge (s_{k,t_j,l} \leftrightarrow v)) \rightarrow (s_{k,t_{j+1},m} \leftrightarrow w) \wedge \overline{c_{0,t_j}} \wedge c_{1,t_j} \wedge q_{p,t_{j+1}} \text{ für } x = R
\end{aligned}$$

Formt man diesen Ausdruck in eine KNF um, so erhält man wieder eine konstante Anzahl an Klauseln pro Ableitungsregel und verwendetem Feld auf dem Band.

Eindeutigkeit des Zustands und Finalzustände

Als Nächstes wollen wir sicherstellen, dass wir uns zu einem beliebigen Zeitpunkt immer nur in einem Zustand befinden. Dafür formulieren wir für alle Zeitpunkte t_j folgende Formel:

$$(q_{0,t_j} \vee \ldots \vee q_{g,t_j}) \wedge \bigwedge_{v \neq w} (\overline{q_{v,t_j}} \vee \overline{q_{w,t_j}})$$

Diese Formel enthält $n^2 + 1$ Klauseln
Als letztes wollen wir noch eine weitere Formel definieren, der garantiert, dass wir uns am Ende in einem Finalzustand befinden. Sei also F die Menge der Finalzustände, dann muss gelten:

$$\bigvee_{q_i \in F} q_{i,t_e}$$

Ergebnis

Wir haben also ein *beliebiges* Entscheidungsproblem aus $\mathcal{NP}$ polynomiell auf eine Instanz von SAT reduziert. Daraus folgt, dass $\mathtt{SAT} \in \mathcal{NPC}$. □

3.3.2 Der schmale Grat

Oft sind sehr ähnlich aussehende Probleme in sehr unterschiedlichen Komplexitätsklassen. Darunter zum Beispiel:

Probleme in $\mathcal{P}$	**Probleme in** $\mathcal{NPC}$
Existenz eines Eulerkreis Ein **Eulerkreis** ist ein Zyklus in einem Graphen, der alle **Kanten** genau einmal enthält.	*Existenz eines Hamilton-Kreises* Ein **Hamilton-Kreis** ist ein Zyklus in einem Graphen, der alle **Knoten** genau einmal enthält.
Existenz unärer Partitionen Gegeben $A = \{a_1, \ldots, a_n\} \subseteq \mathbb{N}$. jeweils **unär** kodiert. Gibt es ein $A' \subseteq A$ mit $\sum A' = \sum A \setminus A'$?	*Existenz binärer Partitionen* Gegeben $A = \{a_1, \ldots, a_n\} \subseteq \mathbb{N}$. jeweils **binär** kodiert. Gibt es ein $A' \subseteq A$ mit $\sum A' = \sum A \setminus A'$?
Graphen-Zweifärbbarkeit Ein Graph ist 2-färbbar, wenn man jedem Knoten eine von **zwei** Farben zuordnen kann, so dass benachbarte Knoten nie dieselbe Farbe haben.	*Graphen-Dreifärbbarkeit* Ein Graph ist 3-färbbar, wenn man jedem Knoten eine von **drei** Farben zuordnen kann, so dass benachbarte Knoten nie dieselbe Farbe haben.

Abbildung 3.6: Gegenüberstellung von $\mathcal{P}$- und $\mathcal{NPC}$-Problemen.

3.3.3 $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$

Es gibt verschiedene Standpunkte zu dem Thema $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$. Man könnte etwa sagen, dass es so viele Probleme in $\mathcal{NPC}$ gibt, dass einfach zu zeigen sein müsste, dass eines davon in $\mathcal{P}$ liegt. Andererseits könnte man auch sagen, dass die Probleme in $\mathcal{NPC}$ aus der Sicht der Komplexitätstheorie alle so ähnlich sind, dass sie auch wahrscheinlich gleich schwierig zu lösen sind. Allgemein wird angenommen, dass $\mathcal{P} \neq \mathcal{NP}$ gilt.

3.3.3.1 Schwierigkeiten

Warum ist es nun so schwierig zu beweisen, wie sich $\mathcal{P}$ zu $\mathcal{NP}$ verhält? Das liegt daran, dass ein Beweis für $\mathcal{P} \neq \mathcal{NP}$ zwar nicht unmöglich ist, aber zumindest besondere Beweisverfahren erfordert. Man spricht hier von *nicht relativierenden Beweisen*. Ein Beweis, der eine Beziehung zwischen zwei Komplexitätsklassen zeigt, ist relativierend, wenn er weiterhin gültig ist, wenn man zu beiden Klassen ein beliebiges Orakel hinzufügt. Hätte man etwa für zwei Klassen K und L gezeigt, dass diese gleich sind, also dass $K = L$ gilt und derselbe Beweis auch $K^O = L^O$ für ein beliebiges Orakel O zeigt, so ist dies ein relativierender Beweis. Dabei ist A^O die Klasse A mit zusätzlichem O-Orakel. Es wurde aber gezeigt, dass es Orakel A und B gibt, für die jeweils gilt: $\mathcal{P}^A = \mathcal{NP}^A$ bzw. $\mathcal{P}^B \neq \mathcal{NP}^B$. Das heißt, dass jeder relativierende Beweis zu einem Widerspruch führen würde, weshalb sich diese Klasse an Beweisen hier nicht verwenden lässt. Die meisten üblichen Beweistechniken wie z. B. die Diagonalisierung sind leider relativierend: Man kann einer Turing-Maschine ein beliebiges Orakel hinzufügen, und diese kann ihr eigenes Halteproblem weiterhin nicht lösen.

3.4 Ausgewählte Probleme in $\mathcal{NP}$

Viele interessante Probleme der Informatik sind in $\mathcal{NP}$, viele davon sogar in $\mathcal{NPC}$. Einige ausgewählte sollen hier (teilweise ohne Beweis) genannt werden.

3.4.1 $\mathcal{NP}$-Vollständigkeit zeigen

Um für eine Sprache L zu zeigen, dass $L \in \mathcal{NPC}$ gilt, reicht es, für eine beliebige Sprache $L' \in \mathcal{NPC}$ zu zeigen, dass $L' \leqslant L$ und $L \in \mathcal{NP}$ gilt, also L' auf L polynomiell reduziert werden kann und L in $\mathcal{NP}$ ist. Da insbesondere für L' schon gezeigt wurde, dass sich alle Probleme aus $\mathcal{NP}$ auf L' polynomiell reduzieren lassen, folgt aus der Transitivität der Reduzierbarkeit sofort, dass dies auch für L gilt. Da $\texttt{SAT} \in \mathcal{NPC}$ bereits gezeigt wurde, können wir nun SAT auf weitere $\mathcal{NP}$-Probleme reduzieren und so zeigen, dass sie in $\mathcal{NPC}$ sind.

3.4.2 3-SAT

Problem 3.33 (3-SAT)**.** *Sei ϕ eine aussagenlogische Formel in KNF, bei der jede Klausel aus höchstens drei Literalen besteht. Hat ϕ eine erfüllende Belegung?*

Beispiel 26 (Eine Beispiel-Instanz für 3-SAT)**.**

$$\underbrace{(a \vee b \vee c)}_{\text{einzelne Klausel}} \wedge (\overline{a} \vee b \vee d)$$

Satz 3.34. *3-SAT* $\in \mathcal{NPC}$.

Beweis. Man sieht einfach, dass diese Sprache in $\mathcal{NP}$ liegt, es muss also noch gezeigt werden, dass $\texttt{3-SAT} \in \mathcal{NPC}$ ist. Wir wollen also nun eine allgemeine aussagenlogische Formel in KNF in eine umformen, die höchstens drei Literale pro Klausel hat und genau dann erfüllbar ist, wenn die ursprüngliche Formel auch erfüllbar ist. Gegeben sei also eine Formel der Form

$$(v_{1,1} \vee v_{1,2} \vee \ldots \vee v_{1,k_1}) \wedge \ldots \wedge (v_{m,1} \vee v_{m,2} \vee \ldots \vee v_{m,k_m})$$

Wir formen die Klauseln nach folgenden Regeln um:

1. Klauseln mit einem Literal der Form (x) werden zu $(x \vee x \vee x)$ umgeformt.
2. Klauseln mit zwei Literalen der Form $(x \vee y)$ werden zu $(x \vee y \vee x)$ umgeformt.
3. Klauseln mit drei Literalen werden unverändert belassen.
4. Für eine Klausel c_i mit $k_i > 3$ Literalen gehen wir folgendermaßen vor: Wir definieren uns neue Literale

   $$a_{i,1}, a_{i,2}, \ldots, a_{i,k_i-3}$$

   und formen mit deren Hilfe die Klausel um:

   $$(v_{i,1} \vee v_{i,2} \vee \cdots \vee v_{i,k_i})$$

   wird zu

   $$(v_{i,1} \vee v_{i,2} \vee a_{i,1}) \wedge (\overline{a_{i,1}} \vee v_{i,3} \vee a_{i,2}) \wedge \cdots \wedge (\overline{a_{i,k_i-3}} \vee v_{i,k_i-1} \vee v_{i,k_i}).$$

In den Fällen 1 bis 3 entstehen keine neuen Literale oder zusätzliche Klauseln, im vierten für eine Klausel mit k Literalen $k-2$ neue Klauseln mit jeweils drei Literalen und insgesamt $k-3$ neue Literale. Zwei benachbarte Klauseln haben dabei jeweils eines der neuen Literale gemeinsam, wobei es in einer der beiden Klauseln jeweils negiert ist. Die neuen Literale bewirken also, dass jeweils eine von zwei benachbarten Klauseln mit Sicherheit `wahr` ist. Da es jedoch $k-2$ Klauseln, aber nur $k-3$ neue Literale gibt, kann keine Belegung der neuen Literale die ganze Formel `wahr` werden lassen, wenn alle anderen Klauseln `falsch` sind. Es muss mindestens eines der schon existierenden Literale `wahr` sein. Dies ist aber genau die Bedingung dafür, dass die ursprüngliche Klausel `wahr` ist. Verfährt man so mit allen Klauseln, erhält man eine Formel, die genau drei Literale pro Klausel hat und genau dann erfüllbar ist, wenn die ursprüngliche Formel erfüllbar ist. Wir haben damit also eine beliebige `SAT`-Instanz auf eine `3-SAT`-Instanz reduziert. Da wir für eine Klausel mit $k > 3$ Literalen $k-2$ Klauseln mit $k-3$ Literalen erzeugen müssen, ist diese Reduktion polynomiell. Daraus folgt, dass `3-SAT` auch $\mathcal{NP}$-vollständig ist. □

3.4.3 CLIQUE

Problem 3.35 (`CLIQUE`). *Sei* $G = (V, E)$ *ein Graph und* $k \in \mathbb{N}$ *mit* $k \leqslant |V|$. *Gibt es ein* $V' \subseteq V$ *mit* $|V'| \geqslant k$, *sodass* $(i, j) \in E$ *für alle* $i \neq j \in V'$ *gilt?*

Satz 3.36. `CLIQUE` $\in \mathcal{NPC}$.

Beweis. Offensichtlich gilt `CLIQUE` $\in \mathcal{NP}$, da für gegebenes V' in $O(V'^2)$ geprüft werden kann, ob V' eine Clique und $|V'| \geqslant k$ ist.

Wir zeigen die Reduktion `CLIQUE` $\geqslant$ `3-SAT`:
Sei $\phi = (x_{11} \vee x_{12} \vee x_{13}) \wedge \cdots \wedge (x_{k1} \vee x_{k2} \vee x_{k3})$ eine `3-SAT`-Instanz mit k Klauseln und x_{i1}, x_{i2}, x_{i3} $(i = 1, \ldots, k)$ jeweils Literale. Wir konstruieren einen Graphen $G = (V, E)$, indem wir für jedes x_{ij} $(i = 1, \ldots, k, j = 1, 2, 3)$ einen Knoten zu V hinzufügen. Wir setzen

$$E = \{(x_{op}, x_{qr}) \mid o \neq q, x_{op} \neq \overline{x_{qr}}\}.$$

E besteht also genau aus den Literal-Paaren, die nicht in der selben Klausel vorkommen und widerspruchsfrei erfüllbar sind (also x_{op} und x_{qr} entweder ein unterschiedliches Atom repräsentieren bzw. dasselbe Atom beide Male negiert bzw. nicht negiert). ϕ ist genau dann erfüllbar, wenn es in G eine Clique der Größe k gibt, da jede Clique genau ein Literal je Klausel beinhaltet und dieses ohne Widersprüche erfüllbar ist. □

Beispiel 27.

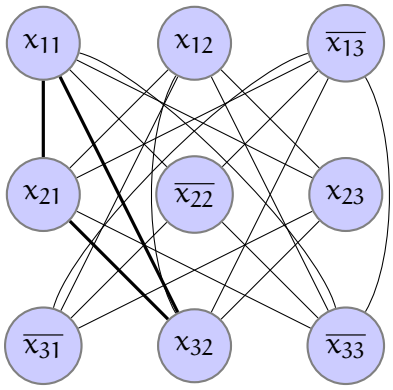


Abbildung 3.7: `CLIQUE`-Graph für $\phi = (a \vee b \vee \overline{c}) \wedge (a \vee \overline{b} \vee c) \wedge (\overline{a} \vee b \vee \overline{c})$. Wir lesen aus der eingezeichneten Clique eine erfüllende Interpretation $a = b =$ `wahr` ab.

3.4.4 Faktorisieren

Ein wichtiges Problem, von dem unter anderem die Sicherheit des RSA-Algorithmus abhängt, ist das Faktorisieren von natürlichen Zahlen. Die Frage nach der Primfaktorzerlegung einer Zahl ist kein Entscheidungsproblem, kann jedoch als solches formuliert werden:

Problem 3.37 (`FACTOR`). *Seien* $n, k, k' \in \mathbb{N}$. *Hat* n *einen Primfaktor* $k' \leqslant k$*?*

Durch Variieren von k lassen sich mit einem Algorithmus, der `FACTOR` entscheidet, die entsprechenden Primfaktoren gewinnen. Es gilt $\texttt{FACTOR} \in (\mathcal{NP} \cap \text{co}\mathcal{NP})$.

3.4.5 Graph-Färbbarkeit

Problem 3.38 (`COLOR`). *Sei* $G = (V, E)$ *ein ungerichteter Graph und* $k \in \mathbb{N}$. *Lassen sich alle* $v \in V$ *mit* k *Farben färben, sodass für alle* $(v_i, v_j) \in E$ v_i *und* v_j *unterschiedliche Farben haben?*

`COLOR` mit $k = 3$ wird auch `3COLOR` oder Graph-Dreifärbbarkeitsproblem genannt:

Problem 3.39 (`3COLOR`). *Sei* $G = (V, E)$ *ein ungerichteter Graph. Lassen sich alle* $v \in V$ *mit drei Farben färben, sodass für alle* $(v_i, v_j) \in E$ v_i *und* v_j *unterschiedliche Farben haben?*

Satz 3.40. $\texttt{3COLOR} \in \mathcal{NP}\text{C}$.

Beweis. Um zu zeigen, dass dieses Problem $\mathcal{NP}$-vollständig ist, wollen wir zeigen, dass `3-SAT` darauf reduzierbar ist. Dass $\texttt{3COLOR} \in \mathcal{NP}$ ist wieder einfach zu erkennen.

Wir wollen aus einer Instanz von `3-SAT` eine Instanz des Graph-Dreifärbbarkeits-Problems erstellen, sodass der entstehende Graph genau dann dreifärbbar ist, wenn die Formel der `3-SAT`-Instanz erfüllbar ist. Bei den Farben legen wir uns auf $\{T, F, O\}$ für `true`, `false` und `other` fest. Die Farben sind grundsätzlich frei wählbar.

3.4.5.1 Hilfsgraph

Es sei ϕ eine 3-SAT-Instanz mit den Literalen $x_1, x_2, \ldots, x_n$. Wir konstruieren zunächst den unter Abbildung 3.8 dargestellten Graphen. Dieser hat drei verbundene Knoten, die mit O, T sowie F beschriftet und als Dreieck angeordnet sind. Damit dieser Graph dreigefärbt werden kann, müssen diese drei Knoten jeweils unterschiedliche Farben haben. Desweiteren erstellen wir für jedes Literal x_i zwei weitere verbundene Knoten, die mit x_i bzw. $\overline{x_i}$ beschriftet und beide mit dem mit O beschrifteten Knoten verbunden sind. Dabei ist die Färbung der Knoten O, T und F jeweils zwar noch nicht vorgegeben; da eine Permutation der Farben die Färbbarkeit eines Graphen jedoch nicht ändert, können wir o. B. d. A. die gewählte Darstellung annehmen.

