Theoretische Informatik 2

Berechenbarkeits- und Komplexitätstheorie

Vorlesungsnotizen

18. Juni 2018

Sebastian Muskalla
Roland Meyer
Elisabeth Neumann

TU Braunschweig

Sommersemester 2018

Teil I.

Entscheidbarkeit & Berechenbarkeit

1. Kontextsensitive Sprachen

In diesem Kapitel werden wir kontextsensitive Sprachen und Turingmaschinen einführen und untersuchen. Die kontextsensitiven Sprachen wurden bereits in der Vorlesung „Theoretische Informatik 1“ definiert, aber nicht weiter vertieft. Wir werden die kontextsensitiven Sprachen hier noch einmal aufgreifen. Einerseits zeigen wir, dass das Wortproblem für kontextsensitive Sprachen entscheidbar ist. Andererseits zeigen wir eine Korrespondenz zwischen kontextsensitiven und linear-beschränkten Turing-Maschinen, angelehnt an die Korrespondenz zwischen kontextfreien Sprachen und Kellerautomaten. Wir gehen auch auf (nicht-beschränkte) Turing Maschinen ein und zeigen eine Korrespondenz mit rekursiv-aufzählbaren Sprachen.

Wir beginnen das Kapitel mit einigen grundlegenden Definitionen, wobei manche schon aus der Vorlesung „Theoretische Informatik 1“ bekannt sind.

1.1 Definition

- Eine **Typ-0-Grammatik** ist eine Grammatik $G = (N, \Sigma, P, S)$, wobei die Produktionen die Form $P \subseteq (N \cup \Sigma)^* \times (N \cup \Sigma)^*$. Typ-0-Grammatiken sind die allgemeinste Form der Grammatiken.

- Die Grammatik ist **kontextsensitiv** oder **Typ-1**, falls für alle Produktionen $\alpha_1 \to \alpha_2$ gilt dass $|\alpha_1| \leqslant |\alpha_2|$. Intuitiv bedeutet diese Einschränkung dass die Produktionen Längeerhaltend sind, also eine Satzform während einer Ableitung nicht kleiner werden kann. Die Produktion $S \to \varepsilon$ ist erlaubt, allerdings darf S dann bei keiner Produktion auf der rechten Seite vorkommen.

- Eine Sprache $\mathcal{L} \subseteq \Sigma^*$ ist **kontextsensitiv**, falls es eine kontextsensitive Grammatik G gibt mit $\mathcal{L}(G) = \mathcal{L}$. Wir verwenden im Folgenden die Abkürzung CSL (context-sensitive languanges) für die kontextsensitiven Sprachen.

- Eine Sprache $\mathcal{L} \subseteq \Sigma^*$ ist **rekursiv aufzählbar**, falls es eine Typ-0-Grammatik G gibt mit $\mathcal{L}(G) = \mathcal{L}$.

1.2 Beispiel

Die folgende Sprache ist kontextsensitiv

$$\mathcal{L} = \{w \in \{a, b, c\}^* \mid \#_a(w) = \#_b(w) = \#_c(w)\}$$

und wird von Typ-1-Grammatik $G = (\{S, R, A, B, C\}, \{a, b, c\}, P, S)$ erzeugt. Die Produktionen P sind gegeben durch

$$
\begin{array}{ll}
S \to \varepsilon \mid R & AB \to BA \\
R \to ABC \mid ABCR & AC \to CA \\
A \to a & BA \to BA \\
B \to b & BC \to CB \\
C \to c & CA \to AC \\
 & CB \to BC.
\end{array}
$$

Intuitiv erlauben die ersten beiden Produktionen uns ein beliebiges Wort aus $(ABC)^*$ zu produzieren. Damit wird garantiert, dass das am Ende erzeugte Wort gleich viele a's, b's und c's enthalten wird. Mithilfe der Produktionen in der linken Spalte können die Buchstaben dann beliebig permutiert werden.

Wie bereits erwähnt, interessieren wir uns für das Wortproblem für kontextsensitive Sprachen. Wie bereits aus „Theoretische Informatik 1" bekannt, ist das Wortproblem wie folgt definiert.

Wortproblem zu $\mathcal{L}$
Gegeben: Wort $w \in \Sigma^*$
Entscheide: Gilt $w \in \mathcal{L}$?

Folgendes Theorem zeigt dass, genau wie im Fall für kontextfreie Sprachen, das Wortproblem für kontextsensitive Sprachen entscheidbar ist.

1.3 Theorem
Für jede kontextsensitive Grammatik G kann man das Wortproblem für $\mathcal{L}(G)$ in Zeit $2^{\mathcal{O}(|w|)}$ lösen.

Beweis:
Wir machen nun Gebrauch von der Länge-erhaltenden Eigenschaft der Produktionen von G. Aufgrund dieser Eigenschaft ist es nicht möglich aus einer Satzform der Länge echt größer $|w|$ noch w abzuleiten. Das heißt, dass wir alle Satzformen der Länge höchstens $|w|$ aufzählen und dann testen können ob sich w darunter befindet.

Die Zeitschranke $2^{\mathcal{O}(|w|)}$ folgt aus folgender Abschätzung für die Anzahl an Satzformen der Länge $\leqslant |w|$.

$$(|N| + |\Sigma| + 1)^{|w|} = 2^{\mathcal{O}(|w|)}$$

Die 1 in der Klammer steht für das leere Wort ε. Man bemerke dass die Grammatik und damit N und Σ nicht Teil der Eingabe des Problems sind. Daher gilt $|N|+|\Sigma|+1 = 2^c$ für eine Konstante c, welche unabhängig von der Eingabe (dem Wort w) ist. □

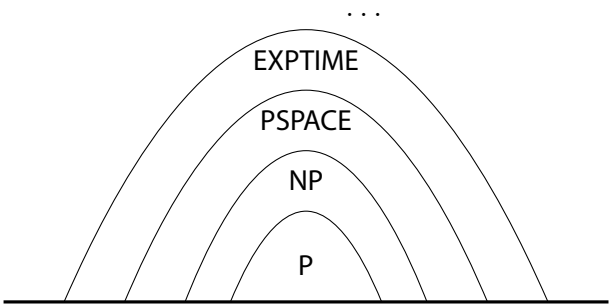
Das Wortproblem für kontextsensitive Sprachen ist schwer:

- Die Korrespondenz mit den linear-beschränkten Turing-Maschinen (siehe weiter unten) zeigt, dass

  $$\text{CSL} = \text{PSPACE},$$

  also, dass die kontextsensitiven Sprachen genau den Problemen entsprechen, welche in Polynomialzeit lösbar sind.

  Komplexitätsklassen:



- Da PSPACE-harte Probleme existieren, gibt es auch kontextsensitive Sprachen $\mathcal{L}(G)$, für die das Wortproblem PSPACE-hart ist.

1.4 Bemerkung
Die PSPACE ist eine in der Praxis wichtige Komplexitätsklasse für Verifikationsprobleme:

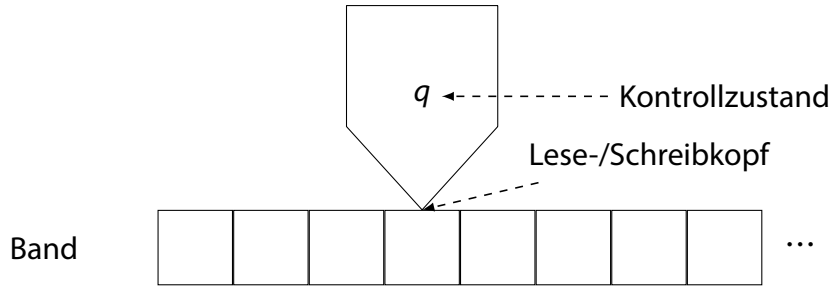
- Erreichbarkeit in Boolschen While-Programmen
- Erreichbarkeit in Multithreaded-Programmen

Die kontextsensitiven Sprachen finden Anwendung in Compilern, wo sie Parameter in Prozeduren und Scopes modellieren.

A) Turingmaschinen

Ziel ist es im Folgenden, Turing-Maschinen und ihre Berechnungen zu definieren.

Intuitiv ist eine Turing-Maschine ein Automat, der auf einem unendlichen Band operiert.



Genau wie dies von endlichen Automaten und Pushdown-Automaten bekannt ist, hat eine Turing-Maschine eine endliche Menge von Kontrollzuständen. Das Band, auf dem sie arbeitet, stellt ihr unbegrenzt viel Speicher zur Verfügung. Ihren Schreib- & Lesekopf darf sie auf diesem Band ohne Einschränkung bewegen (im Gegensatz zu Pushdown-Automaten, die in jedem Schritt nur das oberste Element ihres Stacks anfassen dürfen).

Turing-Maschinen wurden 1936 vom berühmten britischen Mathematiker **Alan Turing** (1912-1954) eingeführt. Sie sind eine sehr erfolgreiche Formalisierung des Begriffs der algorithmischen Lösbarkeit bzw. Entscheidbarkeit. Die **Church-Turing-These**, benannt nach Turing und nach **Alonzo Church** (1903-1995), sagt:

> Alles, was sich intuitiv berechnen lässt, lässt sich mit einer Turing-Maschine berechnen.

Sie lässt sich nicht beweisen – wir quantifizieren schließlich über alle möglichen Berechnungsmodelle – allerdings sind alle anderen Berechnungsmodelle, die man sich bislang überlegt hat, höchstens genauso mächtig wie Turing-Maschinen. Beispielsweise lassen sich reale Programme (Assembler, C, Java, …) durch Turing-Maschinen simulieren (sogar mit kleinem Overhead).

1.5 Definition
Eine **Turing-Maschine** (**TM**), M ist ein Tupel

$$M = (Q, \Sigma, \Gamma, q_0, \delta, Q_F)$$

mit

- Q ist eine endliche Menge von **Kontrollzuständen**,
  - $q_0 \in Q$ ist der **Start- oder Initialzustand**,
  - $Q_F \subseteq Q$ ist die Menge der Endzustände
- Σ ist das endliche, nicht-leere **Eingabealphabet**,
- Γ ist das endliche, nicht-leere **Bandalphabet**,
  - $\Sigma \subseteq \Gamma$,

  - $\sqcup \in \Gamma$ ist das **Leerzeichen** oder **Blank**-Symbol,
  - es gilt $\sqcup \notin \Sigma$,

- Falls die Transitionsfunktion δ von der Form

$$\delta\colon Q \times \Gamma \to Q \times \Gamma \times \{L, R, N\} ,$$

  ist, sprechen wir von einer **deterministischen Turingmaschine** (**DTM**). Hier stehen $\{L, R, N\}$ für die möglichen Bewegungen des Schreib- & Lesekopf: L steht für links, R für rechts und N für neutral (keine Bewegung).

- Andernfalls, wenn δ von der Form

$$\delta \subseteq Q \times \Gamma \times \Gamma \times \{L, R, N\} \times Q ,$$

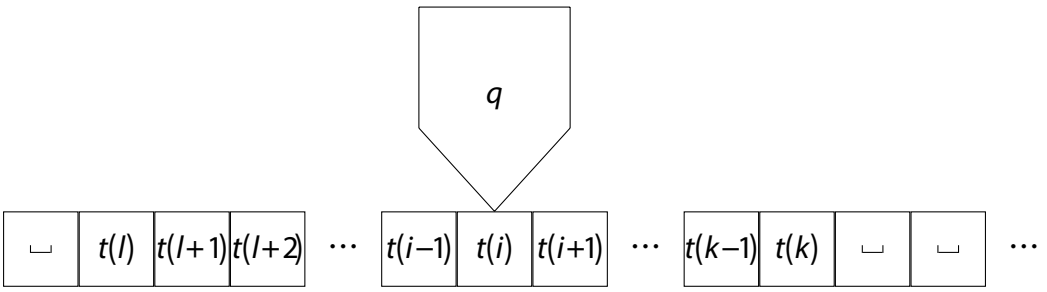
  ist, reden wir von einer **nicht-deterministischen Turingmaschine** (**NTM**).

Die vorherige Definition fixiert die Syntax von Turing-Maschinen. Nun müssen wir die Semantik von Turing-Maschinen, das heißt ihre Berechnungszustände und Berechnungen definieren. Bevor wir dies formal tun, geben wir eine intuitive Erklärung.

Intuitiv besteht die Konfiguration einer Turing-Maschine aus

- einem Kontrollzustand $q \in Q$
- dem Bandinhalt, den man als Funktion $t\colon \mathbb{N} \to \Gamma$ sehen kann, die der n-ten Zelle ihren Inhalt $t(n) \in \Gamma$ zuordnet, und
- der Kopfposition $i \in \mathbb{N}$.

Wir werden hierbei davon ausgehen, dass nur ein endliches Mittelstück des Bandes beschrieben ist, der Rest des Bandes ist mit Blanks gefüllt: $\exists l, k\colon \forall n < l, \forall n > k, : t(n) = \sqcup$.



Sei nun

$$\delta(q, a) = (p, b, d)$$

eine Abbildungsvorschrift gemäß der Transitionsfunktion δ. Dies bedeutet, dass die Maschine M

- im Zustand $q \in Q$,
- wenn an der aktuellen Kopfposition Symbol $a \in \Gamma$ steht,

im nächsten Schritt die folgenden drei Operationen ausführt:

1. Ändere Kontrollzustand zu $p \in Q$.
2. Ersetze den Inhalt der Zelle (derzeit a) durch Symbol $b \in \Gamma$.
3. Bewege den Kopf um eine Position nach links (falls $d = L$), nach rechts (falls $d = R$) oder gar nicht (falls $d = N$).

Wir formalisieren nun die intuitive Definition von Konfigurationen und Berechnungen.

1.6 Definition
Sei $M = (Q, \Sigma, \Gamma, \delta, q_0)$ eine Turing-Maschine.

a) Eine **Konfiguration** von M ist ein Tripel $u\,q\,v \in \Gamma^* \times Q \times \Gamma^*$.

Formal müsste man (u, q, v) schreiben, wir lassen die Klammern aus.

Die Idee hierbei ist, dass u der (bereits besuchte) Bandinhalt links vom Kopf ist, q der aktuelle Kontrollzustand und v der (bereits besuchte) Bandinhalt rechts vom Kopf, wobei das erste Symbol von v der Bandinhalt an der aktuellen Kopfposition ist.

Wie bereits erklärt, betrachten wir ausschließlich Konfigurationen, bei denen nur ein endlicher Teil des Bandes beschrieben ist. Formal identifizieren wir $v = \sqcup v \sqcup = \sqcup\sqcup v \sqcup\sqcup = \ldots = \sqcup^\omega v \sqcup^\omega$.

b) Die **Startkonfiguration** von M für die **Eingabe** $w \in \Sigma^*$ ist die Konfiguration $q_0 w$.

Dies bedeutet, dass sich die Maschine im Startzustand befindet, das Band mit dem Blanks links, dann der Eingabe, und danach mit Blanks rechts gefüllt ist, und der Kopf auf das erste Symbol von w zeigt.

Falls der Input das leere Wort ε ist, ist die Startkonfiguration $q_0 \sqcup$.

c) Die Transitionsfunktion δ induziert eine **Transitionsrelation** zwischen Konfigurationen, die wie folgt definiert ist:

$$
\begin{aligned}
u.a\,q\,b.v &\to u\,q'\,a.c.v\,, && \text{falls } (q', c, L) \in \delta(q, b),\\
u.a\,q\,b.v &\to u.a.c\,q'\,v\,, && \text{falls } (q', c, R) \in \delta(q, b) \text{ und } v \neq \varepsilon,\\
u.a\,q\,b.v &\to u.a\,q'\,c.v\,, && \text{falls } (q', c, N) \in \delta(q, b),\\
u\,q\,b &\to u.c\,q'\,\sqcup && \text{falls } (q', c, R) \in \delta(q, b),\\
q\,b.v &\to q'\,\sqcup.c.v\,, && \text{falls } (q', c, L) \in \delta(q, b).
\end{aligned}
$$

für $a, b, c \in \Gamma, q, q' \in Q, u, v \in \Gamma^*$.

Man sieht, dass die Transitionsrelation $\to$ genau die zuvor beschriebenen Schritte umsetzt. Wir verwenden $\to^*$ um die reflexive, transitive Hülle von $\to$ darzustellen.

d) Eine Konfiguration $u\ q\ v$ heißt **akzeptierend**, falls sie in $\Gamma^* Q_F \Gamma^*$ enthalten ist, also ein akzeptierender Zustand erreicht wurde.

e) Eine **Berechnung** von M auf Eingabe $w \in \Sigma^*$ ist die unendliche Sequenz von Konfigurationen

$$c_0 = q_0 w \to c_1 \to c_2 \to c_3 \to \dots$$

die sich von der Startkonfiguration zu w aus ergibt, in dem man in jedem Schritt einen Nachfolger bezüglich der Transitionsrelation $\to$ nimmt. Falls wir eine deterministische Turingmaschine betrachten ist der Nachfolger einer Konfiguration bezüglich $\to$ immer eindeutig.

Oftmals interessiert man sich nur für einen endlichen Präfix $c_0 \to \dots \to c_k$ einer Berechnung. Dies ist vor allem der Fall wenn die letzte Konfiguration des Präfixes akzeptierend ist.

f) Die Sprache $\mathcal{L}(M)$ der Maschine M ist die Menge aller Wörter w, so dass eine Berechnung von der Startkonfiguration $q_0 w$ zu einer akzeptierenden Konfiguration existiert.

$$\mathcal{L}(M) = \left\{ w \in \Sigma^* \;\middle|\; M \text{ akzeptiert } w \right\} = \left\{ w \in \Sigma^* \;\middle|\; q_0 w \to^* u q' v \in \Gamma^* Q_F \Gamma^* \right\}.$$

B) Varianten von Turing-Maschinen

Mehrband Turing-Maschinen

Bisher haben wir nur Turing-Maschinen mit einem einzigen Band betrachtet. In vielen Fällen gestaltet sich die Konstruktion einer Turing-Maschine zu einem gegebenen Problem aber wesentlich leichter, wenn wir mehr als ein Band zur Verfügung haben.

1.7 Definition
Sei $k \in \mathbb{N}, k > 0$. k-Band-Turing-Maschinen sind analog zu Turing-Maschinen definiert, allerdings haben sie k Bänder und einen Kopf pro Band. Dementsprechend hat die Transitionsfunktion nun die Signatur

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

für deterministische Mehr-Band-Turing-Maschinen bzw.

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

für nicht-deterministische Mehr-Band-Turing-Maschinen. Die Maschine liest in jedem Schritt auf jedem Band die Zelle an der aktuellen Kopfposition, modifiziert diese Zellen und kann die Köpfe unabhängig voneinander bewegen.

In der Initialkonfiguration einer solchen Maschine sind alle Bänder bis auf das erste leer, d.h. gefüllt mit $\ldots \sqcup \sqcup \sqcup \ldots$

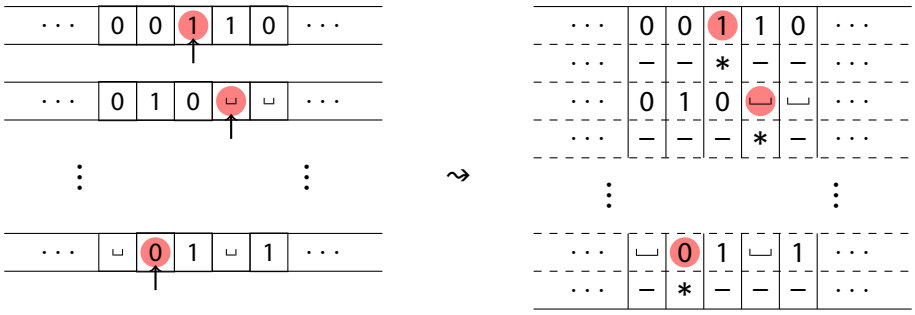
Auch wenn Mehr-Band-Turing-Maschinen auf den ersten Blick mächtiger als die Ein-Band Variante erscheinen, lassen sich, wie wir hier zeigen, mehrere Bänder mithilfe von nur einem simulieren.

1.8 Lemma: Bandreduktion
Zu jeder k-Band-Turing-Maschine M_k gibt es eine Turing-Maschine M, die M_k effizient simuliert. Insbesondere gilt $\mathcal{L}(M_k) = \mathcal{L}(M)$. Falls M_k deterministisch ist, dann ist auch M deterministisch.

Beweis:
M simuliert einen Schritt von M_k durch eine Sequenz von Schritten. Die Idee hierbei ist, den Inhalt der k Bänder von M_k in einem einzigen Band zu speichern. Wir stellen uns vor, dass dieses eine Band in $2k$ Spuren unterteilt ist. Formal ist das Bandalphabet

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

wobei Γ hierbei das Bandalphabet von M_k ist. Die $(2\ell - 1)$-te Komponente eines Buchstaben in Γ' speichert den Inhalt des ℓ-ten Bandes. Die (2ℓ)-te Komponente, die ein Element von $\{*, -\}$ ist, wird verwendet, um die Kopfposition von M_k auf dem ℓ-ten Band durch $*$ zu markieren. Dies ist notwendig, weil M_k die Köpfe auf jedem Band unabhängig voneinander bewegen kann. Es gibt genau ein Vorkommen von $*$ in der 2ℓ-ten Spur, die anderen Einträge sind $-$. Die folgende Abbildung illustriert die Konstruktion.



Ein Schritt von M_k wird von M wie folgt simuliert: M beginnt links beim Symbol $\sqcup$. Hiermit ist wirklich das Symbol $\sqcup \in \Gamma'$ gemeint, nicht ein Vektor, der als Einträge $\sqcup$ hat. M bewegt ihren Kopf nach rechts über das Band, bis sie das erste Symbol $\sqcup$, also den rechten Rand, findet.

Auf dem Weg dahin sammelt M die k Symbole, die sich an den jeweiligen Kopfpositionen befinden und speichert sie im Kontrollzustand. Dies ist möglich, weil diese Symbole wie zuvor besprochen markiert sind.

Sobald M diese Symbole gespeichert hat, kann sie eine Transition von M_k simulieren. Hierzu bewegt sie den Kopf zurück zum Anfang und macht dabei die entsprechenden Änderungen am Bandinhalt, die den Änderungen der Kopfpositionen und Bandinhalten der Transition von M_k entsprechen.

Sobald M wieder links angekommen ist, wechselt M in den entsprechenden Kontrollzustand von M_k. □

Alphabetsreduktion

Reale Computer arbeiten mit Binärzahlen bzw. mit Strings über dem Alphabet $\{0, 1\}$. Man kann auch Turing-Maschinen entsprechend beschränken.

1.9 Theorem: Alphabetsreduktion
Sei $M = (Q, \Sigma, \Gamma, \delta, q_0)$ eine TM. Es gibt eine Abbildung

$$\text{bin}: \Gamma^* \to \{0, 1\}^*$$

und eine TM $M_{\text{bin}} = (Q', \{0, 1\}, \{0, 1, \$, \sqcup\}, \delta', q_0')$ mit

$$w \in \mathcal{L}(M) \subseteq \Sigma^* \qquad \text{gdw.} \qquad \text{bin}(w) \in \mathcal{L}(M_{\text{bin}}) \subseteq \{0, 1\}^* .$$

Wenn M ein Entscheider ist, dann ist auch M_{bin} ein Entscheider.

Beweisskizze:
Wir ordnen jedem Zeichen aus Γ eine Binärkodierung zu. Hierzu benötigen wir $k = \lceil \log_2 |\Gamma| \rceil$ Bits pro Symbol.

Um einen Schritt von M zu simulieren, verhält sich M_{bin} wie folgt:

- Lese die k Bits ab der aktuellen Kopfposition und speichere sie im aktuellen Kontrollzustand.

  (Beachte, dass sich beschränkte Wörter im Kontrollzustand speichern lassen!)

- Wähle die passende Transition von M für den aktuellen Kontrollzustand und das durch die k Bits codierte Symbol aus.

- Gehe zurück und ersetze dabei die k Bits durch die Kodierung des neuen Bandsymbols.

- Bewege den Kopf um k Bits nach links oder rechts, um die Kopfbewegung von M zu simulieren.

- Ändere den Kontrollzustand.

□

Turing-Maschinen mit einseitig unendlichem Band

Wir haben bisher nur Turing-Maschinen betrachtet, die sowohl links als auch rechts unendlich viel Platz auf dem Band haben. Man kann aber beweisen, dass es schon ausreicht wenn das Band nur in eine Richtung unendlich ist. Mit anderen Worten, Turing-Maschinen, deren Band/Bänder nur in eine Richtung unendlich viel Platz zur Verfügung haben sind genauso mächtig wie die TMs mit beidseitig unendlichem Band.

1.10 Definition
Turing-Maschinen mit rechts unendlichem Band sind weitestgehend definiert wie Turing-Maschinen mit beidseitig unendlichem Band, mit folgenden Ausnahmen.

- Das Bandalphabet enthält ein zusätzliches Symbol $\$ \in \Gamma$, den (linken) Endmarker, mit $\$ \notin \Sigma$ und $\$ \neq \sqcup$.

- Der Endmarker darf weder nach links überschritten werden, noch darf er überschrieben werden.

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

- Die Startkonfiguration bei Eingabe $w \in \Sigma^*$ ist $q_0\$w$, wenn q_0 der Startzustand ist. Der Kopf zeigt also bei der Startkonfiguration auf den Endmarker.

Insbesondere sind die Sprache von TMs mit rechts unendlichem Band analog definiert wie bei TMs mit beidseitig unendlichem Band.

Wie bereits erwähnt, führt die Einschränkung nur einseitig unendliche Bänder zu verwenden nicht dazu, dass weniger Sprachen erkannt werden können. Das zugehörige Lemma ist dem Leser als Übung überlassen.

1.11 Lemma
Zu jeder TM $M_{\leftrightarrow}$ mit beidseitig unendlichem Band gibt es eine TM M mit rechts unendlichem Band, die $M_{\leftrightarrow}$ effizient simuliert. Insbesondere gilt $\mathcal{L}(M_{\leftrightarrow}) = \mathcal{L}(M)$.

Beweis: Übungsaufgabe. □

Die Einschränkung auf rechts unendliche Bänder wird in Beweisen weiter hinten im Skript nützlich sein, siehe z.B. Kapitel 10.

C) Korrespondenz von linear-beschränkten Automaten und kontextsensitiven Sprachen

Für die Charakterisierung der kontextfreien Sprachen definieren wir Turing-Maschinen, welche nur den Teil des Bands benutzen, auf dem die Eingabe steht (plus Endmarker links und rechts).

1.12 Definition
Ein **linear-beschränkter Automat** (**LBA**) ist eine nicht-deterministische Turing-Maschine $M = (Q, \Gamma, \Sigma \cup \{\$_L, \$_R\}, q_0, \delta, Q_F)$.

Anstatt der Blank-Symbole stehen nun der **linke Endmarker** $\$_L$ und der **rechte Endmarker** $\$_R$ links bzw. rechts von der Eingabe auf dem Band. Mit den Endmarkern führen wir noch zwei Einschränkungen ein.

1. Der linke (rechte) Endmarker darf nicht nach links (rechts) überschritten werden, d.h. $\forall q \in Q : \nexists\, (q', \$_L, L) \in \delta(q, \$_L)$ und $\forall q \in Q : \nexists\, (q', \$_R, R) \in \delta(q, \$_R)$.
2. Die Endmarker dürfen nicht überschrieben werden, d.h. $\forall q \in Q : \nexists\, (q', a, d) \in \delta(q, \$_L)$ und $\forall q \in Q : \nexists\, (q', a, d) \in \delta(q, \$_R)$ für $a \in \Gamma$, $\$_L \neq a \neq \$_R$ und $d \in \{L, R, N\}$.

Die Sprache eines linear beschränkten Automaten ist die Menge aller Wörter w, so dass eine Berechnung, gemäß der beiden Einschränkungen, von der Startkonfiguration $q_0\$_L w\$_R$ zu einer akzeptierenden Konfiguration existiert.

$$\mathcal{L}(M) = \left\{ w \in \Sigma^* \;\middle|\; q_0\$_L w\$_R \to^* uq'v \in \Gamma^* Q_F \Gamma^* \right\}.$$

1.13 Bemerkung
Ein Band-Kompressions Resultat aus der Komplexitätstheorie zeigt dass man mit linearem Platz (in der Größe des Eingabewortes) auf dem Band die gleiche Berechnungsmächtigkeit hat wie nur mit dem Platz des Eingabewortes. Wir können also bei den Berechnungen von LBAs linear viel Platz in der Größe des Eingabewortes auf dem Band verwenden. Daher kommt auch der Name der linear-beschränkten Automaten.

Folgendes Theorem zeigt dass die LBA-akzeptierten Sprachen genau die kontextfreien Sprachen sind.

1.14 Theorem: Kuroda 1964
Eine Sprache $\mathcal{L} \subseteq \Sigma^*$ wird von einem LBA akzeptiert gdw. sie kontextsensitiv ist.

Beweis:

„$\Leftarrow$" Es sei eine kontextsensitive Sprache $\mathcal{L} = \mathcal{L}(G)$ durch eine Typ-1-Grammatik $G = (N, \Sigma, P, S)$ gegeben. Wir geben einen LBA M an, welcher $\mathcal{L}$ akzeptiert.

Die Eingabe des LBA ist ein Wort $w \in \Sigma^*$ (mit Endmarkern $\$_L, \$_R$ links und rechts von w). Die Idee des Beweises ist die Produktionen der Grammatik rückwärts mithilfe der Turing-Maschine zu simulieren bis das Startsymbol S gefunden wurde. Dazu wählt der LBA nicht-deterministisch eine Produktion $\alpha \to \beta \in P$ aus und sucht nicht-deterministisch ein Vorkommen von β auf dem Band aus. Dann wird das Vorkommen von β auch dem Band durch α ersetzt. Falls $|\alpha| < |\beta|$, werden Buchstaben nach links kopiert um Lücken zu schließen. Wird durch diesen Schritt S erreicht, hält die Turing-Maschine und akzeptiert. Ansonsten wird die beschriebene nicht-deterministische Prozedur wiederholt.

Aufgrund der Länge-erhaltenden Eigenschaft der Produktionen ($|\beta| \geqslant |\alpha|$), benötigt die Turing-Maschine nicht mehr Platz auf dem Band als durch die Eingabe w gegeben. Dadurch verlassen wir den durch die Endmarker beschränkten Platz auf dem Band nicht.

„$\Rightarrow$" Es sei ein LBA $M = (Q, \Gamma, \Sigma \cup \{\$_L, \$_R\}, q_0, \delta, Q_F)$ gegeben mit $\mathcal{L} = \mathcal{L}(M)$. Für diese Richtung des Beweises geben wir eine kontextsensitive Grammatik G an, welche mithilfe der abgeleiteten Satzformen die Konfigurationen von dem LBA M simuliert. Falls das als Eingabe für M gegebene Wort w von M akzeptiert wird, soll w aus den Satzformen der Grammatik herleitbar sein. Da wir aufgrund der Länge-erhaltenden Eigenschaft der Produktionen keine Symbole aus Satzformen löschen können, dürfen die Satzformen nicht länger als $|w|$ werden. Dies wird uns aber durch die lineare Beschränktheit von M garantiert.

Technisch konstruieren wir G wie folgt. Die Nicht-Terminale sind Tupel $(\text{act}_1, a_1), (\text{act}_2, a_2), \ldots, (\text{act}_n, a_n)$, wobei

$\text{act}_1, \text{act}_2 \ldots, \text{act}_n$ den Inhalt des Bandes darstellt und

$a_1, a_2, \ldots, a_n$ das Eingabewort w des LBA.

Die Nicht-Terminale act_i stammen aus dem Alphabet

$$\Delta := \Delta' \cup (Q \times \Delta') \cup (\{\$_L, \$_R\} \times Q \times \Gamma)$$

wobei $\Delta' := \Gamma \cup (\{\$_L, \$_R\} \times \Gamma)$ ist.

Betrachten wir beispielsweise die Konfiguration $\$_L\ x\ q\ y\ a\ z\ \$_R$ für $x, y, z \in \Gamma, a \in \Sigma$, welche während einer Berechnung von M mit Eingabewort $w = abab$ auftritt. Diese Konfiguration würde durch die Satzform

$$((\$_L, x), a).((q, y), b).(a, a).((\$_R, z), b)$$

dargestellt werden. Wir fassen dabei immer den linken Endmarker mit dem linkesten Symbol auf dem Band (bzw. den rechten Endmarker mit dem rechtesten Symbol) so dass die Länge der Eingabe w mit der Länge des Bandinhaltes übereinstimmt.

Wir können nun einfach die Transitionen des LBAs mithilfe der Produktionen simulieren, indem wir nur die aktuelle Konfiguration beachten. Falls zum Beispiel $(q', b, R) \in \delta(q, a)$, erhalten wir die Produktion

$$((q, a), x).(c, y) \to (b, x).((q', c), y) \quad \text{f.a. } c \in \Gamma \text{ und } x, y \in \Sigma.$$

Für die Randfälle, d.h. der Schreib/Lese-Kopf zeigt auf einen der Endmarker, sind Sonderfälle nötig welche wir uns hier kurz anschauen. Für den Rand benutzen wir

- $(q, \$_L, a)$, um darzustellen dass der Schreibe/Lese-Kopf auf dem linken Endmarker steht,
- $(\$_L, q, a)$, falls der Kopf auf dem ersten Symbol auf dem Band steht,
- $(q, \$_R, a)$, falls der Kopf auf dem rechten Endmarker steht und
- $(\$_L, q, a)$, falls der Kopf auf dem letztem Symbol auf dem Band steht.

Die Produktionen, welche wir einführen um die Transitionen des LBA zu simulieren, haben eine leicht abgeänderte Form im Vergleich zu den oben beschriebenen Produktionen. Um beispielsweise $(q', \$_L, R) \in \sigma(q, \$_L)$ zu simulieren, erhalten wir die Produktion

$$((q, \$_L, a), x) \to (\$_L, q, a, x) \quad \text{f.a. } a \in \Gamma \text{ und } x \in \Sigma.$$

Die bisherigen Überlegungen führen zu folgender Konstruktion der Typ-1- Grammatik $G = (N; \Sigma, P, S)$. Die Nicht-Terminal N sind gegeben durch $N = \{S, A\} \cup (\Delta \times \Sigma)$. Die Nicht-Terminale S und A verwenden wir, um die initiale Konfiguration des simulierten LBAs zu bilden und die Nicht-Terminale aus $(\Delta \times \Sigma)$ sind die Tupel, welche wir am Anfang des Beweises beschrieben haben.

Die Produktionen der Grammatik sind gegeben durch

$$\begin{aligned}
P = {} & \{S \to A.((\$_R, a), a) \mid a \in \Sigma\} \\
& \cup \{A \to A.(a, a) \mid a \in \Sigma\} \\
& \cup \{A \to ((q_0, \$_L, a), a) \mid a \in \Sigma\} \\
& \cup \text{Produktionen die } M \text{ simulieren (siehe oben)} \\
& \cup \left\{((q_F, a), b) \to b \;\middle|\; a \in \Delta', q_F \in Q_F, b \in \Sigma\right\} \\
& \cup \{((\$, q_F, a), b) \to b \mid \$ \in \{\$_L, \$_R\}, q_F \in Q_F, a \in \Gamma, b \in \Sigma\} \\
& \cup \left\{(a, b) \to b \;\middle|\; a \in \Delta', b \in \Sigma\right\}.
\end{aligned}$$

Die ersten drei Mengen an Produktionen erlauben es uns Satzformen abzuleiten, welche die initiale Konfigurationen des LBAs M für alle möglichen Eingabeworte $w \in \Sigma^*$ nachbilden. Die vierte Menge haben wir bereits oben beschrieben. Diese Produktionen dienen dazu, die Berechnung des LBAs zu simulieren. Wenn die (simulierte) Berechnung eine akzeptierende Konfiguration erreicht, können wir mit den letzten drei Mengen an Produktionen das Eingabewort w der Berechnung ableiten.

□

1.15 Bemerkung

Die Konstruktion ist analog, wenn wir die Länge-erhaltende Eigenschaft der Grammatik und die Längen-Beschränktheit bei der Turing-Maschine fallen lassen.

1.16 Korollar

Die NTM-akzeptierten Sprachen sind genau die rekursiv aufzählbaren Sprachen.

D) Determinismus

Aus der Vorlesung „Theoretische Informatik 1“ wissen wir bereits, dass deterministische und nicht-deterministische endliche Automaten die gleiche Sprachen akzeptieren (Stichwort Potenzmengenkonstruktion). Für deterministische Kellerautomaten ist bekannt, dass sie nur eine strikte Teilmenge der kontextfreien Sprachen, und damit der Sprachen, welche von nicht-deterministischen Kellerautomaten erkannt werden, akzeptieren.

Dazu stellte Kuroda 1964 zwei Fragen.

(1) Sind die durch deterministische LBAs (DLBAs) akzeptierten Sprachen genau die Sprachen welche von den nicht-deterministischen LBAs (NLBAs) akzeptiert werden? Die Antwort auf diese Frage ist noch offen!

(2) Sind die NLBA-Sprachen abgeschlossen unter Komplement?

Kuroda konnte zeigen, dass wenn (2) nicht gilt, dann auch (1) nicht gilt. Allerdings ist diese Aussage nicht von Nutzen, da von Immermann und Szelepcsényi gezeigt wurde, dass die NLBA-akzeptierten Sprachen tatsächlich unter Komplement abgeschlossen sind. Diesen Satz zeigen wir in einer späteren Vorlesung.