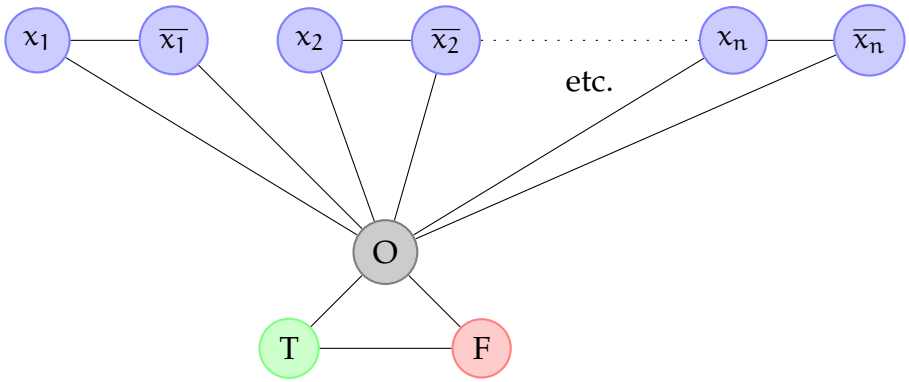


Abbildung 3.8: Hilfsgraph für 3-SAT.

3.4.5.2 Dreieck und Propeller

Die nächste Struktur, die wir betrachten, ist ein einfaches Dreieck, das an zwei Ecken noch jeweils mit einem weiteren Knoten (hier x und y) verbunden ist.

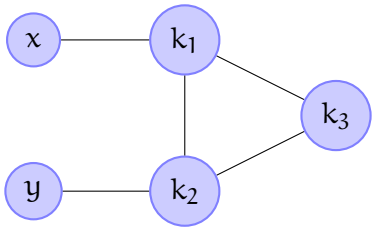


Abbildung 3.9: Dreieck mit verbundenen Ecken.

Angenommen, wir färben zuerst x und y ein. Dann erkennen wir, dass die Farbe von k_3 genau dann eindeutig bestimmt ist, wenn x die gleiche Farbe wie y hat. Nun betrachten wir einen „Propeller“, der aus zwei solchen verbundenen Dreiecken aufgebaut ist.

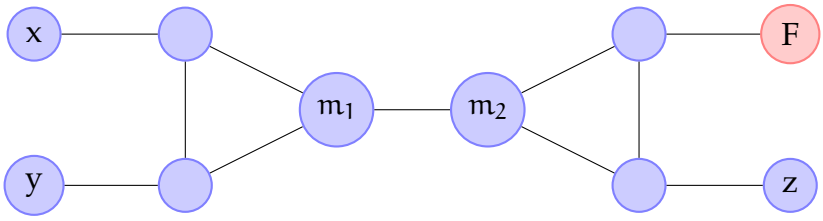


Abbildung 3.10: „Propeller“.

In diesem Graphen sollen nun wieder die Knoten x, y und z jeweils entweder mit T oder F eingefärbt werden. Für ein einzelnes Dreieck gilt, dass der Knoten, der mit dem anderen Dreieck verbunden ist, genau dann in seiner Farbe festgelegt ist, wenn die zwei Knoten, die an dieses Dreieck angefügt wurden, die gleiche Farbe haben. Sind also etwa x, y mit T eingefärbt, so müssen die beiden Knoten, die mit x und y verbunden sind, jeweils mit F und O eingefärbt sein und demnach m_1 auch mit T. Da aber ein Eck des anderen Dreiecks mit einem Knoten verbunden ist, der mit F eingefärbt ist, ist m_2 nur dann festgelegt, wenn z auch mit F eingefärbt ist. Nicht dreifärbbar ist dieses Konstrukt also nur in einem Fall: Wenn alle verbundenen Knoten mit F eingefärbt sind.

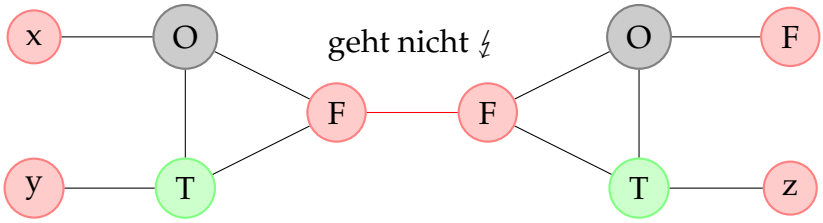


Abbildung 3.11: „Propeller“, Ecken verbunden und ungültig eingefärbt.

Ein „Propeller“ ist also genau dann dreifärbbar, wenn mindestens einer der verbundenen Literal-Knoten nicht die Farbe F hat. Wir verwenden den „Propeller“, um eine einzelne Klausel einer 3-SAT-Instanz zu repräsentieren. Damit wir die entsprechenden Bedingungen für die Knoten, die die Literale darstellen, erfüllen (sie dürfen z. B. nicht mit O eingefärbt werden), sowie mehrere dieser Teilgraphen kombinieren können, verbinden wir entsprechend viele „Propeller“ mit dem zuvor konstruierten Hilfsgraphen. In Abbildung 3.12 ist dabei der Graph mit dem „Propeller“ der Klausel $(x_1 \vee \overline{x_2} \vee x_n)$ dargestellt.

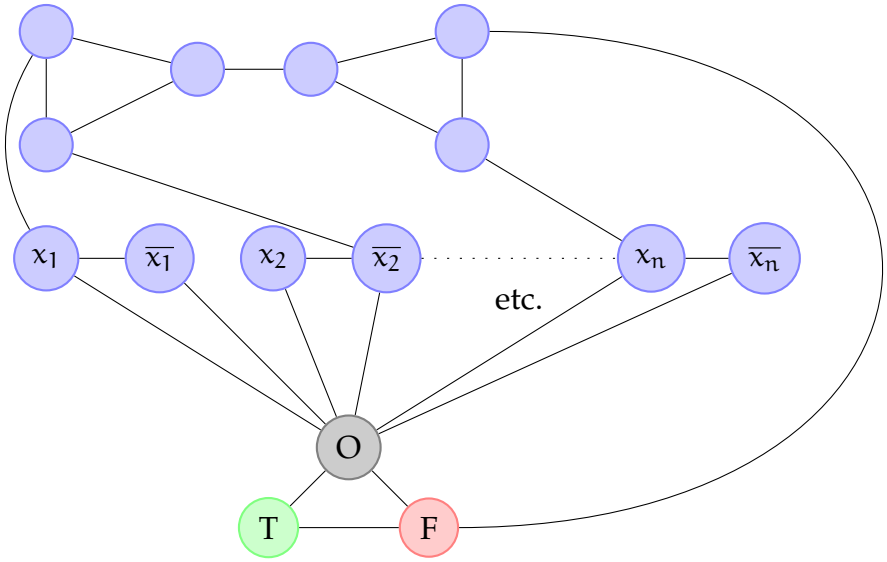


Abbildung 3.12: Hilfsgraph mit einem „Propeller“.

Man sieht nun, dass man weitere solche Propeller anfügen kann, sodass man für jede 3-SAT-Instanz ϕ einen Graphen bauen kann, der genau dann dreifärbbar ist, wenn ϕ erfüllbar ist. Man sieht einfach ein, dass die Reduktion von 3-SAT auf 3COLOR polynomiell ist, weshalb 3-SAT $\leqslant$ 3COLOR und somit 3COLOR $\in \mathcal{NP}$C ist. □

3.4.6 Hamilton-Pfade

Problem 3.41 (HAMPATH). HAMPATH $= \{(G, s, t) \mid G$ *ist ein gerichteter Graph mit einem Hamilton-Pfad von* s *nach* $t\}$

Hinweis.

Analog zum Hamilton-Kreis ist ein Hamilton-Pfad ein Pfad, auf dem jeder Knoten des Graphen genau einmal vorkommt. Ein Hamiltonkreis ist dann ein spezieller Hamilton-Pfad mit $s = t$.

Satz 3.42. HAMPATH $\in \mathcal{NP}$C.

Beweis. Wie in Tabelle 3.6 bereits zu sehen ist (und auch sonst klar sein dürfte), ist HAMPATH $\in \mathcal{NP}$. Wir zeigen nun, dass auch HAMPATH $\in \mathcal{NP}$C gilt, indem wir 3-SAT auf HAMPATH reduzieren. Wir gehen dazu folgendermaßen vor:

1. Sei $\phi = (a_1 \vee b_1 \vee c_2) \wedge \cdots \wedge (a_k \vee b_k \vee c_k)$ eine 3-SAT-Instanz mit k Klauseln. Jedem a, b und c entspricht dabei ein Literal x_i oder $\overline{x_i}$. Für jedes dieser x_i fügen wir einen diamantförmigen Teilgraphen wie in Abbildung 3.13 in G ein.

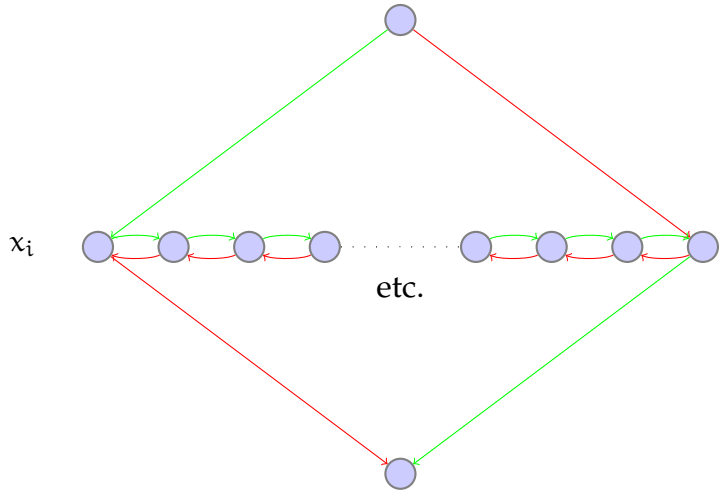


Abbildung 3.13: Für jedes Literal x_i wird ein „Diamant“ erstellt. Es gibt genau einen je Hamilton-Pfad für $x_i = \text{true}$ oder $x_i = \text{false}$.

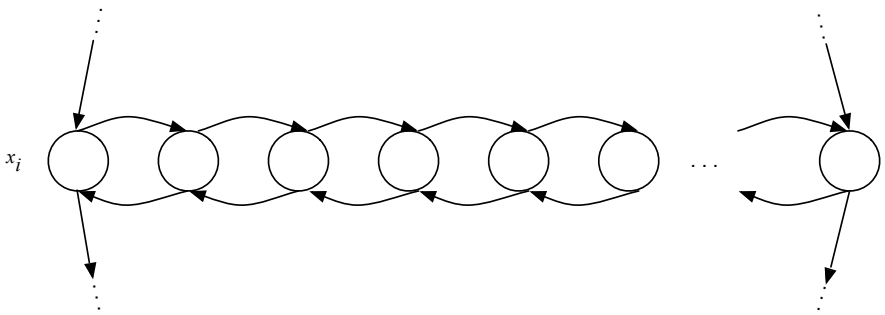


Abbildung 3.14: In einem Diamanten gibt es für jede Klausel, in der das zugehörige Literal vorkommt, ein Knotenpaar. Dazwischen liegen Trennknoten.

Wenn x_i ($\overline{x_i}$) in c_j vorkommt, erweitern wir den Graphen um zwei Kanten wie in Abbildung 3.16 (3.17).

2. Jeder diamantförmige Teilgraph aus Schritt 1 besteht aus $3k+1$ Knoten. Alle Klauseln c_i sind dabei über einen Trennknoten miteinander verbunden, wie Abbildung 3.14 zeigt.

Es bleibt noch zu zeigen, dass es genau dann einen Hamilton-Pfad von s nach t in G gibt, wenn ϕ erfüllbar ist. Angenommen, ϕ ist erfüllbar. Dann gibt es (unter Vernachlässigung der Knoten für die Klauseln) einen Pfad von s nach t. Für jeden diamantförmigen Teilgraphen, der einem Literal x_i entspricht, muss es dazu, wenn x_i eine wahre Belegung zugeordnet wird, einen Pfad im Zick-Zack-Muster geben (falls $x_i = \texttt{falsch}$ anders herum). Die Klausel-Knoten c_j sind durch die in Schritt 2 hinzugefügten Kanten nun ebenfalls Teil des Graphen, wenn die gewählte Belegung des x_i mit der Richtung der Kanten nach c_j übereinstimmt.

Umgekehrt sei nun ein Pfad in G von s nach t gegeben. Es bleibt zu zeigen, dass diese Interpretation ein Modell für ϕ ist. Dies ist dann der Fall, wenn jeder „Diamant“ genau einmal von oben nach unten besucht wird, entsprechend

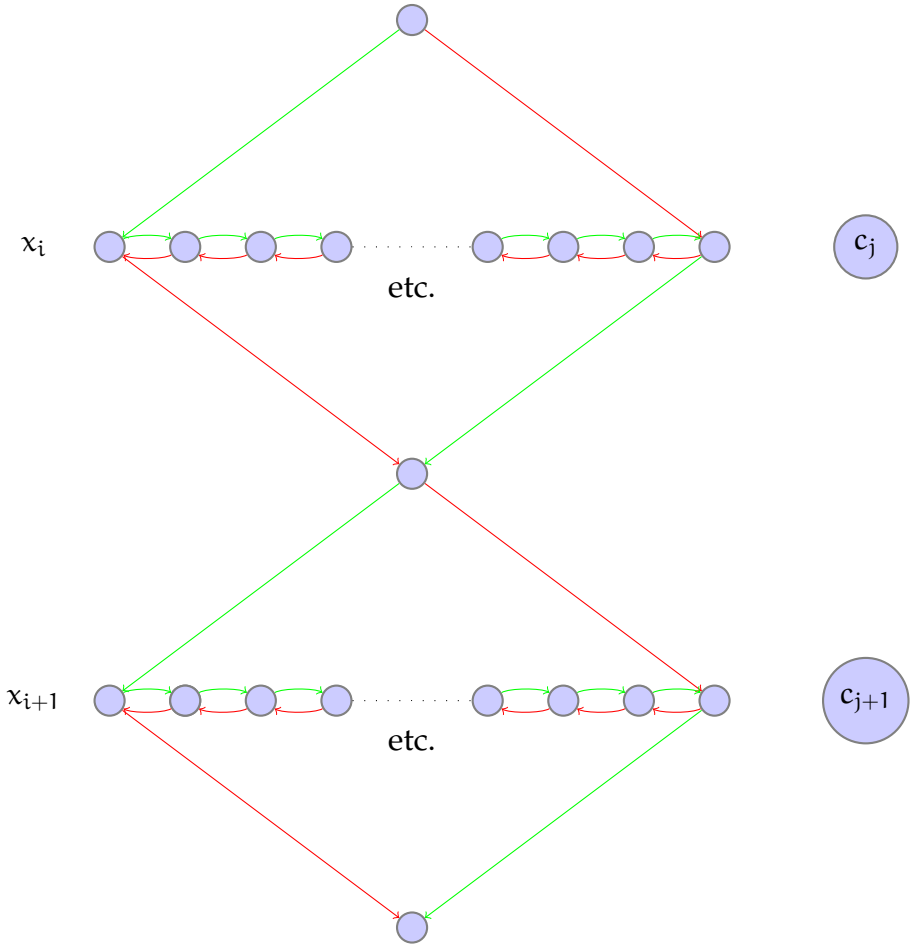


Abbildung 3.15: Für jede Klausel wird ein Klausel-Knoten c_j hinzugefügt.

der Pfadrichtung (Ob „Zick-Zack“ oder „Zack-Zick“ liefert die Interpretation). Man macht sich an einem Beispiel schnell klar, dass es keinen davon abweichenden Pfad geben kann. Angenommen, ein Pfad ginge vom inneren Literal-Knoten a_1 des „Diamanten“ von x_1 zu einem Klausel-Knoten c und von dort zu einem Knoten b_2 im letzten „Diamanten“ x_n, anstatt korrekterweise von a_1 zu a_2 (im x_1-Diamanten). In diesem Fall gäbe es keinen Weg mehr, die Nachbarknoten a_2 und a_3 im x_1-Diamanten zu besuchen. Das ist ein Widerspruch dazu, dass der Pfad ein Hamilton-Pfad ist.

Eine erfüllende Belegung zusammen mit den „Umwegen“ bildet einen Hamilton-Pfad. Gibt es keine erfüllende Belegung, gilt für jedes „Abfahren“ der Variablen, dass eine Klausel (Knoten) unerreichbar bleibt (ohne einen Knoten doppelt zu besuchen). □

3.4.7 VERTEX COVER

Angenommen, man habe ein durch einen Graphen dargestelltes Kommunikationsnetz. Dabei sind die Kommunikationsteilnehmer die Knoten und die Kanten stellen dar, wer wen der Lüge bezichtigt. Wenn also etwa eine Kante zwischen Alice und Bob existiert, so heißt dies, dass die beiden jeweils behaupten, der andere wäre nicht vertrauenswürdig. Man nimmt nun an, es gebe eine Verschwö-

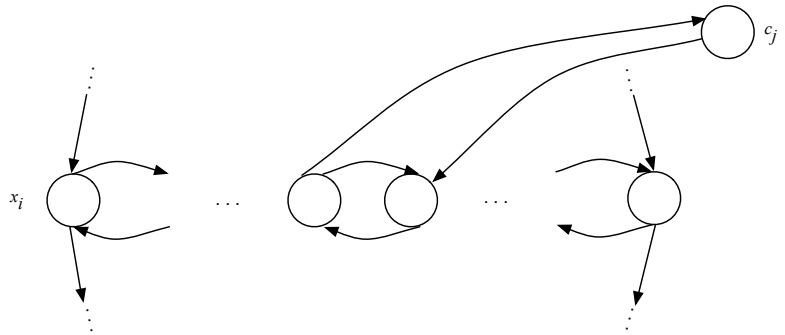


Abbildung 3.16: Ein Knotenpaar wird „nach rechts“ verbunden, wenn x_i in c_j vorkommt.

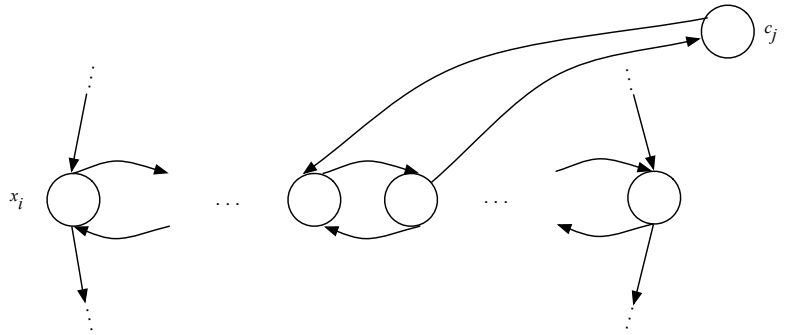


Abbildung 3.17: Ein Knotenpaar wird „nach links“ verbunden, wenn $\overline{x_i}$ in c_j vorkommt.

rung, an der ein gewisser Teil der Kommunikationsteilnehmer beteiligt ist, so dass sich dadurch alle Konflikte erklären lassen. Man sucht für diesen Graphen also die Menge von Knoten, die den Graphen vollständig überdeckt. Diese Menge heißt VERTEX COVER:

Interessant ist dabei vor allem das MINIMAL VERTEX COVER, also ein VERTEX COVER, das eine möglichst geringe Anzahl an Knoten enthält. In der Praxis ist das etwa für Rechnernetze interessant, bei denen einzelne Rechner unterschiedliche Ergebnisse liefern, also manche der Rechner defekt sind. Hier ist es interessant herauszufinden, was die Erklärung für die Probleme ist, die am wenigsten Rechner als defekt markiert. Ein MINMAL VERTEX COVER muss jedoch nicht eindeutig sein. Für einen Graphen, der genau aus zwei verbundenen Knoten besteht, gibt es schon zwei Möglichkeiten. Das zugehörige Entscheidungsproblem, das wir betrachten wollen, lautet nun:

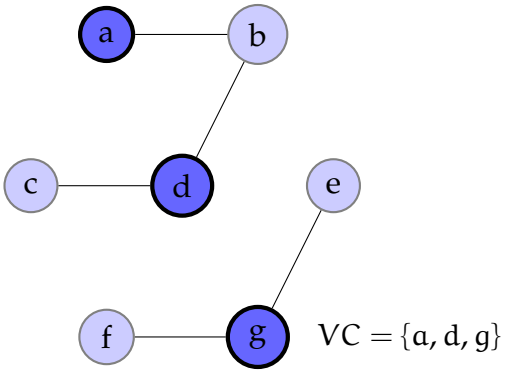


Abbildung 3.18: Beispiel für VERTEX COVER.

Problem 3.43 (VERTEX COVER). *Sei* $G = (V, E)$ *ein Graph und* $k \in \mathbb{N}$. *Gibt es eine Menge* $M \subseteq V$ *mit* $|M| \leqslant k$, *sodass für alle* $(v_i, v_j) \in E$ $v_i \in M$ *oder* $v_j \in M$*?*

Satz 3.44. VERTEX COVER $\in \mathcal{NP}$C.

Beweis. Zuerst einmal sehen wir ein, dass VERTEX COVER $\in \mathcal{NP}$.
Um VERTEX COVER $\in \mathcal{NP}$C zu zeigen, geben wir eine polynomielle Reduktion von 3-SAT auf VERTEX COVER an. Sei ϕ dazu eine Instanz von 3-SAT mit Literalen $x_1, \ldots, x_n$. Wir bilden zunächst wieder einen Hilfsgraphen, der alle Literale enthält, siehe Abbildung 3.19.

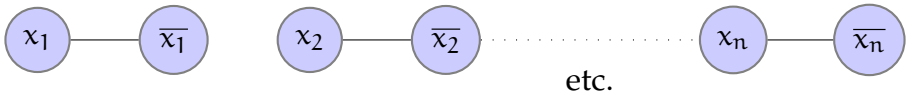


Abbildung 3.19: Hilfsgraph für VERTEX COVER.

Man sieht einfach, dass jedes VERTEX COVER für jedes $x_i \in \{x_1, \ldots, x_n\}$ nur entweder den Knoten für x_i oder den Knoten für $\overline{x_i}$ enthält. Nun betrachten wir wieder die einzelnen Klauseln von ϕ. Für jede Klausel erstellen wir einen neuen, aus einem Dreieck bestehenden Teilgraphen. Die drei Ecken des Dreiecks verbinden wir mit den Knoten, die den Literalen der Klausel entsprechen. In Abbildung 3.20 ist dies wieder am Beispiel der Klausel $(x_1 \vee \overline{x_2} \vee x_n)$ vorgeführt.

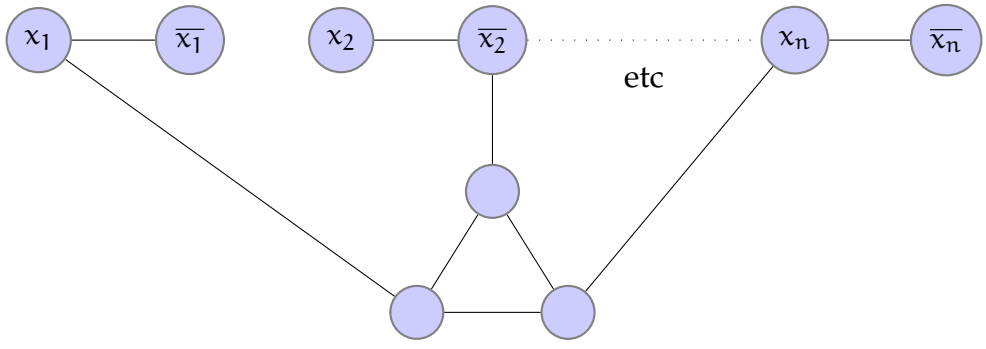


Abbildung 3.20: Hilfsgraph mit einem Dreieck für die Klausel $(x_1 \vee \overline{x_2} \vee x_n)$.

Sei n die Anzahl der Literale und m die Anzahl der Klauseln (also der Dreiecke im Graphen). Dann hat ϕ genau dann eine Lösung, wenn es ein VERTEX COVER mit höchstens $k = n + 2m$ Knoten gibt: Für ein einfaches Dreieck benötigt man zwei Knoten, und da die Ecken der Dreiecke noch mit den Literalen verbunden sind, findet man nur dann eine Überdeckung für ein Dreieck, wenn mindestens eines der verbundenen Literale auch Teil der Überdeckung ist. Das heißt, man muss sich für jedes Literal x_i entscheiden, ob x_i oder $\overline{x_i}$ Teil des VERTEX COVERs ist, und man findet nur dann ein entsprechendes VERTEX COVER, wenn man für jede Klausel mindestens ein Literal ausgewählt hat, das darin vorkommt. Also gibt es genau dann ein VERTEX COVER, wenn ϕ erfüllbar ist. □

3.4.8 3-Dimensional Matching

Problem 3.45 (3DM)**.** *Gegeben drei Mengen* B, G, U *mit* $|B| = |G| = |U| = n$ *und eine dreiwertige Relation* $T \subseteq (B \times G \times U)$*. Gibt es ein Matching* $M \subseteq T$ *aus* n *Tripeln* $(b_1, g_1, u_1), \ldots, (b_n, g_n, u_n)$*, sodass für je zwei Tripel* (b_i, g_i, u_i) *und* (b_j, g_j, u_j) *gilt:* $b_i \neq b_j$, $g_i \neq g_j$ *und* $u_i \neq u_j$*?*

Satz 3.46. 3DM *ist* NP*-vollständig.*

Beweis. In der Übung. □

3.5 Probabilistische Komplexitätsklassen

Die nächste Gruppe von Klassen, die wir betrachten möchten, ist die der *probabilistischen Komplexitätsklassen*. Probleme, die in diesen Klassen liegen, entsprechen am ehesten denen, die man zur Zeit als sinnvoll berechenbar ansieht.

3.5.1 Counting Classes

Bei probabilistischen Komplexitätsklassen betrachten wir Maschinen, die nur mit einer gewissen Wahrscheinlichkeit akzeptieren. Diese Wahrscheinlichkeit hängt dabei von der Anzahl der akzeptierenden Pfade ab. Wir möchten die Anzahl der akzeptierenden Pfade also gerne zählen können.

Definition 3.47 (Counting Turing machine)**.** *Eine* zählende Turing-Maschine *(auch* CTM *für engl.* counting Turing machine*) ist eine nichtdeterministische Turing-Maschine, deren Ausgabe die Anzahl akzeptierender Berechnungspfade ist.*

Definition 3.48 (Klasse #P)**.** *Die Klasse* #P *(gesprochen: sharp* P*) besteht aus den Funktionen, die von polynomiell zeitbeschränkten Counting-Turing-Maschinen berechenbar sind.*

Man sieht sofort ein, dass diese Klasse „mächtiger“ als NP ist. Eine nichtdeterministische Turing-Maschine, die eine Sprache aus NP entscheidet, kann nur entscheiden, ob es einen akzeptierenden Pfad gibt. Die Counting-Turing-Maschine hingegen kann sogar die Anzahl der akzeptierenden Pfade ausgeben. Somit kann

man mit ihrer Hilfe einfach die entsprechende nichtdeterministische Turing-Maschine simulieren. Es ist dabei wichtig zu erkennen, dass #P eine Klasse von Funktionen und keine Klasse von Entscheidungsproblemen ist. $\mathcal{NP} \subseteq \#P$ zu schreiben wäre also Unsinn. Sinnvoller wäre eine Aussage der Form $P^{\mathcal{NP}} = P^{\#P}$. Eine andere Aussage, die man über diese Klasse machen kann, ist, dass $\#P \subseteq \mathcal{PSPACE}$, wobei hier $\mathcal{PSPACE}$ die Menge der Probleme ist, zu deren Berechnung nur polynomiell viel Platz benötigt wird. Interessant ist auch, dass es ein #P-vollständiges Problem gibt: #SAT, welches einer aussagenlogischen Formel die Anzahl der erfüllenden Belegungen zuordnet.

3.5.2 Probabilistische Komplexitätsklassen

Die Grundidee, auf der die probabilistischen Klassen aufbauen, ist, anstelle nichtdeterministischer Verzweigungen probabilistische Verzweigungen zu verwenden. Es wird also jede der Entscheidungen mit einer gewissen Wahrscheinlichkeit getroffen. Eine solche Maschine akzeptiert dann mit einer Wahrscheinlichkeit, die der Anzahl der akzeptierenden Pfade geteilt durch die Anzahl die insgesamt möglichen Pfade entspricht, sofern sich die Maschine in einer entsprechenden Normalform befindet: Die Pfade müssen alle gleich lang sein und alle Verzweigungen müssen genau vom Grad zwei sein.

3.5.3 Die Klasse $\mathcal{RP}$

Definition 3.49 (Probabilistische Turing-Maschine)**.** *Eine* probabilistische Turing-Maschine *(kurz* PTM*) ist eine nichtdeterministische Turing-Maschine, die „ja“ ausgibt, wenn die Mehrheit der Pfade akzeptiert. Falls die Mehrheit der Pfade nicht akzeptiert, hat sie die Ausgabe „nein“. Bei exakt gleich vielen akzeptierenden und nicht akzeptierenden Pfaden hat sie die Ausgabe „weiß nicht“.*

Definition 3.50 (Klasse $\mathcal{RP}$)**.** *Die Klasse $\mathcal{RP}$ (*randomized polynomial time *umfasst die Entscheidungsprobleme, die von einer probabilistische Turing-Maschine in Polynomialzeit mit einseitigem Fehler entschieden werden können. Eine probabilistische Turing-Maschine M akzeptiert x, wenn die Mehrheit der Berechnungspfade x akzeptiert.*

Legt man diese Definition zugrunde, wird gefordert, dass, wenn ein akzeptierender Pfad existiert, mindestens die Hälfte ihrer Pfade akzeptiert. Es gilt $P \subseteq \mathcal{RP} \subseteq \mathcal{NP}$, wobei man hier jeweils vermutet, dass diese Inklusionen echt sind. Ähnlich wie bei der Klasse $\mathcal{NP}$ gibt es auch hier ein Asymmetrie: Die Entscheidung ist lediglich dann mit Sicherheit richtig, wenn eine Eingabe akzeptiert wird.

3.5.4 Polynomprodukt-Inäquivalenz

Ein gutes Beispiel für ein Problem, für das ein $\mathcal{RP}$-Algorithmus existiert, ist das Problem der Polynomprodukt-Inäquivalenz.

Definition 3.51 (Polynomprodukt-Inäquivalenz)**.** *Seien $\{P_1, \ldots, P_n\}$ und $\{Q_1, \ldots, Q_m\}$ Multimengen von multivariablen Polynomen. Ist $\prod_{i=1}^{n} P_i \neq \prod_{i=1}^{m} Q_i$?*

Die Antwort auf dieses Problem ist „ja“, wenn das Produkt der Polynome ungleich ist. Multipliziert man die Polynome einfach aus, steht man vor dem Problem, dass die Anzahl der Monome zu schnell steigen kann. Auch der Ansatz, die Polynome in ihre Primfaktoren zu zerlegen, schlägt fehl: Hier gibt es ebenso das Problem, dass die Anzahl der Monome immer noch zu schnell steigen kann. Als Beispiel hierfür betrachte man etwa die Faktorisierung von $(x^n - 1)$ für $n \in \mathbb{N}$.

3.5.4.1 Probabilistischer Ansatz

Will man dieses Problem probabilistisch angehen, besteht eine Möglichkeit darin, eine zufällige Zahl in die Polynomprodukte einzusetzen und zu überprüfen, ob das Ergebnis identisch ist. Findet man eine Eingabe, für das die Produkte unterschiedlich auswerten, wird das Ergebnis „ja“ ausgegeben. Ist das Ergebnis für beide Produkte gleich, ist es trotzdem möglich, dass die Polynomprodukte unterschiedlich sind, und der Algorithmus gibt „nein“ aus. Möglicherweise hat der Algorithmus gerade einen Schnittpunkt der beiden Produktpolynome erraten. Es ist wiederum die Asymmetrie der Klasse $\mathcal{RP}$ zu betonen.

3.5.4.2 co$\mathcal{RP}$

Für die Klasse co$\mathcal{RP}$, also die Klasse der Komplemente aller Sprachen aus $\mathcal{RP}$, ist diese Asymmetrie gerade invertiert. Betrachtet man etwa die Polynomäquivalenz (im Gegensatz zur Polynom*in*äquivalenz, die wir betrachtet haben), so kann man mit dem selben Algorithmus nur mit Sicherheit „nein“ – die Polynome sind ungleich – antworten. Es existieren also Zeugen für die Ungleichheit von Polynomen, aber man erkennt keine einfachen Zeugen für deren Gleichheit. Daher wird allgemein auch angenommen, dass $\mathcal{RP} \neq$ co$\mathcal{RP}$, aber auch diese Frage ist noch offen.

3.5.5 Die Klasse $\mathcal{ZPP}$

Definition 3.52 (Klasse $\mathcal{ZPP}$)**.** $\mathcal{ZPP} = \mathcal{RP} \cap co\mathcal{RP}$.

$\mathcal{ZPP}$ (*zero-error probabilistic polynomial time*) ist also die Klasse der Probleme, für die es sowohl Algorithmen gibt, die sicher „ja“ ausgeben, als auch solche, die sicher „nein“ ausgeben. Probleme aus $\mathcal{ZPP}$ sind also in erwartet polynomieller Zeit mit definitiv korrekter Antwort entscheidbar.

3.5.5.1 Kasinos

In Anlehnung daran, dass man auch in Kasinos ein Risiko eingeht (auch wenn dieses erheblich höher ist), werden randomisierten Algorithmen im Sinne von Definition 3.50 als „Monte-Carlo-Algorithmen“ bezeichnet. Diese Klasse von Algorithmen, die Probleme aus $\mathcal{ZPP}$ lösen, also Algorithmen, die in erwartet polynomieller Zeit laufen und, wenn sie terminieren, auch mit Sicherheit das richtige Ergebnis liefern, heißen entsprechend „Las-Vegas-Algorithmen“.