Der folgende Satz zeigt, dass (unbeschränkte) deterministischen und nicht-deterministischen Turing-Maschinen die gleiche Sprache akzeptieren.

1.17 Theorem
Eine Sprache $\mathcal{L}$ wird von einer NTM M_1 akzeptiert gdw. $\mathcal{L}$ von einer DTM M_2 akzeptiert wird.

Die Richtung $\Leftarrow$ des Beweises ist trivial, da jede DTM insbesondere auch eine NTM ist.

Für die andere Richtung des Beweises müssen wir eine DTM M_2 konstruieren, welche die Sprache unserer gegebenen NTM M_1 akzeptiert. Man könnte auf die Idee kommen, den Satz mit Hilfe der, aus „Theoretische Informatik I“ bekannten, Potenzmengenkonstruktion zu lösen. Dies wird allerdings nicht auf einfache Art und Weise funktionieren: Angenommen M_1 könnte zu einem Zeitpunkt entweder in der Konfiguration $a\ q\ b$ oder in der Konfiguration $b\ q'\ a$ sein. In der Potenzmengenkonstruktion würden wir dies durch $\{a, b\}\ \{q, q'\}\ \{a, b\}$ repräsentieren. Nun scheint es, als wäre $a\ q'\ a$ eine Möglichkeit für die aktuelle Konfiguration von M_1, dies ist allerdings nicht der Fall.

Ein besserer Ansatz ist es, M_2 so zu konstruieren, dass sie zu einer Eingabe w den **Berechnungsbaum** von M_1 zu w durchsucht. Ein Berechnungsbaum ist dabei wie folgt definiert.

1.18 Definition
Sei M_1 eine NTM und $w \in \Sigma^*$ eine Eingabe. Der **Berechnungsbaum** von M_1 zu w ist ein (potentiell unendlich hoher) Baum, der induktiv wie folgt definiert ist:

- Die Wurzel des Baumes ist markiert mit der Konfiguration $\varepsilon\, q_0\$w$.
- Für jeden Knoten des Baumes, der mit einer Konfiguration $u\, q\, a.v$ markiert ist, hat der Baum einen Kindknoten pro Element von $\delta(q, a)$, der mit der entsprechenden resultierenden Konfiguration markiert ist. Falls $\delta(q, a) = \varnothing$, ist der Knoten ein Blatt.

Falls das Eingabewort w von der NTM M_1 akzeptiert wird, gibt es eine Berechnung, die nach endlich vielen Schritten eine akzeptierenden Konfiguration erreicht. Dementsprechend gibt es in dem Berechnungsbaum einen endlichen Pfad von der Wurzel zu einem Knoten, welcher mit der akzeptierenden Konfiguration markiert ist. Dieser kann durch einen passenden Suchalgorithmus gefunden werden.

Beim Durchsuchen des Berechnungsbaumes gibt es nun (mindestens) zwei Möglichkeiten: Tiefensuche oder Breitensuche. Tiefensuche wird nicht das gewünschte Resultat liefern, da es unendlich lange Pfade in dem Berechnungsbaum geben kann, in denen sich die Tiefensuche verliert. Falls es gleichzeitig auch eine endliche akzeptierende Berechnung zu w gäbe, würden wir diese mit Tiefensuche also eventuell nicht finden.

Unsere Simulation wird also eine Breitensuche im Berechnungsbaum implementieren.

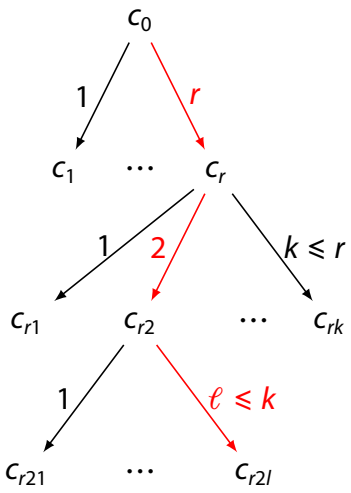
Beweis des Theorems:
Wie bereits erwähnt ist die Richtung von rechts nach links trivial. Für die andere Richtung bemerken wir, dass es zu jeder Konfiguration von M_1 zwar eventuell mehrere, aber in jedem Fall nur endlich viele Nachfolger gibt. Die Zahl der Nachfolger ist beschränkt durch

$$r = \max_{q \in Q} \max_{a \in \Gamma} |\delta(q, a)| \,.$$

Jede Berechnung zu einer festen Eingabe, d.h. jeder Pfad im Berechnungsbaum, lässt sich durch die Wahlen der Nachfolger, die in der Berechnung getroffen wurden, beschreiben.

Eine endliche Berechnung lässt sich somit durch eine endliche Sequenz von Zahlen in $\{1, \ldots, r\}$ charakterisieren.

Betrachte beispielsweise folgenden Berechnungsbaum.



Hierbei haben wir die Wahlen der Nachfolger an die Kanten geschrieben, dies ist eigentlich nicht Bestandteil des Berechnungsbaums. Die rot markierte Berechnung entspricht der Sequenz $r.2.\ell \in \{1, \ldots, r\}^*$.

Beachte, dass nicht jede Sequenz aus $\{1, \ldots, r\}^*$ einer validen Berechnung entspricht, da es zum Teil weniger als r Nachfolger gibt.

Wir konstruieren nun die deterministische Turing-Maschine M_2 mit 3 Bändern. Wie wir in Lemma 1.8 gezeigt haben, kann man deterministische Turing-Maschinen mit mehreren Bändern mit deterministischen Ein-Band-Turing-Maschinen simulieren.

1. Auf dem ersten Band steht die Eingabe w. Diese wird im Laufe der Berechnung nicht verändert.

2. Auf dem zweiten Band wird eine Sequenz aus $\{1, \ldots, r\}^*$ gespeichert.

   M_2 wird alle Sequenzen aus $\{1, \ldots, r\}^*$ der Reihe nach auf eine systematische Art und Weise erzeugen:

   - Sequenzen werden mit aufsteigender Länge generiert.
   - Innerhalb derselben Länge werden die Sequenzen gemäß lexikographischer Sortierung generiert.

   Die Reihenfolge der Sequenzgenerierungen ist also wie folgt:

   $$\varepsilon, 1, 2, \ldots, r, 11, 12, \ldots, 1r, 21, \ldots, 2r, \ldots, r1, \ldots, rr, 111, \ldots$$

   Dies entspricht einem Breitendurchlauf durch den Berechnungsbaum.

3. Das dritte Band wird zur Berechnung genutzt.

Pro Sequenz $s \in \{1, \ldots, r\}^*$ auf dem zweiten Band arbeitet M_2 die folgenden Schritte ab:

- Leere das dritte Band (indem der Inhalt mit Blanks überschrieben wird).
- Kopiere die Eingabe vom ersten auf das dritte Band.
- Simuliere M_1 für $|s|$ Schritte, das heißt einen Schritt pro Eintrag in s.

  Hierbei werden die Einträge von s genutzt, um den Nichtdeterminismus aufzulösen. Sei $s_i \in \{1, \ldots, r\}$ der i-te Eintrag von s, dann wird M_2 im i-ten Schritt den s_i-ten Nachfolger auswählen.
- Falls bei der Simulation eine akzeptierende Konfiguration von M_1 angetroffen wird, akzeptiere.

Wenn es eine Sequenz $s \in \{1, \ldots, r\}^*$ gibt, welche zu einer akzeptierenden Berechnung von M_1 auf Eingabewort w führt, dann wird diese Sequenz auch auf Band 2 generiert. Die Simulation mithilfe dieser Sequenz führt dann zu einer akzeptierenden Konfiguration von M_1 und damit akzeptiert auch M_2 das Wort w. Falls es keine solche Sequenz gibt, akzeptiert M_2 das Wort w auch nicht.

□

1.19 Bemerkung
Diese Konstruktion klappt nicht für LBAs. Weil die Sequenzen $s \in \{1, \ldots, r\}^*$ sehr lang werden können, wird die lineare Beschränktheit der LBAs möglicherweise verletzt.

2. Der Satz von Immermann & Szelepcsényi

Mit dem Satz von Immermann & Szelepcsényi konnte die zweite Frage von Kuroda positiv beantwortet werden. Die kontextsensitiven Sprachen sind tatsächlich abgeschlossen unter Komplement.

Das Resultat wurde unabhängig 1988 und 1987 von Neil Immermann, einem Professor an der University of Massachusetts Amherst, und von Robert Szelepcsényi, einem Studenten in Bratislava, gezeigt. Beide erhielten für den Satz 1995 den Gödel-Preis. Das Resultat führte die fundamentale Technik des **induktiven Zählens** in die Komplexitätstheorie ein.

2.1 Theorem: Immermann & Szelepcsényi, 1988 &1987
Wenn eine Sprache $\mathcal{L} \subseteq \Sigma^*$ kontextsensitiv ist, dann ist auch ihr Komplement $\bar{\mathcal{L}}$ kontextsensitiv.

Sei also $\mathcal{L} = \mathcal{L}(G)$ für eine Typ-1-Grammatik $G = (N, \Sigma, P, S)$. Um die Aussage zu beweisen, konstruieren wir einen NLBA, welcher ein Eingabewort $w \in \Sigma^*$ akzeptiert, falls es **keine Ableitung** $S \Rightarrow_G^* w$ in der Gammatik G gibt.

Wir reduzieren dieses Problem auf **Unerreichbarkeit** in einem Graphen, also ob ein gegebener Knoten in dem Graphen von einem festgelegten Startknoten **nicht erreichbar** ist. Dafür betrachten wir den Graph $\text{Graph}_{|w|}$, welcher wie folgt definiert ist.

2.2 Definition
Der Ableitungsgraph $\text{Graph}_{|w|}$ zu einer Grammatik $G = (N, \Sigma, P, S)$ und einem Wort $w \in \Sigma^*$ hat als Knotenmenge die Menge der Satzform der Länge $\leqslant |w|$ und die Kanten sind durch die Ableitungsrelation $\Rightarrow_G$ der Grammatik gegeben. Formal haben wir

$$\text{Graph}_{|w|} = ((\Sigma \cup N)^{\leqslant |w|}, \{(\alpha, \beta) \mid \alpha \Rightarrow_G \beta\}).$$

Per Definition des Ableitungsgraphen $\text{Graph}_{|w|}$ zu G und w, gilt folgende Aussage.

Es gibt **keine** Ableitung $S \Rightarrow_G^* w$ gdw. es keinen Pfad von S zu w in $\text{Graph}_{|w|}$ gibt.

Der Kern des Beweises ist es also zu zeigen, dass man Unerreichbarkeit in einem Graphen exponentieller Größe (in $|w|$) mit einer nicht-deterministischen, linear-beschränkten Turing-Maschine lösen kann. Man bemerke, dass, aufgrund des Band-Kompressions Resultats, welches wir kurz in der letzten Vorlesung angesprochen haben, wir linear viel Platz (in $|w|$) auf dem Band verwenden können.

Zusammenfassung der bisherigen Einsichten

- Wir wollen einen nicht-deterministischen Algorithmus (bzw. eine NTM) konstruieren, welcher gegeben einem Graphen $\mathcal{G} = (V, \to)$ (in unserem Fall $\text{Graph}_{|w|}$) und zwei Knoten s (hier das Startsymbol S) und t (hier w) entscheidet, ob es keinen Pfad von s nach t in $\mathcal{G}$ gibt.
- Unser Algorithmus darf außerdem nur logarithmisch viel Platz in der Größe von $\mathcal{G}$ auf dem Band verbrauchen. Dies garantiert uns dass unsere konstruierte NTM auch ein NLBA ist. In unserem Fall bedeutet logarithmisch viel Platz in der Größe von $\mathcal{G}$ zu haben, dass man linear viel Platz in $|w|$ hat.

  Diese Aussage gilt da $\text{Graph}_{|w|}$ $(|N| + |\Sigma| + 1)^{|w|} = c^{|w|}$ viele Knoten und damit höchstens $(c^{|w|})^2 = c^{2|w|}$ viele Kanten hat. Logarithmisch viel Platz in der Größe von $\mathcal{G}$ bedeutet also

  $$\log(c^{2|w|}) = 2\log(c)|w|,$$

  was linear in $|w|$ ist.

Idee des Algorithmus

Eine naive Idee wäre, alle von s aus erreichbaren Knoten aufzuzählen und zu überprüfen, dass t sich nicht darunter befindet. Zu Überprüfen, ob ein einzelner Knoten erreichbar ist, wie wir in Kürze sehen werden, mit einem NLBA möglich. Alle solchen Knoten zu speichern kostet allerdings mehr als logarithmisch viel Platz.

Der Beweis löst dieses Problem indirekt, indem er den Algorithmus in zwei Teile zerlegt.

1. Nehmen wir zunächst an, die Anzahl N aller von s aus erreichbaren Knoten wäre bekannt. (Diese Anzahl lässt sich mit logarithmischem Platz speichern.)

   Wir zeigen, dass man unter diese Annahme mit Hilfe von Zählen überprüfen kann, ob t nicht erreichbar ist (mit einem nicht-deterministischen Algorithmus, welcher nur logarithmisch viel Platz in $|\mathcal{G}|$ braucht).

2. Um diese Zahl N zu berechnen, verwenden wir induktives Zählen: Wir berechnen die Anzahl $R(i)$ der in i-Schritten erreichbaren Knoten, unter der Annahme, dass wir $R(i-1)$ kennen. Es gilt $N = R(n)$, da jeder erreichbare Knoten durch einen einfachen Pfad erreicht werden kann.

Schritt 1: Nicht-Erreichbarkeit unter Verwendung von N

Wir gehen im folgenden davon aus, dass

$$N = \left|\{v \in V \mid s \to^* v\}\right|,$$

die Anzahl der von *s* aus erreichbaren Knoten bereits bekannt ist.

2.3 Algorithm: unreach

unreach($\mathcal{G}$,s,t)

```
1:  count := 0
2:  for Knoten v do
3:      Rate, ob v von s aus erreichbar ist
4:      if Ja then
5:          Rate einen Pfad von s nach v der Länge ⩽ n
6:          if Falls das geratene kein gültiger Pfad nach v ist then
7:              return false                    // Erreichbarkeit oder Pfad falsch geraten
8:          end if
9:          if v = t then
10:             return false                                  // t ist erreichbar
11:         end if
12:         count++                                // Erreichbarer Knoten gefunden
13:     end if
14: end for
15: if count ≠ N then
16:     return false                    // für mindestens einen Knoten falsch geraten
17: else
18:     return true                // immer richtig geraten und t wirklich unerreichbar
19: end if
```

Der Algorithmus **unreach** läuft mit (nicht-deterministischem) logarithmischen Platz. Um in Zeile 3 vom Algorithmus die Platzschranke nicht zu verletzen, raten wir den Pfad knotenweise. Es reicht also den aktuellen Knoten und die bisherige Länge des Pfades zu speichern, was mit logarithmischem Platz möglich ist. Die Anzahl der von *s* aus erreichbaren Knoten und die Variable *count* können höchstens so groß wie die Anzahl an Knoten *n* aus $\mathcal{G}$ sein. Binär kodiert benötigen sie also höchstens $\log(n)$ viel Platz.

2.4 Lemma

Sei *N* initialisiert mit der Anzahl der von *s* aus erreichbaren Knoten. Es gibt eine Berechnung zum nicht-deterministischen Algorithmus, die *unreach*$(\mathcal{G}, s, t)$ true zurück gibt, genau dann wenn es keinen Pfad von *s* nach *t* gibt.

Beweis:

Der Algorithmus kann nur dann true zurückgeben, wenn wir genau die erreichbaren Knoten als erreichbar raten:

- Wenn wir einen unerreichbaren Knoten als erreichbar raten, schlägt die Verifikation (Zeile 4 bzw. Zeile 7) fehl, egal welchen Pfad wir raten.
- Wenn wir zu wenige Knoten als erreichbar raten, schlägt die Überprüfung der Anzahl in Zeile 15 fehl.

Wenn t wirklich nicht erreichbar ist, dann gibt es eine Berechnung, nämlich diese Berechnung, die true zurückgibt.

Angenommen es gibt eine Berechnung, die true zurückgibt. Dann kann t nicht erreichbar sein: Wir haben alle erreichbaren Knoten identifiziert, und t war nicht darunter, sonst hätten wir in Zeile 9/10 false zurückgegeben. □

Schritt 2: Induktives Zählen

Wir wollen die Zahl

$$R(i) = \#\{v \in V \mid v \text{ in } \leqslant i \text{ Schritten von } s \text{ aus erreichbar } \}$$

berechnen, und zwar **induktiv**, d.h. unter der Annahme, dass $R(i-1)$ bekannt ist.

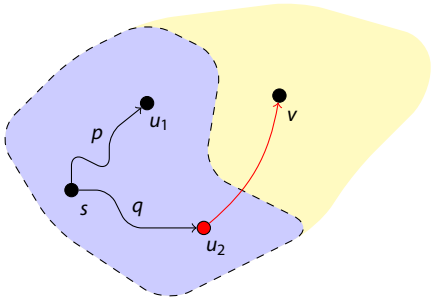
Man beachte:

- Es gilt $R(n) = N$, da jeder erreichbare Knoten auch mit einem einfachen Pfad (der insbesondere Länge $\leqslant n$ hat) erreichbar ist.
- Es gilt $R(0) = 1$, da nur s selbst mit einem Pfad der Länge 0 erreichbar ist.
- Jeder Knoten v, der in i Schritten erreichbar ist, ist Nachfolger eines Knotens u, der in $i-1$ Schritten erreichbar ist.

Unsere Idee ist also wie folgt: Wir zählen alle Knoten v, die in höchstens i Schritten erreichbar sind, wie folgt: Für jeden Kandidaten v gehen wir alle Knoten u durch, die in höchstens $i-1$ Schritten erreichbar sind, und überprüfen, ob $u = v$ gilt oder ob v ein Nachfolger von u ist.

Das Problem hierbei ist, dass wir alle Knoten u nicht gleichzeitig speichern können. Wir verwenden wieder die Idee aus dem ersten Schritt, um sicherzustellen, dass wir genau die Knoten u, die in höchstens $i-1$ Schritten erreichbar sind, auch als in $\leqslant i-1$ Schritten erreichbar raten.

Die folgende Grafik stellt diese Idee dar.



Der blaue Bereich stellt die Knoten dar, die von s aus in höchstens $i - 1$ Schritten erreichbar sind. Nehmen wir an, dass der Algorithmus gerade überprüfen möchte, ob v erreichbar ist. Er sucht dann nach einem Vorgänger, der in $i - 1$ Schritten erreichbar ist. Wenn der Algorithmus nun u_1 betrachtet und den Pfad p korrekt rät, wird er feststellen, dass u_1 kein Vorgänger von v ist. Der Knoten u_2 ist auch in höchstens $i - 1$ Schritten erreichbar und hat eine Kante zu v. Also ist v in i Schritten erreichbar, und der Algorithmus inkrementiert $R(i)$.

2.5 Algorithm: #reach

 # reach($\mathcal{G}$,s)

```
R(0) = 1                                          // Nur s selbst erreichbar in 0 Schritten.
for i = 1, ..., n do
    R(i) := 0                                                    // Initialisierung
    for alle Knoten v do
        // Wir wollen überprüfen, ob v in ⩽ i Schritten erreichbar ist.
        // Wir finden alle Knoten u, die in ⩽ i − 1 Schritten erreichbar sind
        // und überprüfen, ob v Nachfolger ist.
        count := 0
        for alle Knoten u do
            Rate, ob u von s aus in ⩽ i − 1 Schritten erreichbar ist
            if Ja then
                Rate einen Pfad von s nach u der Länge ⩽ i − 1
                if Falls das geratene kein gültiger Pfad nach u ist then
                    return false              // Erreichbarkeit oder Pfad falsch geraten
                end if
                count + +
                if u = v oder u → v then
                    R(i) + +
                    goto nächste Iteration von v-Schleife
                end if
            end if
        end for
        if count ≠ R(i − 1) then
            return false          // Knoten falsch als unerreichbar geraten für ein u
        end if
    end for
end for
return R(n)
```

2.6 Lemma

$\#reach(G, s)$ berechnet für jede Zahl $i \in \{0, \ldots, n\}$ korrekt $R(i)$.

Beweis:

Beweis durch Induktion über i. Der Basisfall $i = 0$ ist klar.

Wie zuvor liefert die Berechnung nur dann nicht false, wenn in jedem Durchlauf genau die erreichbaren Knoten u als erreichbar geraten werden. Im Durchlauf für Knoten v erhöhen wir $R(i)$ in so einer Berechnung genau dann um 1, genau dann, wenn v wirklich erreichbar ist.

Genau dann gibt es nämlich einen Vorgänger von v, der in $\leqslant i - 1$ Schritten erreichbar ist. □

Zusammenfassung

Um zu prüfen ob t in $\mathcal{G}$ von s aus nicht erreichbar ist, führen wir zuerst Algorithmus **# reach(**$\mathcal{G}$**,s)** aus um N, die Anzahl von s aus erreichbaren Knoten, zu berechnen. Danach führen wir **unreach(**$\mathcal{G}$**,s,t)** mit dem zuvor berechneten N aus. Beide nicht-deterministische Algorithmen benötigen nur logarithmisch viel Platz (in $|\mathcal{G}|$) auf dem Band. Damit benötigt der komplette Algorithmus nur $\mathcal{O}(\log(|\mathcal{G}|))$ viel Platz. Wir haben am Anfang von diesem Kapitel argumentiert, dass daraus die Existenz eines NLBA, der $\bar{\mathcal{L}}$ akzeptiert, folgt.

3. Berechenbarkeit

In diesem Kapitel, möchten wir den intuitiven Begriff der berechenbaren Funktionen formalisieren.

Das zugehörige Problem, das **Berechnungsproblem**, ist gegeben durch eine Funktion $f: M_1 \to M_2$. Ziel ist, zu jedem Wert $m \in M_1$ den Funktionswert $f(m) \in M_2$ durch einen Algorithmus zu berechnen.

Berechnungsproblem zu Funktion f
Gegeben: Wert $m \in M_1$
Berechne: Funktionswert $f(m)$

Im Folgenden wollen wir Turing-Maschinen nutzen, um Berechnungsprobleme zu lösen. Wir werden eine Funktion **berechenbar** nennen, wenn sie sich als Turing-Maschine implementieren lässt.

Des Weiteren, führen wir die Begriffe der **entscheidbaren**/**semi-entscheidbaren** Eigenschaften, sowie der **aufzählbaren**/**rekursiv-aufzählbaren** Mengen ein.

Bereits in den 1930er Jahren waren Algorithmen bekannt um manchen Funktionen zu Berechnen (z.B. Algorithmen für das Gaußsche Eliminationsverfahren, Taylor und Newton Approximation, …), allerdings gab es keine allgemeine Definition von Berechenbarkeit. Aber ohne einen solchen Begriff ist es nicht möglich zu zeigen, dass manche Funktionen **nicht berechenbar** sind. In dieser Hinsicht war Turings Definition erfolgreich, da sie scheinbar die Intuition der Berechenbarkeit fassen kann. Diese Annahme ist, wie wir schon in einem vorherigen Kapitel gesehen haben, bekannt als Church-Turing-These.

Diese Annahme wird dadurch gestützt, dass alle bisher vorgeschlagenen Berechnungsmodelle sich auf die Turing-Maschinen reduzieren lassen. Ein paar Beispiele sind

- die primitiv rekursive & μ-rekursive Funktionen (Kurt Gödel 1965, Jacques Herbrand),
- das λ-Kalkül (Alonzo Church 1933, Stephen C. Kleene 1935) und
- die Kombinatorische Logik (Moses Schönfinkel 1924, Haskell B. Curry 1929).

A) Berechenbare Funktionen

Im Folgenden betrachten wir nicht bloß totale Funktionen, sondern auch partielle Funktionen, Funktionen $f\colon \Sigma_1^* \to_p \Sigma_2^*$, die nicht unbedingt zu allen Werten $w \in \Sigma_1^*$ einen Funktionswert haben. Das heißt, dass es erlaubt ist, dass $f(w)$ undefiniert ist.

Intuitiv wollen wir eine solche Funktion $f\colon \Sigma_1^* \to_p \Sigma_2^*$ **berechenbar** nennen, wenn es einen Algorithmus gibt, der eine Eingabe $w \in \Sigma_1^*$ nimmt, und

- falls $f(w)$ definiert ist, nach endlich vielen Schritten akzeptiert und $f(w)$ ausgibt,
- falls $f(w)$ nicht definiert ist, nicht anhält oder nicht akzeptiert.

Umgekehrt, berechnet jeder (deterministische) Algorithmus eine partielle Funktion.

3.1 Beispiel
Betrachte die Funktion $f\colon \Sigma_1^* \to_p \Sigma_2^*$ (für beliebige Σ_1, Σ_2), die auf allen Werten undefiniert ist, d.h. $f(w) =$ undefiniert für alle $w \in \Sigma_1^*$. Diese Funktion wird berechnet durch den folgenden Algorithmus:

```
while true do
    skip
end while
```

3.2 Beispiel
Betrachte die Funktion $f_\pi\colon \mathbb{N} \to \mathbb{N}$, mit

$$f_\pi(n) = \begin{cases} 1 & \text{, falls } n \text{ ein Präfix der Dezimaldarstellung von } \pi \text{ ist,} \\ 0 & \text{, sonst.} \end{cases}$$

Es gilt z.B. $f_\pi(314) = 1, f_\pi(5) = 0$. Diese Funktion ist berechenbar, da es Algorithmen gibt, die π auf beliebig viele Dezimalstellen approximieren. Sei n die Eingabe, und k die Länge der Dezimaldarstellung von n.

Berechne π', eine Approximation von π, die auf $k-1$ Nachkommastellen genau ist.
Vergleiche die erste Stelle von n mit 3 und die weiteren Stellen mit den Nachkommastellen von π'.
Gebe 1 zurück, falls der Vergleich erfolgreich ist, 0 sonst.

3.3 Beispiel
Sei nun die Funktion $g_\pi\colon \mathbb{N} \to \mathbb{N}$, mit

$$f(n) = \begin{cases} 1 & \text{, falls } n \text{ ein Infix der Dezimaldarstellung von } \pi \text{ ist,} \\ 0 & \text{, sonst.} \end{cases}$$

Es ist nicht bekannt, ob diese Funktion berechenbar ist. Der Trick aus dem vorherigen Beispiel, wo wir π genau genug approximieren um die Frage zu beantworten, funktioniert hier nicht. Falls n tatsächlich ein Infix von π ist, werden wir dies mithilfe eines Approximations-Algorithmus feststellen können. Sollte dies allerdings nicht der Fall sein, haben wir keine Abbruchbedingung die uns sagt, dass wir π lange genug approximiert haben und n definitiv kein Infix von π ist.

Falls allerdings die unendlich vielen Ziffern von π zufällig verteilt sind, was bisher weder widerlegt noch bewiesen werden konnte, enthält π jedes Wort aus $\{0, \ldots, 9\}^*$ als Infix. In dem Fall ist die Funktion g_π berechenbar. Dafür definieren wir die Funktion g_π^1, welche für jede Eingabe den Wert 1 ausgibt:

$$g_\pi^1(n) = 1 \text{ für alle } n \in \mathbb{N}.$$

3.4 Beispiel
Sei $f_{\text{PNP}}\colon \{0,1\}^* \to \{0,1\}^*$ die (totale) Funktion, die wie folgt definiert ist:

$$f(w) = \begin{cases} 0 & \text{, falls P = NP,} \\ 1 & \text{, falls P} \neq \text{NP.} \end{cases}$$

Man könnte vermuten, dass diese Funktion nicht berechenbar ist, da unbekannt ist, ob P = NP gilt. Tatsächlich ist diese Funktion jedoch berechenbar – wir haben bloß verlangt, dass es einen Algorithmus *gibt*, nicht dass wir ihn auch *kennen*.

Es ist leicht, zwei Algorithmen anzugeben, die unabhängig von der Eingabe konstant 0 bzw. 1 ausgeben. Einer dieser beiden Algorithmen berechnet f, wir wissen bloß nicht, welcher davon.

Man nennt eine Funktion **effektiv berechenbar**, wenn man den Algorithmus, der die Funktion berechnet, konkret angeben kann. Analog nennt man ein Entscheidungsproblem **effektiv entscheidbar**, wenn man den Entscheidungsalgorithmus für das Problem kennt.

3.5 Beispiel
Sei nun die Funktion $h_\pi\colon \mathbb{N} \to \mathbb{N}$, mit

$$f(n) = \begin{cases} 1 & \text{, falls } \underbrace{5 \ldots 5}_{n\text{–mal}} \text{ ein Infix der Dezimaldarstellung von } \pi \text{ ist,} \\ 0 & \text{, sonst.} \end{cases}$$

Man könnte vermuten, dass, wie im Beispiel 3.3, nicht bekannt ist ob diese Funktion berechenbar ist. Aber aufgrund einer ähnlichen Argumentation wie im vorherigen Beispiel, kann man zeigen, dass h_π tatsächlich berechenbar ist.

Es gibt zwei mögliche Fälle.

1. Falls in der Dezimaldarstellung von π jedes Wort aus 5^* vorkommt, dann definieren wir die Funktion h^1_π, welche für jede Eingabe 1 zurück gibt.

2. Wenn dies nicht der Fall ist, dann gibt es eine Zahl $n_0 \in \mathbb{N}$, so dass alle 5er-Sequenzen in π höchstens Länge n_0 haben. In diesem Fall, definieren wir folgende Funktion

$$h^2_\pi(n) = \begin{cases} 1, \text{ falls } n \leqslant n_0 \\ 0, \text{ sonst} \end{cases}$$

   Einer dieser beiden Fälle wird zutreffen, also existiert ein Algorithmus der h_π berechnet. Wir wissen nur nicht welcher von beiden der Richtige ist.

Für das nächste Beispiel (und um zu zeigen, dass es nicht berechenbare Funktionen gibt), benötigen wir den Begriff der **Abzählbarkeit**.

3.6 Definition
Eine Menge M ist **abzählbar**, wenn sie entweder leer ist oder es eine surjektive Funktion $f\colon \mathbb{N} \to M$ gibt. Mit anderen Worten muss es eine Aufzählung $f(0), f(1), f(2), \ldots$ (für welche nicht unbedingt einen Algorithmus existieren muss) geben, welche die Elemente von M aufzählt. Dabei dürfen Elemente sich in der Aufzählung wiederholen, die Funktion f muss nicht injektiv sein.

3.7 Beispiel
Nun kann man sich die Frage stellen, ob eine analog zu f_π definierte Funktion f_a für jede reelle Zahl $a \in \mathbb{R}$ berechenbar ist. Dies ist nicht der Fall: Es gibt überabzählbar viele reelle Zahlen, aber, wie wir gleich zeigen werden, nur abzählbar viele berechenbare Algorithmen, und dementsprechend nicht für jede reelle Zahl einen Approximationsalgorithmus.

Wir wollen nun den Begriff der Berechenbarkeit mit Hilfe von Turing-Maschinen formalisieren. Dazu modifizieren wir die Definition der Turing-Maschinen, so dass sie nun Funktionen berechnen anstatt Sprachen zu akzeptieren.

3.8 Definition
Sei $f\colon \Sigma_1^* \to_p \Sigma_2^*$ eine partielle Funktion. Wir nennen f **(Turing-)berechenbar**, wenn es eine deterministische Turing-Maschine $M = (Q, \Sigma_1, \Gamma, q_0, \delta, Q_F)$ gibt, so dass für jeder Eingabe $w \in \Sigma_1^*$ gilt dass

$$f(w) = w' \in \Sigma_2^* \quad \text{gdw} \quad q_0 w \to^* \sqcup \ldots \sqcup q_f\, w' \sqcup \ldots \sqcup,$$

wobei $q_f \in Q_F$. Hierbei nehmen wir an, dass $\Sigma_2 \subseteq \Gamma$ im Bandalphabet ist, und $\sqcup$ nicht in Σ_2 vorkommt.

Für partielle Funktionen $f\colon \mathbb{N}^k \to_p \mathbb{N}$, sagen wir dass f Turing-berechenbar ist, falls für jede Eingabe $n_1, \ldots, n_k \in \mathbb{N}$ gilt dass

$$f(n_1, \ldots, n_k) = n \in \mathbb{N} \quad \text{gdw} \quad q_0\, \text{bin}(n_1)\#\ldots\#\text{bin}(n_k) \to^* \sqcup \ldots \sqcup q_f\, \text{bin}(n) \sqcup \ldots \sqcup.$$

wobei $q_f \in Q_F$ und bin(n) die Binärdarstellung (ohne führende Nullen) von n ist.

3.9 Bemerkung

- Wir haben in einer früheren Vorlesung bereits bewiesen, dass man nicht-deterministische Turing-Maschinen determinisieren kann. Daher ist die Verwendung von DTMs in der vorherigen Definition keine Einschränkung (man könnte auch NTMs verwenden). Allerdings ist, im Hinblick auf Funktionen, die Verwendung von DTMs natürlicher.
- Wir können annehmen, dass die Turingmaschine weder ihren Zustand noch ihre Kopfposition ändert nachdem ein akzeptierender Zustand erreicht worden ist. Dies ist der Fall, da für die Akzeptanz einer Turingmaschine die Erreichbarkeit einer akzeptierende Konfiguration ausreicht. Wir sagen dann, dass die Turingmaschine **hält** oder **stehen bleibt**. Im Gegensatz dazu sagen wir, dass die Turingmaschine **stecken bleibt**, wenn sie in einem nicht-akzeptierenden Zustand keine passende Transition zur Verfügung hat.
- Falls die Funktion $f\colon \Sigma_1^* \to_p \Sigma_2^*$ undefiniert auf einer Eingabe w ist, dann darf die zugehörige TM auf der Eingabe w keine Konfiguration der Form, wie sie in der Definition beschrieben ist, erreichen. Die TM muss also
  - stecken bleiben (was nicht möglich ist wenn wir eine DTM vorliegen haben) oder
  - unendlich lange loopen oder
  - in einer Konfiguration stehen bleiben die nicht Beschreibung oben entspricht.

3.10 Beispiel
Die Funktionen f, f_π, f_{PNP}, h_π sind alle Turing-berechenbar.

Wir wollen nun zeigen, dass es auch Funktionen gibt, welche nicht-berechenbar sind. Für diesen Beweis benötigen wir aber zuerst folgendes Resultat.

3.11 Lemma
Es gibt abzählbar viele Turing-Maschinen.

Beweis:
Hierzu nehmen wir O.B.d.A. (ohne Beschränkung der Allgemeinheit) an, dass die Turing-Maschinen der Form $(Q, \Sigma, \Gamma, q_0, \delta, Q_F)$ für

$$Q = \{q_0, q_1, \ldots, q_k\}$$

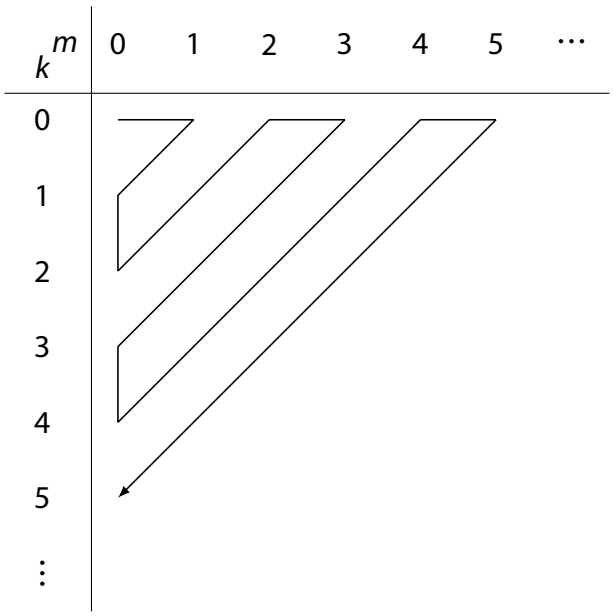
und

$$\Gamma = \Sigma \cup \{a_0, \ldots, a_m, \$, \sqcup\}$$

sind. Diese Annahme ist dadurch gerechtfertigt, dass die von der Maschine akzeptierte Sprache, für welche wir uns interessieren, nicht von den Namen der Kontrollzustände und der Bandsymbole abhängt. Wir können jede beliebige Maschine durch Umbenennen der Symbole in die gewünschte Form überführen, ohne ihre Sprache zu verändern.

Man beachte, dass es für fixierte Zahlen k, m nur endliche viele Maschinen gibt, denn es gibt nur höchstens $(k+2)^2 \cdot (|\Sigma| + 2 + m)^2 \cdot 2$ Möglichkeiten für die Transitionsfunktion δ.

Wir zählen nun alle Turing-Maschinen (der entsprechenden Form) auf, in dem wir das **Cantor'sche Diagonalverfahren** verwenden. Dieses sollte aus dem Beweis, dass $\mathbb{Q}$ abzählbar ist, bekannt sein.



Wir laufen – wie in der Grafik angedeutet – schlangenlinienförmig durch die Tabelle, die einen Eintrag pro Kombination aus m und k hat (und dementsprechend nach rechts und nach unten unendlich ist). Dabei treffen wir jede Zelle $(k, m) \in \mathbb{N}^2$ genau ein Mal.

Wir erhalten unsere gewünschte Abzählung aller Turing-Maschinen, indem wir jedes Mal, wenn wir eine Zelle (k, m) treffen, die endlich vielen Turing-Maschinen für dieses k und dieses m aufzählen.

Wir haben nun bewiesen, dass die Menge der Turing-Maschinen abzählbar ist, es gibt nun also eine Abzählung $M_0, M_1, M_2, \ldots$, in der jede Turing-Maschine vorkommt. □

3.12 Theorem
Es seien Σ_1, Σ_2 beliebige Alphabete. Es gibt nicht-berechenbare Funktionen $f\colon \Sigma_1^* \to_p \Sigma_2^*$.

Beweis:

Der Einfachheit halber beschränken wir uns auf Funktionen $f\colon \mathbb{N} \to_p \mathbb{N}$ (die wie oben erklärt codiert werden können). Der Beweis lässt sich auch für beliebige Alphabete führen.

Wir verwenden in diesem Beweis, dass es unabzählbar viele Funktionen, aber nur abzählbar viele Turing-Maschinen gibt. Um einen Widerspruch zu erhalten nehmen wir an, dass jede Funktion $f\colon \mathbb{N} \to_p \mathbb{N}$ berechenbar ist.

Sei $M_0, M_1, M_2, \ldots$ eine Abzählung aller Turing-Maschinen, die die Anforderungen aus der Definition von „berechenbar" erfüllen, und seien $f_0, f_1, f_2, \ldots$ eine Abzählung der von ihnen berechneten Funktionen. (Beachte: $f_i = f_j$ für $i \neq j$ ist möglich und erlaubt.)

Wir konstruieren nun eine Funktion $f\colon \mathbb{N} \to \mathbb{N}$, und beweisen, dass diese nicht berechenbar ist. Wir definieren

$$f(n) = \begin{cases} 0 & \text{, falls } f_n(n) \text{ undefiniert ist,} \\ f_n(n) + 1 & \text{, sonst.} \end{cases}$$

Angenommen f wäre berechenbar. Da $f_0, f_1, f_2, \ldots$ eine Abzählung aller berechenbaren Funktionen war, gibt es dann einen Index $m \in \mathbb{N}$ mit $f = f_m$. Nun stellen wir allerdings fest, dass sich die Funktionswerte von f und f_m für den Wert m unterscheiden. Falls $f_m(m)$ undefiniert ist, ist $f(m)$ definiert. Falls $f_m(m)$ definiert ist, gilt $f(m) = f_m(m) + 1 \neq f_m(m)$. □

Wir können das Beweisverfahren graphisch als eine in beide Richtungen unendlich große Tabelle darstellen. Hierbei haben wir eine Spalte pro Funktion f_i und eine Zeile pro natürliche Zahl j. In der Zelle (j, i) ist nun der Funktionswert $f_i(j)$ eingetragen.

3.13 Beispiel
Die Tabelle könnte zum Beispiel wie folgt aussehen.

	f	f_0	f_1	f_2	f_3	f_4	$\ldots$
0	0	undef.	2	4	0	undef.	$\ldots$
1	5	1	4	100	0	1	$\ldots$
2	106	0	5	105	0	4	$\ldots$
3	1	2	undef.	0	0	9	$\ldots$
4	17	3	3	115	0	16	$\ldots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Man sieht, dass sich die wie im Beweis konstruierte Funktion f auf der Diagonalen von allen Funktionen f_i unterscheidet, es gilt $f(i) \neq f_i(i)$. Daher heißt das obige Beweisverfahren **Diagonalisierung**. Es sollte vom Beweis des Theorems das besagt, dass die reellen Zahlen nicht abzählbar sind, bekannt sein. Wir werden Diagonalisierung im Laufe dieser Vorlesung noch mehrfach verwenden.

B) Entscheidbarkeit

Wir wollen nun den Begriff der Berechenbarkeit auf Sprachen, also Mengen von Worten, zuschneiden.

3.14 Definition
Eine Menge $A \subseteq \Sigma^*$ (oder $A \subseteq \mathbb{N}$) ist **entscheidbar**, wenn die totale charakteristische Funktion χ_A von A

$$\begin{array}{rcl} \chi_A : \Sigma^* & \to & \{0,1\} \\ w & \mapsto & \begin{cases} 1 & , \text{ falls } w \in A, \\ 0 & , \text{ sonst.} \end{cases} \end{array}$$

berechenbar ist.

Eine Menge $A \subseteq \Sigma^*$ ist **semi-entscheidbar**, wenn die partielle/halbe charakteristische Funktion χ'_A von A

$$\begin{array}{rcl} \chi'_A : \Sigma^* & \to_p & \{1\} \\ w & \mapsto & \begin{cases} 1 & , \text{ falls } w \in A, \\ \text{undefiniert} & , \text{ sonst.} \end{cases} \end{array}$$

berechenbar ist.

3.15 Bemerkung
In der Literatur werden Sprachen

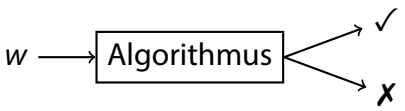
$$A = \{w \in \Sigma^* \mid w \text{ erfüllt die Bedingung für } A\}$$

oft **Entscheidungsprobleme** genannt und damit wie folgt ausgedrückt.

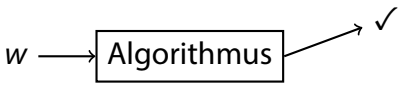
Entscheidungsproblem zu Menge A	
Gegeben:	Wort $w \in \Sigma^*$
Berechne:	Erfüllt w die definierende Bedingung von A?

Da wir bei Entscheidungsproblemen eine Eigenschaft prüfen, wird in der Literatur manchmal der Begriff Entscheidbarkeit/Semi-Entscheidbarkeit für Eigenschaften (Bedingungen, Prädikate) verwendet.

Eine Sprache ist also entscheidbar, wenn es einen Algorithmus (DTM/NTM) gibt, welcher auf jeder Eingabe, sowie bei jeder Berechnung hält und das richtige Ergebnis ausgibt.



Eine Sprache ist semi-entscheidbar, wenn es einen Algorithmus gibt, welcher auf Ja-Instanzen immer hält und das richtige Ergebnis ausgibt. Auf einer Nein-Instanz, kann der Algorithmus loopen, stecken bleiben oder in einer Konfiguration stehen bleiben, welche nicht Definition 3.8 entspricht.



Wenn wir also eine Eingabe w haben, für die wir bereits wissen, dass $w \in A$ gilt, können wir dies verifizieren, indem wir den Algorithmus M auf Eingabe w simulieren: Die Berechnung wird nach endlich vielen Schritten akzeptieren und 1 ausgeben. Wenn wir eine Eingabe $w \notin A$ haben und M simulieren, kann einer der oben beschriebenen Fälle auftreten.

Das bedeutet, dass für eine Eingabe $w \in \Sigma^*$, über die wir noch nichts wissen, lässt sich $w \in A$ nicht durch Simulation entscheiden: Falls M nach endlich vielen Schritten anhält, wissen wir ob $w \in A$ gilt, falls M nicht anhält, wissen wir jedoch nicht,

- ob die Berechnung in der Zukunft noch anhalten wird, und wir bloß noch nicht genügend viele Schritte simuliert haben, oder
- ob die Berechnung niemals halten wird.

Die Turingmaschine, welche wir in Kapitel 1 eingeführt hatten, sind also zunächst bloß Semi-Entscheider (die akzeptieren anstatt 1 auszugeben), also ein Algorithmus, mit dem man die Ja-Instanzen in endlicher Zeit verifizieren kann, allerdings für beliebige Instanzen eventuell unendlich lange simulieren müsste.

Analog zu den entscheidbaren Mengen A, definieren wir nun Turingmaschinen, die auf allen Eingaben und Berechnungen (für NTMs) halten und in einen akzeptierenden Zustand gehen, falls die Eingabe w Element aus A ist und ansonsten einen abweisenden Zustand erreichen.

3.16 Definition
Eine **haltende/totale Turingmaschine** oder ein **Entscheider** M ist ein Tupel

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

wobei Q, Σ, Γ, q_0 und δ wie in Kapitel 1 definiert sind. Die Menge der Kontrollzustände Q enthält nun

- $q_0 \in Q$ ist der **Start- oder Initialzustand**,
- $q_{acc} \in Q$ ist der **akzeptiertende Zustand** und
- $q_{rej} \in Q$ ist der **abweisende Zustand** mit $q_{acc} \neq q_{rej}$,

Die Konfigurationen und die Transitionsrelation (deterministisch/nicht-deterministisch) eines Entscheiders sind wie in Kapitel 1 definiert.

Wir nennen

- Konfigurationen der Form $uq_{acc}v \in \Gamma^* Q \Gamma^*$ **akzeptierend** und
- Konfigurationen der Form $uq_{rej}v \in \Gamma^* Q \Gamma^*$ **abweisend**.

Die Sprache eines Entscheiders M ist definiert durch

$$\mathcal{L}(M) = \left\{ w \in \Sigma^* \mid q_0 w \to^* u\, q_{acc}\, v \in \Gamma^*\, Q\, \Gamma^* \right\}.$$

Wie bereits in 3.9 angemerkt, können wir davon ausgehen, dass die Turingmaschine nach Erreichen eines akzeptierenden oder abweisenden Zustandes weder ihre Kopfposition noch ihren Zustand ändern wird:

$$\forall a \in \Gamma : \delta(q_{acc}, a) = (q_{acc}, a, N) \text{ und } \delta(q_{rej}, a) = (q_{rej}, a, N).$$

Aus diesem Grund nennen wir q_{acc}, q_{rej} auch **Haltezustände**.

Analog dazu nennen wir akzeptierende und abweisende Konfigurationen **haltende Konfigurationen**.

Der Hauptunterschied zu den in Kapitel 1 definierten Turingmaschinen ist, dass ein Entscheider auf **jeder Eingabe** und **in jeder Berechnung** nach **endlich vielen Schritten** in einem der Haltezustände hält.

3.17 Bemerkung
Auch Semi-Entscheider, also die in Kapitel 1 definierten Turingmaschinen, können von der Form

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

sein. Im Gegensatz zu den Entscheidern fordern wir aber nicht, dass jede Berechnung in einer haltenden Konfiguration endet. Eine Eingabe $w \in \Sigma^*$ wird akzeptiert, falls es eine Berechnung auf w gibt, welche in einer akzeptierenden Konfiguration hält. Allerdings wird eine Eingabe $w \in \Sigma^*$ nicht akzeptiert, falls jede Berechnung auf w entweder

1. in einer abweisenden Konfiguration hält, oder
2. unendlich lange loopt, oder
3. in einer nicht-haltenden Konfiguration stecken bleibt (falls keine passende Transition zur Verfügung steht).

3.18 Bemerkung
Theorem 1.17, Lemma 1.8 und Lemma 1.11 gelten auch für Entscheider.

- Zu jedem nicht-deterministischem Entscheider M, gibt es einen deterministischen Entscheider M', so dass $\mathcal{L}(M') = \mathcal{L}(M)$ gilt.
- Zu jedem Mehr-Band-Entscheider M_k gibt es einen Ein-Band-Entscheider M, so dass $\mathcal{L}(M') = \mathcal{L}(M)$.
- Zu jedem Entscheider $M_{\leftrightarrow}$ mit beidseitig unendlichem Band gibt es einen Entscheider M mit rechts unendlichem Band, so dass $\mathcal{L}(M_{\leftrightarrow}) = \mathcal{L}(M)$.

3.19 Beispiel
Jede kontextsensitive Sprache $\mathcal{L}(G)$ ist entscheidbar.

Beweis:
In Kapitel 1 haben wir einen Algorithmus angegeben, der das Wortproblem für kontextsensitive Sprache löst. Für ein gegebenes Wort w, zählen wir alle vom Startsymbol ableitbare Satzformen der Länge $\leqslant |w|$ auf. Befindet sich w darunter, akzeptiert der Algorithmus, ansonsten weist er ab. In jedem Fall terminiert der Algorithmus, da es eine obere Schranke für die aufzuzählenden Satzformen gibt. □

3.20 Theorem
Eine Sprache $A \in \Sigma^*$ ist entscheidbar gdw. A und $\bar{A}$ semi-entscheidbar sind.

Beweis:

"⇒" Klar.

"⇐" Es sei M_A eine TM (Semi-Entscheider) für A und $M_{\bar{A}}$ eine TM für $\bar{A}$. Wir konstruieren einen Algorithmus, welcher A entscheidet und verwenden Church's These um zu argumentieren, dass man den Algorithmus in einen Entscheider für A überführen könnte.

Eingabe: $w \in \Sigma^*$
for $i = 1, 2, 3, \ldots$ **do**
  if M_A akzeptiert Eingabe w in höchstens i Schritten **then**
    return 1
  end if
  if $M_{\bar{A}}$ akzeptiert Eingabe w in höchstens i Schritten **then**
    return 0
  end if
end for

□

Eine andere Herangehensweise an die Entscheidungsprobleme für Sprachen, ist es die Elemente der Sprache aufzuzählen anstatt zu entscheiden ob ein gegebenes Wort Element der Sprache ist.

3.21 Definition
Eine Sprache $A \subseteq \Sigma^*$ ist **rekursiv aufzählbar**, wenn $A = \varnothing$ gilt, oder es eine totale, berechenbare Funktion $f\colon \mathbb{N} \to \Sigma^*$ gibt mit

$$A = \{f(0), f(1), f(2), \ldots\} = \{f(i) \mid i \in \mathbb{N}\} .$$

Beachte, dass $f(i) = f(j)$ für $i \neq j$ erlaubt ist.

Wir sagen, dass A von f aufgezählt wird.

Eine Sprache $A \subseteq \Sigma^*$ ist **rekursiv**, falls sowohl A als auch $\bar{A}$ rekursiv aufzählbar sind.

In der Literatur werden die Begriffe rekursiv und rekursiv aufzählbar für Sprachen und die Begriff entscheidbar und semi-entscheidbar für Eigenschaften verwendet. Diese Unterscheidung ist künstlich und nicht notwendig.

3.22 Theorem
Eine Sprache $A \subseteq \Sigma^*$ ist rekursiv aufzählbar gdw. sie semi-entscheidbar ist.

Beweis:

„⇒“ Die leere Menge ist durch eine Turing-Maschine, die im abweisenden Zustand q_{rej} startet, sogar entscheidbar. Nehmen wir nun an, dass A eine nicht-leere Sprache ist, und dass es eine Funktion $f\colon \mathbb{N} \to \Sigma^*$ wie gefordert gibt. Wir geben Pseudocode für einen Semi-Entscheider für A an, der sich in eine Turing-Maschine überführen lässt.

Eingabe: $w \in \Sigma^*$
for $i = 0, 1, 2, \ldots$ **do**
  Berechne $v = f(i)$.
  Akzeptierte, falls $v = w$.
end for

Aufgrund der unendlichen for-Schleife läuft der Algorithmus eventuell unendlich lange. Beachte, dass das Berechnen von $f(i)$ für jedes i nur endlich viele Schritte dauert, da wir angenommen haben, dass f total und berechenbar ist.

Sei $w \in \Sigma^*$ ein Wort. Falls $w \in A$, dann gibt es einen Index n mit $w = f(n)$, und der Semi-Entscheider akzeptiert, sobald er diesen Index erreicht. Falls $w \notin A$, dann gibt es keinen solchen Index. Der Semi-Entscheider hält in diesem Fall nicht an.

„⇐“ Nehmen wir nun an, dass A rekursiv aufzählbar ist. Falls $A = \varnothing$ ist nichts zu zeigen, wir nehmen also an, dass A nicht-leer ist. Sei M eine Turing-Maschine mit $A = \mathcal{L}(M)$. Wir müssen einen Aufzählungsalgorithmus f konstruieren, der Zahlen aus $\mathbb{N}$ entgegennimmt und Wörter aus A ausgibt. Der Algorithmus soll für jede Eingabe nach endlich vielen Schritten halten, und zu jedem Wort w aus A soll es einen Index m geben mit $f(m) = w$.

Die Idee hierzu ist, dass wir die Wörter $w \in \Sigma^*$ mit den Berechnungsschritten von M „verzahnen“.

Hierzu nummerieren wir die Wörter durch, d.h. es sei $w_0, w_1, w_2, \ldots$ eine Aufzählung aller Wörter in Σ^*. Diese lässt sich erhalten, indem wir für jedes $k = 1, 2, \ldots$ die endlich vielen Wörter der Länge k aufzählen (z.B. innerhalb der gleichen Länge lexikographisch geordnet). Man kann eine totale, berechenbare Funktion $g\colon \mathbb{N} \to \Sigma^*$ mit $g(i) = w_i$ angeben. Dies bedeutet, dass es zu jedem Wort $w \in \Sigma^*$ eine Zahl i gibt mit $g(i) = w$.

Nun simulieren wir die Berechnung von M auf den Wörtern $w_0, w_1, w_2, \ldots$. Dies tun wir nicht Wort für Wort (M ist nur ein Semi-Entscheider, eventuell hält M nicht an!), sondern simultan für alle Wörter.

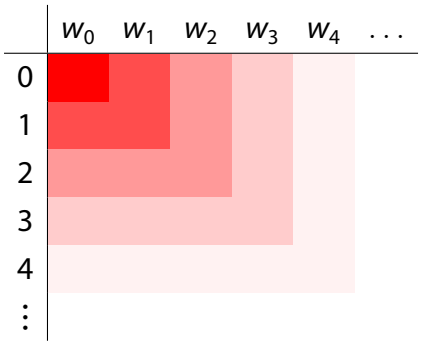
Der folgende Algorithmus $\mathcal{A}$ implementiert dieses Verfahren.

for $i = 0, 1, 2, \ldots$ **do**
  for $j = 0, 1, 2, \ldots, i$ **do**
    Berechne j-tes Wort w
    if M Eingabe w in höchstens i Schritten akzeptiert **then**
      Gebe w **aus**
    end if
  end for
end for

Beachte, dass nur der Algorithmus zwar aufgrund der äußeren for-Schleife unendlich lange läuft, allerdings für jedes $i \in \mathbb{N}$ der Schleifenrumpf in endlicher Zeit abgearbeitet werden kann: Die innere Schleife iteriert nur über endlich viele j, wir haben oben erklärt, dass w_j berechnet werden kann, und die Simulation von M für i Schritte dauert auch nur endlich lange.

Die Arbeitsweise des Algorithmus ist wie folgt: Für $i = 0$ simuliert der Algorithmus M für 0 Schritte auf w_0, für $i = 1$ simuliert der Algorithmus M für jeweils 1 Schritt auf w_0 und w_1, für $i = 2$ simuliert der Algorithmus M für jeweils 2 Schritte auf w_0, w_1, w_2, und so weiter.

Die folgende Graphik repräsentiert dieses Verhalten. Die Spalten entsprechen Wörtern, die Zeilen den Schritten. Der Algorithmus arbeitet in Richtung der blasser werdenden Zellen.



Beachte, dass $\mathcal{A}$ genau die Wörter aus A ausgibt, denn zu jedem solchen Wort $w = w_{j'}$ gibt es auch eine Schrittzahl i', so dass M $w_{j'}$ in i' Schritten akzeptiert. Im Durchlauf für $i = \max\{i', j'\}$ und $j = j'$ wird der Algorithmus $\mathcal{A}$ das Wort w ausgeben. Wörter, die nicht in A liegen werden von M nicht in endlich vielen Schritten akzeptiert.

Um nun den gesuchten Aufzählungsalgorithmus f zu erhalten, zählen wir im obigen Algorithmus die Ausgaben mit. $f(n)$ berechnet nun die n-te Ausgabe von $\mathcal{A}$, indem er die vorherigen verwirft, und nach der n-ten Ausgabe anhält.

```
Eingabe: n ∈ ℕ
  m ← 0
  for i = 0, 1, 2, . . . do
  |  for j = 0, 1, 2, . . . , i do
  |  |  Berechne j-tes Wort w_j
  |  |  if M Eingabe w in höchstens i Schritten akzeptiert then
  |  |  |  m ← m + 1
  |  |  |  if m = n then
  |  |  |  |  Gebe w aus
  |  |  |  |  Akzeptiere
  |  |  |  end if
  |  |  end if
  |  end for
  end for
```

□

3.23 Korollar
Eine Sprache $A \in \Sigma^*$ ist rekursiv gdw. sie entscheidbar ist.

3.24 Korollar
Sei $A \subseteq \Sigma^*$ eine Sprache. Die folgenden Aussagen sind äquivalent:

1. A ist semi-entscheidbar.
2. A ist rekursiv aufzählbar.
3. Es gibt eine TM M mit $\mathcal{L}(M) = A$.
4. Es gibt einen Aufzählungsalgorithmus für A, d.h. A ist der Wertebereich einer totalen berechenbaren Funktion $f\colon \mathbb{N} \to A$.
5. χ'_A ist berechenbar.
6. A ist der Definitionsbereich einer partiellen berechenbaren Funktion $g\colon \Sigma^* \to_p \Sigma_2^*$.

3.25 Bemerkung: Aufzählbarkeit vs. Abzählbarkeit
In dieser Bemerkung wollen wir den Unterschied zwischen Abzählbarkeit und (rekursiver) Aufzählbarkeit klarstellen.

Eine Menge A heißt **abzählbar**, wenn A leer ist oder wenn es eine surjektive Funktion $f\colon \mathbb{N} \to A$ gibt. Erinnerung: Surjektiv bedeutet, dass es zu jedem $m \in A$ einen Index $i \in \mathbb{N}$ gibt mit $m = f(i)$. Injektivität fordern wir dabei nicht, d.h. $f(i) = f(j)$ für $i \neq j$ ist erlaubt. (Dies tun wir, damit auch endliche Mengen gemäß unserer Definition abzählbar sind.)

Eine solche Funktion f heißt auch **Abzählung** von A, wir sehen $f(0)$ als das erste, $f(1)$ als das zweite, usw. Element von A.

Wir haben bereits gesehen oder angemerkt, dass die Menge der reellen Zahlen, die Menge der Sprachen $A \subseteq \Sigma^*$ und die Menge der (partiellen) Funktionen $f\colon \Sigma_1^* \to \Sigma_2^*$ (jeweils für beliebige fixierte Alphabete) nicht abzählbar sind.

Intuitiv ist eine Menge abzählbar, wenn sie höchstens so groß wie die natürlichen Zahlen ist. Jede endliche Menge sowie $\mathbb{N}$, $\mathbb{Z}$ und $\mathbb{Q}$ sind abzählbar.

Jede Sprache $A \subseteq \Sigma^*$ ist abzählbar:

- Die Menge aller Wörter in Σ^* ist abzählbar, denn man kann die Wörter der Lange nach sortiert, und innerhalb der selben Länge lexikographisch sortiert, abzählen.

  Beispielsweise für $\Sigma = \{0, 1\}$:

$n \in \mathbb{N}$	0	1	2	3	4	5	6	7	8	9	10	11	12	$\cdots$
$f(n) \in \{0,1\}^*$	ε	0	1	00	01	10	11	000	001	010	011	100	101	$\cdots$

- Zu jeder abzählbaren Menge A ist auch jede Teilmenge $A' \subseteq A$ abzählbar – man überspringt beim Abzählen die Elemente, die nicht in A' liegen.

  Sei A' nicht-leer (leere Mengen sind nach Definition abzählbar), und $a \in A'$ ein beliebiges Element. Sei f eine Abzählung von A. Wir definieren eine Abzählung g von A' wie folgt:

$$g(n) = \begin{cases} f(n) & \text{, falls } f(n) \in A', \\ a & \text{, sonst.} \end{cases}$$

Durch Kombination der beiden Aussagen erhalten wir, dass A abzählbar ist, in dem wir alle Wörter in Σ^* abzählen und dabei die Wörter, die nicht in A liegen, überspringen. Beachte, dass dies kein Widerspruch dazu ist, dass die *Menge aller Sprachen* nicht abzählbar ist. (Analog für die natürlichen Zahlen: Jede Menge von natürlichen Zahlen ist abzählbar, die Menge aller solcher Mengen ist nicht abzählbar.)

Gemäß dem zuvor bewiesenen Satz ist eine Sprache **(rekursiv) aufzählbar**, wenn es einen Aufzählungsalgorithmus für sie gibt. Dies ist eine echt stärkere Eigenschaft: Wir fordern nun nicht nur, dass es eine beliebige Abzählung gibt, sondern auch, dass wir die Abzählung als Algorithmus implementieren können.

Nicht jede Teilmenge einer rekursiv aufzählbaren Menge ist wieder rekursiv aufzählbar, denn die Sprache Σ^* ist rekursiv aufzählbar (wie im Beweis oben erklärt), allerdings gibt es Sprachen $A \subseteq \Sigma^*$, die nicht semi-entscheidbar, und damit nicht rekursiv aufzählbar sind.

4. Unentscheidbarkeit, universelle Turing-Maschine, Halteproblem & Reduktionen

In Kapitel 3 haben wir nicht-konstruktiv bewiesen, dass es nicht semi-entscheidbare Probleme geben muss. Nun wollen wir endlich konkrete Beispiele für nicht-(semi-)entscheidbare Probleme kennen lernen. Wie wir in diesem Kapitel feststellen werden, handelt es sich hierbei um Verifikationsprobleme, also Probleme, bei denen entschieden werden soll, ob ein gegebenes Programm eine bestimmte Eigenschaft hat. Dies bedeutet, dass die Eingabe des Problems selbst eine Turing-Maschine ist.

In diesem Kapitel werden wir viele verschiedene Konzepte kennen lernen.

A) Da wir uns für Probleme interessieren, bei denen die Eingabe eine Turing-Maschine ist, müssen wir zunächst zeigen, wie man solche Probleme als Wortprobleme auffassen kann. Hierzu müssen wir Turing-Maschinen als Wort **kodieren**.

B) Wenn wir nun ein solches Problem (semi-)entscheiden wollen, müssen wir eine Maschine betrachten, die die Kodierung einer anderen Maschine als Eingabe erhält. Nun liegt es nahe, die kodierte Maschine zu simulieren. Wir zeigen mittels einer **universellen Turing-Maschine**, dass dies möglich ist.

C) Nun können wir beweisen, dass zwei grundlegende Probleme, nämlich das **allgemeine und das spezielle Akzeptanzproblem**, unentscheidbar sind. Diese Probleme fragen danach, ob eine als Eingabe vorliegende Turing-Maschinen auf einer bestimmten Eingabe nach endlich vielen Schritten hält, oder ob ihre Berechnung unendlich lange läuft, ohne zu halten. Für den Beweis benötigen wir sowohl die universelle TM als auch das bereits bekannte Prinzip der Diagonalisierung.

D) Wenn wir nun andere Probleme als unentscheidbar nachweisen wollen, möchten wir nicht von vorne beginnen. Daher werden wir **Reduktionen** einführen, die es uns erlauben, aus der Unentscheidbarkeit der Halteprobleme die Unentscheidbarkeit vieler anderer Probleme zu schlussfolgern.

A) Kodierungen von Turing-Maschinen

Wir wollen Turing-Maschinen als Binärstrings, d.h. als Wörter über dem Alphabet $\{0, 1\}$, auffassen.

4.1 Definition: Kodierung von Turing-Maschinen
Sei $M = (Q, \Sigma, \Gamma, q_0, \delta, \{q_{acc}, q_{rej}\})$ eine TM (für die TM aus Kapitel 1 funktioniert die Konstruktion analog).

Wir gehen o.B.d.A. davon aus, dass die Zustände und Symbole durchnummeriert sind:

$$Q = \{q_0, \ldots, q_n\},$$
$$\Gamma = \{a_0, \ldots, a_m\}.$$

Hierbei gehen wir davon aus, dass q_0 der Startzustand, $q_1 = q_{acc}$, $q_2 = q_{rej}$ und $a_0 = \sqcup$ ist. Wir gehen davon aus, dass dann zunächst das Eingabealphabet Sigma kodiert ist, d.h. $\Sigma = \{a_2, \ldots a_k\}$ für eine Zahl $k > 2$.

Mit diesen Annahmen verbleibt es, die Transitionsfunktion δ zu kodieren. Es sei

$$\delta(q_i, a_j) = (q_{i'}, a_{j'}, d)$$

eine Abbildungsvorschrift gemäß δ. Wir weisen ihr das folgende Wort zu:

$$w_{i,j,i',j',d} = \#\#\text{bin}(i)\#\text{bin}(j)\#\text{bin}(i')\#\text{bin}(j')\#\text{bin}(y) \in \{0, 1, \#\}^* .$$

Hierbei ist bin(−) eine Funktion, die jeder natürlichen Zahl ihre Binärdarstellung bin(n) zuordnet, und $y = 0$ falls $d = L$, $y = 1$ falls $d = R$ und $y = 2$, falls $d = N$.

Um nun M zu codieren, schreiben wir alle Abbildungsvorschriften von δ hintereinander.

$$w = \prod_{i=0}^{n} \prod_{j=0}^{m} w_{i,j,i',j',d} \quad \text{für } \delta(q_i, a_j) = (q_{i'}, a_{j'}, d) .$$

Das Produkt (Π) ist hierbei als Konkatenation der Wörter zu lesen.

Nun haben eine Kodierung von M als Wort $w \in \{0, 1, \#\}^*$. Um das zusätzliche Symbol $\#$ loszuwerden, wenden wir den folgenden Homomorphismus an:

$$0 \mapsto 00, \qquad 1 \mapsto 01, \qquad \# \mapsto 11.$$

Zu jeder Turing-Maschine M sei $\langle M \rangle \in \{0, 1\}^*$ das Wort, das sich ergibt, indem wir M wie oben beschrieben als Wort $w \in \{0, 1, \#\}^*$ kodieren und dann den obigen Homomorphismus anwenden.

4.2 Bemerkung

Wir haben oben stillschweigend die Annahme getroffen, dass der Startzustand q_0 weder der akzeptierende noch der abweisende Zustand ist. Diese Annahme lässt sich leicht dadurch herstellen, dass wir eine Maschine, die direkt abweist oder akzeptiert, so umbauen, dass sie nach einem Schritt abweist oder akzeptiert.

4.3 Bemerkung
Die Kodierung $\langle M \rangle$ einer Turing-Maschine M wird auch als ihre **Gödelnummer** oder **Gödelisierung** bezeichnet. Dieser Begriff ist nach dem Mathematiker **Kurt Gödel** (1906-1972) benannt, welcher eine Kodierung von Wörtern einführte, um den Unvollständigkeitssatz, ein wichtiges Resultat aus der Prädikatenlogik, zu beweisen.

4.4 Bemerkung
Falls die Zustände oder Symbole einer Turing-Maschine nicht von der gewünschten, durchnummerierten Form sind, können wir dies leicht durch Umbenennung herstellen. Es kann dann passieren, dass zwei unterschiedliche Turing-Maschinen M, M' die gleiche Kodierung $\langle M \rangle = \langle M' \rangle$ haben.

Wenn M und M' das gleiche Eingabealphabet Σ haben, gilt jedoch, dass $\langle M \rangle = \langle M' \rangle$ die Gleichheit der Sprachen $\mathcal{L}(M) = \mathcal{L}(M')$ impliziert, in so fern die Nummerierung der Symbole aus Σ konsistent ist.

Nun können wir jede Turing-Maschine als Wort über $\{0, 1\}$ kodieren, allerdings ist nicht jedes Wort über $\{0, 1\}$ auch die Kodierung einer Turing-Maschine. Wir möchten jedes Wort $w \in \{0, 1\}^*$ als Kodierung einer Turing-Maschine sehen, dies wird später lästige Fallunterscheidungen vermeiden.

Hierzu definieren wir $M_\varnothing = \left(\{q_0, q_{acc}, q_{rej}\}, \{0, 1\}, \{0, 1, \$, \sqcup\}, \delta, q_0\right)$ als die Turing-Maschine, die jede Eingabe nach einem Schritt abweist, also deren Transitionsfunktion wie folgt definiert.

$$\begin{aligned} \delta(q_0, \$) &= (q_{rej}, \$, R)\,, \\ \delta(q, a) &= (q, a, R) \quad \forall q \in \{q_0, q_{acc}, q_{rej}\}, \forall a \in \{0, 1, \$, \sqcup\}, (q, a) \neq (q_0, \$)\,. \end{aligned}$$

Es gilt $\mathcal{L}(M_\varnothing) = \varnothing$.

4.5 Definition
Zu jedem Wort $w \in \{0, 1\}^*$ definieren wir M_w als die Turing-Maschine

$$M_w = \begin{cases} M \text{ mit } \langle M \rangle = w, & \text{falls } w \text{ eine valide Kodierung einer Turing-Maschine } M \text{ ist,} \\ M_\varnothing, & \text{sonst.} \end{cases}$$

4.6 Bemerkung
Wie bereits oben angemerkt, gibt es zu einem Wort $w \in \{0, 1\}^*$ mehrere TMs M mit $\langle M \rangle = w$. Wenn man das Eingabealphabet fixiert, haben jedoch all diese Maschinen die gleiche Sprache und und das gleiche Verhalten. Es spielt daher keine Rolle, welche diese Maschinen wir als M_w wählen.

4.7 Bemerkung
Analog zur Kodierung einer TM lässt sich auch eine Eingabe $x \in \Sigma^*$ als Wort über $\{0,1\}$ auffassen. Unter der Annahme, dass die Symbole des Alphabets durchnummeriert, $\Sigma = \{a_2, \ldots a_k\}$, sind lässt sich die Eingabe schreiben als

$$x = a_{i_1} a_{i_2} \ldots a_{i_\ell}$$

mit $i_j \in \{2, \ldots, k\}$ für alle j. Wir können x nun kodieren als

$$x' = \text{bin}(i_1)\#\text{bin}(i_2)\# \ldots \#\text{bin}(i_\ell) \in \{0,1,\#\}^*$$

und erhalten nach Anwenden des Homomorphismus eine Kodierung $\langle x \rangle \in \{0,1\}^*$.

Wir daher werden im Folgenden der Kodierung der Eingabe keine weitere Beachtung schenken.

4.8 Bemerkung
Wir sind in diesem Kapitel davon ausgegangen, dass die Turing-Maschinen wie in unserer initialen Definition deterministisch sind und nur ein Band haben. Andere Sorten von Turing-Maschinen (Nichtdeterministische TMs, Mehrband-TMs) lassen sich kodieren, in dem man sie zunächst in eine DTM mit einem Band überführt und dann kodiert. Sie lassen sich jedoch auch direkt kodieren, was insbesondere bei NTMs wichtig ist, da bei diesen die von uns betrachtete Umwandlung in DTMs nicht effizient ist.

B) Die universelle Turing-Maschine

Im Folgenden werden wir Probleme betrachten, die Turing-Maschinen als Eingaben erwarten. Beispielsweise möchten wir (semi-)entscheiden, ob eine gegebene Eingabe von einer gegebenen Turing-Maschine akzeptiert wird.

Akzeptanzproblem (ACCEPT)	
Gegeben:	Eine Turing-Maschine M und eine Eingabe x
Entscheide:	Akzeptiert M Eingabe x, d.h. gilt $x \in \mathcal{L}(M)$?

Wir können dieses Problem nun als das folgende Wortproblem auffassen:

$$\mathcal{L}_{\text{Accept}} = \left\{ w\#x \in \{0,1,\#\}^* \;\middle|\; w, x \in \{0,1\}^*, x \in \mathcal{L}(M_w) \right\}.$$

Wir identifizieren das Entscheidungsproblem ACCEPT mit der Sprache $\mathcal{L}_{\text{Accept}}$.

Um dieses Problem zu lösen, liegt es nahe, die durch ihre Kodierung gegebene Maschine M_w zu **simulieren**. Dies möchten wir ebenfalls mit einer Turing-Maschine tun. Die Schwierigkeit hierbei ist, dass Zustände und Transitionen des Simulators unabhängig von der Eingabe fixiert werden müssen. Der Simulator muss **universell** in der Lage sein, eine gegeben Turing-Maschine mit eventuell wesentlich mehr Zuständen zu simulieren.

4.9 Theorem: Turing 1936
Man kann eine **universelle Turing-Maschine (UTM)** U konstruieren mit

$$\mathcal{L}(U) = \{w\#x \mid x \in \mathcal{L}(M_w)\} = \mathcal{L}_{\text{ACCEPT}} .$$

4.10 Bemerkung: Einfluss der universellen Turing-Maschine auf die Entwicklung des Computers
Die universelle Turing-Maschine kann als **Interpreter** gesehen werden, der ein gegebenes Programm nimmt und es ausführt. Dies zeigt, dass Turing-Maschinen nicht auf eine bestimmte Funktionalität beschränkt sind. Heutzutage nennt man Programmiersprachen bzw. Computer, die diese Eigenschaft haben **Turing-vollständig**.

Turings Idee einer universellen Turing-Maschine war von von fundamentaler Bedeutung für die Erfindung von dem, was wir heute als **Computer** verstehen.

Die Wissenschaftler, auf deren Arbeit der berühmte Artikel „First Draft of a Report on the EDVAC" (geschrieben 1945 von **John von Neumann** [Neu93]) basiert, hatten zuvor Turings Artikel gelesen und wollten seine Idee praktisch umsetzen. Das Resultat war der EDVAC, der erste Computer, in dem das auszuführende Programm genau wie die Eingabedaten binär codiert elektronisch gespeichert wurde („stored-program computer"). Die beim EDVAC verwendete **von-Neumann-Architektur** ist bis heute die Grundlage für den Aufbau von Computern.

Vorherige Computer waren entweder nicht Turing-vollständig, konnten also nur für bestimmte Berechnungen verwendet werden, oder sie waren es, das Programm wurde allerdings nicht elektronisch gespeichert, sondern durch Veränderung der Hardware eingegeben (z.B. durch entsprechende Veränderung der Verkabelung). In letztere Kategorie fällt der berühmte Computer ENIAC.