3.5.6 $\mathcal{PP}$ und $\mathcal{BPP}$

Die Klasse von Problemen, die wir nun betrachten, sind die eingangs erwähnten, die heutzutage als mit sinnvollem Zeitaufwand berechenbar gelten. Ob dies so bleibt, ist natürlich nicht sicher, da ja auch die Frage, ob $\mathrm{P} \stackrel{?}{=} \mathcal{NP}$ noch nicht geklärt ist.

3.5.6.1 Normalform

Wie schon erwähnt müssen die probabilistischen Turing-Maschinen, die wir im Folgenden betrachten, noch normalisiert werden. Diese Normalisierung garantiert, dass alle Verzweigungen den Verzweigungsgrad zwei haben und alle Pfade gleich lang sind. Im Wesentlichen sorgt dies also dafür, dass solch eine Maschine als vollständiger binärer Baum dargestellt werden kann. So können wir auch von Pfadanzahlen anstatt von Wahrscheinlichkeiten sprechen.

3.5.6.2 $\mathcal{PP}$

Definition 3.53 (Klasse $\mathcal{PP}$)**.** *Die Klasse $\mathcal{PP}$ (*probabilistisch polynomiell*) enthält die Menge aller Entscheidungsprobleme, die von einer probabilistischen Turing-Maschine in polynomieller Zeit gelöst werden.*

Auch wenn diese Klasse schon eher das ist ist, was wir haben möchten, so können wir sie noch nicht direkt verwenden. Man betrachte folgende Überlegung: Sei ϕ eine aussagenlogische Formel, also eine SAT-Instanz. Nun führen wir ein weiteres Literal x_{neu} ein. Das Entscheidungsproblem $\phi \wedge x_{neu}$ liegt nun in $\mathcal{PP}$. Damit ist also $\mathcal{NP} \subseteq \mathcal{PP}$. Wir suchen aber realistisch berechenbare Funktionen, $\mathcal{NP}$ ist uns jedoch schon „zu schwierig“ – wir haben also nichts gewonnen. Problematisch ist, dass wir lediglich verlangen, dass die Mehrheit der Pfade akzeptiert. Diese Aussage ist zu schwach, denn sind immer lediglich minimal mehr akzeptierende Pfade als nicht akzeptierende vorhanden, so wird deren Mehrheit mit steigender Pfadzahl immer unbedeutender.

3.5.6.3 $\mathcal{BPP}$

Die Klasse, auf die wir eigentlich hinaus wollten, ist also nicht $\mathcal{PP}$. Stattdessen definieren wir die Klasse $\mathcal{BPP}$. Dabei steht $\mathcal{B}$ für „bounded“, also beschränkt. Wir beschränken die Anzahl der akzeptierenden Pfade nach unten.

Definition 3.54 (Klasse $\mathcal{BPP}$)**.** *$\mathcal{BPP}$ besteht aus den Entscheidungsproblemen, die durch eine probabilistische Turing-Maschine entscheidbar sind, die mit einer Wahrscheinlichkeit $\frac{1}{2} + \delta$ mit $\delta > 0$ die richtige Antwort gibt.*

Hier wählen wir also eine Konstante δ, die dafür sorgt, dass die Anzahl der akzeptierenden Pfade mit der Gesamtzahl der Pfade hinreichend stark steigt. Erkenntnisse über probabilistische Algorithmen der probabilistischen Turing-Maschinen aus dieser Klasse erlauben es nun, endlich sinnvolle Algorithmen zu finden.

3.5.6.4 $\mathcal{RP}$, $\mathcal{NP}$ und $\mathcal{BPP}$.

Wie verhält sich nun $\mathcal{BPP}$ zu den bisherigen Komplexitätsklassen, insbesondere zu $\mathcal{NP}$? Klar ist, dass z. B. $\mathcal{P}$ komplett in $\mathcal{BPP}$ liegt, da die Klasse $\mathcal{BPP}$ die probabilistische Funktionalität nicht verwenden muss. Indizien sprechen dafür, dass sowohl $\mathcal{BPP} \not\subseteq \mathcal{NP}$ als auch $\mathcal{BPP} \not\supseteq \mathcal{NP}$ gilt. Weiterhin ist es wahrscheinlich, dass $\mathcal{RP} \neq \text{co}\mathcal{RP}$ gilt.

Die Klassen verhalten sich also vermutlich wie in Abbildung 3.21 dargestellt.

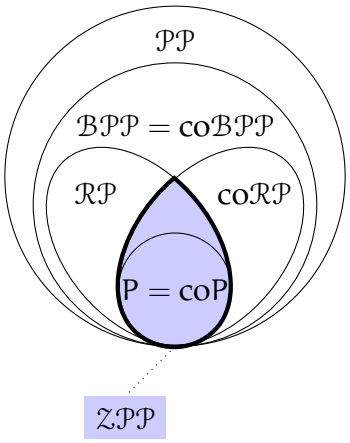


Abbildung 3.21: Komplexitätsklassen in der Übersicht.

3.6 Weitere Klassen

Es gibt viele weitere Komplexitätsklassen, unter anderem etwa

- $\mathcal{EXPTIME}$. So wie $\mathcal{P}$, nur dass man verlangt, dass die Turing-Maschine in $2^{p(|x|)}$ Schritten terminiert.
- $\mathcal{PSPACE}$. Die Sprachen, die mit polynomiellem Platzbedarf entscheidbar sind.
- $\mathcal{IP}$. Die Sprachen, für die ein interaktiver Beweis polynomieller Länge existiert.

Dabei ist zum Beispiel schon gezeigt worden, dass $\mathcal{IP} = \mathcal{PSPACE}$. Es gibt unzählige weitere Beispiele, für die meisten ist noch nicht viel gezeigt worden, mehr dazu findet man im Complexity Zoo.

4. Informationstheorie

4.1 Was ist Information?

Die Informationstheorie befasst sich mit Informationen. Claude Shannon führte zuerst den Informationsbegriff ein, zunächst als „mathematical theory of communication“, also als Theorie der Kommunikation.

Betrachten wir zunächst eine einfache Situation, um den Begriff zu illustrieren: Gegeben sei eine Zufallsquelle, der Sender, der Nachrichten verschickt. Die Zufallsquelle könnte z. B. eine Münze sein, die immer wieder geworfen wird. Weiterhin gibt es einen Beobachter, den Empfänger, der die Nachrichten des Senders beobachtet.

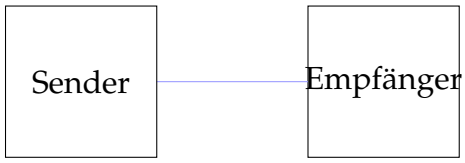


Abbildung 4.1: Sender und Empfänger

Der Shannonsche Informationsbegriff ist nun ein Maß der „Überraschung“, die der Empfänger beim Erhalt der Nachrichten des Senders hat. Das heißt, dass vorhersehbare Nachrichten wenig Information haben, während Nachrichten, die unerwartet sind, also eine geringe Auftrittswahrscheinlichkeit haben, einen hohen Informationsgehalt haben.

4.1.1 Anwendungen

Im Rahmen der Übertragung von Information findet die Informationstheorie ihre praktische Anwendung. Einige Verwendungsbereiche sollen exemplarisch vorgestellt werden:

- Bei der *Quellcodierung* werden die Signale beim Sender an der Quelle, meistens zum Erreichen einer höheren Informationsdichte (also Kompression), codiert. Beispiele für bekannte Kompressionsverfahren sind ZIP oder GZIP.
- Die *Kanalcodierung* dient dazu, Übertragungsfehler bei der Übertragung über einen gestörten Kanal auf Empfänger-Seite beheben zu können. Die Informationsdichte wird also absichtlich verringert. Ein einfaches Beispiel stellt der „Triple Repetition Code“ dar, bei dem jedes Bit dreimal übertragen wird: So kann ein „gekipptes“ Bit pro Dreier-Block korrigiert werden.
- In der *Kryptographie* sucht man Verschlüsselungsalgorithmen, bei denen der Angreifer ohne Schlüssel aus dem Chiffrat möglichst wenig Informationen über den Klartext erhält.

4.1.2 Formale Definitionen

Nach diesen Vorüberlegungen möchten wir nun die Information I für ein Zeichen, das mit Wahrscheinlichkeit p vorkommt, definieren. Sinnvolle Forderungen hierbei sind:

1. Information soll nicht negativ sein: $I_p \geqslant 0$.
2. Ein sicheres Ereignis (also wenn $p = 1$) soll keine Information liefern.
3. Für zwei unabhängige Ereignisse soll sich die Information summieren: $I_{(p_1 \cdot p_2)} = I_{p_1} + I_{p_2}$.
4. Die Information soll robust sein, kleine Änderungen an der Wahrscheinlichkeit sollen also nur kleine Änderungen an der Information bewirken.

Damit ergibt sich folgende Definition für Information:

Definition 4.1 (Information). $I_p = \log_b(\frac{1}{p}) \ \left[= -\log_b(p)\right]$

Im Folgenden sei die Basis $b = 2$.

Beispiel 28.
Das Werfen einer Münze ergibt jeweils mit der Wahrscheinlichkeit $p = \frac{1}{2}$ Kopf bzw. Zahl. Betrachtet man die Information, die ein Wort w über $\Sigma = \{0, 1\}$ mit $|w| = n$ enthält, das durch n-fachen Münzwurf erzeugt wurde, ergibt sich die Wahrscheinlichkeit für ein beliebiges, aber festes Wort mit $p = \frac{1}{2^n}$ und damit $I = -\log_2(\frac{1}{2^n}) = n$.

4.1.3 Der Entropiebegriff

Nachdem wir Information formal definiert haben, betrachten wir den Begriff der Entropie, also ein Maß für die Informationsdichte.

Information. Exkurs: Ursprung des Entropiebegriffs

Der Entropiebegriff hat seinen Ursprung in der Physik. Der informationstheoretische Entropiebegriff ist eng mit dem physikalischen verwandt, die Definition ist identisch. Man betrachte etwa ein sich ausbreitendes Gas:

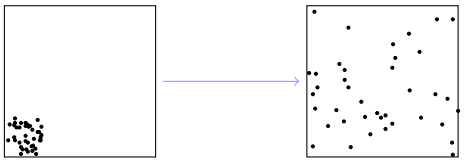


Abbildung 4.2: Entropie nimmt zu

Hier sagt der zweite Hauptsatz der Thermodynamik aus, dass sich das linke System nur zum rechten hin verändern kann und nicht umgekehrt, also die Entropie nur zunehmen kann.

Übrigens ... Maxwells Dämon

Den Zweiten Hauptsatz der Thermodynamik könnte man dadurch verletzen, dass man einen gasgefüllten Raum mit zwei Kammern konstruiert, zwischen denen eine reibungsfreie Schiebetür ist, die ein kleiner Dämon immer dann aufmacht, wenn ein schnelles Teilchen von links oder ein langsames Teilchen von rechts kommt. Damit sortiert er die Teilchen nach ihrer Geschwindigkeit, also Temperatur.

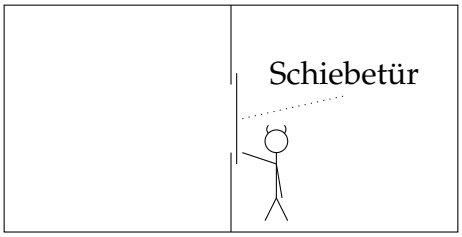


Abbildung 4.3: Maxwells Dämon

Man könnte nun aus der gewonnenen Temperaturdifferenz scheinbar Energie gewinnen (z.B. mit einem Stirling-Motor). Eine mögliche Folgerung daraus ist, dass Datenverarbeitung (hier bspw. das Sortieren der Teilchen) Energie benötigt.

Definition 4.2 (Entropie). *Die Entropie einer diskreten Zufallsvariable X ist (analog zum physikalischem Begriff) definiert durch*

$$H(X) = \sum_{x \in X} p(x) \log(\frac{1}{p(x)}) = \mathbb{E}I(x).$$

Dabei gelten die folgenden Konventionen:

$$0 \cdot \log(0) = 0,\ 0 \cdot \log(\frac{0}{0}) = 0,\ a \cdot \log(\frac{a}{0}) = \infty.$$

Dabei gilt immer $H(X) \geqslant 0$. Für eine andere Basis $b \neq 2$ ergibt sich $H_b = \log_b(2) \cdot H(X)$. Die Entropie wird also bei einem Basiswechsel nur mit einem konstanten Faktor multipliziert. **Vorsicht**, H_α bezeichnet hingegen oft die Rényi-Entropie.

Beispiel 29.
Sei $X = 1$ mit Wahrscheinlichkeit p und $X = 0$ mit Wahrscheinlichkeit $(1-p)$. Dann ist $H(X) = -p\log(p) - (1-p)\log(1-p)$.

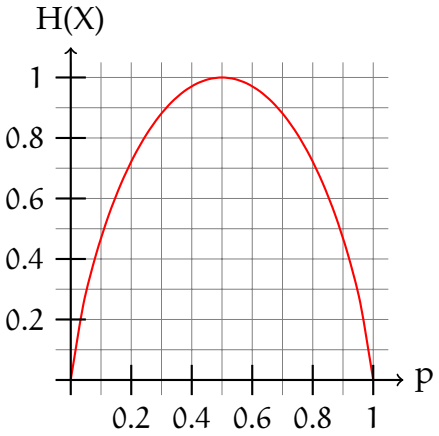


Abbildung 4.4: Entropie von X in Abhängigkeit von p.

Definition 4.3 (Gemeinsame Entropie, Bedingte Entropie)**.** *Die* gemeinsame Entropie *der Zufallsvariablen X, Y mit der gemeinsamen Verteilung $p(x,y)$ ist definiert durch*

$$H(X,Y) = \sum_{x\in X}\sum_{y\in Y} p(x,y)\log(\frac{1}{p(x,y)}).$$

Die bedingte Entropie *der Zufallsvariable Y in Abhängigkeit von X mit gemeinsamer Verteilung $p(x,y)$ ist*

$$\begin{aligned} H(Y|X) &= \sum_{x\in X} p(x)H(Y|X=x) \\ &= -\sum_{x\in X} p(x)\sum_{y\in Y} p(y|x)\log(p(y|x)) \\ &= -\sum_{x\in X, y\in Y} p(x,y)\log(p(y|x)). \end{aligned}$$

Korollar 4.4. *Es gilt*

$$H(X,Y) = H(X) + H(Y|X) = H(Y) + H(X|Y).$$

Beweis. Es gilt $\log(p(x,y)) = \log((p(x)p(y|x))) = \log(p(x)) + \log(p(y|x))$. □

Daraus kann man folgern, dass $H((X,Y)|Z) = H(X|Z) + H(Y|X,Z)$. Achtung: im Allgemeinen gilt $H(X|Y) \neq H(Y|X)$.

4.1.4 Transinformation

Der Teil der ursprünglichen Information, der tatsächlich bei der Quelle ankommt, wird als Transinformation bezeichnet.

Definition 4.5 (Transinformation)**.** *Die* Transinformation *ist definiert durch*

$$\begin{aligned} I(X;Y) &= H(X) - H(X|Y) \\ &= H(Y) - H(Y|X). \end{aligned}$$

Alle vorgestellten Entropien und die Transinformation kann man in einem Schaubild in Beziehung setzen (vgl. Abbildung 4.5). Von der Entropie der Quelle $H(X)$ geht ein Teil verloren, die sog. Äquivokation $H(X|Y)$. Gleichzeitig kommt durch das Rauschen auf dem Kanal auch neue Entropie in Form von Fehlinformation $H(Y|X)$ hinzu. Alle bei der Senke ankommenden Informationen bilden die Entropie $H(Y)$, doch nur die Transinformation $I(X;Y)$ ist der Teil, der tatsächlich von der Quelle gesendet wurde. Die Totalinformation $H(X,Y)$ beschreibt alles, was über den Kanal gesendet wird.