Der deutsche Computerpionier Konrad Zuse hatte bereits 1936 die Idee, sowohl Daten als auch Programm elektronisch zu speichern (was zu Zeiten des 2. Weltkriegs den Wissenschaftlern um John von Neumann nicht bekannt war), der vom ihm gebaute Computer Z3 konnte allerdings aufgrund des Fehlens von bedingten Anweisungen (*if-then-else*) nicht auf einfache Art und Weise als UTM benutzt werden.

Beweisskizze für Theorem 4.9:
U verwendet zusätzlich zum Eingabeband drei weitere Bänder, nämlich Programm-, Daten-

und Zustandsband. Wir haben in Kapitel 1 kurz angesprochen, dass sich eine äquivalente 1-Band-TM konstruieren ließe.

Für eine gegebene Eingabe $w\#x$ überprüft U zunächst, ob w die valide Kodierung einer Turing-Maschine ist.

Hierzu muss überprüft werden, ob w das Bild eines Wortes $w' \in \{0, 1, \#\}^*$ gemäß des Homomorphismus ist. In diesem Fall kann die Maschine w' berechnen und auf dem zweiten Band speichern, das wir im folgenden **Programmband** nennen werden.

Nun ist zu überprüfen, ob $w' = \prod_{i=0}^{n} \prod_{j=0}^{m} w_{i,j,i',j',d}$ gilt, das heißt ob es zu jeder Kombination aus Zustand und Symbol eine korrekt kodierte Transition gibt.

Falls eine dieser Überprüfungen fehlschlägt, weist U ab, denn dann gilt $M_w = M_\varnothing$, und somit $\mathcal{L}(M_w) = \varnothing$. Wenn alle Überprüfungen erfolgreich sind, kopiert U den hinteren Teil der Eingabe, also x, auf das dritte Band, das **Datenband**. Um die Kodierung der Eingabe wollen wir uns hierbei keine Gedanken machen, wir gehen also davon aus, dass $\{0, 1\}$ das Eingabealphabet und $\{0, 1, \$, \sqcup\}$ das Bandalphabet von M_w ist, siehe Bemerkung 4.7.

Nun wird noch das vierte Band, das **Zustandsband** mit $0 = \text{bin}(0)$, der Kodierung des Initialzustands von M_w beschrieben.

Die Simulation von M_w kann nun begonnen werden. Ein Schritt von M_w wird wie folgt simuliert:

- Suche auf dem Programmband nach der Transition für den im Zustandsband gespeicherten Kontrollzustand und das Bandsymbol, auf den der Kopf im Datenband zeigt.
- Wende die Transition an, d.h. ändere entsprechende den Inhalt des Datenbandes, die Kopfposition und ersetze den Inhalt des Zustandsbandes durch die Kodierung des neuen Kontrollzustandes.

U akzeptiert bzw. weist während der Simulation ab, wenn M_w akzeptieren oder abweisen würde. Hierzu wird nach jedem Schritt überprüft, ob das Zustandsband die Kodierung des akzeptierenden oder abweisenden Zustands speichert. □

C) Unentscheidbarkeit mittels Diagonalisierung

Mithilfe der universellen Turingmaschine und einem Diagonalisierungsverfahren, werden wir nun Turings berühmtes Resultat nachweisen. Der Satz besagt, dass das Akzeptanzproblem zwar semi-entscheidbar, aber nicht entscheidbar ist.

4.11 Bemerkung
Turing zeigte initial, dass das **Halteproblem** unentscheidbar ist.

Halteproblem (HP)	
Gegeben:	Eine Turing-Maschine M, eine Eingabe x
Entscheide:	Hält die Berechnung von M auf x nach endlich vielen Schritten?

Wir identifizieren das Halteproblem mit der folgenden Sprache

$$\mathrm{HP} = \left\{ w\#x \;\middle|\; w, x \in \{0,1\}^*, M_w \text{ hält auf Eingabe } x \right\} \subseteq \{0,1,\#\}^* .$$

Es ist nicht schwierig die Turingmaschine M_w in eine andere Turingmaschine M'_w zu überführen, so dass

$$x \in \mathcal{L}(M_w) \text{ gdw. } M'_w \text{ hält auf } x.$$

Da das Akzeptanzproblem auch eine Variante des Halteproblems ist, können wir Turings Resultat ebenso mithilfe des Akzeptanzproblems beweisen.

4.12 Theorem: Turing 1936
Das Akzeptanzproblem ACCEPT ist semi-entscheidbar, allerdings nicht entscheidbar.

Beweis:
Um zu zeigen, dass ACCEPT semi-entscheidbar ist, kann analog zur Konstruktion der universellen Turing-Maschine U eine Turing-Maschine U' konstruieren mit $\mathcal{L}(U') = \mathcal{L}_{\text{Accept}}$. Für eine Eingabe $w\#x$ simuliert U' die Maschine M_w auf der Eingabe x. Falls M_w hält (also akzeptiert oder abweist), akzeptiert U'.

Der andere Teil der Aussage ist das wichtige Resultat: Es zeigt, dass die Klasse der entscheidbaren Sprachen eine echte Teilklasse der semi-entscheidbaren Sprachen ist.

Wir nehmen an, $\mathcal{L}_{\text{Accept}}$ sei entscheidbar, und leiten einen Widerspruch her. Sei H ein Entscheider mit $\mathcal{L}(H) = \mathcal{L}_{\text{Accept}}$. Eine Eingabe $w\#x$ wird von H also nach endlich vielen Schritten akzeptiert, wenn $x \in \mathcal{L}(M_w)$, und andernfalls wird sie nach endlich vielen Schritten abgewiesen.

Wir konstruieren eine neue Maschine D, die eine Eingabe w erwartet und

- hält und akzeptiert falls $w \notin \mathcal{L}(M_w)$ und
- hält und abweist falls $w \in \mathcal{L}(M_w)$.

Wir können den Entscheider D mithilfe von H konstruieren. Dafür simuliert D zunächst H auf der Eingabe $w\#w$ und invertiert am Ende die Antwort von H, also

- falls H abweist, akzeptiert D,

- falls H akzeptiert, weist D ab.

Betrachten wir nun die Eingabe $\langle D \rangle$ für Maschine D, wir fragen uns also, ob die Maschine D ihrer eigene Kodierung akzeptiert.

Fall 1: D akzeptiert $\langle D \rangle$:
Nach Konstruktion von D weist also H die Eingabe $\langle D \rangle \# \langle D \rangle$ ab. Da $\mathcal{L}(H) = \mathcal{L}_{\text{Accept}}$, bedeutet dies, dass $\langle D \rangle \notin \mathcal{L}(D)$ - ein Widerspruch.

Fall 2: D weist $\langle D \rangle$ ab:
Nach Konstruktion von D akzeptiert H also die Eingabe $\langle D \rangle \# \langle D \rangle$. Da $\mathcal{L}(H) = \mathcal{L}_{\text{Accept}}$, bedeutet dies, dass $\langle D \rangle \in \mathcal{L}(D)$ - ein Widerspruch.

In beiden Fällen erhalten wir einen Widerspruch, die Maschine D, und damit auch die Maschine H, kann also nicht existieren. □

4.13 Bemerkung
Im obigen Beweis haben wir das Prinzip der Diagonalisierung verwendet.

	w_{M_1}	w_{M_2}	w_{M_3}	...	w_{M_D}	...
M_1	accept	accept	reject		reject	
M_2	reject	reject	reject		accept	
M_3	reject	accept	accept		accept	
⋮						
M_D	accept	accept	accept		?	
⋮						

Der Eintrag in der Tabelle zu Entscheider M_i und Eingabe w_{M_j} ist „accept“, falls M_i Eingabe w_{M_j} akzeptiert und „reject“, sonst. Die im Beweis konstruierte Maschine D invertiert die Diagonale dieser Tabelle. Der Widerspruch wird bei dem Eintrag „?“ hergeleitet.

Im Beweis der Unentscheidbarkeit des Halteproblems haben wir zur Konstruktion des Widerspruchs nur die Diagonale, also eine Eingabe der Form $\langle D \rangle \# \langle D \rangle$ benötigt. Damit ist beweisen, dass sogar das **spezielle Akzeptanzproblem** unentscheidbar ist.

Spezielles Akzeptanzproblem (SACCEPT)
Gegeben: Eine Turing-Maschine M
Entscheide: Akzeptiert M die Eingabe $\langle M \rangle$?

$$\mathcal{L}_{\text{SACCEPT}} = \left\{ w \in \{0,1\}^* \mid w \in \mathcal{L}(M_w) \right\}.$$

4.14 Korollar
Das spezielle Akzeptanzproblem SACCEPT ist semi-entscheidbar, aber nicht entscheidbar.

D) Reduktionen

Um weitere Unentscheidbarkeitsresultate zu zeigen, möchten wir nicht in jedem Fall einen separaten Beweis durch Diagonalisierung führen, oder – wie im Fall des speziellen Akzeptanzproblems – einen Beweis genau inspizieren. Stattdessen wollen wir bereits gezeigte Resultate verwenden, um aus der Unentscheidbarkeit z.B. des Akzeptanz- oder Halteproblems die Unentscheidbarkeit weiterer Probleme abzuleiten.

Hierzu betrachten wir **Reduktionen**. Um zu zeigen, dass ein Problem B unentscheidbar ist, beweisen wir, dass im Problem bereits ein bereits als Unentscheidbar bekanntes Problem A als Spezialfall eingebettet ist. Man sagt, dass Problem A auf Problem B reduziert. Ein Entscheidungsverfahren für B kann somit nicht existieren, denn es würde auch als Entscheider für das unentscheidbare Problem A fungieren.

4.15 Definition
Es seien $A \subseteq \Sigma_1^*, B \subseteq \Sigma_2^*$ Sprachen. Problem A ist **(many-one-)reduzierbar** auf Problem B, geschrieben als $A \leqslant B$, falls es eine totale und berechenbare Funktion

$$f\colon \Sigma_1^* \to \Sigma_2^*$$

gibt, so dass für alle $x \in \Sigma_1^*$ gilt

$$x \in A \quad \text{gdw. } f(x) \in B\,.$$

Eine solche Funktion f heißt **(Many-One-)Reduktion**. Wenn $A \leqslant B$ gilt, dann ist B *mindestens so schwer wie* A.

Das folgende Lemma formalisiert die obige Diskussion dazu, dass sich Reduktionen für Unentscheidbarkeitsbeweise nutzen lassen.

4.16 Lemma
Wenn $A \leqslant B$ gilt und B (semi-)entscheidbar ist, dann ist auch A (semi-)entscheidbar.

Beweis:
Wir betrachten zunächst den Fall, dass B entscheidbar ist.

Sei f eine Reduktion von A auf B, und sei M_f eine Turing-Maschine, die f berechnet. Sei M_B ein Entscheider für B.

Wir konstruieren einen Entscheider M_A für A wie folgt: Auf einer Eingabe x verhält M_A sich zunächst wie M_f, um innerhalb von endlich vielen Schritten auf einem zusätzlichen Band $f(x)$ zu schreiben.

Nun verhält sich M_A wie M_B, wobei das Band, auf dem $f(x)$ steht, als Eingabeband für M_B genutzt wird. Wenn M_B akzeptiert oder abweist, akzeptiert bzw. weist M_A ab.

Da $x \in A$ genau dann, wenn $f(x) \in B$, ist M_A tatsächlich ein Entscheider für A.

Falls B semi-entscheidbar ist und M_B ein Semi-Entscheider für B, so ist M_A ein Semi-Entscheider für A. □

Üblicherweise wird die Kontraposition des obigen Lemmas verwendet.

4.17 Korollar
Wenn $A \leqslant B$ und A nicht (semi-)entscheidbar ist, dann ist auch B nicht (semi-)entscheidbar.

Beweis: Angenommen B wäre semi-entscheidbar, dann mit dem obigen Lemma auch A. □

4.18 Bemerkung
Es gibt einen alternativen Beweis des Lemmas, der ohne die explizite Konstruktion eines Entscheiders auskommt.

Wenn B entscheidbar ist, dann ist die totale charakteristische Funktion χ_B berechenbar. Des Weiteren gibt es eine berechenbare Reduktion f von A auf B.

Die Verkettung berechenbarer Funktionen ist erneut berechenbar. Damit ist die charakteristische Funktion von A

$$\chi_A = \chi_B \circ f$$

berechenbar, und somit A entscheidbar.

4.19 Bemerkung
In der Literatur werden zum Teil andere Sorten Reduktionen betrachtet, z.B. sogenannte Turing-Reduktionen. Wir verwenden hier Many-One-Reduktionen, zum Einen, weil Sie sich im Gegensatz zu Turing-Reduktionen auch dazu verwenden lassen, semi- und co-semi-entscheidbare Probleme zu untersuchen, zum Anderen weil wir diese Sorte Reduktion im nächsten Teil der Vorlesung zu Komplexitätstheorie erneut verwenden werden.

Als Beispiel für die Anwendung von Reduktionen betrachten wir eine weitere Variante des Halteproblems.

Halten auf leerem Eingabeband (HP_ε)	
Gegeben:	Eine Turing-Maschine M
Entscheide:	Hält die Berechnung von M auf dem leeren Wort ε?

$$\mathrm{HP}_\varepsilon = \left\{ w \in \{0,1\}^* \mid M_w \text{ hält auf } \varepsilon \right\}.$$

HP_ε ist unentscheidbar, dies beweisen wir, in dem wir das allgemeine Halteproblem HP auf HP_ε reduzieren.

4.20 Lemma
$\mathrm{HP} \leqslant \mathrm{HP}_\varepsilon$.

Beweis:
Wir definieren eine Reduktion f, die HP auf HP_ε reduziert. Sei $w\#x$ eine Eingabe für das allgemeine Halteproblem.

Wir konstruieren eine Maschine M_w^x, die sich wie folgt verhält:

- Sie löscht die Eingabe (d.h. überschreibe den Inhalt des Eingabeband mit $\sqcup$),
- sie schreibt x auf das Eingabeband, und dann
- verhält sie sich wie M_w.

Die Kodierung $\langle M_w^x \rangle$ dieser Maschine ist der Funktionswert unserer Reduktion f,

$$\begin{array}{rccc} f & : & \{0,1,\#\}^* & \to & \{0,1\}^* \\ & & w\#x & \mapsto & \langle M_w^x \rangle . \end{array}$$

Man kann nachweisen, dass f tatsächlich berechenbar ist. Die Ideen, die nötig sind, um dies formal zu beweisen, lassen sich in der Konstruktion der universellen Turing-Maschine finden.

Es gilt: M_w hält auf Eingabe x genau dann, wenn M_w^x auf einer beliebigen Eingabe hält. Hierzu ist zu beachten, dass sich M_w^x unabhängig von der Eingabe immer wie M_w mit Eingabe x verhält.

Insbesondere gilt also: M_w hält auf Eingabe x genau dann, wenn M_w^x auf Eingabe ε hält. □

Oben wurde bereits erwähnt, dass das Akzeptanzproblem ebenso wie das Halteproblem unentscheidbar ist. Dies beweisen wir nun unter der Verwendung von Reduktionen.

4.21 Theorem
Die folgenden Probleme sind semi-entscheidbar, aber nicht entscheidbar.

a) Das **allgemeine Akzeptanzproblem**

$$\text{ACCEPT} = \left\{ w\#x \in \{0,1,\#\}^* \;\middle|\; w, x \in \{0,1\}^*, x \in \mathcal{L}(M_w) \right\}.$$

b) Das **Selbstakzeptanzproblem** oder **spezielle Akzeptanzproblem**

$$\text{SELF-ACCEPT} = \left\{ w \in \{0,1\}^* \;\middle|\; w \in \mathcal{L}(M_w) \right\}.$$

c) Das **Leeres-Wort-Akzeptanzproblem**

$$\text{ACCEPT}_\varepsilon = \left\{ w \in \{0,1\}^* \;\middle|\; \varepsilon \in \mathcal{L}(M_w) \right\}.$$

Beweis:
ACCEPT ist die Sprache der universellen Turing-Maschine, und damit semi-entscheidbar.

Um zu zeigen, dass SELF-ACCEPT und $\text{ACCEPT}_\varepsilon$ semi-entscheidbar sind, können wir Reduktionen in Kombination mit Lemma 4.16 verwenden.

- SELF-ACCEPT $\leqslant$ ACCEPT:
  Betrachte die Reduktion, die eine Instanz w von SELF-ACCEPT nimmt und die Instanz $w\#w$ von ACCEPT ausgibt.
- $\text{ACCEPT}_\varepsilon \leqslant$ ACCEPT:
  Betrachte die Reduktion, die eine Instanz w von $\text{ACCEPT}_\varepsilon$n nimmt und die Instanz $w\# = w\#\varepsilon$ von ACCEPT ausgibt.

Um die Unentscheidbarkeit der Probleme zu beweisen, können wir die Kontraposition von Lemma 4.16, also Korollar 4.17, verwenden, und für jede Variante des Akzeptanzproblems eine Reduktion des jeweiligen Halteproblems angeben.

Wir beweisen hier exemplarisch HP $\leqslant$ ACCEPT, die anderen beiden Beweise funktionieren analog.

Gegeben sei eine Instanz $w\#x$ des Halteproblems. Ziel ist es, eine Instanz $w'\#x$ des Akzeptanzproblems zu erzeugen, so dass gilt

$$w\#x \in \text{HP} \quad \text{gdw.} \quad w'\#x \in \text{ACCEPT}.$$

(Theoretisch dürfte die Reduktion auch x verändern, wir werden jedoch sehen, dass dies nicht notwendig ist.)

Zur gegebenen, durch w kodierten, Maschine M_w konstruieren wir eine Maschine M'_w, die

- sich zunächst wie M_w verhält,

- falls M_w akzeptiert, akzeptiert und
- falls M_w abweist, ebenfalls akzeptiert.

Es gilt: M'_w akzeptiert Eingabe x genau dann, wenn M_w auf Eingabe x hält. Die Funktion, die eine Instanz $w\#x$ des Halteproblems entgegennimmt und die Instanz $\langle M'_w \rangle \# x$ des Akzeptanzproblems zurückgibt, ist die gesuchte Reduktion.

Die Berechnung von $\langle M'_w \rangle$ lässt sich durch eine Manipulation der Kodierung von w erreichen: Alle Transitionen, die von einem nicht-haltenden Zustand nach q_{rej} führen, werden so abgeändert, dass sie stattdessen nach q_{acc} führen.

Daher ist $w\#x \mapsto \langle M'_w \rangle \# x$ eine berechenbare Reduktion, die HP $\leqslant$ ACCEPT beweist. □

5. Das Postsche Korrespondenzproblem & der Satz von Rice

Im letzten Kapitel haben wir gesehen, dass wir Probleme indirekt als unentscheidbar nachweisen können, in dem wir ein bereits als unentscheidbar bekanntes Problem auf sie reduzieren. In diesem Kapitel wollen wir zwei wichtige Resultate zur Unentscheidbarkeit kennen lernen, nämlich die Unentscheidbarkeit des **Postschen Korrespondenzproblems (PCP)**, und den **Satz von Rice**. In beiden Fällen beweisen wir die Unentscheidbarkeit durch eine aufwendige Reduktion des Halteproblems.

Das Postsche Korrespondenzproblem ist, im Gegensatz zu den bisher betrachteten unentscheidbaren Problemen, nicht über Turing-Maschinen definiert. Wenn man ein Problem als unentscheidbar nachweisen möchte, ist es oft technisch einfacher, das PCP statt das Halteproblem zu reduzieren.

Der Satz von Rice sagt, dass fast alle Probleme, bei denen es darum geht zu entscheiden, ob die Sprache einer Turing-Maschine eine bestimmte Eigenschaft hat, unentscheidbar sind.

A) Das Postsche Korrespondenzproblem (PCP)

5.1 Definition
Das **Postsche Korrespondenzproblem (PCP – Post's correspondence problem)** ist das wie folgt definierte Entscheidungsproblem:

Postsches Korrespondenzproblem (PCP)	
Gegeben:	Eine endliche Sequenz von Tupeln aus Wörtern $(x_1, y_1), \ldots, (x_k, y_k)$
Entscheide:	Gibt es eine endliche, nicht-leere Sequenz von Indizes $i_1 \ldots i_n$ mit $x_{i_1} x_{i_2} \ldots x_{i_n} = y_{i_1} y_{i_2} \ldots y_{i_n}$?

Man kann sich das Postsche Korrespondenzproblem wie folgt vorstellen: Gegeben sind Sorten von Dominosteinen, wobei jede Sorte Dominostein j oben mit x_j und unten mit y_j beschriftet ist. Nehmen wir nun an, dass wir von jeder Sorte beliebig viele Dominosteine zur Verfügung haben. Ziel ist es, eine Reihe aus Dominosteinen zu bilden, so dass die Wörter, die sich aus den oberen bzw. unteren Beschriftungen ergeben übereinstimmen.

5.2 Beispiel
Betrachte die Instanz

$$K = \underbrace{(1, 101)}_{1}, \underbrace{(10, 00)}_{2}, \underbrace{(011, 11)}_{3} .$$

Es handelt sich hierbei um eine Ja-Instanz, denn 1323 ist eine Sequenz wie gewünscht; es gilt

$$1.011.10.011 = 101.11.00.11\ .$$

1	011	10	011
101	11	00	11
1	3	2	3

Im Rest dieses Teilkapitels wollen wir den folgenden Satz beweisen.

5.3 Theorem: Post 1946
Das PCP ist unentscheidbar.

Zum Beweis wollen wir das Akzeptanzproblem auf das PCP reduzieren. Direkt wäre dies schwierig, wir definieren zunächst eine modifizierte Version des PCP, und führen dann den Beweis in zwei Schritten.

Modifiziertes Postsches Korrespondenzproblem (MPCP)

Gegeben: Eine endliche Sequenz von Tupeln aus Wörtern $(x_1, y_1), \ldots, (x_k, y_k)$

Entscheide: Gibt es eine endliche, nicht-leere Sequenz von Indizes $i_1 \ldots i_n$ mit $x_{i_1} x_{i_2} \ldots x_{i_n} = y_{i_1} y_{i_2} \ldots y_{i_n}$ und $i_1 = 1$?

Wir zeigen zunächst, dass sich das modifizierte PCP auf das PCP reduzieren lässt. Wenn wir dann später bewiesen haben, dass MPCP unentscheidbar ist, muss auch PCP unentscheidbar sein, ansonsten würden wir über die Reduktion ein Entscheidungsverfahren für das MPCP erhalten.

5.4 Lemma
MPCP $\leqslant$ PCP.

Beweis:
Sei $K = (x_1, y_1), \ldots, (x_k, y_k)$ eine gegebene MPCP-Instanz. Sei Σ das Alphabet, das die Buchstaben beinhaltet, die in den Wörtern x_j und y_j vorkommen. Wir gehen o.B.d.A. davon aus, dass die Symbole \$ und # nicht in Σ vorkommen.

Für jedes Wort $w = a_1 \dots a_m \in \Sigma^*$ definieren wir drei Varianten:

$$\bar{w} = \#a_1\#a_2\# \dots \#a_m\# \,,$$
$$\grave{w} = \#a_1\#a_2\# \dots \#a_m \,,$$
$$\acute{w} = a_1\#a_2\# \dots \#a_m\# \,.$$

Zur gegeben Instanz K konstruieren wir nun die Instanz

$$f(K) = (\bar{x_1}, \grave{y_1}), (\acute{x_1}, \grave{y_1}), (\acute{x_2}, \grave{y_2}), \dots, (\acute{x_k}, \grave{y_k}), (\$, \#\$) \,.$$

Die Funktion f, die eine MPCP-Instanz K nimmt, und die PCP-Instanz $f(K)$ zurückgibt, ist berechenbar. Wir müssen zeigen, dass sie in der Tat eine Reduktion ist, d.h. dass gilt

K hat eine Lösung mit $i_1 = 1$ gdw. $f(K)$ hat eine beliebige Lösung.

Wir zeigen beide Richtungen.

„⇒“ Sei $i_1 \dots i_n$ eine Lösung für K mit $i_1 = 1$, dann ist die folgende Sequenz eine Lösung für $f(K)$:

$$1, i_2 + 1, i_3 + 1, \dots, i_n + 1, k + 2 \,.$$

„⇐“ Wenn Instanz $f(K)$ eine Lösung hat, dann hat sie auch eine Lösung mit minimaler Länge. Sei $i_1 \dots i_n \in \{1, \dots, k + 2\}^*$ eine solche minimale Lösung für $f(K)$. Es muss aufgrund der Konstruktion von $f(K)$ gelten:

- $i_1 = 1$, denn kein anderes Paar hat in beiden Komponenten $\#$ als erstes Symbol,
- $i_n = k + 2$, denn kein anderes Paar hat in beiden Komponenten das selbe letzte Symbol.

Da wir annehmen, dass $i_1, \dots, i_n$ eine minimale Lösung ist, kommen 1 und $k + 2$ nicht innerhalb der Sequenz erneut vor. Angenommen, dies wäre der Fall, dann könnten wir die Lösungen in zwei Teile aufspalten, von denen einer erneut eine Lösung für die Instanz ist, ein Widerspruch zur Minimalität.

Also gilt $i_2 \dots i_{n-1} \in \{2, \dots, k + 1\}^*$. Die Sequenz

$$1, i_2 - 1, \dots, i_{n-1} - 1$$

ist eine Lösung für K, deren erster Eintrag wie gefordert 1 ist.

□

Nun zeigen wir, dass die modifizierte Version des PCPs unentscheidbar ist, in dem wir das Akzeptanzproblem auf sie reduzieren.

5.5 Proposition
ACCEPT $\leqslant$ MPCP.

Beweis:
Sei $w\#x$ eine Eingabe für das Akzeptanzproblem, wobei w die Turing-Maschine $M_w = (Q, \Sigma, \Gamma, \delta, q_0, \{q_{acc}, q_{rej}\})$ kodiert, und $x = a_1 \dots a_n \in \Sigma^*$ die Eingabe für diese Maschine ist.

Unser Ziel ist es, eine Funktion anzugeben, die aus einer solchen Eingabe eine Sequenz von Paaren $K = (x_1, y_1), \dots, (x_k, y_k)$ berechnet, so dass gilt

$$M_w \text{ akzeptiert } x \qquad \text{gdw.} \qquad K \text{ hat eine Lösung mit } i_1 = 1.$$

Die Turing-Maschine M_w akzeptiert Eingabe x genau dann wenn es eine endliche Sequenz von Konfigurationen

$$c_0 \to \dots \to c_t$$

gibt mit $c_0 = q_0\, x$ und $c_t = u\, q_{acc}\, v$.

Wir werden sicher stellen, dass in diesem Fall die MPCP-Instanz eine Lösung der Form

$$\#c_0\#c_1\#\dots\#c_t\#c_t'\#\dots\#q_{acc}\#\#$$

hat. Dies bedeutet, dass die Lösung in der MPCP-Instanz die Sequenz der Konfigurationen von M_w für Eingabe x, also die Berechnung, kodiert.

Das Alphabet der PCP-Instanz ist $\Gamma \uplus Q \uplus \{\#\}$. Das erste Paar ist $(\#, \#q_0x\#)$, kodiert also die initiale Konfiguration von M_w.

Ziel ist es, dass sowohl die x-Sequenz (also die Sequenz der x_{i_j} in einer Lösung $i_1 \dots i_n$) als auch die y-Sequenz die Berechnung der Turing-Maschine kodieren. Die x-Sequenz fällt dabei einen Schritt zurück: Wenn das Anhängen eines Paares (x_i, y_i) in der y-Sequenz zur Konfiguration c_ℓ beitragt, trägt es in der x-Sequenz zur Konfiguration $c_{\ell-1}$ bei. Dieser Versatz erlaubt es uns, sicherzustellen, dass Konfiguration $c_{\ell+1}$ der Nachfolger der Konfiguration c_ℓ gemäß der Transitionsrelation auf Konfigurationen der TM ist.

Die folgende Darstellung illustriert dies. Die in Klammern angegebenen Zahlen j geben an, aus welchem Index i_j der Lösung die entsprechenden Symbole kommen.

$$x\text{-Sequenz}: \quad \overbrace{\#}^{1}\overbrace{q_0a_1}^{2}\overbrace{a_2}^{3}\ldots\overbrace{a_n}^{n+1}\overbrace{\#}^{n+2}$$

$$y\text{-Sequenz}: \quad \underbrace{\#\ q_0\ \ a_1\ \ a_2\ \ \ldots\ \ \ a_n\ \#}_{1}\ \ \underbrace{q_1b}_{2}\underbrace{a_2}_{3}\ldots\underbrace{a_n}_{n+1}\underbrace{\#}_{n+2}$$

Zusätzlich zum oben angegebenen ersten Paar sind die folgenden Paare in der konstruierten PCP-Instanz:

- Kopierregeln:

$$(a, a) \text{ für jedes } a \in \Gamma \cup \{\#\},$$

- Transitionsregeln:

$$\begin{aligned}
(qa, q'b), &\text{ falls } \delta(q, a) = (q', b, N),\\
(qa, bq'), &\text{ falls } \delta(q, a) = (q', b, R),\\
(cqa, q'cb), &\text{ falls } \delta(q, a) = (q', b, L) \text{ für alle } c \in \Gamma,\\
(q\#, bq'\#), &\text{ falls } \delta(q, \sqcup) = (q', b, R),\\
(q\#, q'b\#), &\text{ falls } \delta(q, \sqcup) = (q', b, N),\\
(cq\#, q'cb\#), &\text{ falls } \delta(q, \sqcup) = (q', b, L) \text{ für alle } c \in \Gamma,
\end{aligned}$$

- Löschregeln:

$$(aq_{acc}, q_{acc}) \text{ und } (q_{acc}a, q_{acc}) \text{ für alle } a \in \Gamma,$$

- Abschlussregel:

$$(q_{acc}\#\#, \#).$$

Zu jeder akzeptierenden Berechnung von M_w für Eingabe x lässt sich eine Lösung der MPCP-Instanz konstruieren:

- Die Lösung beginnt mit dem initialen Paar.
- Danach werden die Kopier- und Transitionsregeln benutzt, um die Berechnung der Maschine bis zur ersten akzeptierenden Konfiguration zu simulieren.
- Nun kann man mit den Löschregeln den Bandinhalt löschen: In jeder Konfiguration entfernt man das Symbol links oder rechts vom Kontrollzustand.

  Wir erhalten Sequenzen der folgenden Form.

$$x\text{-Sequenz}: \quad \#c_0\#c_1\#\ldots\#c_t\#\overbrace{\ldots}^{\text{Löschen}}\#$$
$$y\text{-Sequenz}: \quad \#c_0\#c_1\#\ldots\#c_t\#\underbrace{\ldots}_{\text{Löschen}}\#q_{acc}\#$$

- Durch Verwenden der Abschlussregel werden die beiden Sequenzen gleich.

Es lässt sich zeigen, dass jede Lösung der MPCP-Instanz die obige Form haben muss, also eine akzeptierende Berechnung von M_w kodiert. □

Durch eine einfache Reduktion lässt sich beweisen, dass bereits das sogenannte 0, 1-PCP, also das PCP für Instanzen, bei denen die Wörter x_i und y_i über dem Alphabet $\{0, 1\}$ sind, unentscheidbar ist.

5.6 Bemerkung
Für $k \in \mathbb{N}, k > 0$ sei PCP_k das PCP für Instanzen, die aus genau k Paaren (x_i, y_i) bestehen. Es ist bekannt, dass PCP_9 unentscheidbar ist und dass PCP_2 entscheidbar ist. Dementsprechend ist auch PCP_k für $k > 9$ unentscheidbar, und PCP_1 ist trivial. Die Probleme PCP_3 bis PCP_8 sind offen.

B) Der Satz von Rice

Wir möchten nun den **Satz von Rice**, ein allgemeineres Resultat zur Unentscheidbarkeit, beweisen. Er sagt aus, dass *jede* nicht-triviale Eigenschaft des Verhaltens von Turing-Maschinen unentscheidbar ist. Dies besagt letztlich, dass Unentscheidbarkeit der Regelfall bei Verifikationsproblemen ist, und nicht bloß eine Ausnahme.

5.7 Definition
Sei Σ ein Alphabet. Wir bezeichnen mit RE(Σ) die Menge aller rekursiv-aufzählbaren (semientscheidbaren) Sprachen über Σ, also die Menge aller Sprachen $\mathcal{L}$, zu denen es eine TM M mit $\mathcal{L}(M) = \mathcal{L}$ gibt. (RE steht für *recursively enumerable*.)

Eine **Eigenschaft** P der Sprachen in $RE(\Sigma)$ ist eine Funktion

$$P\colon \text{RE}(\Sigma) \to \{0, 1\} \;\hat{=}\; \mathbb{B} = \{\mathit{false}, \mathit{true}\}\,.$$

Wir sagen, dass eine Sprache $\mathcal{L} \in \text{RE}(\Sigma)$ Eigenschaft P hat, falls $P(\mathcal{L}) = 1$ gilt.

Eine Eigenschaft heißt **trivial**, falls P eine konstante Funktion ist, also $P(L) = 0$ für alle $\mathcal{L} \in \text{RE}(\Sigma)$ oder $P(L) = 1$ für alle $\mathcal{L} \in \text{RE}(\Sigma)$, ansonsten heißt sie **nicht-trivial**.

Dass eine Eigenschaft P nicht-trivial ist, bedeutet, dass es jeweils mindestens eine Sprache gibt, die die Eigenschaft hat, und eine Sprache, die sie nicht hat. Es gibt in diesem Fall also $\mathcal{L}, \mathcal{L}'$ mit $P(\mathcal{L}) = 1$ und $P(\mathcal{L}') = 0$.

Eine Eigenschaft P zu entscheiden, bedeutet für eine gegebene Sprache $\mathcal{L}$ algorithmisch zu entscheiden, ob $P(\mathcal{L}) = 1$ gilt. Hierzu muss die Sprache eine endliche Darstellung besitzen, die wir als Eingabe des Algorithmus nutzen können. Da wir über semi-entscheidbare Sprachen sprechen, liegt es nahe, eine Sprache $\mathcal{L}$ durch eine Turing-Maschine M mit $\mathcal{L} = \mathcal{L}(M)$ darzustellen, oder genauer gesagt, durch die Kodierung $\langle M \rangle$ einer solchen Maschine.

Wir sehen eine Eigenschaft P also als Sprache

$$P = \left\{ w \in \{0,1\}^* \mid P(\mathcal{L}(M_w)) = 1 \right\},$$

die Eigenschaft zu entscheiden, bedeutet, diese Sprache zu entscheiden.

Man beachte, dass wir Eigenschaften von Sprachen, nicht Eigenschaften von Turing-Maschinen, betrachten. Ob eine Kodierung in P liegt, darf also nur von der Sprache $\mathcal{L}$ der Maschine abhängen und muss ansonsten unabhängig von der Maschine sein. Entweder gilt $P(\mathcal{L}(M_w)) = 1$, und damit $w \in P$, für alle w mit $\mathcal{L}(M_w) = \mathcal{L}$, oder es gilt $P(\mathcal{L}(M_w)) = 0$, und damit $w \notin P$, für alle w mit $\mathcal{L}(M_w) = \mathcal{L}$.

5.8 Beispiel
Die folgenden Eigenschaften sind nicht-triviale Eigenschaften von semi-entscheidbaren Sprachen:

- $\mathcal{L} = \mathcal{L}(M_w)$ ist endlich,
- $\mathcal{L} = \mathcal{L}(M_w)$ ist regulär,
- $\mathcal{L} = \mathcal{L}(M_w)$ ist kontextfrei,
- $\mathcal{L} = \mathcal{L}(M_w)$ ist entscheidbar,
- $10110 \in \mathcal{L}$, d.h. M_w akzeptiert Eingabe 10110,
- $\mathcal{L} = \Sigma^*$, d.h. M_w ist universell.

Die folgenden beiden Eigenschaften sind Eigenschaften von semi-entscheidbaren Sprachen, allerdings trivial:

- L ist Bild einer totalen berechenbaren Funktion,
- L ist nicht-semi-entscheidbar.

Die folgenden Eigenschaften sind Eigenschaften von Turing-Maschinen, nicht Eigenschaften ihrer Sprachen:

- M_w hat 481 Kontrollzustände,
- Die Berechnung von M_w auf Eingabe 10110 hält nach höchstens 10 Schritten,
- M_w ist ein Entscheider,
- Es gibt eine kleinere TM mit der selben Sprache.

Für jedes dieser Beispiele lassen sich Maschinen M_w und $M_{w'}$ finden, deren Sprache gleich ist, $\mathcal{L}(M_w) = \mathcal{L}(M'_w)$, wobei M_w die gewünschte Eigenschaft hat und $M_{w'}$ sie nicht hat.

5.9 Theorem: Rice 1953
Jede nicht-triviale Eigenschaft der semi-entscheidbaren Sprachen ist unentscheidbar.

Beweis:
Sei P eine beliebige nicht-triviale Eigenschaft der semi-entscheidbaren Sprachen über einem Alphabet Σ. Wir gehen o.B.d.A. davon aus, dass $P(\varnothing) = 0$ gilt, der Beweis für den anderen Fall funktioniert analog. (Beachte, dass $\varnothing$ natürlich rekursiv aufzählbar ist.)

Da P nicht-trivial ist, gibt es auch eine semi-entscheidbare Sprache $\mathcal{L}$ mit $P(\mathcal{L}) = 1$. Sei K eine Turing-Maschine mit $\mathcal{L}(K) = \mathcal{L}$.

Wir reduzieren das Akzeptanzproblem auf die Eigenschaft P, also auf die Sprache

$$P = \left\{ w \in \{0,1\}^* \mid P(\mathcal{L}(M_w)) = 1 \right\},$$

woraus folgt, dass P unentscheidbar sein muss.

Sei $w\#x$ eine Eingabe für das Akzeptanzproblem, bestehend aus der Kodierung der Maschine M_w und der Eingabe x. Wir konstruieren eine Maschine $M^K_{w,x}$, deren Sprache $\varnothing$ ist, falls M_w Eingabe x nicht akzeptiert, und deren Sprache $\mathcal{L}$ ist, falls M_w x akzeptiert. Eigenschaft P für diese Maschine zu entscheiden, bedeutet also, dass man entscheidet, ob M_w auf x hält.

Für eine Eingabe y verhält sich $M^K_{w,x}$ wie folgt:

1. Speichere Eingabe y auf einem separaten Band
2. Ersetze den Inhalt des Eingabebands durch das fixierte Wort x. Dadurch, dass x fixiert ist, lässt sich dies in die Transitionsfunktion kodieren.
3. Simuliere M_w auf Eingabe x.
4. Falls M_w Eingabe x akzeptiert, also die obige Simulation in einer akzeptierenden Konfiguration hält, simuliere K auf Eingabe y.

   Akzeptierte genau dann, wenn K Eingabe y akzeptiert.

1. Fall: $M_w$akzeptiert Eingabe x.

In diesem Fall hält die Simulation in Schritt 3., und es gilt $y \in \mathcal{L}\left(M^K_{w,x}\right)$ genau dann, wenn $y \in \mathcal{L}(K)$ gilt. Da K eine Maschine für die Sprache $\mathcal{L}$ mit Eigenschaft P war, ist dies der Fall genau dann, wenn $y \in \mathcal{L}$.

2. Fall: M_w akzeptiert Eingabe x nicht.

In diesem Fall wird Schritt 4. nicht erreicht, und somit akzeptiert $\mathcal{L}\left(M^K_{w,x}\right)$ keine Eingabe y, unabhängig von der konkreten Eingabe.

Im ersten Fall, wenn M_w x akzeptiert, gilt also $\mathcal{L}\left(M^K_{w,x}\right) = \mathcal{L}(K) = \mathcal{L}$, und im zweiten Fall, in dem M_w x nicht akzeptiert, gilt $\mathcal{L}\left(M^K_{w,x}\right) = \varnothing$.

Zusammenfassend erhalten wir:

- Wenn wenn M_w auf x hält, gilt $P\left(\mathcal{L}\left(M^K_{w,x}\right)\right) = P(\mathcal{L}) = 1$.
- Wenn wenn M_w auf x nicht hält, gilt $P\left(\mathcal{L}\left(M^K_{w,x}\right)\right) = P(\varnothing) = 0$.

Dies schließt den Beweis ab: Angenommen wir hätten einen Entscheider für P, dann würde dieser Entscheider angewandt auf $M^K_{w,x}$ das Akzeptanzproblem für die Eingabe $w\#x$ entscheiden, dies ist ein Widerspruch. □

Der Satz von Rice sagt nur aus, dass P unentscheidbar ist. Es gibt nicht-triviale Eigenschaften die semi-entscheidbar oder co-semi-entscheidbar (jedoch nicht beides gleichzeitig) sind, hierüber trifft der obige Satz keine Aussage. Es gibt eine Variante des Satzes, die etwas über die Semi-Entscheidbarkeit von Eigenschaften aussagt, die wir hier ohne Beweis angeben wollen.

Wir nennen eine Eigenschaft P der semi-entscheidbaren Sprachen **monoton** wenn für alle Sprachen $\mathcal{L}, \mathcal{L}' \in \mathrm{RE}(\Sigma)$ gilt, das $\mathcal{L} \subseteq \mathcal{L}'$ impliziert, dass $P(\mathcal{L}) \leqslant P(\mathcal{L}')$. Andernfalls heißt P **nicht-monoton**.

Monotonie einer Eigenschaft P bedeutet, dass zu jeder Sprache mit $\mathcal{L}$ mit Eigenschaft P auch jede größere Sprache $\mathcal{L}' \supseteq \mathcal{L}$ die Eigenschaft P hat.

5.10 Theorem: Rice 1956
Jede nicht-monotone Eigenschaft der semi-entscheidbaren Sprachen ist nicht-semi-entscheidbar.

6. Unentscheidbare Probleme kontextfreier Sprachen

Wir wollen den Teil der Vorlesung zu Entscheidbarkeit abschließen, in dem wir die aus „Theoretische Informatik I“ bekannten kontextfreien Sprachen untersuchen. Im Gegensatz zu den semi-entscheidbaren Sprachen sind manche Probleme für kontextfreie Sprachen entscheidbar.

- Sei $\mathcal{L}$ eine kontextfreie Sprache, die durch eine kontextfreie Grammatik oder einen Pushdown-Automaten gegeben ist. Das Wortproblem, also gegeben ein Wort w, entscheide, ob $w \in \mathcal{L}$ gilt, ist mit Hilfe des CYK-Algorithmus entscheidbar.
- Das Leerheitsproblem, also gegeben eine Grammatik G, entscheide ob $\mathcal{L}(G) = \varnothing$ gilt, ist entscheidbar.

Es gibt jedoch auch viele Probleme, deren Eingabe eine kontextfreie Sprache beinhaltet, die unentscheidbar sind. Wir wollen einige dieser Probleme in diesem Kapitel betrachten und als unentscheidbar nachweisen. Im Beweis der Unentscheidbarkeit werden wir jeweils das PCP reduzieren.

6.1 Theorem
Die Eingabe der Probleme sind zwei kontextfreie Grammatiken G_1, G_2. Die folgenden Probleme sind unentscheidbar:

a) Gilt $\mathcal{L}(G_1) \cap \mathcal{L}(G_2) \neq \varnothing$?

b) Gilt $|\mathcal{L}(G_1) \cap \mathcal{L}(G_2)| = \infty$?

c) Ist $\mathcal{L}(G_1) \cap \mathcal{L}(G_2)$ kontextfrei?

d) Gilt $\mathcal{L}(G_1) \subseteq \mathcal{L}(G_2)$?

e) Gilt $\mathcal{L}(G_1) = \mathcal{L}(G_2)$?

Beweis:

a) Wir reduzieren das $0, 1$-PCP. Sei

$$K = (x_1, y_1), \ldots, (x_k, y_k)$$

eine gegebene PCP Instanz mit $x_i, y_i \in \{0, 1\}^*$ für alle i.

Wir konstruieren zwei Grammatiken G_1, G_2 über dem Terminalalphabet $\{0, 1, \$, a_1, \ldots, a_k\}$. Bei den Symbolen $a_1, \ldots, a_k$ handelt es sich um ein Symbol a_i pro Paar (x_i, y_i) in K.

Grammatik G_1 hat die folgenden Produktionsregeln:

$$\begin{aligned}
&S \to A\$B\,,\\
&A \to a_1Ax_1 \mid \ldots \mid a_kAx_k\,,\\
&A \to a_1x_1 \mid \ldots \mid a_kx_k\,,\\
&B \to y_1^{\text{reverse}}Ba_1 \mid \ldots \mid y_k^{\text{reverse}}Ba_k\,,\\
&B \to y_1^{\text{reverse}}a_1 \mid \ldots \mid y_k^{\text{reverse}}a_k\,.
\end{aligned}$$

Hierbei ist zu einem Wort $w = w_1 \ldots w_m$ das Wort w^{reverse} durch Umkehrung der Reihenfolge der Buchstaben in w definiert, also $w^{\text{reverse}} = w_k \ldots w_1$.

Die von G_1 erzeugte Sprache ist

$$\mathcal{L}(G_1) = \left\{a_{i_n} \ldots a_{i_1}x_{i_1} \ldots x_{i_n}\$y_{j_m}^{\text{reverse}} \ldots y_{j_1}^{\text{reverse}}a_{j_m} \ldots a_{j_1} \;\middle|\; n, m \geqslant 1, i_s, j_t \in \{1, \ldots, k\}\forall s, t\right\}.$$

Grammatik G_2 hat die folgenden Produktionsregeln:

$$\begin{aligned}
&S \to a_1Sa_1 \mid \ldots \mid a_kSa_k \mid T\,,\\
&T \to 0T0 \mid 1T1 \mid \$\,.
\end{aligned}$$

Die von G_2 erzeugte Sprache ist also

$$\mathcal{L}(G_2) = \left\{u\,v\,\$v^{\text{reverse}}u^{\text{reverse}} \;\middle|\; v \in \{0,1\}^*, u \in \{a_1, \ldots, a_k\}^*\right\}.$$

Es gilt, dass K eine nicht-leere Lösung $i_1 \ldots i_n$ hat, genau dann, wenn der Schnitt $\mathcal{L}(G_1) \cap \mathcal{L}(G_2)$ nicht-leer ist, nämlich das Wort

$$a_{i_n} \ldots a_{i_1}x_{i_1} \ldots x_{i_n}\$y_{i_n}^{\text{reverse}} \ldots y_{i_1}^{\text{reverse}}a_{i_n} \ldots a_{i_1}$$

enthält. Die Funktion f mit $f(K) = (G_1, G_2)$ ist also die gesuchte Reduktion des PCPs auf die Nicht-Leerheit des Schnittes.

b) Falls eine PCP-Instanz eine Lösung hat, hat sie unendlich viele Lösungen: Zu jeder Lösung $s = i_1 \ldots i_n$ sind auch $s^2 = i_1 \ldots i_ni_1 \ldots i_n, s^3, s^4, \ldots$ Lösungen.

Die Reduktion f aus Teil a) ist also auch eine Reduktion auf die Unendlichkeit des Schnittes.

c) Um zu zeigen, dass die Kontextfreiheit des Schnittes unentscheidbar ist, zeigen wir , dass die Nicht-Kontextfreiheit des Schnittes unentscheidbar ist. Hiermit ist das folgende Problem gemeint: Gegeben Grammatiken G_1, G_2, entscheide ob $\mathcal{L}(G_1) \cap \mathcal{L}(G_2)$ nicht kontextfrei ist.

Angenommen die Nicht-Kontextfreiheit wäre entscheidbar, dann wäre auch die Kontextfreiheit entscheidbar – Die Klasse der Entscheidbaren Probleme ist abgeschlossen unter Komplement.

Die Reduktion f aus Teil a) ist auch eine Reduktion auf Nicht-Kontextfreiheit.

Wenn der Schnitt leer ist (also die gegebene PCP-Instanz keine Lösung hat), dann gilt $\mathcal{L}(G_1) \cap \mathcal{L}(G_2) = \varnothing$, und $\varnothing$ ist kontext-frei.

Es bleibt zu zeigen, dass wenn der Schnitt nicht leer ist (und die PCP eine Lösung hat), dass dann $\mathcal{L}(G_1) \cap \mathcal{L}(G_2)$ nicht kontextfrei ist. Hierzu kann das aus „Theoretische Informatik I" bekannte Pumping-Lemma für kontextfreie Sprachen verwendet werden

d) Wir reduzieren die Leerheit des Schnittes auf das Inklusionsproblem. (Wir haben in a) gezeigt, dass die Nicht-Leerheit des Schnittes unentscheidbar ist, damit ist auch das Komplementproblem, also die Leerheit des Schnittes, unentscheidbar.)

Hierfür ist wichtig, dass $\mathcal{L}(G_1)$ und $\mathcal{L}(G_2)$ sogenannte deterministische kontextfreie Sprachen sind: Es ist möglich, deterministische Pushdown-Automaten A_1, A_2 zu konstruieren mit $\mathcal{L}(A_1) = \mathcal{L}(G_1), \mathcal{L}(A_2) = \mathcal{L}(G_2)$. Die Klasse der deterministischen kontextfreien Sprachen ist abgeschlossen unter Komplement: Man erhält einen deterministischen Pushdown-Automaten für die Komplementsprache, in dem man die finalen Zustände invertiert. (Man erinnere sich daran, dass die Klasse der kontextfreie Sprachen nicht abgeschlossen unter Komplement ist!)

Man kann also eine Grammatik $\overline{G_2}$ konstruieren mit $\mathcal{L}\left(\overline{G_2}\right) = \overline{\mathcal{L}(G_2)} = \{0, 1, \$, a_1, \ldots, a_k\}^* \setminus \mathcal{L}(G_2)$ (indem man den invertierten Pushdown-Automaten in eine kontextfreie Grammatik transformiert).

Es gilt

$$\mathcal{L}(G_1) \cap \mathcal{L}(G_2) = \varnothing \quad \text{gdw.} \quad \mathcal{L}(G_1) \subseteq \mathcal{L}\left(\overline{G_2}\right).$$

Die Funktion g mit $g(G_1, G_2) = (G_1, \overline{G_2})$ ist also die gesuchte Reduktion.

e) Es lässt sich leicht eine Grammatik G_3 konstruieren mit $\mathcal{L}(G_3) = \mathcal{L}(G_1) \cup \mathcal{L}(G_2)$.

Es gilt

$$\mathcal{L}(G_1) \subseteq \mathcal{L}(G_2) \quad \text{gdw.} \quad \underbrace{\mathcal{L}(G_1) \cup \mathcal{L}(G_2)}_{\mathcal{L}(G_3)} = \mathcal{L}(G_2).$$

Also ist h mit $h(G_1, G_2) = (G_3, G_2)$ eine Reduktion von Sprachinklusion auf Sprachgleichheit. Sprachinklusion ist wie in Teil d) gezeigt unentscheidbar.

□

6.2 Bemerkung
Die Sprachen $\mathcal{L}(G_1), \mathcal{L}(G_2)$, die wir im Beweis des obigen Theorems verwendet haben sind – wie im Beweis von Teil d) angemerkt – deterministisch. Daher sind die Probleme aus den Teilen a) - d) auch für deterministische kontextfreie Grammatiken unentscheidbar.

Sprachgleichheit ist für deterministische kontextfreie entscheidbar, wie 2001 von Sénizergues bewiesen wurde [Sé01; Sé02]. Hierfür wurde ihm 2002 der Gödel-Preis verliehen.

6.3 Bemerkung
Die Grammatiken G_1, G_2 aus dem Beweis des obigen Theorems lassen sich auch verwenden, um zu zeigen, dass die folgenden Probleme unentscheidbar sind.

- Gegeben eine kontextfreie Grammatik G,
  - Ist G eindeutig?
  - Ist $\overline{\mathcal{L}(G)}$ kontext-frei?
  - Ist $\mathcal{L}(G)$ regulär?
  - Ist $\mathcal{L}(G)$ deterministisch kontextfrei?
  - Gilt $\mathcal{L}(G) = T^*$?
- Gegeben eine kontextfreie Grammatik G und eine reguläre Sprache R (z.B. repräsentiert durch einen NFA), gilt $\mathcal{L}(G) = R$?

Teil II.

Komplexitätstheorie

7. Zeit- und Platzkomplexität

In diesem Kapitel werden wir den **Zeit-** und **Platzverbrauch** von Turing-Maschinen definieren. Darauf aufbauend führen wir dann die **grundlegenden Komplexitätsklassen** ein, das heißt die Klassen der Probleme, die sich mit einem vorgegebenen Zeit- oder Platzverbrauch lösen lassen.

Davon abgeleitet definieren wir dann die **robusten Komplexitätsklassen**. Im Gegensatz zu den grundlegenden Klassen sind diese robust gegenüber Veränderungen am Berechnungsmodel. Beispielsweise ist die Klasse P der in Polynomialzeit lösbaren Probleme eine solche robuste Klasse. Es spielt keine Rolle, ob man zur Definition von P Einband- oder Mehrband-Turing-Maschinen verwendet, man erhält in beiden Fällen die selbe Klasse.

7.1 Bemerkung
In diesem Teil der Vorlesung werden wir – sofern nicht explizit anders spezifiziert – davon ausgehen, dass alle betrachteten Turing-Maschinen Entscheider sind, also dass alle Berechnungen zu allen Eingaben nach endlich vielen Schritten halten.

A) Zeitkomplexität

Zunächst wollen wir den **Zeitverbrauch** von Turing-Maschinen definieren und daraus abgeleitet die beiden grundlegenden **Zeitkomplexitätsklassen** $\mathrm{DTIME}_k(t)$ und $\mathrm{NTIME}_k(t)$ einführen.

7.2 Definition: Zeitverbrauch
Sei M eine Turing-Maschine (potentiell nicht-deterministisch, potentiell mit mehreren Bändern). Sei $x \in \Sigma^*$ eine Eingabe für M.

a) Der **Zeitverbrauch** (oder die Rechenzeit) von M für Eingabe x ist

$$\mathrm{Time}_M(x) = \max\{n \mid n \text{ ist Länge einer haltenden Berechnung von } M \text{ auf } x\} .$$

Die Länge einer Berechnung $c_0 \to c_1 \to \ldots$ ist hierbei der erste Index $n \in \mathbb{N}$, so dass c_n eine haltende Konfiguration ist. Man beachte, dass, falls M eine DTM ist, es genau eine Berechnung zu Eingabe x gibt.

Falls M auf einer Eingabe x eine nicht-haltende Berechnung hat, schreiben wir $\mathrm{Time}_M(x) = \infty$. Wenn M ein Entscheider ist, wird dies nie auftreten.

b) Für eine Zahl $n \in \mathbb{N}$, definieren wir die **Zeitkomplexität von** M als

$$\mathrm{Time}_M(n) = \max\{\mathrm{Time}_M(x) \mid |x| = n\} .$$

Wir messen also das Worst-Case-Verhalten von M auf Eingaben der Länge n.

c) Sei $t\colon \mathbb{N} \to \mathbb{N}$ eine Funktion. Wir sagen, dass M t-**zeitbeschränkt** ist, wenn $\mathrm{Time}_M(n) \leqslant t(n)$ für alle $n \in \mathbb{N}$.

Nun können wir die grundlegenden Zeitkomplexitätsklassen definieren. Zu jeder Zeitschranke, also einer Funktion $t\colon \mathbb{N} \to \mathbb{N}$, betrachtet man die Klasse der Probleme, die sich innerhalb dieser Zeitschranke lösen lassen.

7.3 Definition: Grundlegende Zeitkomplexitätsklassen
Sei $t\colon \mathbb{N} \to \mathbb{N}$ eine Zeitschranke. Wir definieren

$$\mathrm{DTIME}_k(t) = \{\mathcal{L}(M) \mid M \text{ ist eine } k\text{-Band DTM, ein Entscheider und } t\text{-zeitbeschränkt}\},$$
$$\mathrm{NTIME}_k(t) = \{\mathcal{L}(M) \mid M \text{ ist eine } k\text{-Band NTM, ein Entscheider und } t\text{-zeitbeschränkt}\}.$$

7.4 Bemerkung
Wir schreiben nur $\mathrm{DTIME}(t)$ bzw. $\mathrm{NTIME}(t)$ anstatt $\mathrm{DTIME}_1(t)$ bzw. $\mathrm{NTIME}_1(t)$.

Sei T eine Menge von Zeitschranken $t\colon \mathbb{N} \to \mathbb{N}$. Dann schreiben wir $\mathrm{DTIME}(T)$ für $\bigcup_{t\in T} \mathrm{DTIME}(t)$ (analog für NTIME und Mehrbandmaschinen).

Oft betrachten wir z.B. $\mathrm{DTIME}(\mathcal{O}(t))$, wenn wir multiplikative Konstanten ignorieren wollen.

Wir könnten nun diese grundlegenden Komplexitätsklassen untersuchen, d.h. wir könnten zum Beispiel die Klasse $\mathrm{DTIME}\left(n^2\right)$ der in quadratischer Zeit lösbaren Probleme betrachten. Wie bereits angemerkt sind diese Klassen nicht robust, es gilt z.B. $\mathrm{DTIME}_1(n) \neq \mathrm{DTIME}_2(n)$.

7.5 Bemerkung
Es ergibt wenig Sinn, sublineare Zeitschranken, also Funktionen $t\colon \mathbb{N} \to \mathbb{N}$ mit $t(n) < n$ für mindestens ein n, zu betrachten. Dies würde bedeutet, dass die Turing-Maschine nicht einmal in der Lage ist, ihre gesamte Eingabe zu lesen.

B) Platzkomplexität

Wir möchten analog zu DTIME und NTIME Platzkomplexitätsklassen einführen.

Im Gegensatz zum Zeitverbrauch ist es durchaus interessant, Probleme zu betrachten, die sich mit sublinearem Platzverbrauch lösen lassen. Hierbei wollen wir allerdings den Platzverbrauch der Eingabe nicht mitzählen, sondern nur den zusätzlichen Speicher, den die Maschine benötigt.

Daher betrachten wir hier Maschinen mit einem speziellen Eingabeband. Das Eingabeband ist **read-only**, die Eingabe darf also nicht verändert werden. Formal bedeutet read-only, dass die Transitionsfunktion die Form

$$\delta(q, (a_1, a_2, \ldots, a_k)) = (q', (a_1, b_1, \ldots, b_k), D)$$

für $a_i, b_i \in \Gamma, q, q' \in Q$ und $D \in \{R, L, N\}^k$ haben muss, falls das erste Band das Eingabeband ist (analog für eine nicht-deterministische Transitionsfunktion). Die anderen Bänder bezeichnen wir als **Arbeitsbänder**. Wir messen nur den Platzverbrauch der Maschine auf ihren Arbeitsbändern.

7.6 Definition: Platzverbrauch
Sei M eine Turing-Maschine (potentiell nicht-deterministisch, potentiell mit mehreren Arbeitsbändern) mit einem speziellen read-only Eingabeband. Sei $x \in \Sigma^*$ eine Eingabe für M.

a) Sei c eine Konfiguration, die während einer Berechnung von M zu Eingabe x auftritt. Der **Platzverbrauch von** M **in Konfiguration** c ist

$$\text{Space}_M(c) = \max\{|w| \mid w \text{ ist der Inhalt eines Arbeitsbandes in Konfiguration } c\} .$$

Hierbei zählen wir Blank-Symbole, die am Anfang und am Ende des Bandinhaltes auftreten können, nicht mit.

b) Der **Platzverbrauch von** M **zu Eingabe** x ist

$$\text{Space}_M(x) = \max\{\text{Space}_M(c) \mid c \text{ ist Konfiguration in einer Berechnung von } M \text{ zu Eingabe } x\} .$$

Falls der Platzverbrauch von M auf x unbeschränkt ist, schreiben wir $\text{Space}_M(x) = \infty$.

c) Für eine Zahl $n \in \mathbb{N}$, definieren wir die **Platzkomplexität von** M als

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

Wir messen also das Worst-Case-Verhalten von M auf Eingaben der Länge n.

d) Sei $s\colon \mathbb{N} \to \mathbb{N}$ eine Funktion. Wir sagen, dass M s-**platzbeschränkt** ist, wenn $\text{Space}_M(n) \leqslant s(n)$ für alle $n \in \mathbb{N}$.

7.7 Definition: Grundlegende Platzkomplexitätsklassen
Sei $s\colon \mathbb{N} \to \mathbb{N}$ eine Platzschranke. Wir definieren die grundlegenden Platzkomplexitätsklassen.

$$\text{DSPACE}_k(s) = \{\mathcal{L}(M) \mid M \text{ ist eine DTM mit } k \text{ Arbeitsbändern, ein Entscheider und } s\text{-platzbeschränkt}\} ,$$
$$\text{NSPACE}_k(s) = \{\mathcal{L}(M) \mid M \text{ ist eine NTM mit } k \text{ Arbeitsbändern, ein Entscheider und } s\text{-platzbeschränkt}\} ,$$

Wir verwenden die selbe vereinfachte Notation wie bei den Zeitkomplexitätsklassen.

Man beachte, dass der Zeitverbrauch den Platzverbrauch beschränkt: In jedem Schritt kann eine Maschine höchstens eine neue Zelle beschreiben. Daraus folgt zum einen, dass $\text{Space}_M(x) = \infty$ nicht auftreten kann, wenn M ein Entscheider ist. Zum anderen beweist es das folgende Lemma:

7.8 Lemma
Für jede Funktion $t\colon \mathbb{N} \to \mathbb{N}$ und jede Zahl k gilt

$$\text{DTIME}_{k+1}(t) \subseteq \text{DSPACE}_k(t),$$
$$\text{NTIME}_{k+1}(t) \subseteq \text{NSPACE}_k(t)\ .$$

Insbesondere gilt also

$$\text{DTIME}(t) \subseteq \text{DSPACE}(t), \qquad \text{NTIME}(t) \subseteq \text{NSPACE}(t),\ .$$

Der unterschiedliche Index k bzw. $k+1$ ergibt sich daraus, dass wir bei DTIME alle Bänder, bei DSPACE aber nur die Arbeitsbänder zählen.

7.9 Beispiel
Betrachte die Sprache der Wörter mit gleich vielen as und bs, also

$$\mathcal{L} = \left\{x \in \{a,b\}^* \;\middle|\; \text{Anzahl von } a \text{ und } b \text{ in } x \text{ ist gleich} \right\}.$$

Es gilt $\mathcal{L} \in \text{DSPACE}(\mathcal{O}(\log n))$, $\mathcal{L}$ lässt sich also deterministisch mit logarithmischem (insbesondere also sublinearem) Platzverbrauch lösen.

Wir konstruieren eine DTM M mit read-only-Eingabe und einem Arbeitsband, die $\mathcal{L}$ löst: Auf dem Arbeitsband speichert M einen Zähler. Nun geht die Maschine die Eingabe x von links nach rechts durch:

- Für jedes Vorkommen von Buchstabe a wird der Zähler inkrementiert (+1),
- Für jedes Vorkommen von Buchstabe b wird der Zähler dekrementiert (-1).

Nachdem die Eingabe gelesen wurde akzeptiert die Maschine genau dann, wenn der Zähler Wert 0 hat.

Es ist klar, dass M tatsächlich $\mathcal{L}$ entscheidet.

Wenn der Zähler als Binärzahl gespeichert wird, hat M nur logarithmischen Platzverbrauch: Der Wert des Zähler ist durch $-n$ nach unten und n nach oben beschränkt, mit $n = |x|$. Es werden also $\lceil \log n \rceil + 1$ viele Bits (Zellen) benötigt, um den Zähler zu speichern.

C) Die robusten Komplexitätsklassen

7.10 Definition

Wir definieren nun die **robusten Komplexitätsklassen**.

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

Die Klasse P umfasst alle Probleme, die sich deterministisch in Polynomialzeit lösen lassen. Ein Problem $\mathcal{L}$ ist also in P, wenn es einen konstanten Exponenten gibt, so dass es sich in $\mathrm{DTIME}\left(\mathcal{O}\left(n^K\right)\right)$ lösen lässt. Wichtig ist, dass der Exponent konstant und damit unabhängig von der Länge der Eingabe ist.

7.11 Bemerkung

Man kann auch noch größere Klassen betrachten, zum Beispiel zu jeder Zahl $m \in \mathbb{N}, m > 0$ die Klassen mEXP und mEXPSPACE, die Klassen der Probleme, die sich mit m-fach exponentiellem Zeit-/Platzverbrauch lösen lassen. Beispielsweise ist 2EXP die Klasse der Probleme $\mathcal{L}$ für die es einen Exponenten k gibt, so dass $\mathcal{L}$ in $\mathrm{DTIME}\left(2^{2^{\mathcal{O}\left(n^k\right)}}\right)$ ist.

Die Klasse ELEMENTARY ist die Klasse der **elementaren** Probleme, also Probleme, die sich für eine Konstante m in m-fach exponentieller Zeit lösen lassen. Es gilt

$$\mathrm{ELEMENTARY} = \bigcup_{\substack{m\in\mathbb{N},\\ m>0}} \mathrm{mEXP}\ .$$

(Wir werden später sehen, dass es keine Rolle spielt, ob man in der Definition von ELEMENTARY deterministische oder nichtdeterministische Klassen, und ob man Zeit- oder Platzkomplexitätsklassen verwendet). Wichtig ist, dass auch hier die Zahl m konstant ist und nicht von der Eingabegröße abhängt.

Es gibt nicht-elementare Probleme, zum Beispiel Probleme, bei denen die Laufzeit eines jeden Entscheidungsverfahrens $f(n)$-fach exponentiell ist, wobei $f(n)$ eine Funktion in der Eingabegröße ist.

Im Allgemeinen sieht man P als die Klasse der **effizient lösbaren** Probleme an. Da für jedes solche Problem der Exponent konstant ist, kann man auch sehr große Instanzen noch mit erträglichem Zeitaufwand lösen.

In unseren Definitionen der grundlegenden Klassen wie z.B. DTIME_k haben wir die Anzahl der Bänder spezifiziert. Die robusten Komplexitätsklassen wie z.B. P haben wir dann allerdings über Ein-Band-TMs definiert. Das folgende Lemma setzt die Zeit- und Platzklassen für Ein- und Mehrband-Maschinen in Relation.

7.12 Lemma
Für jede Funktion $f\colon \mathbb{N} \to \mathbb{N}$ und jede Zahl $k \in \mathbb{N}$ gilt

$$\mathrm{DTIME}_k(f) \subseteq \mathrm{DTIME}_1(f \cdot f), \qquad \mathrm{DSPACE}_k(f) \subseteq \mathrm{DSPACE}_1(f),$$
$$\mathrm{NTIME}_k(f) \subseteq \mathrm{NTIME}_1(f \cdot f), \qquad \mathrm{NSPACE}_k(f) \subseteq \mathrm{NSPACE}_1(f).$$

Beweis:
Zu einer Maschine M mit k-Bändern lässt sich eine äquivalente TM M' mit nur einem Band konstruieren, siehe Lemma 1.8.

Pro Schritt von M muss M' einmal das gesamte Band durchgehen. Da M pro Schritt maximal eine Zelle auf jedem Band neu beschreiben kann, ist die Zeitschranke f auch eine obere Schranke für den Platzverbrauch von M. Die 1-Band-TM braucht also maximal $f(n)$ viele Schritte pro Schritt von M auf einer Eingabe der Länge n, und damit insgesamt $f(n) \cdot f(n)$ viele Schritte.

Um zu sehen, dass eine Platzschranke f für M auch eine Platzschranke für M' ist, beobachte man, dass der Inhalt des einen Bandes von M' so lange ist wie der längste Inhalt eines Bandes von M.

In Lemma 1.8 haben wir 1-Band-TMs betrachtet, mit einem analogen Beweis kann man allerdings auch das Lemma für Maschinen mit Arbeitsband zeigen. □

7.13 Korollar
Die Definitionen der robusten Komplexitätsklassen sind unabhängig von der Anzahl der verwendeten Bänder, z.B. gilt

$$\mathsf{P} = \bigcup_{k' \in \mathbb{N}} \bigcup_{k \in \mathbb{N}} \mathsf{DTIME}_{k'}\left(\mathcal{O}\left(n^k\right)\right),$$
$$\mathsf{L} = \bigcup_{k' \in \mathbb{N}} \mathsf{DSPACE}_{k'}(\mathcal{O}(\log n)).$$

D) Komplementklassen

Wir wollen auch Komplementklassen kennen lernen.

Zunächst erinnern wir uns daran, dass zu einem Problem (also einer Sprache) $\mathcal{L} \subseteq \Sigma^*$ das Komplementproblem als

$$\overline{\mathcal{L}} = \Sigma^* \setminus \mathcal{L}$$

definiert war. Dies bedeutet, dass die Ja- und die Nein-Instanzen vertauscht werden.

7.14 Definition
Sei $\mathcal{C}$ eine Klasse von Problemen. Dann ist die **Komplementklasse** von $\mathcal{C}$

$$\mathsf{co}\mathcal{C} = \left\{\overline{\mathcal{L}} \;\middle|\; \mathcal{L} \in \mathcal{C}\right\}$$

die Klasse aller Komplementprobleme zu Problemen in $\mathcal{C}$.

Man beachte, dass co$\mathcal{C}$ nicht das Komplement von $\mathcal{C}$ ist, sondern die Komplemente der Probleme in $\mathcal{C}$ beinhaltet.

7.15 Beispiel
Sei

$$\mathsf{UNSAT} = \{F \text{ eine Boolsche Formel} \mid F \text{ nicht erfüllbar}\}.$$

Dann ist UNSAT $\in$ coNP, da $\overline{\mathsf{UNSAT}} = \mathsf{SAT} \in \mathsf{NP}$.

Das Ziel dieser Vorlesung sind es

1. Die definierten Komplexitätsklassen zu verstehen, insbesondere welche Probleme sie enthalten und welche Form die zugehörigen Algorithmen haben, und
2. die Zusammenhänge zwischen den Komplexitätsklassen zu verstehen.

Ein einfaches Resultat zu Punkt 2 ist, dass die deterministischen Klassen mit ihren Komplementklassen übereinstimmen.

7.16 Lemma
Für jede Funktion $f\colon \mathbb{N} \to \mathbb{N}$ und alle $k \in \mathbb{N}$ gilt

$$\mathrm{DSPACE}_k(f) = \mathrm{coDSPACE}_k(f), \qquad \mathrm{DTIME}_k(f) = \mathrm{coDTIME}_k(f)\,.$$

Beweis:
Ein deterministischer Entscheider für ein Problem wird zu einem deterministischen Entscheider für das Komplementproblem, in dem man die Rollen von q_{acc} und q_{rej} vertauscht. Die resultierende Maschine hat den gleichen Zeit- und Platzverbrauch. □

7.17 Korollar
Es gilt

$$\mathrm{L} = \mathrm{coL}, \qquad \mathrm{PSPACE} = \mathrm{coPSPACE}, \qquad \mathrm{P} = \mathrm{coP}. \qquad .$$

7.18 Definition
Eine Komplexitätsklasse $\mathcal{C}$ heißt abgeschlossen unter Komplement, falls für alle $L \in \mathcal{C}$ gilt dass $\overline{L} \in \mathcal{C}$.

7.19 Behauptung

$$\begin{aligned} \mathcal{C} = \mathrm{co}\mathcal{C} \quad &\text{gdw.} \quad \mathcal{C} \text{ abgeschlossen unter Komplement} \\ &\text{gdw.} \quad \overline{\mathcal{C}} \text{ abgeschlossen unter Komplement} \end{aligned}$$

Ob die nichtdeterministische Komplexitätsklassen abgeschlossen unter Komplement sind, also z.B. $\mathrm{NL} \stackrel{?}{=} \mathrm{coNL}, \mathrm{NPSPACE} \stackrel{?}{=} \mathrm{coNPSPACE}, \mathrm{NP} \stackrel{?}{=} \mathrm{coNP}$, gilt, ist jeweils eine nicht-triviale Fragestellung. Wir werden uns später mit dieser Frage beschäftigen und feststellen, dass die ersten beiden Gleichheiten gelten. Das Problem $\mathrm{NP} \stackrel{?}{=} \mathrm{coNP}$ ist offen, man glaubt allerdings, dass die beiden Klassen unterschiedlich sind.

Weitere Resultate zu Punkt 2 sind die folgenden Inklusionen innerhalb der Komplexitätsklassen.

7.20 Lemma
Für jede Funktion $f\colon \mathbb{N} \to \mathbb{N}$ gilt

1. $\mathrm{DTIME}(f(n)) \subseteq \mathrm{NTIME}(f(n))$,
2. $\mathrm{DSPACE}(f(n)) \subseteq \mathrm{NSPACE}(f(n))$,

3. DTIME($f(n)$) ⊆ DSPACE($f(n)$) und NTIME($f(n)$) ⊆ NSPACE($f(n)$),
4. DSPACE($f(n)$) ⊆ DTIME$\left(2^{\mathcal{O}(f(n))}\right)$ und NSPACE($f(n)$) ⊆ NTIME$\left(2^{\mathcal{O}(f(n))}\right)$

Beweis:

Zu 1. und 2. Jede DTM kann auch als NTM mit dem gleichen Platz- und Zeitverbrauch gesehen werden.

Zu 3. Da wir in jedem Berechnungsschritt nur eine Zelle besuchen können, kann die Turing-Maschine in $f(n)$ Berechnungsschritten auch nur höchstens $f(n)$ verschiedene Zellen besuchen und beschreiben.

Zu 4. Sei $M \in$ DSPACE($f(n)$) ein Entscheider, welcher $f(n)$ Platz auf dem Arbeitsband benötigt. Da M ein Entscheider ist, muss er auf jeder Eingabe und jeder Berechnung halten. Nehmen wir der Einfachheit halber erstmal an, dass M deterministisch ist. Für eine Berechnung von M bedeutet dies, dass es für jede Konfiguration genau eine eindeutige Nachfolger-Konfiguration hat. Falls während einer Berechnung eine Konfiguration c zum zweiten Mal erreicht wird, bedeutet das,

- dass es erstens eine Schleife von c nach c in der Konfigurationsfolge gibt und
- zweitens, dass diese Schleife nicht mehr verlassen werden kann, da sich M beim zweiten Betreten von c so verhalten muss wie beim ersten Betreten.