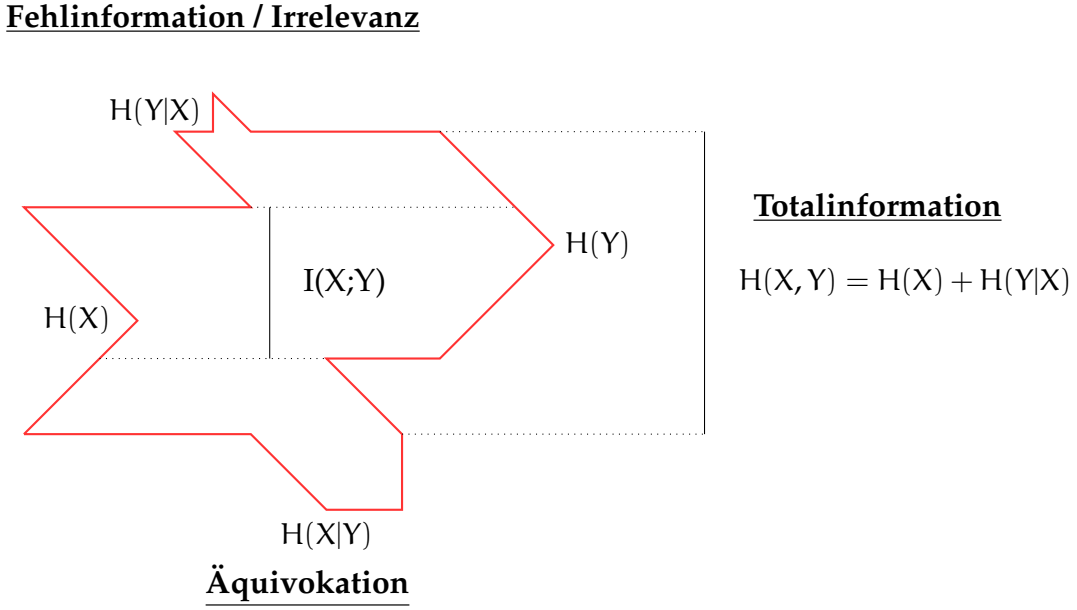


Abbildung 4.5: Zusammenhang zwischen Transinformation und Entropie.

4.2 Die Kolmogorow-Komplexität

Nachdem wir Entropie und Information im informationstheoretischen Sinn definiert haben, widmet sich die Kolmogorow-Komplexität der Frage nach der kürzesten Repräsentation von Information, insbesondere im Zusammenhang mit Algorithmen und Turing-Maschinen. Die Idee ist, Information als die kürzeste konkrete Repräsentation eines Objekts aufzufassen. Dies wird auch als algorithmischer Informationsgehalt oder Beschreibungskomplexität bezeichnet.

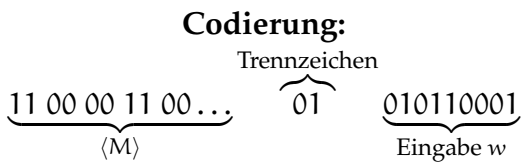


Abbildung 4.6: Für $x = \langle M \rangle w$ kann sich das Problem ergeben, dass nicht klar ist, wo $\langle M \rangle$ aufhört und w beginnt. Deshalb wird bei der Darstellung von $\langle M \rangle$ jedes Bit doppelt ausgegeben und „01“ als Trennsymbol verwendet.

Ähnlich wie in Kapitel 2.1 vorgestellt, gehen wir in diesem Abschnitt von einer binären Repräsentation $\langle M \rangle$ der Turing-Maschine M aus. $\langle M \rangle$ repräsentiert allerdings nicht notwendigerweise eine Gödelnummer, sondern eine beliebige, aber feste Codierung. Bei Konkatenation $\langle M \rangle w$ von $\langle M \rangle$ mit einem binären Wort w ist nicht klar, wo die Darstellung von M aufhört und wo w beginnt. Deshalb wird ab sofort für $\langle M \rangle$ jedes Bit doppelt ausgegeben und 01 als Trennsymbol zu w verwendet, siehe Abbildung 4.6.

Definition 4.6 (Kolmogorow-Komplexität)**.** *Es sei x ein binäres Wort. Die minimale Beschreibung $d(x)$ von x ist die kürzeste Zeichenkette $\langle M \rangle w$, für die die Turing-Maschine M mit Eingabe w unter Ausgabe von x auf dem Band hält. Falls mehrere solcher Zeichenketten existieren, wird die lexikographisch erste davon gewählt. Die Kolmogorow-Komplexität $K(x)$ ist $K(x) = |d(x)|$.*

Beispiel 30.
Die naive Darstellung der Zeichenkette 0^n würde n Zeichen benötigen. Es geht jedoch auch kürzer:

```
for i = 0 to n-1
print("0")
```

Für hinreichend große n benötigt diese Pseudocode-Darstellung weniger als n Zeichen.

Intuitiv sollte die Kolmogorow-Komplexität eines Worts nicht viel größer sein als seine Darstellung. Dies ist auch der Fall.

Satz 4.7. *Es gibt ein c, sodass für alle $x \in \{0,1\}^*$ gilt: $K(x) \leqslant |x| + c$.*
Die Kolmogorow-Komplexität für jedes Wort x ist also höchstens konstant größer als $|x|$, c ist universal für alle Wörter.

Beweis. Es sei M die Turing-Maschine, die bei der Eingabe w das Wort w ausgibt und c die Länge von $\langle M \rangle$ in der hier verwendeten Codierung. M ist also die Turing-Maschine, die die Identität berechnet. □

Wenn das gleiche Wort doppelt ausgegeben wird, wächst die Kolmogorow-Komplexität nur um einen konstanten Wert im Vergleich zur einfachen Ausgabe.

Satz 4.8. *Es gibt ein c, sodass für alle $x \in \{0,1\}^*$ gilt: $K(xx) \leqslant K(x) + c$.*

Beweis. Es sei M die Turing-Maschine, die als Eingabe $\langle N\rangle w$ erwartet, wobei N eine Turing-Maschine ist und w die Eingabe für N. M führt N auf w aus, liest das Ergebnis x und gibt xx aus. Die Beschreibung von xx ist $\langle M\rangle d(x)$, wobei $|\langle M\rangle|$ eine Konstante c ist und $|d(x)|$ die Kolmogorow-Komplexität von x. Damit ergibt sich $K(xx) \leqslant K(x) + c$. □

Es stellt sich die Frage, inwiefern die Kolmogorow-Komplexität von der Beschreibungssprache abhängt. Gibt es Beschreibungen, die signifikant besser geeignet sind als die auf Turing-Maschinen basierenden? Man kann zeigen, dass dies nicht der Fall ist.

Satz 4.9. *Für jede Beschreibungssprache* p *existiert eine Konstante* c *mit*

$$\forall x : K(x) \leqslant K_p(x) + c.$$

Beweis. Idee: Es gibt eine Turing-Maschine, die die Beschreibungssprachen ineinander übersetzt. Diese Maschine hat konstante Größe und kann Teil der Beschreibung sein. □

Satz 4.10. *Für alle* n *gibt es unkomprimierbare Wörter der Länge* n.

Beweis. Über dem Alphabet $\{0, 1\}$ gibt es genau 2^n Wörter der Länge n. Für alle kürzeren Beschreibungen w' von einem Wort w mit $|w| = n$ gilt offensichtlich $|w'| \leqslant n-1$. Da es aber nur $\sum_{i=0}^{n-1} 2^i = 1+2+4\cdots+2^{n-1} = 2^n - 1$ Wörter mit Länge $\leqslant n-1$ gibt, gibt es mindestens ein w mit $|w| = n$, für das es keine Beschreibung mit kürzerer Länge gibt. □

Fast alle Wörter sind nicht komprimierbar. Dies gilt insbesondere für zufällige Zeichenketten.

4.3 Quellcodierungen

4.3.1 Verlustbehaftete Kompression

Bei Quellcodierungen, also Kompressionsverfahren, unterscheidet man grundsätzlich zwischen verlustbehafteter und verlustfreier Kompression. Verlustbehaftete Kompressionsverfahren sind, wie der Name schon sagt, Kompressionsverfahren, bei denen Information verloren geht. Typische Beispiele für solche Verfahren sind etwa JPEG 2000 oder MP3. Ein solches Verfahren lässt dabei üblicherweise „unwichtige“ Informationen weg und übernimmt nur die wesentlichen Merkmale. Solche Verfahren sind jedoch nicht Gegenstand dieser Vorlesung und wir konzentrieren uns daher auf verlustfreie Kompression.

4.3.2 Codierungen

Aus den Definitionen von Information und Entropie und allgemeinen Überlegungen können wir einige Anforderungen an eine Quellcodierung stellen. Zum einen muss die Länge der Codierung eines Zeichens abhängig von der Häufigkeit des Auftretens dieses Zeichens sein. Weiterhin muss der Code eindeutig decodierbar sein, wenn die Codierungen eines Zeichens nicht-feste Länge haben (vgl. Beispiel 31).

Beispiel 31.

Zeichen	A	B	C	D
Auftrittswahrscheinlichkeit	$\frac{1}{16}$	$\frac{3}{16}$	$\frac{3}{16}$	$\frac{9}{16}$
Codierung	0	01	10	11

Hier ist ADA $\widehat{=} 0110 =$ BC, die Codierung ist also nicht eindeutig.

4.3.3 Präfix-Codes

Eindeutigkeit ist also eine Eigenschaft, die schon bei der Code-Konstruktion beachtet werden muss. Eine Klasse von Codes, die eindeutig ist, ist die der sogenannten Präfix-Codes[1].

Definition 4.11 (Präfix-Code). *Sei* $c = c_1 \dots c_n$ *eine Codierung. Wenn für alle* $c' = c_1 \dots c_l$ $(1 \leqslant l < k)$ *gilt, dass* c' *keine Codierung ist, heißt* c *präfixfrei oder Präfix-Code.*

Ein Codewort darf also kein Präfix besitzen, das selbst wieder ein Codewort ist. Daraus folgt, dass man solche Codierungen eindeutig von links nach rechts lesend decodieren kann. Ein Präfix-Code wird durch einen Baum repräsentiert: Wenn man ein Wort binär codieren will, so verwendet man einen binären Baum. Dessen Blätter sind die zu codierenden Zeichen, die Äste werden, wenn sie nach links verzweigen mit 0, nach rechts mit 1 beschriftet. Die Zeichen auf dem Ästen vom Wurzelknoten zum Blatt stellen die Codierung des Zeichens dar. Ein Präfix-Code für die Zeichen $\{\sigma_1, \dots, \sigma_7\}$ lässt sich beispielsweise durch den Baum in Abbildung 4.7 darstellen.

[1] Sinnvoller mag der Begriff *präfix-freier Code* sein, der allerdings in der Literautur nicht verwendet wird.

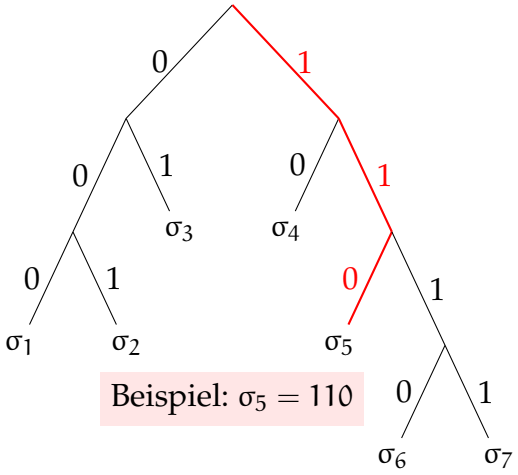


Abbildung 4.7: Ein Präfix-Code.

Dieses Verfahren berücksichtigt aber noch nicht die Auftrittswahrscheinlichkeit der einzelnen Zeichen, weshalb eine solche Codierung in der Regel nicht optimal ist.

Information.

Man unterscheidet zwischen *gedächtnislosen Quellen*, bei denen die Auftrittswahrscheinlichkeit eines Zeichens nicht von den vorhergehenden abhängt, und *Quellen mit Gedächtnis*, bei denen dies der Fall ist.

Betrachten wir nun wieder die folgenden vier Zeichen und ihre Auftrittswahrscheinlichkeit:

Zeichen	A	B	C	D
Auftrittswahrscheinlichkeit	$\frac{1}{16}$	$\frac{3}{16}$	$\frac{3}{16}$	$\frac{9}{16}$

Jedes Zeichen ist hier mit 2 Bit codiert, der Erwartungswert für die codierte Länge eines Zeichens ist also demnach 2. Man findet aber bessere Codierungen, etwa:

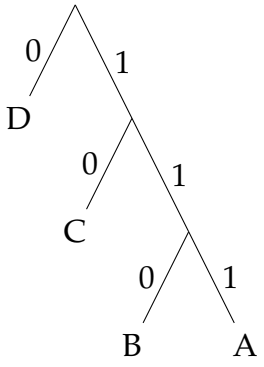


Abbildung 4.8: Eine optimale Codierung.

Unter Berücksichtigung der neuen Codierungslängen der Zeichen berechnet sich der Erwartungswert für die Codierungslänge eines einzelnen Zeichens nun als $\frac{1}{16} \cdot 3 + \frac{3}{16} \cdot 3 + \frac{3}{16} \cdot 2 + \frac{9}{16} \cdot 1 = \frac{27}{16} < 2$. Im Vergleich zu Beispiel 31 ist diese Codierung präfixfrei.

Wir stellen im Folgenden zwei Codierungen vor, von denen eine optimal bezüglich des Erwartungswerts der Zeichenlänge ist.

4.3.3.1 Die Shannon-Fano-Codierung

Die Shannon-Fano-Codierung wird nach folgendem Muster erstellt:

1. Sortiere die vorkommenden Symbole absteigend nach ihrer Häufigkeit.
2. Teile die Symbole in zwei Teilgruppen auf, sodass die aufsummierten Wahrscheinlichkeiten beider Gruppen möglichst gleich sind.
3. Hänge nun die beiden entstehenden Gruppen als Blätter an eine neue Wurzel.
4. Verfahre nun rekursiv: Ersetze die Gruppen solange jeweils durch den Baum, der beim Anwenden des Verfahrens auf sie jeweils entsteht, bis alle Blätter einzelne Zeichen sind.

Der resultierende Baum ist die Codierung, die allerdings nicht immer optimal ist.

Beispiel 32.
Wir wollen nun anhand der folgenden Verteilung eine Codierung mittels des Shannon-Fano-Verfahrens bestimmen.

Symbol	A	B	C	D	E
Absolute Häufigkeit	15	7	6	6	5

Es ergibt sich folgender Baum:

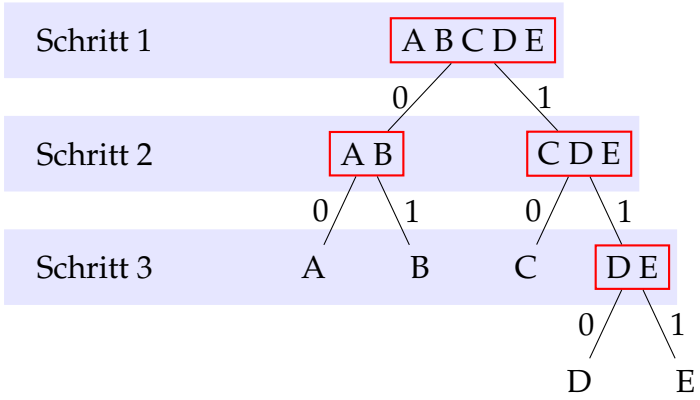


Abbildung 4.9: Shannon-Fano-Codierung.

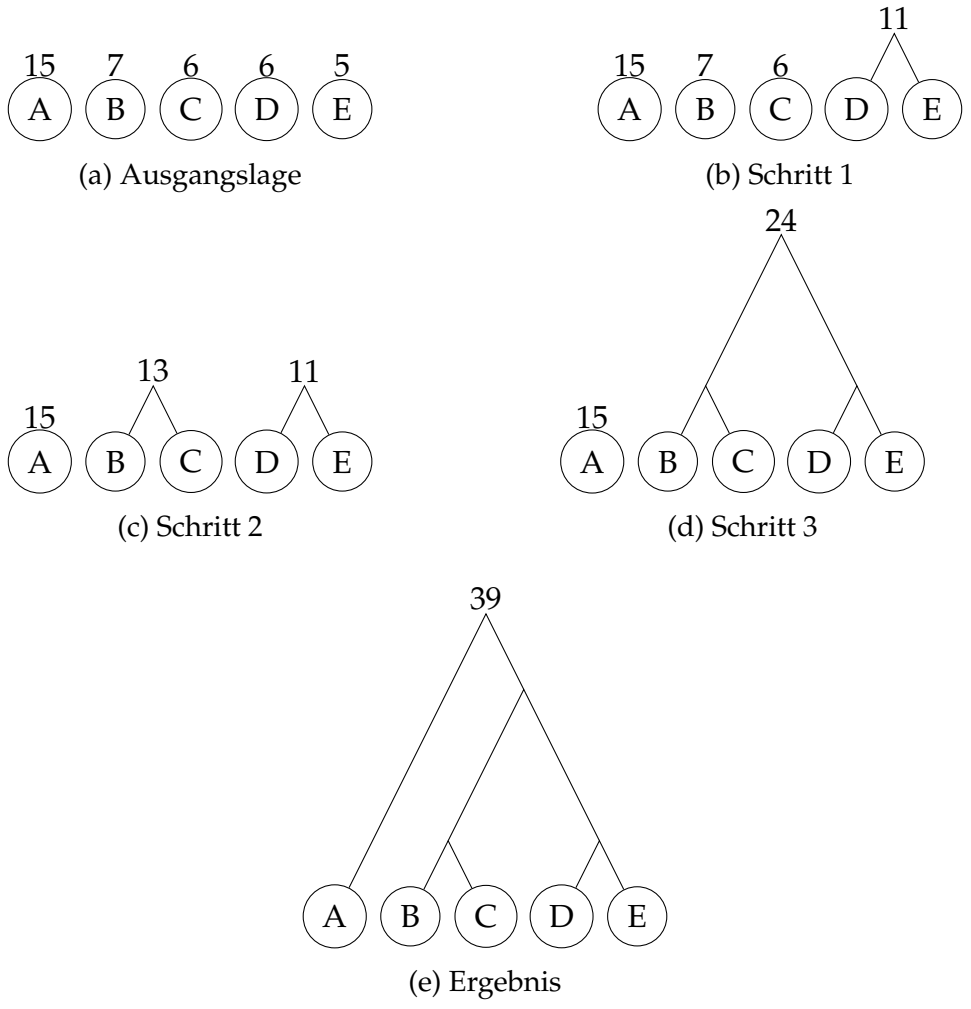
Für die erwartete Länge eines Zeichens ergibt sich $\frac{15}{39} \cdot 2 + \frac{7}{39} \cdot 2 + \frac{6}{39} \cdot 2 + \frac{6}{39} \cdot 3 + \frac{5}{39} \cdot 3 = \frac{89}{39}$.