Daraus folgt, dass in keiner Berechnung eines (deterministischen) Entscheiders eine Konfiguration zweimal vorkommen kann. Also ist die maximale Länge einer Berechnung durch die Anzahl der möglichen Konfigurationen beschränkt. Da eine Konfiguration aus drei Komponenten, nämlich dem Zustand der TM, dem $f(n)$-beschränkten Bandinhalt und der Kopfposition auf dem Band, besteht, gibt es maximal

$$|Q| * 2^{f(n)} * f(n) = 2^{\mathcal{O}(f(n))}$$

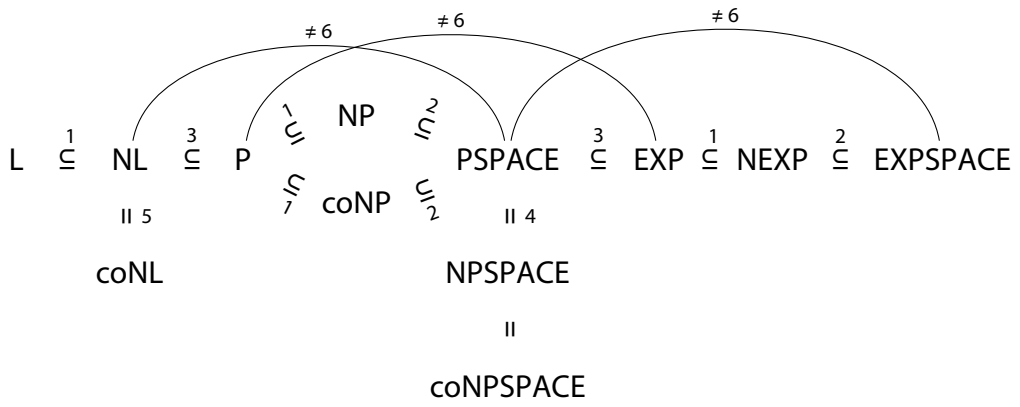
viele unterschiedliche Konfigurationen von M.

Für nicht-deterministische Entscheider M ist der Beweis analog. Hier fordern wir, dass M auf jeder möglichen Berechnung zu der Eingabe hält. Falls aber in einer Berechnung eine Konfiguration c zum zweiten Mal gesehen wird, kann sich M immer wieder wie beim ersten Betreten von c verhalten und somit in eine nicht-haltende Schleife von c nach c laufen.

□

8. Eine Landkarte der Komplexitätstheorie

Unser Studium der Relationen zwischen den Klassen wird schlussendlich folgendes Bild ergeben.



1. Wie bereits gezeigt ist Nichtdeterminismus mächtiger als Determinismus. Es gilt also $L \subseteq NL$, $P \subseteq NP$ und $EXP \subseteq NEXP$. Analog kann man $P \subseteq coNP$ beweisen.

2. Wir haben bereits gesehen, dass jede Zeitschranke auch eine Platzschranke ist. Dieses Resultat lässt sich erweitern, um zusätzlich sogar zu zeigen, dass sich nicht-deterministische Maschinen mit Zeitverbrauch t durch deterministische Maschinen mit Platzverbrauch t simulieren lassen.

3. Eine Maschine mit Platzschranke t lässt sich – selbst wenn sie nicht-deterministisch ist – durch eine deterministische Maschine mit Zeitschranke 2^t simulieren.

4. Der **Satz von Savitch** zeigt, dass $NSPACE(f) \subseteq DSPACE(f \cdot f)$. Daraus folgt $PSPACE = NPSPACE$, und damit, da $PSPACE = coPSPACE$ gilt, auch $NPSPACE = coNPSPACE$. Der Satz zeigt jedoch nicht $L = NL$, ob dies gilt, ist nach wie vor offen.

5. Der **Satz von Immerman und Szelepcsényi** zeigt, dass $NL = coNL$ gilt.

6. Die **Hierarchie-Resultate für Zeit und Platz** zeigen, dass man mit exponentiell mehr Zeit bzw. Platz echt mehr Probleme lösen kann. Daher gilt $NL \subsetneq PSPACE$, $P \subsetneq EXP$ und $PSPACE \subsetneq EXPSPACE$.

Ob die anderen Inklusionen echte Inklusionen oder Gleichheiten sind, ist bislang offen. Insbesondere ist das berühmte $P \stackrel{?}{=} NP$-Problem sowie $NP \stackrel{?}{=} coNP$ ungelöst. Man vermutet, dass alle Inklusionen strikt sind.

9. L und NL

Nun wollen wir die robusten Komplexitätsklassen im Detail untersuchen, beginnend mit L und NL. Wir wollen Sprachen (Probleme) kennen lernen, die sich mit logarithmisch viel Platz (nichtdeterministisch bzw. deterministisch) lösen lassen. Intuitiv, handeln die Sprachen in L vom Arithmetik und die Probleme in NL von Pfaden. Ob L = NL gilt, ist offen, aber wir versuchen die beiden Klassen voneinander abzugrenzen. Hierzu werden wir Sprachen in NL identifizieren, von denen man glaubt, dass sie nicht in L liegen. Diese Sprachen werden durch den Begriff der **Vollständigkeit** für die Klasse NL charakterisiert. Eine Sprachen ist vollständig für eine Komplexitätsklasse, wenn sie erstens in der Klasse enthalten ist (mit den gegebenen Ressourcen gelöst werden kann) und zweitens **hart** für die Klasse ist.

A) Reduktionen und Vollständigkeit

Formal nennen wir eine Sprache NL-**hart** oder NL-**schwer**, wenn sie mindestens so schwer ist wie jede andere Sprache in NL.

Wenn man zeigen könnte, dass eine dieser Sprachen nicht in L ist, wären damit alle diese Sprachen nicht in L. Falls eine dieser Sprachen in L liegt, liegen sie alle in L, und damit würde L = NL gelten.

Im Folgenden lernen wir Techniken kennen, um Sprachen als schwer nachzuweisen. Was bedeutet es, zu zeigen, dass eine Sprache mindestens so schwer ist, wie jede andere Sprache in NL? Um dies formal zu definieren möchten wir wieder Many-One-Reduktionen nutzen.

Die Many-One-Reduktionen, die wir im Teil der Vorlesung zu Entscheidbarkeit kennen gelernt haben, sind zu mächtig. Wir müssen vermeiden, dass die Reduktion (also die Turing-Maschine, welche die Reduktion berechnet) bereits einen Teil der Berechnung der Lösung des Problems vornimmt.

9.1 Definition
Sei R eine Klasse von Funktionen. Eine Sprache (oder Problem) $A \subseteq \Sigma_1^*$ heißt R-**many-one-reduzierbar** auf eine Sprache $B \subseteq \Sigma_2^*$, falls es eine Funktion $f \in \Sigma_1^* \to \Sigma_2^*$ mit $f \in R$ gibt, so dass für alle $x \in \Sigma_1^*$ gilt:

$$x \in A \quad \text{gdw.} \quad f(x) \in B\,.$$

Wir nennen f die **Reduktion** und schreiben $A \leqslant_m^R B$.

9.2 Definition
Sei $\mathcal{C}$ eine Komplexitätsklasse, R eine Menge von Funktionen und B eine Sprache.

a) Die Sprache B heißt $\mathcal{C}$-**schwer bezüglich** R-**many-one Reduktionen** (oder $\mathcal{C}$-**hart bezüglich** R-**many-one Reduktionen**) falls sich alle $A \in \mathcal{C}$ mit mit R-many-one-Reduktionen auf B reduzieren lassen:

$$\forall A \in \mathcal{C}\colon A \leqslant_m^R B\,.$$

Intuitiv bedeutet dies, dass B mindestens so schwer wie jede Sprache in $\mathcal{C}$ ist.

b) B heißt $\mathcal{C}$-**vollständig bezüglich** R-**many-one Reduktionen** falls

- B in $\mathcal{C}$ liegt („Membership“) und
- B $\mathcal{C}$-schwer bezüglich R-many-one Reduktionen ist („Hardness“).

Intuitiv bedeutet dies, dass B das schwerste Problem in $\mathcal{C}$ ist.

Wenn $B \in \mathcal{C}$ ist, sagen wir auch, dass $\mathcal{C}$ eine **obere Schranke** für die Härte von B ist. Wenn B $\mathcal{C}$-schwer ist, dann ist $\mathcal{C}$ eine **untere Schranke** für die Härte von B.

Damit Reduktionen nützlich sind, sollten sie zwei Eigenschaften erfüllen.

(1) Wenn wir zwei Komplexitätsklassen vergleichen, gibt es eine, von der wir vermuten, dass sie die mächtigere der beiden Klassen ist. Die Reduktionen, die wir betrachten, sollten schwächer sein, als diese Klasse. Ansonsten kann ein signifikanter Teil der Berechnung der Sprache, welche wir reduzieren wollen, bereits durch die Reduktion berechnet werden.

Für eine Komplexitätsklasse $\mathcal{C}$, die wir betrachten, sollte gelten: Wenn Problem A auf Problem B R-many-one-reduzierbar ist, und $B \in \mathcal{C}$ gilt, dann sollte auch $A \in \mathcal{C}$ folgen.

Anders formuliert: Die Komplexitätsklasse $\mathcal{C}$ sollte **abgeschlossen unter** R-**Many-One-Reduktionen** sein.

(2) Reduzierbarkeit sollte eine **transitive** Relation sein.

Insbesondere sollte, wenn A eine $\mathcal{C}$-schwere Sprache ist und $A \leqslant_m^R B$ gilt, auch B $\mathcal{C}$-schwer sein.

9.3 Bemerkung
Die Reduktionen, die wir im zweiten Teil der Vorlesung betrachtet haben, waren R-Many-One-Reduktionen für R gleich der Klasse der totalen berechenbaren Funktionen.

In der Komplexitätstheorie werden insbesondere zwei Klassen von Funktionen R verwendet:

- die Reduktionen, die in Polynomialzeit berechenbar sind ($\leqslant_m^{poly}$) und
- die Reduktionen, die mit logarithmischem Platz berechenbar sind ($\leqslant_m^{log}$).

9.4 Definition
Eine Funktion $f\colon \Sigma_1^* \to \Sigma_2^*$ ist **logspace-berechenbar**, wenn es eine deterministischen Turing-Maschine mit speziellem Ein- und Ausgabeband gibt, wobei

- das Eingabeband mit Alphabet Σ_1 read-only ist,
- das Ausgabeband mit Alphabet Σ_2 write-only ist (sobald ein Symbol geschrieben ist, bewegt sich der Kopf nach rechts) und
- das Arbeitsband mit Alphabet Γ read-write ist.

Des Weiteren, ist die Turing-Maschine

- total,
- hält zu jeder Eingabe $x \in \Sigma_1^*$ nach endlich vielen Schritten und hat $f(x) \in \Sigma_2^*$ auf das Ausgabeband geschrieben und
- der Speicherverbrauch ist durch $\mathcal{O}(\log n)$ beschränkt.

In der Literatur heißt diese DTM auch **Logspace-Transducer**.

Analog ist eine Funktion $f\colon \Sigma_1^* \to \Sigma_2^*$ **in Polynomialzeit berechenbar**, wenn es eine deterministischen Turing-Maschine wie im obigen Fall gibt, die allerdings $\mathcal{O}\left(n^k\right)$-zeitbeschränkt für eine von der Eingabegröße unabhängige Konstante k ist (anstatt $\mathcal{O}(\log n)$-platzbeschränkt zu sein).

9.5 Bemerkung
Man beachte, dass wir den Platzverbrauch auf Ein- und Ausgabeband nicht mitzählen, allerdings haben wir durch die read-only- bzw. write-only-Anforderung sichergestellt, dass der Logspace-Transducer diese Bänder nicht zum Rechnen zweckentfremden kann.

9.6 Definition
Eine Sprache $A \subseteq \Sigma_1^*$ heißt **logspace-(many-one-)reduzierbar** auf eine Sprache $B \subseteq \Sigma_2^*$, wenn A R-many-one-reduzierbar auf B ist mit R gleich der Klasse der logspace-berechenbaren Funktionen. Wir schreiben $A \leqslant_m^{log} B$.

Durch Einsetzen der Definition erhalten wir die folgende explizitere Definition: Eine Sprache $A \subseteq \Sigma_1^*$ heißt **logspace-(many-one-)reduzierbar** auf eine Sprache $B \subseteq \Sigma_2^*$, wenn es eine logspace-berechenbare Funktion $f\colon \Sigma_1^* \to \Sigma_2^*$ gibt, so dass für alle $x \in \Sigma^*$ gilt:

$$x \in A \quad \text{gdw.} \quad f(x) \in B$$

Eine Sprache $A \subseteq \Sigma_1^*$ heißt **in Polynomialzeit reduzierbar (many-one-)reduzierbar** oder **polynomiell reduzierbar** auf eine Sprache $B \subseteq \Sigma_2^*$, wenn A R-many-one-reduzierbar auf B ist mit R gleich der Klasse der in Polynomialzeit berechenbaren Funktionen. Wir schreiben $A \leqslant_m^{poly} B$.

Intuitiv sind die logspace-Reduktionen die Reduktion, die zur Klasse L korrespondieren, und die polynomiellen Reduktion die Reduktionen, die zur Klasse P korrespondieren.

Wir werden später sehen, dass L $\subseteq$ P und sogar NL $\subseteq$ P gelten. Dementsprechend ist jede logspace-Reduktion auch eine polynomielle Reduktion. Zu zeigen, dass etwas logspace-reduzierbar ist, ist eine stärkere Aussage, als zu zeigen, dass es in Polynomialzeit reduzierbar ist.

9.7 Lemma
Falls $A \leqslant_m^{log} B$, dann auch $A \leqslant_m^{poly} B$.

Es ist allerdings nicht bekannt, ob logspace-Reduktionen **echt stärker** als polynomielle Reduktionen sind.

Wenn wir die Klassen L und NL untersuchen möchten, ergibt es wenig Sinn, Polynomialzeit-Reduktionen zu betrachten. Wie oben in Punkt (1) erläutert sind die Reduktionen mächtiger als die Klassen und erlauben es damit, die Berechnung der Lösung des zu reduzierenden Problems in die Reduktion zu verlagern. Wir werden uns daher zunächst auf die logspace-Reduktionen konzentrieren. Polynomialzeit-Reduktionen haben aber im Wesentlichen dieselben prinzipiellen Eigenschaften wie z.B. Transitivität.

Wir beweisen nun, dass logspace-Reduzierbarkeit eine transitive Relation ist, also Punkt (2) oben erfüllt.

9.8 Lemma
Es seien $f : \Sigma_1^* \to \Sigma_2^*$ und $g : \Sigma_2^* \to \Sigma_3^*$ logspace-berechenbare Funktionen. Dann ist auch ihre Verkettung $g \circ f$: $\Sigma_1^* \to \Sigma_3^*$ eine logspace-berechenbare Funktion.

Insbesondere folgt aus $A \leqslant_m^{log} B$ und $B \leqslant_m^{log} C$ auch $A \leqslant_m^{log} C$.

Beweis:
Es seien M_f und M_g die Turing-Maschinen, die die Berechnung von f und g in logarithmischem Platzverbrauch berechnen. Sei $x \in \Sigma_1^*$ eine Eingabe.

Idee: Berechne $f(x)$ durch Simulation von M_f, verwende danach M_g auf Eingabe $f(x)$ zur Berechnung von $g(f(x))$.

Problem: Wir haben bei logspace-berechenbaren Funktionen nur den Platzverbrauch auf den Arbeitsbändern beschränkt. Die Ausgabe $f(x)$ von M_f kann mehr als logarithmisch viel

Platz brauchen, und ist damit zu groß, um auf einem Arbeitsband zwischengespeichert werden zu können.

Lösung: Berechne $f(x)$ bitweise on-demand und verwende den Speicherplatz wieder.

Wichtig hierfür ist,

- dass die Ausgabe $f(x)$ höchstens polynomiell groß ist und
- dass jede Zelle von $f(x)$ logspace-berechenbar ist.

Behauptung: $f(x)$ **ist höchstens polynomiell groß**

Eine Konfiguration von M_f zu Eingabe x ist gegeben durch

- Kontrollzustand,
- Eingabe x,
- Inhalt der Arbeitsbänder,
- Inhalt des Ausgabebands,
- Kopfpositionen auf den Bändern.

Wir beobachten, dass zum Einen das Eingabeband read-only ist und sich während der Berechnung nicht verändert; wir brauchen diese also nicht weiter betrachten. Zum anderen ist das Ausgabeband write-only, M_f darf sein Verhalten also nicht davon abhängig machen, was bereits auf das Ausgabeband geschrieben wurde. Wenn wir nun untersuchen wollen, nach wie vielen Schritten M_f hält, können wir sowohl den Inhalt der Ausgabe als auch die Kopfposition in der Ausgabe vernachlässigen.

Essentiell für eine Konfiguration ist der Inhalt der Arbeitsbänder. Da M_f logarithmischen Platzverbrauch hat, gibt es Konstanten d', d'', so dass der Inhalt jeden Arbeitsbandes durch $d' \cdot \log n + d''$ beschränkt ist.

Wir haben für jede Zelle des Arbeitsbandes nur $|\Gamma|$ viele Möglichkeiten, es gibt also

$$|\Gamma|^{d' \cdot \log n + d''}$$

viele Möglichkeiten für den Bandinhalt.

Insgesamt gibt es höchstens

$$\underbrace{|Q|}_{\text{Kontrollzustände}} \cdot \underbrace{k \cdot |\Gamma|^{d' \cdot \log n + d''}}_{\text{Inhalt der Arbeitsbänder}} \cdot \underbrace{n}_{\text{Kopfposition in der Eingabe}} \cdot \underbrace{k \cdot d' \cdot \log n + d''}_{\text{Kopfpositonen in den Arbeitsbändern}}$$

viele Konfigurationen zu Eingabe x, wobei hier k die Anzahl der Bänder ist.

Wichtig ist, das $|Q|, |\Gamma|, k, d', d''$ Konstanten, also unabhängig von der Eingabe x, sind. Des Weiteren können wir $|\Gamma|^{d' \cdot \log n + d''}$ umschreiben zu

$$2^{\log|\Gamma| \cdot d' \cdot \log n + d''} = \left(2^{d' \cdot \log n} \cdot 2^{d''}\right)^{\log|\Gamma|} = \left(n^{d'} \cdot 2^{d''}\right)^{\log|\Gamma|}.$$

Dieser Ausdruck ist in $\mathcal{O}\left(n^d\right)$ für eine geeignete Konstante d, also polynomiell.

Angenommen M_f würde eine Konfiguration während der Berechnung wiederholen. Dies würde bedeuten, dass M_f in eine Schleife läuft. (M_f ist deterministisch und wird daher diese Konfiguration wieder und wieder besuchen!) Wir erhalten einen Widerspruch zur Annahme, dass M_f für alle Eingaben nach endlich vielen Schritten hält, also insbesondere für x.

M_f hält also nach höchstens polynomiell vielen Schritten. Da pro Schritt maximal eine Zelle der Ausgabe geschrieben werden kann, ist damit auch die Länge der Ausgabe durch ein Polynom beschränkt.

Behauptung: Für jede Zahl i ist die Stelle $(f(x))_i$ logspace-berechenbar

Wir konstruieren eine Turing-Maschine M_i, die die i-te Zelle $(f(x))_i$ von $f(x)$ berechnet.

M_i verhält sich zunächst wie M_f. Zusätzlich hält sich M_i einen Zähler *count* (in Binärkodierung), der initial mit i belegt ist. Solange $count > 0$ gilt, wird jedes Mal wenn M_f eine Zelle der Ausgabe schreiben möchte, diese Ausgabe verworfen, aber der Zähler dekrementiert ($count - -$). Wenn $count = 0$ gilt, wird als nächstes M_f die i-te Zelle der Ausgabe schreiben. M_i schreibt diese Zelle und akzeptiert.

Falls M_f hält, bevor die i-te Zelle geschrieben wurde, schreibt M_i ein Leerzeichen als Ausgabe und hält.

Beweis der eigentlichen Aussage

Wir konstruieren nun , eine Maschine, die $g \circ f$ mit logarithmischen Platzverbrauch. Die Idee ist, dass wir M_g auf Eingabe $f(x)$ simulieren. Wir stellen uns vor, dass $M_{g \circ f}$ ein Band hätte, auf dem $f(x)$ steht.

Dies ist aus den bereits erläuterten Gründen nicht wirklich der Fall. In Wirklichkeit hält sich $M_{g \circ f}$ einen Zähler, in welcher Zelle dieses imaginären Bandes M_g ist. Wenn immer M_g auf die i-te Zelle von $f(x)$ zugreifen möchte, berechnen wir diesen Wert.

Wenn $M_{g \circ f}$ schreibt die selbe Ausgabe und akzeptiert wie die Simulation von M_g auf $f(x)$. Also berechnet $M_{g \circ f}$ in der Tat $g(f(x))$.

Gemäß unserer obigen Diskussion ist der Wert des Zählers i durch $\mathcal{O}\left(n^d\right)$ beschränkt, wenn wir ihn Binär speichern, benötigen wir also nur $\mathcal{O}\left(\log(n^d)\right) = \mathcal{O}(d \cdot \log n) = \mathcal{O}(\log n)$ viele Zellen. Die Werte $(f(x))_i$ können mit logarithmischem Platz berechnet werden, und da sie nur

eine einzige Zelle belegen, können wir sie auch jeweils speichern. Wichtig ist, dass wir den Speicherplatz für die aktuelle Zelle und die entsprechende Berechnung wiederverwenden konnen. Insgesamt erhalten wir, dass $M_{g \circ f}$ nur logarithmisch viel Platz braucht. □

Wir wollten dass unsere Reduktionen nicht zu mächtig sind (Punkt (1) oben). Das folgende Lemma zeigt, dass für die Klasse L die logspace-Reduktionen bereits zu mächtig sind.

9.9 Lemma
Sei Σ ein endliches Alphabet.

a) Eins Sprache $\mathcal{L} \subseteq \Sigma^*$ ist in L genau dann, wenn $\mathcal{L} \leqslant_m^{log} \{1\}$.
Hier bezeichnet $\{1\}$ die Sprache $\{1\} \subseteq \{0,1\}^*$.

b) Jede Sprache $\mathcal{L} \subseteq \Sigma^*$ in L mit $\mathcal{L} \neq \varnothing$ und $\mathcal{L} \neq \Sigma^*$ ist bereits L-vollständig (bezüglich logspace-many-one-Reduktionen).

Beweis: Übungsaufgabe. □

Es ist also nur sinnvoll, Reduktion zu betrachten, die schwächer sind als die betrachtete Klasse.

Viele Klassen sind abgeschlossen unter logspace-Reduktionen

9.10 Lemma
Sei $A \leqslant_m^{log} B$. Wenn B in L bzw. NL bzw. P ist, dann ist auch A in L bzw. NL bzw. P.

Das folgende Lemma zeigt, dass sich schwere Probleme tatsächlich dazu eignen, die Klassen voneinander abzugrenzen. Sollte es möglich sein, ein hartes Problem in einer Klasse mit weniger Aufwand zu lösen, dann fallen die entsprechenden Klassen zusammen.

9.11 Lemma
Sei A eine Sprache.

a) Falls A NL-schwer bezüglich logspace-many-one-Reductions ist und $A \in$ L gilt, dann folgt NL = L.

b) Falls A P-schwer bezüglich logspace-many-one-Reductions ist und $A \in$ NL gilt, dann folgt NL = P.

B) PATH ist NL-vollständig

Die wichtigsten Probleme in der Klasse NL sind Pfadprobleme, also Probleme, bei denen es darum geht, die (Nicht-)Existenz von Pfaden in Graphen zu untersuchen.

Pfadexistenz (PATH)	
Gegeben:	Gerichteter Graph $G = (V, R)$, Quellknoten $s \in V$, Zielknoten $t \in V$
Entscheide:	Gibt es einen Pfad von s nach t in G?

Um dieses Problem als Wortproblem aufzufassen, müssen wir festlegen, wie ein gerichteter Graph G kodiert werden soll. Wir können davon ausgehen, dass die Knoten $V = \{1, \ldots, n\}$ durchnummeriert sind. Dies erlaubt es uns, einen Knoten i durch die Binärdarstellung bin(i) zu repräsentieren. Die Kanten im Graphen können wir auf verschiedene Arten kodieren (z.B. als Matrix oder Adjazenzliste). Wir gehen hier von einer Kodierung als Adjazenzliste aus, zu jedem Knoten i gibt es also einen Liste

$$\text{adj}_i = i \to j_1, j_2, \ldots, j_{k_i}$$

wobei $j_1, \ldots j_{k_i}$ die Knoten sind, zu denen i eine ausgehende Kante hat. Die Zahlen $i, j_1, \ldots j_{k_i}$ können wir wieder wie üblich binär kodieren.

Insgesamt kodieren wir dann einen Graphen G mit n Knoten als Wort

$$\text{adj}_1; \ldots; \text{adj}_n \,.$$

Im Folgenden werden wir einen Graphen mit seiner Kodierung identifizieren. Wir können PATH damit als Wortproblem

$$\text{PATH} = \{G\#s\#t \mid G \text{ kodiert einen Graphen, in dem es einen Pfad von } s \text{ nach } t \text{ gibt}\}$$

auffassen.

9.12 Lemma
PATH ist in NL.

Beweis:
Wir geben einen Algorithmus an, der PATH löst und sich als nicht-deterministische Turing-Maschine mit logarithmischen Platzverbrauch implementieren lässt.

Zunächst beobachtet man, dass, wenn es einen Pfad von s nach t gibt, es auch einen **einfachen** Pfad gibt, also einen Pfad, der keinen Knoten doppelt beinhaltet: Ein Pfad mit Wiederholungen lässt sich durch Entfernen der Schleifen in einen einfachen Pfad mit der selben Quelle

und dem selben Ziel transformieren. Es gibt daher, wenn es einen Pfad gibt, einen Pfad der Länge höchstens $n = |V|$, da Pfade mit Länge echt größer als n zwangsläufig Wiederholungen beinhalten.

Unser Algorithmus wird eine Sequenz von Knoten raten und verifizieren, dass es sich hierbei um einen validen Pfad von s nach t handelt. Da wir nur logarithmisch viel Speicher verwenden möchten, ist es uns nicht möglich, die komplette Sequenz zu speichern. Wir raten daher die Sequenz Knoten für Knoten, und vergessen immer alle außer die beiden aktuellsten Knoten. Dies erlaubt es uns, den Speicherplatz wiederzuverwenden.

Um sicherzustellen, dass wir nicht zu lange raten – wir möchten ein Entscheidungsverfahren, das garantiert terminiert - halten wir einen Zähler, den wir bei jedem geratenen Knoten inkrementieren. Wenn der Wert n überschreitet, wissen wir, dass der geratene Pfad nicht mehr einfach sein kann und brechen ab.

```
count = 0
current = s
while count < n do
    count + +
    Rate Knoten new ∈ {1, . . . , n}
    if new ist Nachfolger von current then
        if new = t then
            return true
        end if current = new
    else
        return false
    end if
end while
return false
```

Die Überprüfung, ob *new* Nachfolger von *current* ist, lässt sich durch Durchgehen der Adjazenzliste in der Eingabe durchführen.

Der Algorithmus ist korrekt: Wenn es einen einfachen Pfad von s nach t gibt, gibt es eine Berechnung des Algorithmus, in dem genau dieser Pfad geraten wird und der Algorithmus damit true zurückgibt. Wenn der Algorithmus eine Berechnung hat, die true zurückgibt, dann gibt es auch einen Pfad von s nach t, nämlich den in dieser Berechnung geratenen.

Es verbleibt zu argumentieren, dass der Algorithmus mit logarithmisch viel Speicherplatz implementiert werden kann.

Der Zähler *count* ist in seinem Wert durch n beschränkt. Zudem werden zwei Knoten *current* und *new* gebraucht, die ebenfalls als Zahl in $\{1, \ldots, n\}$ gespeichert werden können. Für die

Überprüfung, ob der neue Knoten Nachfolger des alten ist, werden gegebenenfalls weitere Zeiger in die Eingabe benötigt.

Wenn man all diese Zahlen in Binärdarstellung speichert, werden jeweils höchstens log n Bits (Zellen) benötigt. Nun beachte man, dass die Eingabe mindestens Länge n hat, da wir davon ausgehen können, dass jeder Knoten in $\{1, \ldots, n\}$ eine Adjazenzliste hat, die mindestens eine Zelle belegt. □

Wir haben nun bewiesen, dass PATH in NL lösbar ist. Als nächstes wollen wir PATH als erstes Problem, welches NL-vollständig ist, nachweisen.

Die Schwierigkeit hierbei ist zu zeigen, dass sich *jedes* Problem aus NL auf PATH reduzieren lässt. (Wir haben noch kein anderes Problem als vollständig nachgewiesen, welches wir reduzieren könnten.)

Sobald die Härte von PATH bewiesen ist, können wir die Härte anderer Problem durch die Reduktion von PATH beweisen. Andererseits können wir von anderen Problemen beweisen, dass sie in NL lösbar sind, indem wir sie auf PATH reduzieren

9.13 Theorem
PATH ist NL-vollständig (bezüglich logspace-many-one-Reduktionen).

Beweis:
Wir müssen zeigen, dass

1. PATH in NL liegt und
2. dass PATH NL-hart ist (bezüglich logspace-many-one-Reduktionen).

Den ersten Punkt haben wir bereits in Lemma 9.12 nachgewiesen.

Für den zweiten Teil, müssen wir jedes andere Problem in NL auf PATH reduzieren. Sei $\mathcal{L} \in$ NL ein solches Problem. Es gibt eine NTM M mit durch $\mathcal{O}(\log n)$-beschränkten Platzverbrauch, die $\mathcal{L}$ entscheidet, also $\mathcal{L}(M) = \mathcal{L}$.

Wir zeigen, dass es eine Funktion f_M gibt, die mit logarithmischem Platz berechenbar ist, mit: Für eine Eingabe x und den zugehörigen Funktionswert $f_M(x) = G\#s\#t$ gilt:

$$M \text{ akzeptiert } x \quad \text{gdw.} \quad \text{es gibt in } G \text{ einen Pfad von } s \text{ nach } t\,.$$

Hierbei ist G ein gerichtet Graph und s und t sind Knoten in G.

Der Graph G ist der Konfigurationsgraph von M zu Eingabe x. Seine Knoten sind Konfigurationen mit Eingabe x und einem Bandinhalt, der durch $\mathcal{O}(\log n)$ beschränkt ist. Für zwei

Konfigurationen c_1, c_2 beinhaltet der Graph eine Kante (c_1, c_2) genau dann, wenn c_2 ein mögliche Nachfolgekonfiguration von c_1 ist. Der Quellknoten s ist die Startkonfiguration von M, d.h. initialer Kontrollzustand, Eingabe x und leeres Arbeitsband.

Wir gehen o.B.d.A. davon aus, dass M eine eindeutige akzeptierende Konfiguration hat: Kontrollzustand q_{acc}, Eingabe x, ein leeres Arbeitsband und Kopfposition auf dem Eingabeband steht auf dem ersten Symbol der Eingabe. Diese Konfiguration ist der Zielknoten t. Zu jedem Problem in NL gibt es eine solche Maschine: Zu einer Maschine M', die die Annahme nicht erfüllt, können wir eine Maschine M, die die gleiche Sprache akzeptiert und im wesentlichen den gleichen Platz- und Zeitverbrauch hat, konstruieren, die am Ende der Berechnung vor den Akzeptieren den Bandinhalt löscht und den Kopf auf dem Eingabeband auf das erste Symbol der Eingabe bewegt. M akzeptiert x genau dann, wenn es eine akzeptierende Berechnung zu x gibt, also einen Pfad in G von s nach t.

Es verbleibt zu zeigen, dass f_M mit logarithmischem Platz berechnet werden kann. Die Schwierigkeit hierbei ist es, den Graphen auszugeben.

Jede Konfiguration wird dargestellt durch Kontrollzustand, Kopfpositionen auf beiden Bändern, und den Inhalt des Arbeitsbandes. Die Eingabe x bleibt während der Berechnung unverändert und muss nicht kodiert werden, lediglich die Kopfposition der Maschine im Eingabeband ist von Interesse.

Die Schwierigkeit bildet die Kodierung des Arbeitsbandes. Der Kontrollzustand benötigt nur eine Konstante Anzahl Zellen zur Kodierung, die Kopfpositionen lassen sich als Binärzahlen jeweils in $\log|x|$ Bits speichern. Da der Platzverbrauch von M durch $\mathcal{O}(\log n)$ beschränkt ist, gibt es eine Konstante d, so dass sich der Inhalt des Arbeitsbandes durch ein Wort der Länge $d \cdot \log|x|$ darstellen lässt.

Insgesamt erhalten wir Konstanten d', d'', so dass jede Konfiguration von M sich als ein Wort der Länge $d' \cdot \log|x| + d''$ darstellen lässt.

Die DTM für f_M zählt alle Wörter der Länge $d' \cdot \log|x| + d''$ auf und überprüft jeweils, ob Sie die valide Kodierung einer Konfiguration sind. Die Wörter, die diesen Test bestehen, werden ausgegeben.

Die Ausgabe der Kanten lässt sich ähnlich realisieren: Es werden Paare (c_1, c_2) von solchen Wörtern aufgezählt. Wenn beide valide Konfigurationen sind, und c_2 ein Nachfolger von c_1 gemäß der Transitionsrelation von M ist, wird die Kante ausgegeben.

Für die Ausgabe des Graphen müssen also maximal zwei Konfigurationen gleichzeitig gespeichert werden. Die Berechnung von f_M lässt sich mit einem Arbeitsband, dessen Platz durch $\mathcal{O}(\log n)$ beschränkt ist, umsetzen. □

Man kann das Pfadproblem sogar auf sogenannte **azyklische** Graphen, also auf Graphen, in denen es keine Kreise $c \to c_0 \to \ldots \to c_k \to c$ gibt, einschränken. Es bleibt NL-vollständig.

Pfadexistenz in azyklischen Graphen (ACYCLICPATH)	
Gegeben:	Gerichteter azyklischer Graph $G = (V, R)$, Knoten $s, t \in V$
Entscheide:	Gibt es einen Pfad von s nach t in G?

9.14 Lemma
Das Problem ACYCLICPATH ist NL-vollständig.

Bereits im Beweis, dass PATH NL-schwer ist, sieht man ein Indiz für dieses Lemma: Der Teil des Konfigurationsgraphen von M, der von der Initialkonfiguration aus erreichbar ist, muss azyklisch sein, da M sonst kein Entscheider ist. Da es aber nicht-azyklische Teile des Graphen, die nicht erreichbar sind, geben kann, ist dies noch kein formaler Beweis. Eine Reduktion von PATH auf ACYCLICPATH ist eine Übungsaufgabe.

C) 2SAT ist NL-vollständig

Wir interessieren uns insbesondere für die Komplexität von Problemen, deren Eingabe die „Theoretische Informatik I" bekannten endlichen Automaten bzw. die aus „Einführung in die Logik" bekannten aussagenlogischen Formeln sind. Hier wollen wir die Erfüllbarkeit von aussagenlogischen Formeln in konjunktiver Normalform betrachten.

Sei $x_0, x_1, x_2, \ldots$ eine abzählbar unendliche Menge von Aussagenvariablen. Ein **Literal** L ist von der Form x_i (positiv) oder $\neg x_i$ (negativ). Eine **Klausel** ist eine Disjunktion von Literalen,

$$C = L_1 \vee \ldots L_k \,.$$

Eine **aussagenlogische Formel in konjunktiver Normalform (CNF)** ist eine Konjunktion von Klauseln

$$F = C_1 \wedge \ldots \wedge C_n \,.$$

Belegungen und die Auswertung von Formeln sind dabei wie üblich definiert. Wir identifizieren $0 \mathrel{\widehat{=}} \mathit{false}$, $1 \mathrel{\widehat{=}} \mathit{true}$.

Erfüllbarkeit von aussagenlogischen Formeln in CNF (SAT)	
Gegeben:	Eine Formel F in CNF
Entscheide:	Ist F erfüllbar, d.h. gibt es eine Belegnung φ mit $\varphi(F) = \mathit{true}$?

SAT steht für *Satisfiablity*, also Erfüllbarkeit.

Man interessiert sich insbesondere für die Abhängigkeit der Härte dieses Problems von diversen Parametern, z.B. der Größe der Klauseln. Sei $k > 0$ eine natürliche Zahl. Eine Formel F ist in k-CNF, wenn sie in CNF ist, und jede Klausel aus höchstens k Literalen besteht.

Erfüllbarkeit von aussagenlogischen Formeln in k-CNF (kSAT)	
Gegeben:	Eine Formel F in k-CNF
Entscheide:	Ist F erfüllbar, d.h. gibt es eine Belegnung φ mit $\varphi(F) = true$?

Man beachte, dass k nicht Teil der Eingabe, sondern fixiert ist. Wir werden uns später mit SAT und kSAT für $k > 2$ beschäftigen. 1SAT ist trivial. In diesem Kapitel betrachten wir 2SAT.

9.15 Theorem
Das Problem 2SAT ist NL-vollständig.

Es sind zwei Dinge zu zeigen:

- „Membership": Wir zeigen zunächst, dass 2SAT in coNL enthalten ist, also dass das Komplementproblem $\overline{\text{2SAT}}$ in NL liegt. 2SAT $\in$ NL folgt dann mit dem Satz von Satz von Immerman und Szelepcsényi, welcher aussagt, dass NL = coNL.
- „Hardness": 2SAT ist coNL-schwer. Wir reduzieren $\overline{\text{ACYCLICPATH}}$. Genau wie ACYCLICPATH, ist auch $\overline{\text{ACYCLICPATH}}$ NL-schwer.