4.3.3.2 Der Huffman-Code

Der Erwartungswert für die codierte Länge eines Zeichens in Beispiel 32 ist jedoch nicht optimal. Diese Forderung wird von Huffman-Codes, die mit folgendem Algorithmus erzeugt werden können, erfüllt:

1. Erstelle aus jedem der Symbole einen Baum, der genau dieses Symbol als einzigen Knoten enthält. Diese Bäume bilden einen Wald.
2. Wiederhole die folgenden Schritte solange, bis der Wald nur noch ein einziger Baum ist:
   - Wähle die beiden Bäume, deren summierte Auftrittswahrscheinlichkeit aller Blätter am kleinsten ist.
   - Erstelle einen neuen Baum mit neuer Wurzel und den Wurzeln der beiden Bäume aus dem vorherigen Schritt als Kinder.

Beispiel 33.
Wir betrachten nun die Huffman-Codierung für die Eingabe aus Beispiel 32:



(a) Ausgangslage (b) Schritt 1 (c) Schritt 2 (d) Schritt 3 (e) Ergebnis

Abbildung 4.10: Konstruktion eines Huffman-Baums.

Die erwartete Länge eines Zeichens ist hier $\frac{15}{39} \cdot 1 + \frac{7}{39} \cdot 2 + \frac{6}{39} \cdot 2 + \frac{6}{39} \cdot 3 + \frac{5}{39} \cdot 3 = \frac{74}{39} < \frac{89}{39}$.

Satz 4.12. *Huffman-Codes sind optimal, d. h. der Erwartungswert für die Zeichenlänge der Codierung ist minimal.*

Beweis. Sei $\Sigma = \{\sigma_0 \ldots, \sigma_{n-1}\}$ das Alphabet, für das der Code erstellt werden soll. Die Auftrittswahrscheinlichkeit eines Zeichens σ bezeichnen wir mit $p(\sigma)$, die Länge des Pfades zu σ, also die Codierungslänge, mit $l(\sigma)$. Es gilt, folgenden Term zu minimieren:

$$\mathbb{E} = \sum_{\sigma \in \Sigma} p(\sigma) \cdot l(\sigma)$$

Um zu zeigen, dass dieser Term bei einem Huffman-Baum minimal ist, zeigen wir drei Lemmata:

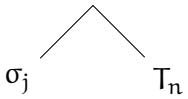
Lemma 4.13. *Jeder innere Knoten in einem optimalen Baum hat zwei Kind-Knoten.*

Beweis. Angenommen, ein innerer Knoten hat nur einen Kind-Knoten. Dann könnte man den inneren Knoten einfach durch den Kind-Knoten ersetzen und verringerte damit insgesamt den Erwartungswert $\mathbb{E}$. Das ist jedoch ein Widerspruch zur Annahme, dass der Baum optimal ist. ↯

□

Lemma 4.14. *Wenn σ_j und σ_k die beiden Symbole mit der kleinsten Auftrittswahrscheinlichkeit sind, so haben sie in einem optimalen Baum denselben Vater-Knoten.*

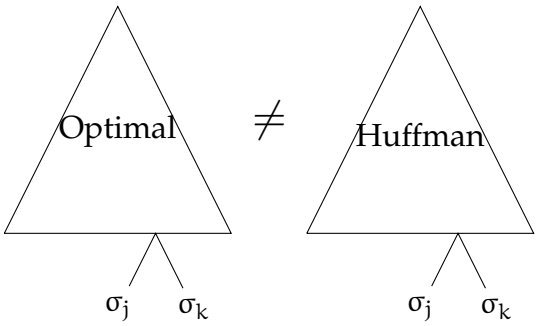
Beweis. Es sei o. B. d. A. $l(\sigma_j) \geqslant l(\sigma_k)$ und σ_j habe im Baum den Nachbarbaum T_n, also



Die Summe der Wahrscheinlichkeiten von T_n ist definitionsgemäß größer als oder gleich groß wie die von σ_k (da σ_j und σ_k die geringste Auftrittswahrscheinlichkeit haben). Vertauscht man nun einfach T_n mit σ_2, so ist der Erwartungswert für die Zeichenlänge der Codierung beim so resultierenden Baum kleiner als oder gleich groß wie vorher. Der alte Baum kann also nicht der optimale gewesen sein. ↯ □

Lemma 4.15. *Der Huffman-Baum ist optimal.*

Beweis. Es seien σ_j und σ_k die beiden Symbole mit der geringsten Auftrittswahrscheinlichkeit und o. B. d. A. $l(\sigma_j) \geqslant l(\sigma_k)$. Nehmen wir also an der optimale Baum wäre nicht aus dem oben beschrieben Verfahren hervorgekommen. Untersuchen wir dann das kleinste Gegenbeispiel.



Nun ersetzen wir den Teilbaum $\overset{\sigma_j}{\wedge}\overset{\sigma_k}{}$ in beiden Bäumen durch ein σ' mit $p(\sigma') = p(\sigma_j) + p(\sigma_k)$. Der Teilbaum $\overset{\sigma_j}{\wedge}\overset{\sigma_k}{}$ existiert, da σ_j und σ_k nicht auf unterschiedlicher Höhe hängen dürfen, wenn sie aber auf gleicher Höhe hängen und nicht benachbart sind, so kann man ohne Beeinflussung von $\mathbb{E}$ den Baum so umhängen, dass sie benachbart sind. Wir haben nun unter der Annahme, dass das oben das kleinste Gegenbeispiel war, ein kleineres konstruiert (da Huffman *und* optimaler Baum sich bei der Ersetzung nicht ändern). Das widerspricht der Annahme, dass es überhaupt ein Gegenbeispiel gibt. $\lightning$ □

□

4.4 Kanalcodierungen

Nun wollen wir uns mit Kanalcodierungen befassen. Das sind im Allgemeinen fehlererkennende oder -korrigierende Codes, die verwendet werden, um bei der Übertragung auftretende Fehler wie beispielsweise zufällige gekippte Bits zu erkennen und zu korrigieren.

Man unterscheidet zwischen zwei Klassen von Codes:

- Block-Codes: Codewörter haben feste Länge und aufeinanderfolgende Blöcke werden unabhängig voneinander codiert.
- Faltungs-Codes: Es wird ein Teil der bereits codierten Zeichen im Puffer gehalten, und entsprechend diesen ein neu eingelesenes Zeichen codiert. Diese Art von Codierung ist besonders für kontinuierliche Quellen wie Radio geeignet.

Wir befassen uns im Folgenden nur mit Block-Codes.

4.4.1 Block-Codes

Block-Codes haben folgende Eigenschaften:

- Der Code ist immer über einem endlichen Alphabet $\mathbb{Q}$, wobei vor allem $\mathbb{Q} = \{0, 1\}$ interessant ist.

- Ein Block-Code ist eine Teilmenge $C \subseteq Q^n$ für ein $n \in \mathbb{N}$, d. h. alle Blöcke haben dieselbe Länge.

Für $|C| = 1$ heißt C trivial, da es nur ein Codewort gibt. Für $|Q| = 2$ heißt C binär, für $|Q| = 3$ terniär, etc. Sei p eine Primzahl und $q = p^r$ $(r \in \mathbb{N})$. $\mathbb{F}_q$ ist ein endlicher Körper mit q Elementen. Wir betrachten im Folgenden nur $\mathbb{F} = \mathbb{F}_q = \mathbb{F}_2 = \{0, 1\}$. Sei also $\mathbb{F} = \mathbb{F}_2$. Man sieht, dass $\mathbb{F}_2^n$ gerade die Bitvektoren, also binären Wörter, der Länge n enthält.

Es stellt sich die Frage, wie ein Codewort wieder decodiert werden kann, wenn es bei der Übertragung einen Fehler gegeben hat. Ein Ansatz ist Maximum-Likelihood-Decoding.

Definition 4.16 (Maximum-Likelihood-Decoding). *Für eine Nachricht $x \in \mathbb{F}_2^n$ wählt das Maximum-Likelihood-Decoding das Codewort $y \in C$ aus, sodass*
$\Pr[y \text{ gesendet} | x \text{ empfangen}]$ *maximiert wird.*

In unserem Fall bedeutet dies, dass das Codewort gewählt wird, für das der Abstand zur übertragenen Nachricht minimal ist. Dazu führen wir mit der Hamming-Distanz eine Metrik ein, welche den Unterschied zwischen zwei Codewörtern festlegt und eine Quantifizierung von Fehlern erlaubt.

Definition 4.17 (Hamming-Distanz). *Für $x, y \in \{0, 1\}^n$ ist*

$$d(x, y) := \sum_{i=1}^{n} (1 - \delta_{x_i, y_i}) = |\{i \mid i = 1, \ldots, n, x_i \neq y_i\}|$$

die Hamming-Distanz zwischen x und y.

Damit is die Hamming-Distanz zwischen x und y die Anzahl der Zeichen in x, die sich von denen in y unterscheiden.

Definition 4.18 (Hamming-Kugel). *Es sei $B_\rho(x)$ die Menge aller Wörter y mit $d(x, y) \leqslant \rho$. Wir nennen B_ρ die Hamming-Kugel um x mit Radius ρ.*

Jetzt können wir die Minimaldistanz eines Codes definieren, welche den minimalen Abstand zwischen zwei Codewörtern beschreibt.

Definition 4.19 (Minimaldistanz). *Die Minimaldistanz eines nichttrivialen Block-Codes C ist*

$$m(C) = min_{c_1, c_2 \in C, c_1 \neq c_2} d(c_1, c_2).$$

Dabei ist hier $d(c_1, c_2)$ die Hamming-Distanz. Eine weitere wichtige Größe im Zusammenhang mit Codes ist die Rate. Sie beschreibt das Verhältnis von Informationssymbolen zur Länge der Codewörter.

Definition 4.20 (Informations-Rate). *Für einen Code $C \subseteq Q^n$ heißt*

$$R(C) = \frac{\log(|C|)}{\log(|Q|^n)} = \frac{\log(|C|)}{n \cdot \log(|Q|)}$$

die (Informations-) Rate von C.

Für $|Q| = 2$ ergibt sich die Rate also zu $\frac{\log(|C|)}{n}$. Die Rate eines Codes ist immer kleiner als 1, da durch eine Kanalcodierung immer Redundanz zu den Daten hinzugefügt wird.

Definition 4.21 (Perfekter Code)**.** *Ein Code C mit ungerader Minimaldistanz $m(C)$ heißt* perfekt, *falls es für jedes $x \in Q^n$ genau ein $c \in C$ gibt, sodass $d(x, c) \leqslant \frac{m(C)-1}{2}$.*

Für perfekte Codes gilt also, dass man mit Maximum-Likelihood-Decoding jedes empfangene Wort eindeutig einem Codewort zuordnen kann. Allerdings muss dieses Codewort nicht unbedingt das gesendete Codewort sein.

Lemma 4.22. *Ein Block-Code C mit Minimaldistanz $m(C) = d$ kann entweder bis zu $d-1$ Fehler erkennen oder bis zu $\lfloor \frac{d-1}{2} \rfloor$ Fehler korrigieren.*

Beweisskizze. Der Abstand zwischen zwei Codewörtern beträgt d Bits, daher kann jedes Codewort, das an $d-1$ Stellen gekippt ist, als fehlerhafte Nachricht identifiziert werden. Allerdings ist bei $d-1$ gekippten Stellen nicht mehr klar, welches Codewort ursprünglich gesendet wurde, daher kann keine Korrektur erfolgen. Falls nur $\lfloor \frac{d-1}{2} \rfloor$ Bits gekippt sind, gibt es eine eindeutige Zuordnung zu einem Codewort (vgl. Abbildung Abbildung 4.11). □

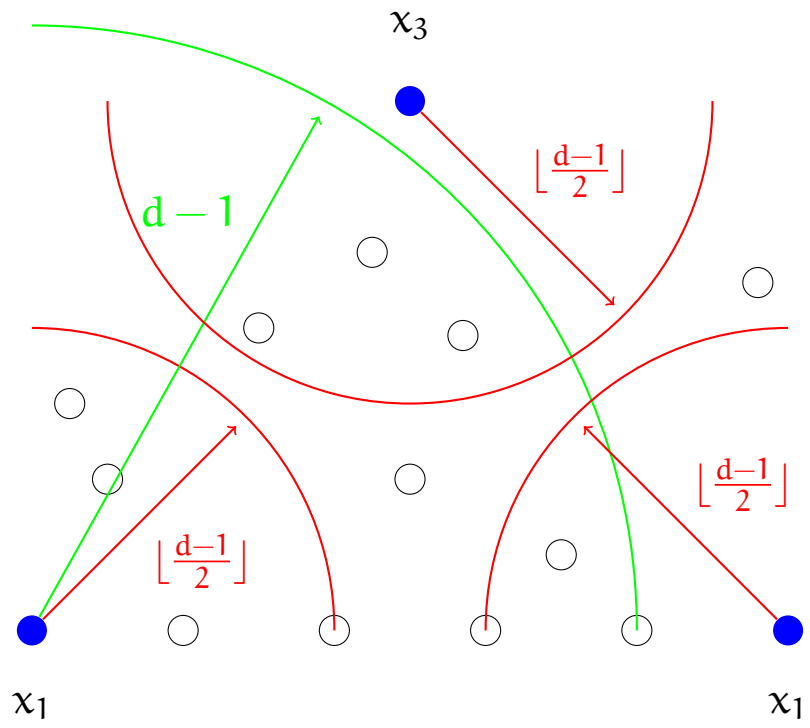


Abbildung 4.11: Beweisskizze.

Wir werden im Folgenden einige Codierungen betrachten. Dabei gibt es einige Eigenschaften, die zu beachten sind:

1. Der Code sollte eine hohe Rate haben, also möglichst wenig Redundanz.

2. Die Minimaldistanz sollte groß sein, um eine möglichst hohe Korrekturleistung zu erzielen.

3. Das Decodieren sollte effizient sein.

In Abbildung 4.12 erkennt man, dass (1) und (2) konkurrierende Ziele sind.

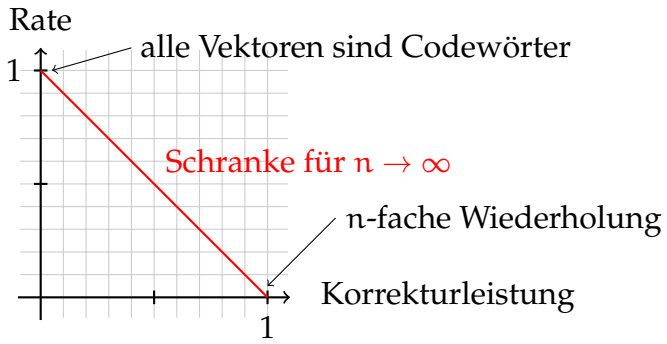


Abbildung 4.12: Zusammenhang von Rate und Korrekturleistung.

Definition 4.23 (Linearer Code). *Ein linearer* $[n, k]$*-Block-Code* C *ist ein Untervektorraum von* $\mathbb{F}_q^n$ *der Dimension* k*.*

Definition 4.24 (Hamming-Metrik). *Für* $x \in \mathbb{F}_q^n$ *definieren wir die Hamming-Metrik (Hamming-Gewicht,* L_0*-Norm)*

$$\mathrm{wgt}(x) = d(x, 0) = \sum_{i=1}^{n} (1 - \delta_{x_i,0}) = |\{i \mid i = 1, \ldots, n, x_i \neq 0\}|.$$

Korollar 4.25. *Für Block-Codes gilt* $d(x, y) = \mathrm{wgt}(x - y)$.