9.16 Lemma
2SAT ist in coNL.

Für eine gegebene 2CNF F konstruieren wir einen Graphen $G_F = (V_F, E_F)$ wie folgt:

- Es gibt für jede Variable x, die in F vorkommt zwei Knoten $x, \neg x$, also einen pro Literal.

$$V_F = \{x, \neg x \mid x \text{ ist Variable in } F\}$$

- Es gibt Kanten $\alpha \to \beta$ und $\neg\beta \to \neg\alpha$ falls $\neg\alpha \vee \beta$ eine Klauseln in F ist. Für Klauseln die aus nur einem Literal α bestehen, erhalten wir die Kante $\neg\alpha \to \alpha$.

$$E_F = \bigcup_{\substack{(\neg L_1 \vee L_2) \text{ is a} \\ \text{clause of } F}} \{(L_1, L_2), (\neg L_2, \neg L_1)\} \cup \bigcup_{\substack{(L) \text{ is a} \\ \text{clause of } F}} \{(\neg L, L)\}$$

Mit $\neg L$ ist hiermit das negierte Literal gemeint, also $\neg(x) = \neg x$ und $\neg(\neg x) = x$. Die Kanten von G_F korrespondieren zu Implikationen. Die Klausel $\neg L_1 \vee L_2$ ist in der Tat logisch äquivalent zu den Implikationen $L_1 \to L_2$ und $\neg L_2 \to \neg L_1$. Für Klauseln, die aus einem Literal bestehen, gilt

$$L \Leftrightarrow L \vee L \Leftrightarrow \neg L \to L \,.$$

Pfade in G_F entsprechen also ebenfalls Implikationen, denn Implikation ist transitiv.

Man beachte auch die folgende Symmetrie in G_F: Es gilt $L_1 \to L_2$ gdw. $\neg L_2 \to \neg L_1$.

Wir betrachten ein Beispiel für die Konstruktion

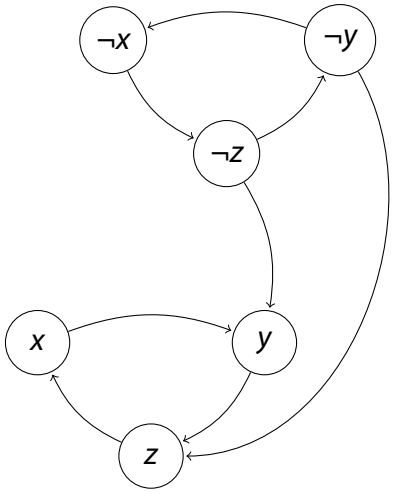
9.17 Beispiel
Sei

$$F = (\neg x \vee y) \wedge (\neg y \vee z) \wedge (x \vee \neg z) \wedge (z \vee y) \,.$$

Wir konstruieren G_F und erhalten

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

Die Folgende Abbildung stellt den Graphen dar.



Das Folgende Lemma ist wichtig, um die Korrektheit unserer Reduktion nachzuweisen.

9.18 Lemma
Eine 2CNF Formel F ist unerfüllbar genau dann, wenn es eine Variable x gibt, so dass es in G_F einen Pfad von x nach $\neg x$ und einen Pfad von $\neg x$ nach x gibt.

$$F \text{ unerfüllbar} \quad \text{gdw.} \quad \exists x\colon x \to_G^* \neg x, \neg x \to_G^* x$$

Beweis:
Angenommen, die beiden Pfade existieren, aber es gäbe dennoch eine erfüllende Belegung φ für F. O.b.d.A. nehmen wir $\varphi(x) = 1$ an.

Wir erhalten $\varphi(\neg x) = 0$. Da es einen Pfad von x nach $\neg x$ gibt, gibt es eine Kante $L \to L'$ auf dem Pfad mit $\varphi(L) = 1$ und $\varphi(L') = 0$. Dieser Kante entspricht die Klausel $\neg L \vee L'$ in der Formel. Diese Klausel wird ausgewertet zu

$$\varphi(\neg L \vee L') = \min\bigl(1 - \varphi(L), \varphi(L')\bigr) = 0.$$

Daher gilt $\varphi(F) = 0$, ein Widerspruch dazu, dass φ die Formel F erfüllt.

Der Fall $\varphi(x) = 0$ lässt sich ähnlich behandeln, hierbei muss der Pfad von $\neg x$ nach x betrachtet werden.

Für die andere Richtung nehmen wir an, dass es die Pfade nicht gibt, und konstruieren eine erfüllende Belegung φ. Dies erledigt der folgenden Algorithmus.

while es gibt noch ein Literal ohne Wahrheitswert **do**
  Wähle Literal L, so dass es keinen Pfad von L nach $\neg L$ gibt
  for Literal L', das von L aus erreichbar ist, also $L \to^* L'$ **do**
    Setze $\varphi(L') = 1$.
  end for
  for Literal L', von dem aus $\neg L$ erreichbar ist, also $L' \to^* \neg L$ **do**
    Setze $\varphi(L') = 0$.
  end for
end while

Wir müssen begründen, dass die resultierende Belegung φ eine wohldefinierte Belegung ist, die F erfüllt.

Zunächst beobachte, dass ein Literal L' und seine Negation $\neg L'$ im gleichen Durchlauf der While-Schleife belegt werden. Wenn es einen Pfad $L \to^* L'$ gibt, dann gibt es aufgrund der Symmetrie im Graphen auch einen Pfad $\neg L' \to^* \neg L$.

- Die resultierende Belegung weist jedem Literal einen Wahrheitswert zu: Sei L ein Literal, dem noch kein Wahrheitswert zugewiesen ist. Dann ist auch $\neg L$ noch nicht belegt. Falls wir L nicht auswählen können (weil es einen Pfad von L nach $\neg L$ gibt), dann können wir $\neg L$ auswählen. Falls es einen Pfad von $\neg L$ nach L gäbe, würden wir einen Widerspruch zur Annahme erhalten.

- Die resultierende Belegung ist wohldefiniert. Angenommen es gäbe ein Literal L', so dass L und $\neg L'$ den selben Wahrheitswert haben. Da die beiden im selben Durchlauf ihren Wert erhalten, sagen wir im Durchlauf, in dem Literal L gewählt wird, gibt es entweder Pfade von L sowohl nach L' als auch nach $\neg L'$, oder $\neg L$ ist sowohl von L' als auch $\neg L'$ erreichbar ist. Wir behandeln den ersten Fall, der zweite ist analog.

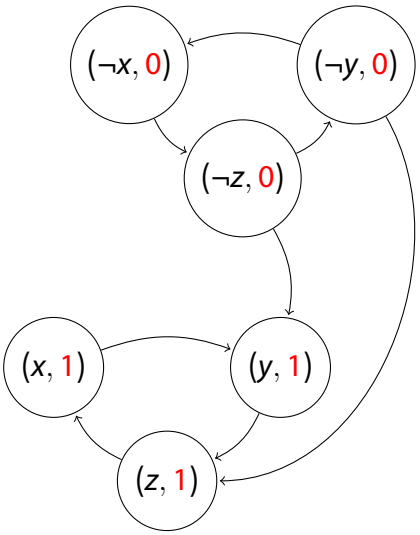
  Es gelte also $L \to^* L'$ und $L \to^* \neg L'$. Aufgrund der Symmetrie im Graphen gilt dann auch $\neg L' \to \neg L$ und $L' \to^* \neg L$. Wir erhalten $L \to^* L' \to \neg L$, ein Widerspruch zur Wahl von L im Algorithmus.

- Die resultierende Belegung erfüllt F. Wenn L ein Literal ist, das auf 1 gesetzt ist, dann gibt es kein von L aus erreichbares Literal, das auf 0 gesetzt ist. Alle im Graph kodierten Implikationen sind also erfüllt, damit werden alle Klauseln und auch F zu 1 ausgewertet.

□

9.19 Beispiel

Wir setzen das obige Beispiel fort und konstruieren eine erfüllende Belegung wie im Beweis.



Wir können das Lemma nun beweisen. Wie bereits gesagt beweisen wir, dass das Komplementproblem in NL ist.

Beweis:
Der folgende Algorithmus löst $\overline{\text{2SAT}}$ in NL.

Eingabe: 2CNF F
Ausgabe: true falls F unerfüllbar

```
Konstruierte G_F.
for Variable x in F do
    if G#x#¬x ∈ PATH und G#¬x#x ∈ PATH then
        return true
    end if
end for
return false
```

□

9.20 Lemma
2SAT is coNL-hart (bezüglich logspace-Reduktionen).

Beweis:
Wir reduzieren $\overline{\text{ACYCLICPATH}}$ auf 2SAT.

Sei $G\#s\#t$ eine $\overline{\text{ACYCLICPATH}}$ Instanz. Wir konstruieren eine 2-CNF formal F wie folgt: Die Variablen von F sind die Knoten des Graphen

$$Vars = \{x \mid x \text{ ist Knoten von } G\}$$

Für jede Kante $x \to y$ des Graphen G führen wir eine Klausel $\neg x \vee y$ ein. Darüber hinaus fügen wir die Klauseln s und $\neg t$ für Quell- und Zielknoten ein.

Es gilt:

F erfüllbar ist genau dann, wenn es keinen Pfad von s nach t in G gibt.

Der Beweis dieser Aussage sei der Leserin/dem Leser als Übungsaufgabe überlassen.

Die Konstruktion der Formel lässt sich mit logarithmischem Platz realisieren. □

D) NIM ist in L

Wir möchten uns nun einem Problem anschauen, welches in L liegt. **NIM** ist ein 2-Spieler Spiel, bei dem die Spieler abwechselnd Streichhölzer von einem Spielbrett wegnehmen. Die

Streichhölzer sind in mehrere Reihen aufgeteilt und während eines Zuges darf ein Spieler beliebig viele, aber mindestens eines, der Streichhölzer aus einer Reihe wegnehmen. Der Spieler, der das letzte Streichholz nimmt, gewinnt.

Folgendes Beispiel stellt einen möglichen Ablauf eines NIM-Spiels dar. Begonnen wir mit zwei Reihen mit jeweils zwei Streichhölzer und einer Reihe mit nur einem Streichholz. Es spielt Spieler P_1 gegen Spieler P_2. Spiele P_1 beginnt und am Ende nimmt Spieler P_1 auch das letzte Streichholz und gewinnt.

$$221 \xrightarrow{P_1} 220 \xrightarrow{P_2} 120 \xrightarrow{P_1} 110 \xrightarrow{P_2} 010 \xrightarrow{P_1} 000$$

1901 konnte der Mathematiker Charles Leonard Bouton zeigen, dass, abhängig von der Startkonfiguration auf dem Brett, einer der beiden Spieler eine Gewinnstrategie hat. Eine solche Strategie garantiert, dass der Spieler gewinnt, unabhängig davon welche Züge der Gegner macht. 1940 wurde der erste Computer für NIM entwickelt und 1951 konnte der Computer Nimrod den damaligen Wirtschaftsminister Ludwig Erhard auf der Berliner Industrieausstellung in einem NIM-Spiel schlagen.

Formal ist das NIM-Problem wie folgt gegeben.

NIM
Gegeben: $(R_1, \ldots, R_k)$, R_i binärkodiert für $i = 1, \ldots k$ (k Reihen von Streichhölzer)
Entscheide: Hat Spieler P_1 eine Gewinnstrategie für diese Startkonfiguration?

9.21 Theorem: Bouton
NIM ist in L.

Um diesen Satz zu beweisen, benötigt man, wie für 2SAT ein tieferes Verständnis der Semantik des Spiels.

9.22 Definition
Wir nehmen an, dass die Konfigurationen als Matrix aufgeschrieben werden, wobei

- die Zeilen der Matrix gleich $R_1, \ldots, R_k$ sind und
- jede Zeile in Least-Significant-Bit-First Darstellung gegeben ist.

Dann heißt eine Konfiguration **balanciert**, falls jede Spalte eine gerade Anzahl an Einsen aufweist. Zum Beispiel ist die Konfiguration 221 nicht balanciert, da in der ersten Spalte der Matrixdarstellung

$$\begin{array}{rl} & 0\ 1 \\ 221 \simeq & 0\ 1 \\ & 1\ 0 \end{array}$$

nur eine ungerade Anzahl an Einsen vorhanden ist.

9.23 Lemma

1. Zu jeder unbalancierten Konfiguration, gibt es einen Spielzug der zu einer balancierten Konfiguration führt.
2. Für jede balancierte Konfiguration, führt jeder Spielzug zu einer unbalancierten Konfiguration.

Der Beweis dieser Aussage sei der Leserin/dem Leser als Übungsaufgabe überlassen.

Am obigen Beispiel erkennt man, dass sich balancierte und unbalancierte Positionen abwechseln:

$$\text{unb} \xrightarrow{P_1} \text{b} \xrightarrow{P_2} \text{unb} \xrightarrow{P_1} \text{b} \xrightarrow{P_2} \text{unb} \xrightarrow{P_1} \text{b} = 000$$

9.24 Lemma
Spieler P_1 hat eine Gewinnstrategie gdw. die initiale Konfiguration unbalanciert ist.

Beweis:
Falls die Startkonfiguration unbalanciert ist, zieht Spieler P_1 während des Spieles immer so, dass die Folgekonfiguration balanciert ist. Das heißt, dass der Gegner P_2 immer nur bei balancierten Konfigurationen ziehen kann und dabei, unabhängig von seinem Spielzug, nur unbalancierte Folgekonfigurationen erreichen kann. Insbesondere, kann der Gegner also nicht die gewinnende Konfiguration, die nur aus Nullen besteht, erreichen da diese balanciert ist.

Falls die Startposition balanciert ist, hat Spieler P_2 eine Gewinnstrategie indem er immer zu balancierten Folgekonfigurationen zieht. □

Mit diesem Lemma, können wir nun Satz 9.21) beweisen.

Beweis:
Um zu entscheiden ob Spieler P_1 eine Gewinnstrategie hat, müssen wir prüfen, ob die Startkonfiguration unbalanciert ist. Es reicht je ein Bit für jede Spalte um zu prüfen, ob sie gerade viele Einsen enthält. Die Spalte wird durchlaufen und für jede Eins, wird das Bit der betreffenden Spalte geflippt. Zu Beginn initialisieren wir jedes Bit mit 0 und prüfen am Ende ob mindestens eines der Bits eine 1 enthält. Dies lässt sich mit logarithmischem Platz realisieren.
□

10. P

Unser nächsten Ziel ist es, zu verstehen, welche Probleme in polynomieller Zeit gelöst werden können. Wir beginnen mit der Klasse P, also der Klasse der Probleme, die durch **deterministische Turing-Maschinen** in **Polynomialzeit gelöst werden können**.

Üblicherweise lassen sich Probleme in P als Auswertungen oder Überprüfungen formulieren. Dies beinhaltet das Prüfen der Korrektheit von Beweisen oder die Evaluation einer Funktionen an einem bestimmten Wert. Konkret werden wir ein erstes P-vollständiges Problem betrachten. Der Beweis der Vollständigkeit geht auf Ladner zurück.

Das Circuit-Value-Problem

Unser Ziel ist es, das **Circuit-Value-Problem (CVP)** zu definieren. Wir beginnen mit der Definition von Schaltkreisen (Circuits).

10.1 Definition
Ein **Boolescher Schaltkreis (Circuit)** ist ein Sequenz von endlich vielen Zuweisungen der Form

$$P_k = 0 \mid 1 \mid P_i \vee P_j \mid P_i \wedge P_j \mid \neg P_i,$$

mit $i, j \in \mathbb{N}$ und $i, j < k$.

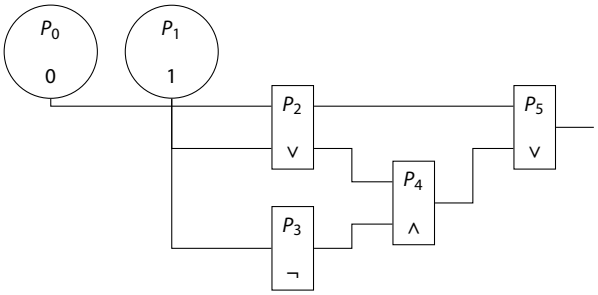
Jedes P_k darf dabei nur ein Mal definiert werden, also nur ein mal auf der linken Seite einer Zuweisung stehen.

10.2 Beispiel
Betrachte den Schaltkreis, der durch folgende Zuweisungen gegeben ist.

$$\begin{aligned} C: \; & P_0 = 0, \\ & P_1 = 1, \\ & P_2 = P_1 \vee P_0, \\ & P_3 = \neg P_1, \\ & P_4 = P_3 \wedge P_2, \\ & P_5 = P_2 \vee P_4. \end{aligned}$$

Wir können Schaltkreise als gerichtete Graphen auffassen. Dabei sind die P_k, welchen ein Wert 0 oder 1 zugewiesen wird die Inputsignale. Die anderen P_k stellen Gatter dar, die bis zu zwei

Inputs haben und diese mit $\neg$, $\vee$ oder $\wedge$ kombinieren. Dies erinnert dann an Schaltkreise aus der Elektrotechnik. Für obiges Beispiel ergibt sich der folgende Graph:



Zu beachten ist, dass P_2 zwei Ausgänge hat. Es ist also erlaubt, dass ein P_i auf der rechten Seite von mehreren P_k mit $k > i$ verwendet wird.

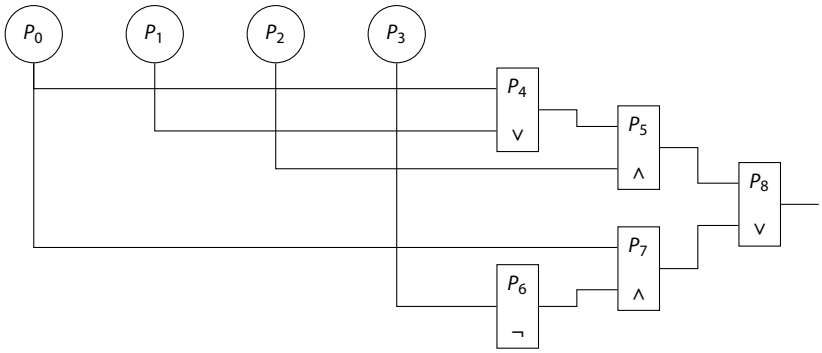
10.3 Bemerkung
Die Tatsache, dass der Wert eines P_i mehrfach verwendet werdet darf, ist ein wichtiger Unterschied zu Booleschen Formeln.

Wenn wir eine Boolesche Formel gegeben haben, so können wir diese als einen *symbolischen* Schaltkreis modellieren, in dem jedes P_i nur ein Mal verwendet wird. Beispielsweise können wir die Formel $F = ((a \vee b) \wedge c) \vee (\neg d \wedge a)$ wie folgt als Schaltkreis interpretieren:

$$
\begin{aligned}
C: \; & P_0 = a, \\
& P_1 = b, \\
& P_2 = c, \\
& P_3 = d, \\
& P_4 = P_0 \vee P_1, \\
& P_5 = P_4 \wedge P_2, \\
& P_6 = \neg P_3, \\
& P_7 = P_6 \wedge P_0, \\
& P_8 = P_5 \vee P_7.
\end{aligned}
$$

Wie man sieht handelt es sich hierbei um einen symbolischen Schaltkreis, da die Werte von P_0, P_1, P_2 und P_3 (bzw. a, b, c und d) noch nicht bekannt sind.

Wir sehen, dass dieser Schaltkreis eine baumartige Struktur hat, da jedes P_i, das nicht einer Variable entspricht, höchstens ein Mal auf einer rechten Seite verwendet wird.



Wenn wir nun eine Variablenbelegung für $a, b, c,$ und d gegeben haben, so können wir den symbolischen Schaltkreis durch Ersetzen der Variablen einen Schaltkreis umwandeln. Das Berechnen des Werts von P_8 entspricht dann dem Auswerten der Booleschen Formel an der gegebenen Belegung.

Man beachte, dass das Auswerten von Booleschen Formeln ein Problem in L ist, während das Auswerten von beliebigen Schaltkreisen P-vollständig ist, wie wir gleich sehen werden.

Das zu einem Schaltkreis assoziierte algorithmische Probleme ist das Ausrechnen der Werte der P_i.

Circuit Value Problem (CVP)
Gegeben: Ein boolscher Schaltkreis C als Liste von Zuweisungen $P_0, \ldots, P_\ell$.
Entscheide: Ist der Wert von P_ℓ gleich 1?

10.4 Beispiel
Der Schaltkreis aus Beispiel 10.2 lässt sich wie folgt auswerten: $P_0 = 0, P_1 = 1$ sind gegeben. Nun wertet man der Reihe nach aus: $P_2 = 1, P_3 = 0, P_4 = 0, P_5 = 1$. Intuitiv geht man durch die Liste der Zuweisungen. Wenn man auf ein neues, unausgewertetes P_k trifft, sucht man die Werte der P_i mit $i < k$, die in der Zuweisung von P_k verwendet werden. Da diese schon ausgewertet wurden, kann man P_k auswerten. Diese Idee werden wir im ersten Teils des folgenden Theorems verwenden, um zu zeigen, dass CVP in P liegt.

10.5 Theorem: Ladner, 1975
CVP ist P-vollständig (bezüglich logspace-many-one-Reduktionen).

Wir teilen den Beweis des Theorems in zwei Schritte auf. Im ersten Schritt zeigen wir, dass CVP in P liegt („Membership"). Im zweiten Schritt, dass CVP auch P-schwer ist („Hardness").

10.6 Lemma: „Membership"
CVP liegt in P.

Beweis:
Wir konstruieren eine Turing-Maschine M, welche die Idee des oben erwähnten Algorithmus umsetzt. Der Einfachheit halber verwenden wir eine TM die nur ein rechts unendliches Band hat, siehe dazu B) in Kapitel 1. Sei der gegebene Schaltkreis C durch die Zuweisungen an $P_0, \ldots, P_\ell$ definiert. Wir nehmen an, dass die Zuweisungen getrennt durch ein Zusatzsymbol $\#$ auf dem Eingabeband von M stehen.

Nun wertet M die P_i von links nach rechts aus. Dazu speichert sie die bereits berechneten Werte auf einem Arbeitsband, ebenfalls durch $\#$-Symbole getrennt.

\$	P_0	$\#$	P_1	$\#$	P_2	$\#$	$\cdots$

Angenommen, wir haben $P_0, \ldots, P_i$ schon ausgewertet und wollen nun P_{i+1} auswerten. Dann bewegt M seinen Kopf zu P_{i+1} auf dem Eingabeband und liest die Zuweisung. Falls $P_{i+1} = 0$, so schreibt die Maschine $0\#$ auf das Arbeitsband und fährt mit P_{i+2} fort, analog für $P_{i+1} = 1$.

Falls $P_{i+1} = P_j \wedge P_k$ mit $j, k \leqslant i$, so überträgt M die Indices j und e auf weitere Arbeitsbänder. Nun läuft M durch das erste Arbeitsband, um die bereits berechneten Werte von P_i und P_e zu lesen; Hierbei werden die gespeicherten Indizes zum Finden der richtigen Position verwendet. Sobald der erste Wert gelesen wurde, wird er im Kontrollzustand gespeichert. Wenn auch der zweite Wert gelesen wurde, kann M auf das Arbeitsband $P_j \wedge P_k\#$ schreiben und mit P_{i+2} fortfahren. Die Fälle $P_{i+1} = P_j \vee P_k$ und $P_{i+1} = \neg P_j$ werden analog behandelt.

Sobald der Wert von P_ℓ berechnet wurde, akzeptiert M wenn dieser Wert 1 ist und weist ansonsten ab.

Im schlimmsten Fall muss M für das berechnen eines jeden P_i einmal komplett durch die bereits berechneten Werte laufen. Dazu werden höchstens $\mathcal{O}(\ell)$ Schritte benötigt. Insgesamt ergibt sich ein Zeitaufwand von $\mathcal{O}\left(\ell^2\right)$. □

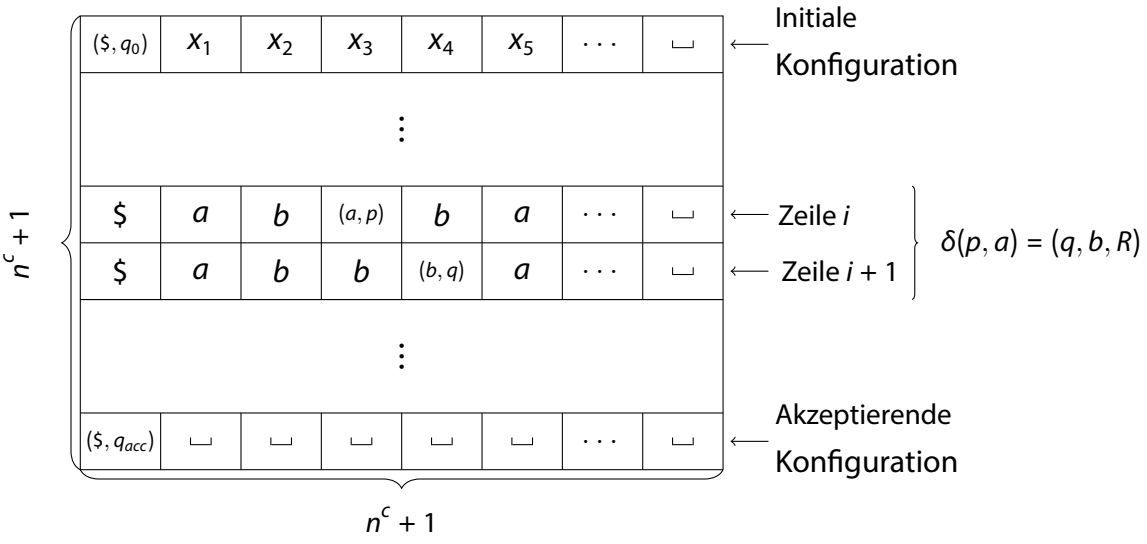
Es verbleibt zu zeigen, dass CVP P-schwer ist. Die Schwierigkeit liegt hierbei darin, dass wir bislang kein weiteres P-schweres Problem kennen, welches wir reduzieren könnten. Wir müssen also tatsächlich zeigen, wie man jedes Problem aus P auf CVP reduzieren kann.

10.7 Lemma: „Hardness“
CVP ist P-schwer (bezüglich logspace-many-one-Reduktionen).

Beweis:
Wir reduzieren ein beliebiges Problem A aus P auf CVP. Das Problem A ist die Sprache einer deterministischen Turing-Maschine M, die n^c-zeitbeschränkt ist. Hierbei ist c eine Konstante. Des Weiteren nehmen wir an, dass M nur ein rechts unendliches Band verwendet.

Im Folgenden beschreiben wir die Idee des Beweises. Sei Q die Zustandsmenge von M, Γ das Bandalphabet und δ die Transitionsrelation. Ferner, sei x ein Input von M der Länge n. Eine Berechnung von M auf x ist eine Folge von höchstens $n^c + 1$ Konfigurationen. Zudem belegt jede dieser Konfigurationen höchstens $n^c + 1$ Zellen auf dem Band von M. Wir können die Konfigurationen also in eine Matrix der Größe $(n^c + 1) \times (n^c + 1)$ schreiben.

Die i-te Zeile der Matrix beschreibt hierbei die i-te Konfiguration. Dies entspricht einem String der Länge $n^c + 1$ über $(\Gamma \cup (\Gamma \times Q))$. Man beachte, dass nur ein Eintrag pro Zeile in $(\Gamma \times Q)$ liegen darf. Dieser beschreibt die Kopfposition und den aktuellen Zustand. Die j-te Spalte der Matrix beschreibt den Inhalt der j-ten Zelle von M in jeder Konfiguration. Ein Sprung von Zeile i zu $i + 1$ entspricht dem Ausführen einer Transition.



Man kann erkennen, dass sich der Bandinhalt pro ausgeführter Transition nur an wenigen Stellen ändert. Die Berechnung von M ist also in gewissem Sinne **lokal**. Diese Lokalität erlaubt es uns, die Matrix und damit die Berechnung von M als Schaltkreis zu kodieren.

Konstruktion: Wir werden zuerst die Variablen des Schaltkreises und deren Idee erläutern:

- P_{ij}^a ist 1, wenn in der i-ten Konfiguration in Zelle j das Symbol a steht.
- Q_{ij}^p ist 1, wenn in der i-ten Konfiguration der Kopf von M auf Zelle j steht und M im Zustand p ist.

Wir erhalten diese Variablen für $0 \leqslant i, j \leqslant n^c$, $a \in \Gamma$, $p \in Q$.

Wir beginnen mit der Kodierung der ersten Zeile der Matrix, der initialen Konfiguration. Wir setzen $P_{00}^{\$} = 1$. Dies sichert uns, dass das Symbol \$ zu Beginn ganz links auf dem Band von M steht. Weiter setzen wir $P_{00}^b = 0$ für alle $b \in \Gamma \setminus \{\$\}$. Hiermit stellen wir sicher, dass an der Stelle nur das \$-Symbol steht und kein anderes Symbol.

Um die Eingabe x zu kodieren, setzen wir für $j = 1, \ldots, n$: $P^{x_j}_{0j} = 1$ und $P^{b}_{0j} = 0$, wobei $b \in \Gamma \setminus \{x_j\}$. Nun fehlen noch die $\sqcup$-Symbole, die den Rest des Bandes von M füllen. Dazu setzen wir für $j = n + 1, \ldots, n^c$: $P^{\sqcup}_{0j} = 1$ und $P^{b}_{0j} = 0$, wobei $b \in \Gamma \setminus \{\sqcup\}$.

Kodieren wir nun die Kopfposition und den initialen Zustand von M. Wir setzen $Q^{q_0}_{00} = 1$, denn der Kopf von M steht zunächst in Zelle 0 und M ist im initialen Zustand q_0. Weiter setzen wir $Q^{q}_{00} = 0$ für $q \in Q \setminus \{q_0\}$ und $Q^{q}_{0j} = 0$ für $j = 1, \ldots, n^c$, $q \in Q$.

Als nächstes kodieren wir die Transitionen von Konfiguration (Zeile) i nach $i + 1$. Auf Grund der vorher angesprochenen Lokalität, wird jede Zuweisung einer Variable nur konstant viele andere Variablen enthalten. Wir beginnen mit dem Bandinhalt. Dieser kann sich in Zelle j ändern, wenn der Kopf von M in Konfiguration i auf Zelle j stand und das passende Symbol für eine Transition gelesen hat. Dann schreibt M in Zelle j. Falls der Kopf nicht in Zelle j stand, bleibt der Bandinhalt in Zelle j derselbe. Zusammengefasst ergibt sich:

$$P^{b}_{(i+1)j} = \underbrace{\bigvee_{\delta(p,a)=(q,b,d)} \left(Q^{p}_{ij} \wedge P^{a}_{ij}\right)}_{M \text{ schreibt in Zelle } j.} \vee \underbrace{\left(P^{b}_{ij} \wedge \bigwedge_{p \in Q} \neg Q^{p}_{ij}\right)}_{\text{Kopf steht nicht in Zelle } j.}.$$

Den Zustand von M können wir anhand der Transition ändern. Für die Kopfposition ergibt sich Folgendes: Der Kopf steht in Konfiguration $i + 1$ in Zelle j, wenn er in Konfiguration i in Zelle $j - 1$ stand und M ihn nach rechts bewegt hat, oder wenn er in Konfiguration i in Zelle $j + 1$ stand und M ihn nach links bewegt hat.

$$Q^{q}_{(i+1)j} = \underbrace{\bigvee_{\delta(p,a)=(q,b,R)} \left(Q^{p}_{i,j-1} \wedge P^{a}_{i,j-1}\right)}_{M \text{ bewegt den Kopf nach rechts.}} \vee \underbrace{\bigvee_{\delta(p,a)=(q,b,L)} \left(Q^{p}_{i,j+1} \wedge P^{a}_{i,j+1}\right)}_{M \text{ bewegt den Kopf nach links.}}.$$

Man beachte, dass man Zelle 0 nur von rechts erreichen kann, also nur durch Kopfbewegung nach links, und Zelle n^c nur von links, also nur durch Kopfbewegung nach rechts. Die entsprechenden Zuweisungen ändern sich dann geringfügig.

Wir können ohne Beschränkung der Allgemeinheit annehmen, dass jede Berechnung von M damit endet, dass M ihren Kopf zum Startmarker \$ bewegt. (Wir können jedes M, das nicht von dieser Form ist, durch eine entsprechende Routine erweitern, ohne die akzeptierte Sprache zu verändern. Es wird dann lediglich linear mehr Zeit benötigt.) Demnach ergibt sich: M akzeptiert x genau dann, wenn $Q^{q_{acc}}_{n^c 0} = 1$. Dies ist die letzte Zuweisung im Schaltkreis. Daher akzeptiert x genau dann, wenn der Schaltkreis, der über die P^{a}_{ij} und Q^{p}_{ij} definiert wurde, eine positive CVP-Instanz ist.

Diesen Schaltkreis kann man in logarithmischem Platz konstruieren, es handelt sich hierbei also um eine logspace-Reduktion. □

In der Definition von Circuits haben wir in den Zuweisungen an Variablen P_k nur binäre Konjunktionen und Disjunktionen erlaubt. Im Beweis haben wir komplexere Formeln als rechte Seite von Zuweisungen verwendet. Durch das Einführen von Hilfsvariablen lassen sich diese erweiterten Zuweisungen in die initial geforderte Form bringen. Hierbei ist wie bereits erwähnt die Lokalität wichtig: Jede der komplexen Formeln verwendet nur konstant viele Variablen. Um genau zu sein, hängt die Größe dieser Formeln von der Turing-Maschine M die wir kodieren ab (ihrer Bandsymbole, ihrer Kontrollzustände und ihrer Transitionen), nicht jedoch von der Größe der Eingabe x.

10.8 Aufgabe
Beweise, dass die Konstruktion aus dem Beweis von Lemma 10.7 sich tatsächlich mit einer logspace-Reduktion umsetzen lässt.

A) Kontextfreie Sprachen und Dynamische Programmierung

Wir wollen nun ein weiteres Problem aus P kennen lernen.

Aus der Vorlesung „Theoretische Informatik 1“ sind uns die kontextfreien Sprachen bekannt. Die kontextfreien Sprachen sind jene Sprachen, welche sich durch kontextfreie Grammatiken darstellen lassen. Zur Erinnerung, die Produktionen von kontextfreien Grammatiken haben die Form $X \to \alpha$, wobei $X \in N$ ein einzelnes Nichtterminal ist und $\alpha \in (N \cup \Sigma)^*$ ein beliebiges Wort aus Terminalen und Nichtterminalen.

Wir werden in diesem Kapitel sehen, dass das Wortproblem für eine kontextfreie Sprache in P liegt. Für den Polynomialzeit-Algorithmus werden wir die algorithmische Technik des **dynamischen Programmierens** einführen.

Dynamisches Programmieren

Die zentrale Idee bei dieser algorithmischen Technik ist es, das gegebene Problem in gleichartige Teilprobleme zu zerlegen, diese zu lösen um dann die Lösung des Gesamtproblems aus den Teillösungen zu berechnen. Um zu vermeiden, dass Lösungen von Teilproblemen mehrfach berechnet werden, speichern wir sie in einer Tabelle um sie bei Bedarf auslesen zu haben.

Ein typisches Problem, bei dessen Lösung dynamische Programmierung zum Einsatz kommt, ist die Berechnung der Fibonacci-Zahlen. Die n-te Fibonacci Zahl, $\mathrm{fib}(n)$, für eine natürliche Zahl $n \geqslant 3$ ist rekursiv definiert durch

$$\mathrm{fib}(n) = \mathrm{fib}(n-1) + \mathrm{fib}(n-2),$$

mit den Anfangswerten $\mathrm{fib}(1) = \mathrm{fib}(2) = 1$.

Um fib(n) (für $n \geqslant 3$) zu berechnen, müssen wir also erst die Teilprobleme fib($n-1$) und fib($n-2$) lösen. Bei der naiven Implementierung werden Lösungen von Teilproblemen mehrfach berechnet.

10.9 Beispiel
Bei der Berechnung von fib(5)

$$\begin{aligned}\text{fib}(6) &= \text{fib}(5) + \text{fib}(4)\\ &= (\text{fib}(4) + \text{fib}(3)) + (\text{fib}(3) + \text{fib}(2))\\ &= \Big((\underline{\text{fib}(3)} + \text{fib}(2)) + \underline{\text{fib}(3)}\Big) + \Big(\underline{\text{fib}(3)} + \text{fib}(2)\Big)\\ &= \ldots\end{aligned}$$

wird der Wert von fib(3) drei Mal ermittelt.

Um die Mehrfachberechnung zu vermeiden, speichern wir die Lösung des Teilproblems nach der ersten Berechnung in einer Tabelle und schauen den Wert bei Bedarf nach. Für die Berechnung der n-ten Fiboncci Zahl, werden wir eine eindimensionale Tabelle tab der Länge n verwenden. Wir initialisieren tab(1) = tab(2) = 1 und setzen tab(i) = tab($i-1$) + tab($i-2$) für $3 \leqslant i \leqslant n$ und füllen so die Tabelle von links nach rechts. Am Ende der Berechnung speichert die i-ten Zelle den Wert von fib(i).