Beweis.

$$\begin{aligned} d(x, y) &= |\{i \mid i = 1, \ldots, n, x_i \neq y_i\}| \\ &= |\{i \mid i = 1, \ldots, n, x_i - y_i \neq 0\}| \\ &= \mathrm{wgt}(x - y). \end{aligned}$$

□

Da aber für lineare Block-Codes mit $x, y \in C$ auch $x - y \in C$, entspricht die Minimaldistanz also dem Vektor $c \in C$ mit dem kleinsten Hamming-Gewicht. Es gibt zwei Möglichkeiten, einen solchen Code zu beschreiben:

- Ist C eine linearer $[n, k]$-Code, so können wir C als Kern einer $\mathbb{F}_q^{(n-k)\times n}$-Matrix H angeben:

  $$C = \mathrm{Ker}(H) = \{x \in \mathbb{F}_q^n \mid H \cdot x = 0\}$$

  Dabei heißt H Prüfmatrix oder Parity-Check-Matrix.

- Beschreibung über Codierungsabbildung: Für einen $[n, k]$-Code C können wir eine $\mathbb{F}_q^{n\times k}$-Matrix G angeben sodass

$$C = \text{Bild}(G) = \{y \in \mathbb{F}_q^n \mid \exists x \in \mathbb{F}^k : y = G \cdot x.\}$$

  G bildet Informationswörter auf Codewörter ab.

Dabei ist die Parity-Check-Matrix die wichtigere Beschreibungsart, vor allem hinsichtlich Fehlererkennung und Fehlerkorrektur.
Weiterhin gilt für gegebene G, H, dass $H \cdot G = 0$, jede Spalte von G ist also ein gültiges Codewort.

Definition 4.26 ((Fehler)-Syndrom). *Für $x \in \mathbb{F}^n$ heißt $s = H \cdot x$ das (Fehler)-Syndrom von x.*

Dabei hängt s nur von einem additiven Fehler ab, nicht aber vom Codewort selbst: Ist $x = c + e$, so ist

$$H \cdot x = H \cdot (c + e) = H \cdot c + H \cdot e = H \cdot e = s.$$

4.4.1.1 Parity-Codes

Die Idee hinter Parity-Codes ist einfach: Man fügt einem Informationswort ein Bit hinzu, das angibt, ob die Quersumme des Informationswortes gerade oder ungerade ist (engl. *parity*), um so einfache Bitfehler zu erkennen.

Beispiel 34 (Even Parity).
Seien

$$G = \begin{pmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}, H = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \end{pmatrix}, x = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix}.$$

Dann ist $c = G \cdot x = \begin{pmatrix} 1 & 0 & 1 & 0 & \mathbf{0} \end{pmatrix}$. Das letzte Bit ist das so genannte Parity-Bit, welches hier aussagt, dass die Quersumme der vorigen Bits gerade ist.

Parity-Codes sind $[n, n-1]$-Codes, codieren also $n-1$ Informationsbits in Wörter der Länge n. Die Rate eines Parity-Codes ergibt sich als:

$$R(C) = \frac{n-1}{n} = 1 - \frac{1}{n}.$$

Das heißt also, $R(C) \to 1$ für $n \to \infty$.

Für die Minimaldistanz ergibt sich $m(C) = 2$. Parity-Codes können also einen einzelnen Bitfehler erkennen, aber keine Fehler korrigieren. Ist C ein $[n, k]$-Code mit ungerader Minimaldistanz d und Prüfmatrix H, so können wir den parity-erweiterten Code $\bar{C}$ definieren durch die Prüfmatrix

$$\bar{H} = \begin{pmatrix} 1 & 1 & \dots & 1 \\ & & & 0 \\ & H & & \vdots \\ & & & 0 \end{pmatrix}$$

$\bar{C}$ hat Minimaldistanz $d + 1$, da alle Gewichte in $\bar{C}$ gerade sind.

4.4.1.2 Hamming-Codes

Hamming-Codes sind eine Familie von dual-projektiven Codes.

Definition 4.27 (Dualer Code). *Sei C ein $[n,k]$-Code. Dann nennen wir*

$$C^\perp = \{y \in \mathbb{F}_q^n \mid \forall x \in C : x \cdot y^T = 0\}$$

den Dual-Code von C.

Information.

$C^\perp$ ist nicht mit dem Komplementärraum bei Vektorräumen über $\mathbb{R}$ zu verwechseln. Bei endlichen Körpern kann $C \cap C^\perp$ mehr als die Null enthalten und sogar $C = C^\perp$ sein.

Definition 4.28 (Projektiver Code). *C heißt projektiv, wenn alle Spalten seiner Generatormatrix G paarweise linear unabhängig sind.*

Im Folgenden sei $n = 2^k - 1$ für $k \in \mathbb{N}$.

Definition 4.29 (Hamming-Code). *Hamming-Codes sind $[n, n-k]$-Codes, für die je zwei Spalten der Prüfmatrix H linear unabhängig sind.*

Beispiel 35.
Der $[7,4]$-Hamming-Code hat folgende Matrizen:

$$H = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}, \quad G = \begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Als Rate für Hamming-Codes ergibt sich $R(C) = \frac{n-k}{n} = 1 - \frac{k}{n}$ und für die Minimaldistanz $m(C) = 3$.

Satz 4.30. *Hamming-Codes sind perfekt.*

Beweis. Sei C ein $[n, n-k]$-Hamming-Code. Betrachte Hamming-Kugeln $B_1(c)$, also diejenigen mit Radius 1 um die einzelnen Codewörter $c \in C$. Dann ist

$$|B_1(c)| = 1 + 2^k - 1 = 2^k.$$

C hat Dimension $n-k$, also gibt es $|C| = 2^{n-k}$ Codewörter. Für $c_1 \neq c_2$ sind $B(c_1)$ und $B(c_2)$ disjunkt. Die Hamming-Kugeln-Packung hat $2^k \cdot |C|$ Elemente

$$2^k \cdot |C| = 2^k \cdot 2^{n-k} = 2^n = |\mathbb{F}^n|.$$

□

Bei Hamming-Codes ist also ein effizientes Decodieren von 1-Bit-fehlerbehafteten Wörtern möglich. Gegeben ein Wort c. Hat die Prüfmatrix H die systematische Form wie in Beispiel 35, berechnet man zuerst $s = H \cdot c$, wobei s das Syndrom ist. Man identifiziert s^T mit einem Spaltenvektor aus H, dessen Position dem Index des gekippten Bits entspricht. Gibt es mehrere solche Spalten-Vektoren, ist eine eindeutige Korrektur nicht möglich. Die redundanten Bits werden entfernt (also die Positionen des Codeworts, deren Zeilen der Generatormatrix mehr als eine 1 enthalten) und man erhält das ursprüngliche Wort w.

Beispiel 36.
Sei $c = (1\ 0\ 0\ 1\ 1\ 1\ 1)^T$ das empfangene Wort. Dann bestimmen wir das Syndrom s durch

$$s = H \cdot c = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}.$$

Wir identifizieren das Syndrom $(0\ 0\ 1)^T$ eindeutig mit der ersten Spalte von H. Der Fehler befindet sich also an der ersten Stelle von c_w. In der dritten, fünften, sechsten und siebten Zeile von G steht genau eine 1, weshalb sich $w = (c_{w_3}\ c_{w_5}\ c_{w_6}\ c_{w_7})^T = (0\ 1\ 1\ 1)^T$ ergibt.

4.4.1.3 Weitere Verfahren

Es gibt noch viele weitere Block-Codierungsverfahren, die effizientes Decodieren ermöglichen:

- Repetition-Codes,
- Fourier- und Polynom-Codes: Reed-Solomon-Codes, BCH,
- Algebraische Geometrie-Codes: Goppa-Codes,
- d-reguläre Expander-Graphen: LDPC-Codes.

4.4.2 Verbindung zur Komplexitätstheorie

Problem 4.31 (`COSET-WEIGHTS`)**.** *Gegeben: Prüfmatrix H und ein Syndrom s und eine Zahl k. Frage: Gibt es ein e mit $\mathrm{wgt}(e) \leqslant k$, sodass $H \cdot e = s$?*

Satz 4.32 (Berlekamp, McEliece, van Tilborg)**.** `COSET-WEIGHTS` *ist $\mathcal{NP}$-vollständig.*

Beweis. Zunächst ist klar, dass `COSET-WEIGHTS` $\in \mathcal{NP}$: Wir raten einen Vektor $e \in \mathbb{F}_2^n$, prüfen ob $H \cdot e = s$ und ob $\mathrm{wgt}(e) \leqslant k$. Dies geht in $O(n^2)$ (wird dominiert von Matrix-Multiplikation). Wir zeigen durch Reduktion von `3DM` (Siehe Kapitel 3.4.8) auf `COSET-WEIGHTS`, dass `COSET-WEIGHTS` $\mathcal{NP}$-vollständig ist.

Sei (B, G, U, T) mit $|B| = |G| = |U| = n$ eine Instanz von `3DM`. Wir definieren $H \in \mathbb{F}_2^{3n \times |T|}$ folgendermaßen:

- Wir identifizieren die Zeilenindizes von H mit Elementen der Mengen B, G und U.
- Für jedes Tripel $(b, g, u) \in T$ führen wir einen Spalte c in H ein, die jeweils genau an den Stellen b, g und u eine 1, sonst überall 0 stehen hat. Wir können deshalb Spaltenindizes von H mit Tripeln $(b, g, u) \in T$ identifizieren.

Wir setzen $k = n$ und $s = (1, 1, \ldots, 1)^T \in \mathbb{F}_2^{3n}$. Für eine Lösung $e \in \mathbb{F}_2^{|T|}$ muss $\mathrm{wgt}(e) \geqslant n$ gelten, da $\mathrm{wgt}(s) = 3n$ und jede Spalte ein Hamming-Gewicht von 3 hat. Wir suchen also nach einem e mit Gewicht genau n. Wir erhalten damit: Die `3DM`-Instanz (B, G, U, T) ist genau dann erfüllbar, wenn die `COSET-WEIGHTS`-Instanz (H, s, k) erfüllbar ist. Da Spalten von H und Elemente von T sich gegenseitig entsprechen, können wir aus einem Vektor $e \in \mathbb{F}_2^{|T|}$ mit Gewicht n dadurch ein Matching M konstruieren, indem wir die Spalten genau die Tripel $(b, g, u) \in T$ in das Matching M aufnehmen, wenn in e an der Stelle (b, g, u) eine 1 steht. Umgekehrt können wir aus einem Matching M einen erfüllenden Vektor e konstruieren indem wir für jedes $(b, g, u) \in M$ in den Vektor e an Position (b, g, u) eine 1 schreiben und überall sonst Nullen. Damit ist die Reduktion abgeschlossen. □

Beispiel 37.
Sei $|B| = |G| = |U| = 2$ und $T = \{(1,1,2), (2,1,2), (2,2,1)\}$. Es ist also $n = 2$. Die Matrix H und das Syndrom s sehen folgendermaßen aus:

$$H = \begin{array}{c} b_1 \\ b_2 \\ g_1 \\ g_2 \\ u_1 \\ u_2 \end{array} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ \hline 1 & 1 & 0 \\ 0 & 0 & 1 \\ \hline 0 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}, s = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

Eine Lösung dieser Instanz ist $e = (1\ 0\ 1)^T$ mit $\mathrm{wgt}(e) = 2$.

4.5 Kryptographie und Informationstheorie

Informationstheorie kann auch auf die Kryptographie angewendet werden: Man versucht kryptographische Verfahren zu finden, bei denen das Chiffrat keine bzw. möglichst wenig Information über den Klartext enthält.

Definition 4.33 (Grundlagen der Kryptographie)**.**

- M *bezeichnet den Klartext (*message*), der verschlüsselt werden soll.*
- C *bezeichnet das Chiffrat (*ciphertext*), also den verschlüsselten Klartext.*
- K *bezeichnet den Schlüssel (*key*).*

M_i, C_i, K_i $(i \geqslant 0)$ *bezeichnet das* i*-te Zeichen von* M, C, K.

4.5.1 Einfache Verschlüsselungsverfahren: Substitutionschiffren

Im Folgenden soll eine einfache Klasse von Verschlüsselungsverfahren vorgestellt werden, bei der Buchstaben nach bestimmten Regeln durch andere Buchstaben ersetzt werden. Wir vereinbaren, dass jedem Buchstaben des Alphabets Zahlenwerte von 0 bis 25 zugeordnet werden, also $A \hat{=} 0, \ldots, Z \hat{=} 25$. Weiterhin definieren wir die Addition *modulo* 26: $x + y = (x + y) \mod 26$. Damit können wir nun mit Buchstaben „rechnen“: $X + Y = V$ $((23 + 24) \mod 26 = 21)$.

4.5.1.1 Die Caesar-Verschlüsselung

Bei der Caesar-Verschlüsselung wird jeder Buchstabe des Klartexts um einen konstanten Wert „verschoben“, man spricht daher auch von einer Verschiebechiffre.

Klartext M	…	A	B	C	E	F	G	H	…
Chiffrat C	…	W	X	Y	Z	A	B	C	…

Abbildung 4.13: Caesar-Verschlüsselung mit $K = 22$. Der i-te Buchstabe geht auf den $(i + 22 \mod 26)$-ten Buchstaben über.

Diese Art von Verschlüsselung ist unsicher, denn aus dem Chiffrat kann sehr viel Information über den Klartext gewonnen werden, da sich die Auftrittswahrscheinlichkeit der Zeichen nicht ändert. Durch Kenntnis der Verteilung der Buchstaben in deutschen Wörtern kann der Schlüssel bei hinreichend großem Chiffrat schnell ermittelt werden. Natürlich ist dieses Verfahren auch deshalb unsicher, weil es nur 26 mögliche Schlüssel gibt. Diese grundlegende Problematik ist bei allen monoalphabetischen Substitutionschiffren zu beobachten.

4.5.1.2 Die Vigenère-Verschlüsselung

Die Vigenère-Verschlüsselung, welche lange Zeit als „le chiffre indéchiffrable“ (dt. „die unentzifferbare Verschlüsselung“) galt, funktioniert ebenso wie die Caesar-Verschlüsselung nach dem Substituionsprinzip: Für einen Klartext M ergibt sich das j-te Zeichen C_j des Chiffrats aus dem j-ten Zeichen des Klartextes M_j durch $C_j = M_j + K_{j \mod |K|} \mod 26$. Hier ist der Schlüssel ein Wort, und jeder Buchstabe des Schlüssels bestimmt die Größe der Verschiebung.

Beispiel 38.

Klartext M	DIESERTEXTISTHOCHKRITISCH
Schlüssel K	ONYXONYXONYXONYXONYXONYXO
Chiffrat C	RVCPSERBLGGPHUMZVXPFHVQZV

Abbildung 4.14: Verschlüsselung von „DIESERTEXTISTHOCHKRITISCH“ mit Schlüssel „ONYX“.

Doch auch dieses Verfahren ist unsicher: für einen Schlüssel der Länge n wird jeder n-te Buchstabe des Klartextes mit demselben Schlüssel verschlüsselt. Damit ist auch dieses Verfahren, wenn auch in deutlich geringerem Maße, für eine Häufigkeitsanalyse anfällig.

4.5.2 Perfekte Sicherheit

Was macht also ein informationstheoretisch sicheres (*information-theoretically secure*) Verschlüsselungsverfahren aus? Um dies zu klären, führen wir den Begriff der perfekten Sicherheit ein. Dabei ist anzumerken, dass selbst ein Verfahren, das diese Bedingung erfüllt, nicht automatisch sicher ist. Gegen Anwendungsfehler etwa schützt auch perfekte Sicherheit nicht.

Definition 4.34 (Perfekte Sicherheit)**.** *Sei* M *ein Klartext, sowie* C *das dazugehörige Chiffrat. Sei nun* $p(M)$ *die Wahrscheinlichkeit, den Klartext zu erraten, sowie* $p(M|C)$ *die Wahrscheinlichkeit, den Klartext zu erraten, wenn* C *bekannt ist. Ein Verschlüsslungsverfahren heißt perfekt sicher, wenn*

$$p(M|C) = p(M)$$

gilt.

Die Kenntnis des Chiffrats ändert also nichts an der Wahrscheinlichkeit, den Klartext erraten zu können.

Korollar 4.35. *Bei einem perfekt sicheren Verschlüsselungsverfahren gilt* $p(C|M) = p(C)$.

Beweis. Aus dem Satz von Bayes folgt:

$$p(C|M) = \frac{p(M|C) \cdot p(C)}{p(M)} = p(C)$$

da $p(M|C) = p(M)$ nach Definition 4.34. □

Man kann bei einem solchen Verfahren also auch mit bekanntem Klartext keine Aussage über das Chiffrat machen. Um dies bei endlich langen Klartexten zu erreichen, muss die Anzahl der verwendeten Schlüssel größer als oder gleich groß wie die Anzahl der Klartexte sein.

Satz 4.36 (Perfekte Sicherheit nach Shannon)**.** *Sei* $\tilde{M} = \{M^1, \ldots, M^n\}$ *eine Menge von Klartexten und* $\tilde{C} = \{C^1, \ldots, C^n\}$ *eine Menge von Chiffraten. Sei weiterhin* $G = (V, E)$ *mit* $V = \tilde{M} \cup \tilde{C}$ *und* $E = \{(M, C) \in \tilde{M} \times \tilde{C} \mid M$ *kann Klartext von* C *sein.*$\}$ *ein Graph. Ein Verfahren, das einem* $M^j \in \tilde{M}$ *ein* $C^k \in \tilde{C}$ *zuordnet ist genau dann perfekt sicher, wenn* G *vollständig und bipartit ist und alle Paare* $(M, C) \in E$ *gleich wahrscheinlich sind.*

Beweis. Leichte Übung. □

Anschaulich gesprochen bedeutet das, dass jedes Chiffrat zu jedem Klartext entschlüsselt werden kann.

Beispiel 39.

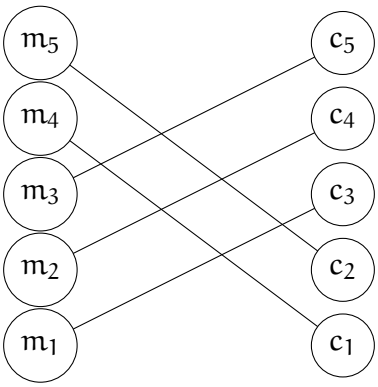


Abbildung 4.15: Graph für eine Caesar-Verschlüsselung.

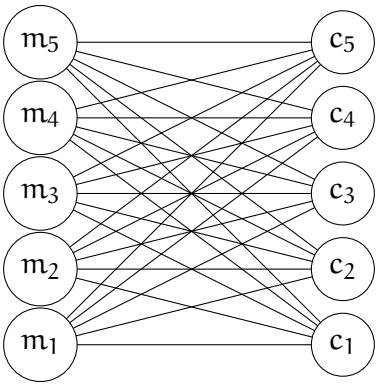


Abbildung 4.16: Vollständiger bipartiter Graph für eine perfekt sichere Verschlüsselung.

4.5.2.1 One-Time-Pads: Ein sicheres Kryptosystem

Basierend auf einem Verschlüsselungssystem von Gilbert Vernam lässt sich ein beweisbar sicheres Verschlüsselungsverfahren entwerfen.

Definition 4.37 (One-Time-Pad). *Sei* M *ein Klartext und* K *eine gleichverteilt zufällige Zeichenfolge mit* $|K| = |M|$ *und* K, M *im Dualsystem (also* $K_i, M_i \in \{0, 1\}$*). Dann ist das* i*-te Zeichen des Chiffrats* $C_i = P_i \oplus K_i$*, wobei* $\oplus$ *die bitweise* XOR*-Verknüpfung ist.*

Man sieht leicht, dass $M_i = C_i \oplus K_i$ gilt. One-Time-Pads sind allerdings nur sicher, solange der Schlüssel genau so lange wie der Klartext ist und nur einmal verwendet wird. One-Time-Pads sind also ein Spezialfall der Vigenère-Verschlüsselung, bei der Schlüssel und Klartext gleich lang sind.

Satz 4.38. *One-Time-Pads sind perfekt sicher.*

Beweis. Zur Übung. □

Übrigens ...

Während des Kalten Krieges führte die mehrfache Verwendung von Schlüsseln durch die Sowjetunion dazu, dass die Vereinigten Staaten im VENONA-Projekt Teile der sowjetischen Kommunikation entschlüsseln konnten. Moderne Implementierungen verwenden technische Maßnahmen um sicherzustellen, dass ein Schlüssel nur ein einziges mal verwendet wird.

4.5.2.2 Zusammenhang zur Entropie

Betrachen wir den Klartext M und den Schlüssel K jeweils als Zufallsquellen, dann existieren auch $H(M)$ und $H(K)$. Diese Betrachtungsweise erlaubt es uns nun, auch hier Zusammenhänge zu untersuchen. Man erkennt zum Beispiel, dass man die Äquivokation nun als Sicherheitsmaß betrachten kann. Interessant sind dabei jeweils

$$H(K|C) = -\sum_{C,K} p(C,K)\log(p(K|C))$$

$$H(M|C) = -\sum_{C,M} p(C,M)\log(p(M|C))$$

und wie sich diese Begriffe zu $H(K)$ bzw. $H(M)$ verhalten.

Satz 4.39. *Für perfekte Sicherheit gilt* $H(M|C) = H(M)$.

Beweis. Es ist

$$p(C|M) = \frac{p(C,M)}{p(M)}.$$

Mit der Definition von perfekter Sicherheit ($p(C|M) = p(C)$) sieht man, dass $p(C,M) = p(C)\cdot p(M)$ gilt. Das heißt also, dass $p(C)$ und $p(M)$ stochastisch unabhängig sind. Untersuchen wir also nun $H(M|C)$, so folgt:

$$\begin{aligned} H(M|C) &= -\sum_{C,M} p(C,M)\log(p(M|C)) \\ &= -\sum_{C,M} p(C)\cdot p(M)\log(p(M)) \\ &= -\underbrace{\sum_{C} p(C)}_{1}\cdot\underbrace{\sum_{M} p(M)\log(p(M))}_{-H(M)} = H(M) \end{aligned}$$

Umgekehrt also auch:

$$H(M|C) = H(M) \Rightarrow p(M|C) = p(M).$$

□

Auch die Transinformation, also gegenseitige Information, verhält sich wie erwartet:

$$I(M;C) = H(M) - H(M|C) = 0.$$

4.5.3 Moderne Kryptographie

Ein interessantes kryptographisches Problem ist folgendes: Kann man über eine Telefon-Verbindung gerecht eine Münze werfen, also ein zufälliges Bit ermitteln, sodass keiner der beiden Gesprächspartner das Ergebnis bestimmen kann? Die triviale Möglichkeit, dass einer der beiden Gesprächsteilnehmer eine Münze wirft, reicht offenbar nicht aus, denn er könnte ja behaupten, ein beliebiges Ergebnis wäre herausgekommen. Nach einer Idee von Manuel Blum kann folgendes Protokoll genutzt werden:

1. Alice steckt ein zufälliges Bit b in einen Tresor.
2. Alice schickt Bob den Tresor, dieser kann ihn aber ohne Schlüssel nicht öffnen.
3. Bob sendet Alice ein zufälliges Bit b'.
4. Alice sendet Bob den Schlüssel des Tresors.
5. Nun ist $b \oplus b'$ das Ergebnis des Münzwurfes.

Dabei legen sich beide auf ihr jeweiliges Bit fest, bevor sie das des anderen kennen. Außerdem ist $b \oplus b'$ schon dann zufällig, wenn einer der beiden ehrlich ist, also sein Ergebnis echt zufällig wählt.

4.5.3.1 Allgemeines zu Bit-Commitments

Wie funktioniert nun der „Tresor“? Der Tresor ist offenbar das zentrale Element in diesem Algorithmus. Ein Verfahren, das einen solchen Tresor implementiert, heißt *Bit-Commitment*. Es besteht dabei aus zwei Phasen:

- `c = commit(b)` — Das Bit b wird durch das Commitment c festgelegt, ohne jedoch den Wert zu offenbaren.
- `unveil(c)` — Deckt das Commitment c auf, gibt also aus, was b ist. Dies entspricht dem Öffnen des Tresors.

Um die Sicherheitseigenschaften definieren zu können, benötigen wir vernachlässigbare Funktionen.

Definition 4.40 (Vernachlässigbare Funktion)**.** *Eine Funktion* $f : \mathbb{N} \to \mathbb{R}$ *heißt* vernachlässigbar, *wenn zu jeder positiven Konstante* $c \in \mathbb{N}$ *eine Schwelle* $m_c \in \mathbb{N}$ *existiert, ab der für alle* $n > m_c$ *gilt:*

$$|f(n)| < \frac{1}{n^c}.$$

Für ein Bit-Commitment-Verfahren gibt es zwei Sicherheitseigenschaften:

Definition 4.41 (Eigenschaften von Commitment-Verfahren)**.** *Ein Commitment* c *ist*

- binding, *wenn gilt:*

$$\Pr[\texttt{unveil}(\texttt{commit}(b)) \neq b] \leqslant \varepsilon$$

  Es kann also nur genau der Wert aufgedeckt werden, auf den committet wurde.

- hiding, *wenn gilt:*

$$\Pr[b|c] - \Pr[(1-b)|c] \leqslant \varepsilon$$

  Ein Commitment auf die 0 ist ununterscheidbar von einem Commitment auf die 1.

Dabei ist ε *eine vernachlässigbare Funktion.*

Man spricht von informationstheoretischem Binding (*unconditional binding*), wenn selbst ein in jeder Form unbeschränkter Sender das Bit nicht anders aufdecken kann, als es committet wurde. Informationstheoretisches Hiding (*unconditional hiding*) heißt, dass die Nachricht in `commit(b)` statistisch unabhängig von b ist, also keinerlei Information über b enthält. Man sieht, dass diese beiden Kriterien sich ausschließen: Wenn eine solches Verfahren im informationstheoretischen Sinne binding ist, so muss die Nachricht von `commit` also eindeutig zuordenbar sein — damit ist sie jedoch nicht statistisch unabhängig von b. Wenn andererseits die Nachricht, die `commit` erzeugt, statistisch unabhängig von b ist, müssen mit der gleichen Argumentation `unveil`-Aufrufe existieren, die ein beliebiges Ergebnis haben, was allerdings nicht binding ist. Also kann im informationstheoretischen Sinne ein solches Verfahren nicht gleichzeitig binding und hiding sein.

Daher ist es nötig, dass es für eine Eigenschaft des Bit-Commitments „schwierig“ ist, sie zu brechen, statt unmöglich. Hier kommen komplexitätstheoretische Annahmen ins Spiel.

4.5.3.2 Mathematisches Werkzeug

Satz 4.42. *Sei* p *eine Primzahl. Dann ist* $\mathbb{Z}/p\mathbb{Z}$ *ein Körper.* $\mathbb{Z}/p\mathbb{Z}$ *wird auch als* $\mathbb{F}_p$ *oder* $\mathrm{GF}(p)$ *bezeichnet.* $\mathbb{F}_p^*$ *bezeichnet die multiplikative Gruppe* $(\mathbb{F}_p \setminus \{0\}, \cdot)$*. Sie hat* $p-1$ *Elemente.* $g \in \mathbb{F}_p^*$ *wird Erzeuger genannt, wenn* $\langle g \rangle = \{g^0, g^1, \dots g^{p-2}\} = \mathbb{F}_p^*$*,* g *also* $\mathbb{F}_p^*$ *erzeugt. Insbesondere gibt es für jedes* $y \in \mathbb{F}_p^*$ *ein* x*, sodass* $g^x = y$*.*
x *wird diskreter Logarithmus von* y *zur Basis* g *genant.*

Beweis. Ohne Beweis. □

Definition 4.43 (DLOG)**.** *Sei* p *eine Primzahl,* g *ein Erzeuger von* $\mathbb{F}_p^*$ *und* $h \neq 0 \in \mathbb{F}_p^*$*. Das Finden von einem* x*, sodass*

$$g^x = h$$

wird Diskreter-Logarithmus-Problem (DLOG-Problem) genannt.

Vermutung.
DLOG $\notin \mathcal{P}$.

Man geht davon aus, dass es keinen effizienten Algorithmus gibt, um eine Lösung für das DLOG-Problem zu finden. Daher verwendet man den diskreten Logarithmus als injektive *Einweg-Funktion*.

Definition 4.44 (Einwegfunktion)**.** *Es bezeichne* U_n *die Gleichverteilung auf* $\{0,1\}^n$ *für* $n \in \mathbb{N}$. *Eine Funktion* $f : \{0,1\}^* \to \{0,1\}^*$ *wird Einweg-Funktion genannt, wenn*

1. f *in Polynomialzeit berechnet werden kann und*
2. *wenn für jeden probabilistischen Polynomialzeit-Algorithmus* A *und hinreichend großes* n
$$\Pr[A(f(U_n, 1^n)) \in f^{-1}(f(U_n))] \leqslant \varepsilon$$
gilt, wobei ε *eine vernachlässigbare Funktion in* n *ist.*

Wir verlangen also für eine Einwegfunktion, dass das Finden von Urbildern schwierig ist.

Gegeben eine injektive Einwegfunktion f, kann man sich auf ein Bit b festlegen, indem man zufällige Vektoren x und r wählt mit $b = \langle x, r\rangle$, wobei $\langle\cdot\rangle$ das innere Produkt beschreibt, und dann als `commit(b)` $(f(x), r)$ zurückgibt. Als `unveil` verwendet man einfach (b, x, r), womit der Empfänger einfach überprüfen kann ob $b = \langle x, r\rangle$. Dabei ist dieses Verfahren informationstheoretisch binding, aber nur komplexitätstheoretisch hiding.

4.5.3.3 Pedersen-Commitments

Definition 4.45 (Pedersen-Commitments)**.** *Es sei* G *eine zyklische Gruppe,* g *und* h *Erzeuger von* G *und* m *der zu committende Wert.*

- `commit`(m) = *Wähle eine Zufallszahl* r *und berechne* $c = g^m \cdot h^r$.
- `unveil`(c) = *Decke* m *und* r *auf.*

Satz 4.46. *Pedersen-Commitments sind informationstheoretisch hiding.*

Beweis. Es ist zu zeigen, dass es für alle m, r und $m' \neq m$ auch ein r' gibt, sodass $c = g^m \cdot h^r = g^{m'} \cdot h^{r'}$:

$$\begin{aligned} c &= g^m \cdot h^r \\ &= g^{m'} \cdot g^{-m'+m} \cdot h^r \\ &= g^{m'} \cdot h^{\log_h(g)\cdot(-m'+m)} \cdot h^r \\ &= g^{m'} \cdot h^{r+\log_h(g)\cdot(-m'+m)} \end{aligned}$$

□

Satz 4.47. *Pedersen-Commitments sind komplexitätstheoretisch binding* (computationally binding)*.*

Beweis. Aus dem Beweis zu Satz 4.46 folgt, dass man, um $c = g^m \cdot h^r$ als m' aufzudecken, $r' = r + \log_h(g) \cdot (-m' + m)$ berechnen müsste. Dies ist mindestens so schwierig wie das Lösen einer Instanz des DLOG-Problems. □

A. Anhang

A.1 Philosophisches: Die vier Grundgragen der Philosophie

Immanuel Kant hat sich vier Grundfragen der Philosophie aufgestellt und versucht zu beantworten. Interessanterweise können wir mit dem Wissen, das wir aus der Vorlesung „Theoretische Grundlagen der Informatik“ mitgenommen haben, auch Aussagen zu diesen Fragen machen.

A.1.1 Was kann ich wissen?

Mit dieser Frage beschäftigt sich die Erkenntnistheorie. Unter dem, was im Rahmen dieser Vorlesung behandelt wurde, ist zu dieser Frage natürlich sofort der Gödelsche Unvollständigkeitssatz zu nennen: Wir können aus einem formalen System entweder nicht alles Korrekte beweisen, oder das System ist nicht widerspruchsfrei. Die Widerspruchsfreiheit der Mathematik ist auch nicht beweisbar. Wie wir gesehen haben, sind die Implikationen aus diesen Sätzen enorm.
Schließlich kann man natürlich auch den Standpunkt des Solipsismusses vertreten: Nur das Ich existitert, die Außenwelt ist nur Bewusstseinsinhalt ohne eigene Existenz. Die eigene Existenz ist dabei nach Descartes' berühmten Grundsatz „Cogito ergo sum“ — „Ich denke, also bin ich“ — das einzige, über das man wirklich sicher sein kann.

A.1.2 Was soll ich tun?

Diese Frage wird von der Ethik behandelt. Auf diese Frage hat Kant mit seinem Kategorischen Imperativ eine recht formale Antwort gegeben:

> „Handle nur nach derjenigen Maxime, durch die du zugleich wollen kannst, dass sie ein allgemeines Gesetz werde.“

Man kann nun auch überlegen, ob man diese Frage mit den Mitteln der Informatik beantworten kann. Könnte man etwa durch Simulation die bestmöglichte Handlung ermitteln? Was ist diese Überhaupt, also was wäre die Zielfunktion? All dies hängt auch noch von der vierten Frage, der nach dem Menschenbild, ab.

A.1.3 Was darf ich hoffen?

Mit dieser Frage beschäftigt sich die Religionsphilosophie. Darunter fallen auch Fragen wie: Existiert ein persönlicher Gott? Wie entsteht Bewusstsein? Gibt es hier einen Zusammenhang zum Kategorischen Imperativ?

A.1.4 Was ist der Mensch?

Mit dieser Frage setzt sich die Anthropologie auseinander. Hier stellt sich vorallem die Frage nach den Grenzen: Wann fängt ein Mensch an zu leben? Wann genau ist er tot? Wann begann die Menscheit als Spezies? Auch hier bietet die Informatik vielleicht wieder die Möglichkeit, durch Simulation diese Fragen zumindest teilweise zu beantworten.

Sätze

Definitionen

Probleme

Glossar

A | B | C | D | E | G | H | I | K | L | M | N | O | P | R | S | T | U | V | W | Z

A

B

C