Widmen wir uns nun dem Wortproblem für kontextfreie Sprachen. Dazu fixieren wir eine kontextfreie Grammatik $\mathcal{G}$ in Chomsky Normalform. Zur Erinnerung, eine Grammatik ist in Chomsky Normalform, falls alle Produktionen die Form $A \to BC$ oder $A \to a$ haben, wobei $A, B, C \in N$ Nichtterminale sind und $a \in \Sigma$ ein Terminal ist. Das Wortproblem bezüglich der Sprache der Grammatik $\mathcal{L}(\mathcal{G})$ ist dann folgendermaßen definiert.

Wortproblem für $\mathcal{L}(\mathcal{G})$	
Gegeben:	Ein Terminalwort $w \in \Sigma^*$.
Entscheide:	Gilt $w \in \mathcal{L}(\mathcal{G})$?

Wir fragen uns also ob man das Wort $w = a_1 a_2 \ldots a_n$ vom Startsymbol S der Grammatik $\mathcal{G}$ ableiten kann.

In der Vorlesung „Theoretische Informatik 1“ haben wir bereits einen Algorithmus, der das Wortproblem für Grammatiken in Chomsky Normalform löst, kennengelernt. Der CYK-Algorithmus füllt dazu eine $(n \times n)$- Tabelle tab.

Die Subprobleme bestimmen für jeden Infix $v = a_i \ldots a_j$ von w (für $1 \leqslant i \leqslant j \leqslant n$) jene Nichtterminale A, für die $A \Rightarrow^* v$ gilt, also von denen aus v abgeleitet werden kann. Diese

Liste von Nichtterminalen wird in tab(i, j) abgespeichert. Wir starten mit Infixen der Länge 1 und erhöhen die Länge in jedem Schritt. Dabei verwenden wir die Einträge der kurzen Infixe um die Einträge für die längeren Infixe zu bestimmen. Genauer gesagt, gehen wir wie folgt vor.

- Angenommen wir haben schon bestimmt, welche Nicht-Terminale die Infixe der Länge $\leqslant k-1$ erzeugen.
- Um zu bestimmen, ob A das Infix v der Länge k erzeugt, spalte $v = a_i \ldots a_{i+k}$ in zwei nichtleere Teile $v_1 = a_i \ldots a_{i+l}$ und $v_2 = a_{i+l+1} \ldots a_{i+k}$. Es gibt $k-1$ Möglichkeiten v zu teilen.

  Wir betrachten alle Regeln $A \to BC$ und prüfen, ob B v_1 generiert und C v_2 generiert. Falls ja, fügen wir A dem Eintrag für tab$(i, i+k)$ hinzu.

Falls am Ende der Berechnung das Startsymbol S in tab$(1, n)$ gespeichert ist, akzeptiert der Algorithmus das Wort w.

10.10 Beispiel
Sei $\mathcal{G} = (\{S, A, B, C\}, \{a, b\}, P, S)$ mit Produktionen P:

$$\begin{aligned} S &\to AB|BC \\ A &\to BA|a \\ B &\to CC|b \\ C &\to AB|a \end{aligned}$$

und $w = baaba$.

Dann füllt der CYK-Algorithmus die Tabelle wie folgt.

	1	2	3	4	5
1	$\{B\}$	$\{S, A\}$	$\varnothing$	$\varnothing$	$\{A, C, S\}$
2		$\{A, C\}$	$\{B\}$	$\{B\}$	$\{A, C, S\}$
3			$\{A, C\}$	$\{S, C\}$	$\{B\}$
4				$\{B\}$	$\{A, S\}$
5					$\{A, C\}$

Wir können schließen, dass $w \in \mathcal{L}(\mathcal{G})$ gilt, da $S \in \{A, C, S\} = \text{tab}(1, 5)$.

Der folgende Pseudocode implementiert den CYK-Algorithmus.

for $i = 1, \ldots, n$ **do**
  $\text{tab}(i, i) \leftarrow \{A \in N \mid A \to a_i \in P\}$
end for
for $k = 2, \ldots, n$ **do**
  for $i = 1, \ldots, (n - k) + 1$ **do**
    $\text{tab}(i, (i + k) - 1) \leftarrow \varnothing$
    for $m = 1, \ldots, k - 1$ **do**
      $\text{tab}(i, (i + k) - 1) \leftarrow \text{tab}(i, (i + k) - 1)$
        $\cup \{A \in N \mid A \to BC \text{ mit } B \in \text{tab}(i, (i + m) - 1)$
          und $C \in \text{tab}((i + m), (i + k) - 1)\}$
    end for
  end for
end for
if $S \in \text{tab}(1, n)$ **then**
  return true
end if
return false

Der Algorithmus verwendet drei geschachtelte Schleifen. Die erste Schleife ist für die Länge des Infixes, die zweite Schleife für den Start des Teilwortes und die dritte Schleife für die Spaltposition. Jede Schleife durchläuft maximal die Werte von 1 bis $|w|$ und damit benötigt der Algorithmus $\mathcal{O}\left(|w|^3\right)$ Schritte.

10.11 Theorem
Das Wortproblem für kontextfreie Sprachen ist in P.

Pseudo-polynomielle Algorithmen

Wir betrachten nun das Beispiel einer dynamischen Programmierung, die auf den ersten Blick in polynomieller Zeit läuft. Wir werden allerdings feststellen, dass man bei Laufzeitanalysen genau auf die Kodierung der Eingabe achten muss.

Wir betrachten dazu das Rod-Cutting Problem.

Rod Cutting	
Gegeben:	Eine Länge $k \in \mathbb{N}$, eine Funktion $w : \{1, \ldots k\} \mapsto \mathbb{N}$ und eine Schranke m.
Entscheide:	Gibt es $i_1, \ldots i_n \in \mathbb{N}$ mit $\sum_{j=1}^{n} i_j = k$, so dass $\sum_{j=1}^{n} w(i_j) \geqslant m$?

Wir fragen uns also, wenn wir einen Stock der Länge k gegeben haben, ob wir ihn zerschneiden können, so dass die Summe der Werte der Einzelteile mindestens so groß wie Schranke m ist.

Wir können dieses Problem mit dynamischer Programmierung lösen. Dazu verwenden wir eine Liste tab der Länge k. In dem i-ten Eintrag speichern wir den maximalen Wert, den wir für einen Stock der Länge i erhalten können. Falls am Ende der Berechnung $\text{tab}(k) \geqslant m$ gilt, akzeptiert der Algorithmus, ansonsten weist er ab.

Um die Tabelle zu befüllen, gehen wir wie im CYK-Algorithmus vor.

- Wir initialisieren zuerst $\text{tab}(1) = \text{w}(1)$.
- Angenommen wir hätten bereits den maximalen Wert für die Längen $1, \ldots, i-1$ bestimmt. Um den optimalen Wert für einen Stock der Länge i zu bestimmen, betrachten wir alle möglichen Spaltpositionen für den Stock.

  $$\text{tab}(i) = \max_{1 \leqslant l \leqslant i-1} \{\text{w}(i), \text{tab}(l) + \text{tab}(i-l)\}$$

  Der Eintrag $\text{w}(i)$ steht für den Fall, wo wir den höchstmöglichen Wert für i erzielen indem wir den Stock nicht zerteilen.

Dieser Algorithmus benötigt $\mathcal{O}\left(k^2\right)$ Schritte. Auf den ersten Blick könnte man also vermuten, dass Rod Cutting, wie auch das Wortproblem für kontextfreie Sprachen, in P liegt. Es gibt allerdings einen entscheidenden Unterschied. Der CYK-Algorithmus hat Laufzeit $\mathcal{O}\left(|w|^3\right)$, also kubisch in der **Länge** der Eingabe w. Der Algorithmus für Rod Cutting hingegen ist quadratisch im **numerischen Wert** von k. Wir nehmen bei den Laufzeitbetrachtungen an, dass die Eingabe binär kodiert ist. Wir brauchen für die Kodierung von k nur $n_k = \log(k)$ Bits und haben daher eine Laufzeit von $\mathcal{O}\left((2^{n_k})^2\right) = \mathcal{O}\left(2^{2n_k}\right)$ in der Länge der Kodierung von k.

Wir sprechen in diesem Fall von einer pseudo-polynomiellen Laufzeit. Formal ist eine Laufzeit pseudo-polynomiell, wenn sie polynomiell in der Länge der Eingabe **und** in dem numerischen Wert der größten Zahl der Eingabe ist. Alternativ, sagen wir dass eine Laufzeit pseudo-polynomiell ist, wenn sie polynomiell in der Länge der Eingabe ist, falls diese (bzw. die enthaltenen Zahlen) **unär** kodiert wäre.

11. NP

Betrachten wir nun die Klasse NP. Wir wollen verstehen, welche Probleme in **Polynomialzeit** durch **nicht-deterministische Turing-Maschinen** gelöst werden können.

Wie oben schon beschrieben, bestehen typische Problem in P daraus, ein gegebene Eingabe (einen Beweis, eine Formel, einen Schaltkreis, …) zu prüfen. Durch die Hinzunahme von Nicht-Determinismus ist es nun möglich, Probleme zu konstruieren, bei denen die Maschine den Beweise finden muss. Intuitiv können solche Probleme in zwei Schritten gelöst werden: (1) ein Beweis wird nicht-deterministisch geraten, (2) der geratene Beweis wird überprüft. Wir werden sehen, dass sich diese Intuition mit Hilfe von Zertifikaten formulieren lässt und eine alternative Charakterisierung der Klasse NP ergibt.

Wir zeigen jedoch zunächst, dass das SAT Problem (bekannt aus Kapitel **??**) NP-vollständig ist. Tatsächlich war SAT (und auch kSAT für $k \geqslant 3$, insbesondere also 3SAT) das erste Problem, welches von Cook und Levin als NP vollständig nachgewiesen wurde.

A) NP-vollständige Probleme

Wir werden zunächst zeigen, dass SAT NP-vollständig ist. Im Beweis möchten wir auf die Ideen aus dem Beweis der P-Hardness des CVP zurückgreifen (siehe Kapitel Section 10). Hierzu benötigen wir einige Hilfsmittel.

11.1 Definition
Ein Boolscher Schaltkreis **mit variablem Input** ist ein Boolscher Schaltkreis, in welchem wir die zusätzliche Zuweisung

$$P_k = \ ?$$

erlauben. Diese gibt an, dass wir den Wert für P_k nicht kennen, er kann also 0 oder 1 sein.

Sei C ein Schaltkreis mit variablen Inputs $P_1, \ldots, P_k$ und $y_1, \ldots, y_k \in \{0, 1\}$ eine Sequenz von Wahrheitswerten. Wir schreiben $C(y_1, \ldots, y_k)$ für den Schaltkreis, den wir erhalten, wenn wir anstatt der Zuweisung $P_i = ?$ die Zuweisung $P_i = y_i$ für $i = 1, \ldots, k$ einsetzen. Weiter schreiben wir $C(y_1, \ldots, y_k) = 1$, falls der Schaltkreis $C(y_1, \ldots, y_k)$ eine positive CVP-Instanz ist.

Wir haben schon festgestellt, dass das Auswerten von boolschen Formeln effizienter ist (nämlich in L) als das Auswerten von Schaltkreisen (P-schwer). Das Testen der Erfüllbarkeit hat jedoch für beide Modelle dieselbe Komplexität.

11.2 Bemerkung
Sei C ein Schaltkreis mit Zuweisungen $P_1, \ldots, P_\ell$ und variablen Inputs $P_1, \ldots, P_k$. Wir konstruieren eine boolsche Formel F_C wie folgt: Für jede Zuweisung P_i erhalten wir eine Variable x_i. Wenn P_i eine Zuweisung der Form $P_i = E$ ist, wobei $E \neq ?$, fügen wir eine Klausel $x_i \leftrightarrow E$ zu F_C hinzu. Um den Ausdruck E in der Formel F_C zu verwenden, ersetzen wir in E alle P_i durch die entsprechenden Variablen x_i. Für die letzte Zuweisung P_k hängen wir dann noch die Variable x_k mit einer Konjunktion an:

$$F_C = x_k \wedge \bigwedge_{P_i=E, E\neq ?} (x_i \leftrightarrow E).$$

Nun gilt, dass F_C erfüllbar ist, genau dann, wenn es $y_1, \ldots, y_k \in \{0, 1\}$ gibt, sodass $C(y_1, \ldots, y_k) = 1$.

11.3 Beispiel
Verwendet man die obige Konstruktion für folgenden Schaltkreis C:

$$\begin{aligned} P_0 &= ? \\ P_1 &= ? \\ P_2 &= P_0 \vee P_1 \\ P_3 &= \neg P_1 \\ P_4 &= P_2 \wedge P_3, \end{aligned}$$

ergibt sich die Formel $F_C = (x_2 \leftrightarrow x_0 \vee x_1) \wedge (x_3 \leftrightarrow \neg x_1) \wedge (x_4 \leftrightarrow x_2 \wedge x_3) \wedge x_4$. Erfüllende Inputs $P_0 = 1$ und $P_1 = 0$ für C lassen sich nun zu einer erfüllenden Belegung von F_C erweitern: Die Variablen x_0 und x_1 werden wie die Zuweisungen P_0 und P_1 auf 1 und 0 gesetzt. Der Wert der verbleibenden Variablen kann dann errechnet werden: $x_2 \to 1, x_3 \to 1, x_4 \to 1$.

11.4 Bemerkung
Die konstruierte Formel ist – anders als wir dies für SAT – fordern noch nicht in Konjunktiver Normalform. Sie kann allerdings mit der Tseitin-Transformation TODO in diese überführt werden.

11.5 Theorem: Cook 1971, Levin 1973
SAT ist NP-vollständig.

Wie für das CVP, teilen wir auch hier den Beweis in zwei Schritte auf. Der erste Schritt, **Membership** in NP, ist dabei schnell einzusehen: Ein Algorithmus für SAT rät zu einer gegebenen Formel F eine Belegung und prüft nach, ob diese erfüllend ist. Dazu muss man die geratene

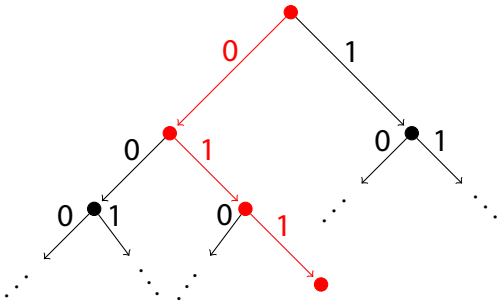


Abbildung 1: Der binäre Berechnungsbaum von M auf Input x. Der rot markierte Pfad wird durch den String $y = 011$ beschrieben.

Belegung nur in die Formel einsetzen und diese auswerten. Dieses Verfahren läuft in Polynomialzeit, daher gilt SAT $\in$ NP.

Im zweiten Schritt zeigen wir, dass SAT NP-schwer ist.

11.6 Lemma
SAT ist NP-schwer.

Beweis:
Sei A ein Problem aus NP. Dann gibt es eine nicht-deterministische Turing-Maschine M mit $A = \mathcal{L}(M)$, die n^c-Zeit beschränkt ist. Wir könne annehmen, dass sich M während einer Berechnung immer höchstens zwischen zwei Transitionen entscheiden muss (binär verzweigt). (Andernfalls könnten wir die gegebene Maschine so transformieren, dass dies gilt, in dem wir eine Entscheidung zwischen k Verzweigungen durch mehrere aufeinander folgende Entscheidungen zwischen jeweils nur 2 Verzweigungen kodieren. Der zusätzliche Zeitaufwand ist polynomiell.)

Sei x nun ein Input von M. Eine Berechnung von M auf x ist ein Pfad im binären Berechnungsbaum. Solch ein Pfad wird durch einen String $y \in \{0,1\}^{n^c}$ beschrieben. Dabei steht die i-te Stelle y_i für die Wahl der Transition in Schritt i. Ein Beispiel ist in Abbildung 1 gegeben.

Sei M' nun die deterministische Turing-Maschine, die als Input $x\#y$ mit $|y| = n^c$ erwartet und M auf x simuliert. Jede nicht-deterministische Entscheidung von M wird dann durch y aufgelöst: M' wählt in der Simulation von M in Schritt i die y_i-te Transition aus. Es gilt daher: M akzeptiert x genau dann, wenn es ein $y \in \{0,1\}^{n^c}$ gibt, sodass M' die Eingabe $x\#y$ akzeptiert.

Da M' deterministisch ist, können wir nun, wie im Satz von Ladner (Theorem 10.5), einen Schaltkreis C konstruieren. Dieser modelliert die Berechnung von M' auf $x\#y$. Es gilt dann: M' akzeptiert $x\#y$ genau dann, wenn C eine positive CVP-Instanz ist. Seien $P_1, \ldots, P_{n^c}$ die Zuweisungen, welche den String y modellieren. Wenn wir diese unbestimmt machen, also durch die Zuweisungen $P_1 = ?, \ldots, P_{n^c} = ?$ ersetzen, erhalten wir einen neuen Schaltkreis

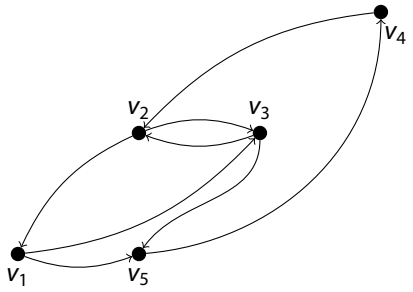
C'. Für diesen gilt dann: M akzeptiert x genau dann, wenn es ein $y \in \{0,1\}^{n^c}$ gibt, sodass $C'(y_1, \ldots, y_{n^c}) = 1$.

Nun übersetzen wir den Schaltkreis C' in die Formel $F_{C'}$, wie in Bemerkung 11.2. Mit der Formel erhalten wir dann folgende Äquivalenz: M akzeptiert x genau dann, wenn $F_{C'}$ erfüllbar ist. Also haben wir ein beliebiges Problem A in NP auf die Erfüllbarkeit einer boolschen Formel zurückgeführt. □

Betrachten wir nun ein Problem aus der Welt der Graphen: Wir suchen nach einem hamiltonschen Kreis, einem Kreis, der alle Knoten eines Graphen genau ein mal besucht. Wir werden zeigen, dass auch dieses Problem NP-vollständig ist. Dazu verwenden wir eine Reduktion von SAT. Widmen wir uns aber zunächst den benötigten Definitionen:

11.7 Definition
Sei $G = (V, E)$ ein gerichteter Graph mit $|V| = n$. Ein **Hamiltonscher Kreis** in G ist ein Pfad $v_{i_1}, \ldots, v_{i_n}$, sodass $v_{i_j} \neq v_{i_k}$ für $j \neq k$ und es gibt eine Kante von v_{i_n} nach v_{i_1}. In anderen Worten: Es handelt sich um einen Kreis, welcher jeden Knoten genau ein mal besucht.

11.8 Beispiel
Der folgende Graph hat den hamiltonschen Kreis: v_1, v_3, v_5, v_4, v_2.



Die Frage, ob es zu einem gegebenen Graphen einen hamiltonschen Kreis gibt, formulieren wir im folgenden Problem:

Hamiltonian Cycle	
Gegeben:	Ein gerichteter Graph $G = (V, E)$.
Entscheide:	Gibt es einen hamiltonschen Kreis in G?

11.9 Theorem
Hamiltonian Cycle ist NP-vollständig.

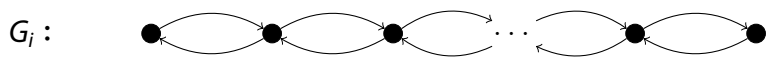
Der Beweis gliedert sich wieder in zwei Schritte. Wie bei SAT kann man auch hier die Zugehörigkeit zu NP schnell einsehen: Sei $G = (V, E)$ der gegebene Graph. Eine nicht-deterministische

Turing-Maschine für Hamiltonian Cycle rät ein Folge $v_1, \ldots, v_n$ von Knoten in V und schreibt diese auf das Band. Nun wird verifiziert, dass es sich dabei um einen hamiltonschen Kreis handelt: Zuerst wird getestet, ob alle Knoten vorkommen, also ob je zwei der geratenen Knoten verschieden sind. Dies kann in $\mathcal{O}\left(n^2\right)$ ermittelt werden. Dann wird getestet, ob es von v_i nach v_{i+1}, mit $i = 1, \ldots, n-1$ und von v_n nach v_1 je eine Kante gibt. Ist dies erfüllt, akzeptiert die Turing-Maschine. Falls nicht, weist sie ab. Die Maschine läuft in $\mathcal{O}\left(n^2\right)$ Zeit.

11.10 Lemma
Hamiltonian Cycle ist NP-schwer.

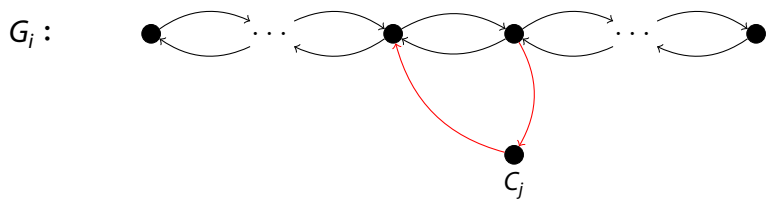
Beweis:
Wir reduzieren SAT auf Hamiltonian Cycle. Sei dazu F eine SAT-Instanz, also eine Formel in CNF. Seien $C_1, \ldots, C_m$ die Klauseln von F und $x_1, \ldots, x_k$ die Variablen. Die Formel F hat dann die Form $F = C_1 \wedge \cdots \wedge C_m$.

Schritt 1: Für jede Variable x_i konstruieren wir einen Graphen G_i, welcher die Belegung der Variable simuliert. Sei dazu m_i die Anzahl der Klauseln, die x_i oder $\neg x_i$ enthalten. Der Graph G_i hat $2m_i + 2$ Knoten, die wie folgt miteinander verbunden sind:

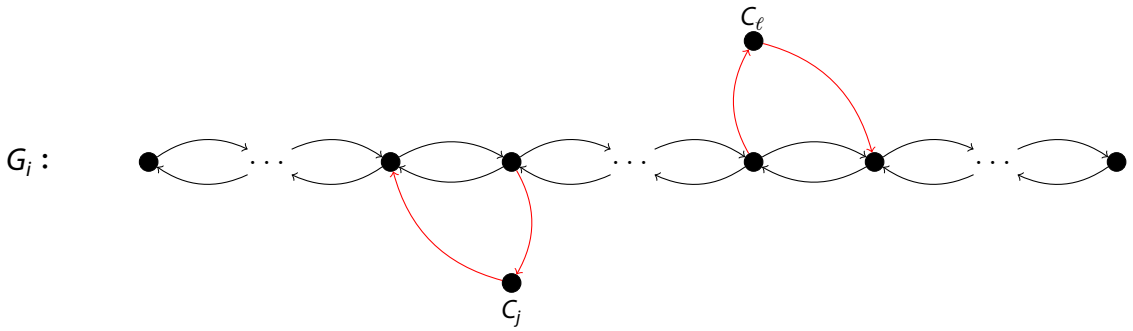


Die Belegung von x_i wird durch einen Durchlauf durch G_i simuliert: Ein Durchlauf von rechts nach links entspricht der Belegung true, ein Durchlauf von links nach rechts entspricht der Belegung false.

Schritt 2: Da man für die Erfüllbarkeit von F alle Klauseln von F erfüllen muss, werden wir die Klauseln nun in die Graphen G_i einbauen. Sei C_j eine Klausel, die x_i enthält. Dann fügen wir einen neuen Knoten in G_i ein. Das Durchlaufen durch diesen Knoten in einem Pfad soll bedeuten, dass wir die Klausel mit der aktuellen Belegung der Variablen x_i erfüllen. Also verbinden wir den Knoten so mit G_i, dass er nur in einem Durchlauf von rechts nach links (true) besucht werden kann. Diese Kanten werden wir im Folgenden rot markieren, um sie von den vorherigen Kanten unterscheiden zu können.



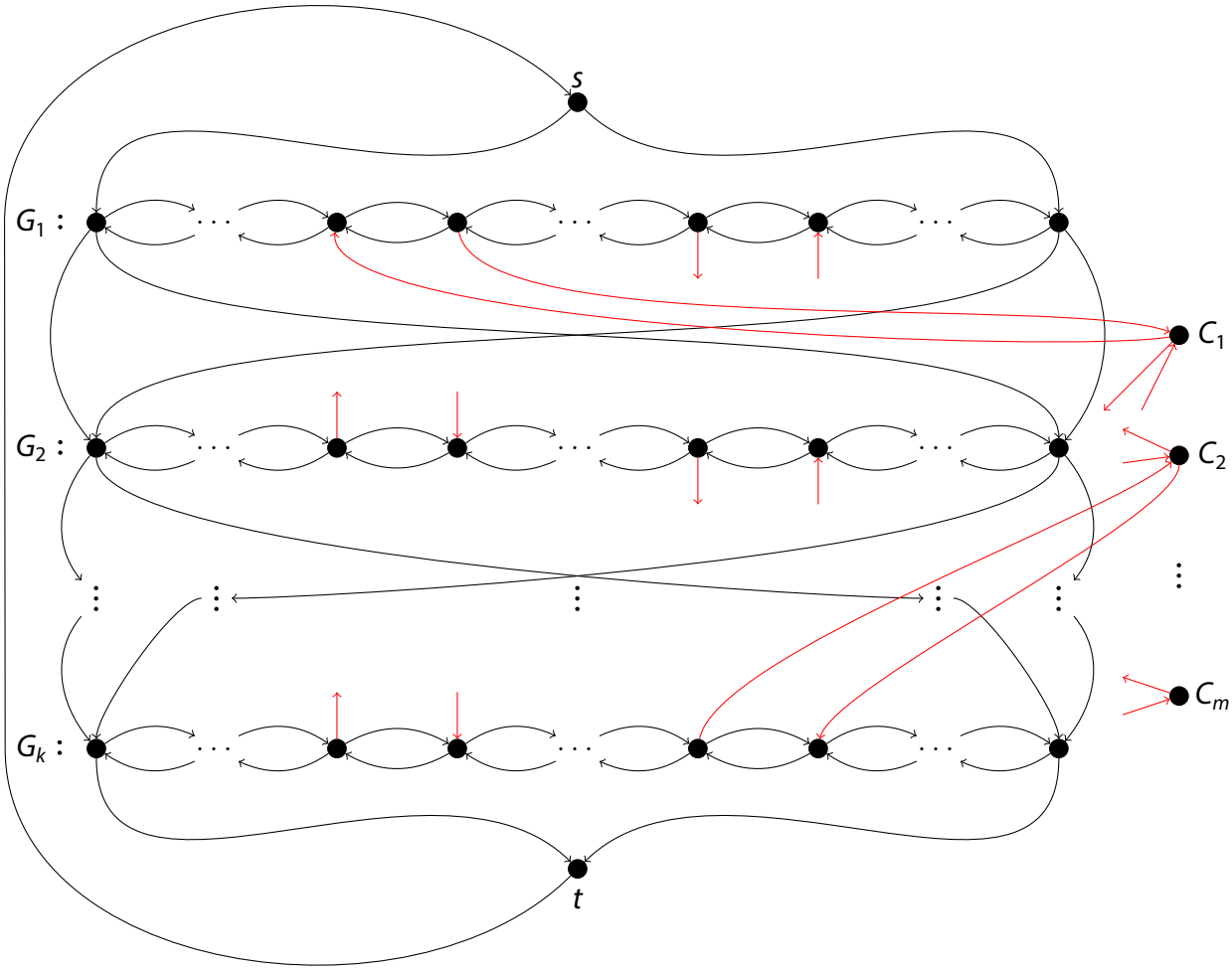
Sei nun C_ℓ eine Klausel, welche $\neg x_i$ enthält. Auch hier fügen wir einen neuen Knoten ein und verbinden diesen so mit G_i, dass er nur durch einen Durchlauf von links nach rechts (false) besucht werden kann:



Da jede Klausel, die x_i oder $\neg x_i$ beinhaltet mit dem Graphen an zwei Knoten verbunden wird und wir später zusätzlich zwei Knoten an den Rändern brauchen, benötigen wir die oben erwähnten $2m_i + 2$ Knoten.

Schritt 3: Nun setzen wir alle Graphen G_i zusammen. Das heißt wir konstruieren einen Graphen G, der aus allen G_i besteht. Dabei werden die Knoten, welche für die Klauseln eingeführt wurden aber nur ein mal für alle Graphen G_i eingeführt und wie oben verbunden. Ein Klauselknoten hat dann eine Verbindung zu mehreren G_i, nämlich genau zu jenen, sodass x_i als positives oder negatives Literal in der Klausel vorkommt.

Zudem führen wir Verbindungen von G_i zu G_{i+1} für $i = 1, \ldots, k-1$ ein. Diese ermöglichen es, nach einem Durchlauf durch G_i in beliebiger Richtung, den Durchlauf durch G_{i+1} in beliebiger Richtung fortzuführen. Intuitiv entspricht dies der Wahl der Belegung von x_i und x_{i+1}. Sobald man x_i auf true oder false gesetzt hat, setzt man x_{i+1} auf true oder false. Die Belegung von x_{i+1} ist nicht von x_i abhängig.



Als letzten Teil der Konstruktion fügen wir einen Initialknoten s und einen Finalknoten t ein und eine Kante (t, s), um einen Kreis zu schließen. Intuitiv beginnt ein hamiltonscher Kreis in G dann in s und endet in t.

Nehmen wir nun an, dass G einen hamiltonschen Kreis hat. Also ist es möglich, durch alle Knoten, auch die Knoten für die Klausen zu laufen. Dies entspricht dann einer erfüllenden Belegung von F. Analog kann jede erfüllende Belegung von F in einen hamiltonschen Kreis von G übersetzt werden. □

B) Zertifikate

Viele Probleme aus NP können mit einem nicht-deterministischen Algorithmus gelöst werden, der zunächst eine „Lösung“ rät und danach deterministisch überprüft, dass richtig geraten wurde. Anders ausgedrückt ist das Verifizieren von Ja-Instanzen leicht, wenn ein poly-

nomiell großes **Zertifikat** gegeben ist, mit dessen die Antwort überprüft werden kann. Beispielsweise ist beim SAT-Problem das Zertifikat eine erfüllende aussagenlogische Belegung.

Wir wollen nun sehen, dass sich in der Tat jedes Problem aus NP auf diese Art und Weise lösen lässt. Intuitiv gesprochen zeigen wir, dass man jedes Problem aus NP mit einem Algorithmus lösen kann, der zunächst ein Zertifikat rät, danach aber (unter Verwendung des geratenen Zertifikats) deterministisch weiter rechnet. Im Gegensatz dazu können theoretisch bei den bisher betrachteten Algorithmen sich nicht-deterministische und deterministische Phasen der Berechnung beliebig abwechseln.

Wir beginnen zunächst damit, das Konzept der Zertifikate zu formalisieren.

11.11 Definition
Ein **Verifizierer** ist eine deterministische, totale Turing-Maschine $\mathcal{V}$ mit zwei Eingabebändern, einem über Alphabet Σ und einem zweiten Zertifikatsband über $\{0, 1\}$.

Die Sprache von $\mathcal{V}$ ist die Menge aller Wörter $x \in \Sigma^*$, so dass es ein **Zertifikat** $y \in \{0, 1\}^*$ gibt, so dass $\mathcal{V}$ die Eingabe (x, y) akzeptiert (d.h. $\mathcal{V}$ akzeptiert, wenn initial das erste Eingabeband x und das zweite Eingabeband y beinhaltet),

$$\mathcal{L}(\mathcal{V}) = \left\{x \in \Sigma^* \;\middle|\; \exists y \in \{0, 1\}^* \colon \mathcal{V} \text{ akzeptiert } (x, y)\right\}.$$

Wir nennen ein solches $\mathcal{V}$ auch einen Verifizierer für $\mathcal{V}$.

Im folgenden sind wir in **Polynomialzeit-Verifizierer** interessiert. Im Gegensatz zu normalen Verifizierern ist hier ein Wort x nur in der Sprache, wenn es ein polynomiell langes Zertifikat y gibt, also y mit $|y| \in \mathcal{O}\left(|x|^k\right)$, wobei die Konstante k von x unabhängig ist. Desweiteren verlangen wir, dass $\mathcal{V}$ auf einer Eingabe x und einem polynomiell großen Zertifikat auch in polynomieller Zeit hält.

11.12 Theorem
Die Klasse NP sind genau die Probleme, die von Polynomialzeit-Verifizierern erzeugt werden: Es gilt $\mathcal{L} \in$ NP genau dann, wenn es einen Polynomialzeit-Verifizierer $\mathcal{V}$ mit $\mathcal{L}(\mathcal{V}) = \mathcal{L}$ gibt.

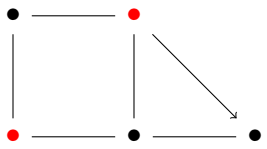
Bevor wir den Satz zeigen, möchten wir zunächst einiger Anwendung des Resultats sehen. Wie bereits oben angesprochen kann man bei SAT eine erfüllende Belegung, welche für Ja-Instanzen existieren muss, als Zertifikat verwenden.

11.13 Beispiel

Independent Set	
Gegeben:	Ein gerichteter Graph $G = (V, E)$ und eine k
Entscheide:	Gibt es in G ein Independent Set der Größe k?

Hierbei ist ein **Independent Set** der Größe k eine Menge $V' \subseteq V$ aus k Knoten, so dass es keine Kante zwischen zwei Knoten aus V' gibt.

Im folgenden Graphen bilden die rot markierten Knoten ein Independent Set der Größe 2.



Ein Polynomialzeit-Verifizierer für Independent Set überprüft, ob auf dem Zertifikatsband (die Kodierung von) k Knoten steht, so dass es zwischen diesen Knoten keine Kanten gibt. Das Zertifikat ist also das Independent Set selbst. Mit Hilfe von Theorem 11.12 ist also gezeigt, dass Independent Set in NP liegt.

11.14 Beispiel

Subset Sum	
Gegeben:	Ein Menge von Zahlen $N = \{n_1, \ldots, n_k\}$ und eine Zahl S
Entscheide:	Gibt es eine Teilmenge $N' \subseteq N$, so dass die Summe der Zahlen in N' die Zahl S ergibt, $S = \sum_{n \in \mathbb{N}'} n$?

Ein Polynomialzeit-Verifizierer für Subset Sum überprüft, ob auf dem Zertifikatsband (die Kodierung von) Zahlen steht, die in der ursprünglichen Menge enthalten sind, berechnet ihre Summe und vergleicht das Ergebniss mit der gegebenen Zahl S. Das Zertifikat ist also die gesuchte Teilmenge. Mit Hilfe von Theorem 11.12 ist also gezeigt, dass Subset Sum in NP liegt.

11.15 Bemerkung

Tatsächlich sind beide Probleme, Independent Set und Subset Sum, sogar NP-vollständig.

Beweis von Theorem 11.12:

Wir müssen zeigen, wie sich eine polynomiell zeitbeschränkte NTM in einen Polynomialzeit-Verifizierer umwandeln lässt und umgekehrt.

Nehmen wir zunächst an, dass ein Polynomialzeit-Verifizierer $\mathcal{V}$ gegeben ist. Wir konstruieren eine NTM $\mathcal{M}$, die sich auf Eingabe x wie folgt verhält:

1. Rate auf einem Arbeitsband das Zertifikat y mit $|y| \in \mathcal{O}\left(|x|^k\right)$. Hierbei ist k die Konstante für $\mathcal{V}$.
2. Simuliere $\mathcal{V}$ auf (x, y).

Es ist leicht einzusehen, dass $\mathcal{M}$ nur polynomiell viel Zeit braucht und $\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{V})$ gilt.

Für die andere Richtung nehmen wir an, dass eine NTM $\mathcal{M}$ gegeben ist, die höchstens polynomiell viel Zeit braucht. Wir können ohne Beschränkung der Allgemeinheit annehmen, dass es in jedem Schritt höchstens zwei mögliche Transitionen gibt. (Ähnliche Annahmen haben wir bereits in anderen Beweisen verwendet und begründet.)

Wir wissen außerdem, dass es eine Konstante k gibt, so dass jede Berechnung von $\mathcal{M}$ zu einer Eingabe x in höchstens $\mathcal{O}\left(|x|^k\right)$ Schritten hält. Wir konstruieren einen Polynomialzeit-Verifizierer $\mathcal{V}$, der sich auf Eingabe x wie folgt verhält:

1. Lese in Schritt i den Wert $y_i \in \{0, 1\}$, d.h. den i-ten Eintrag des Zertifikatsbands.
2. Falls dieser Eintrag nicht existiert, weise ab.
3. Ansonsten simuliere $\mathcal{M}$ auf der aktuellen Konfiguration und verwende y_i um den Nichtdeterminismus aufzulösen: Falls $y_i = 0$, wähle die erste Transition, falls $y_i = 1$ die zweite.
4. Wenn das Ergebnis eine akzeptierende oder abweisende Konfiguration ist, akzeptiere oder weise ab.

Da $\mathcal{V}$ höchstens einen Schritt von $\mathcal{M}$ pro Eintrag auf dem Zertifikatsband simuliert und wir nur polynomiell lange Zertifikate betrachten, ist klar, dass $\mathcal{V}$ höchstens polynomiell lange läuft. Wir argumentieren, dass $\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{V})$ gilt:

- Wenn $x \in \mathcal{L}(\mathcal{M})$ gilt, dann gibt es eine akzeptierende Berechnung zu x polynomielle Länge. Wenn wir die Wahlen von Transitionen, die entlang dieser Berechnung getroffen werden, zu einer Sequenz y kodieren und als Zertifikat verwenden, wird $\mathcal{V}$, gestartet auf (x, y), genau diese Berechnung von $\mathcal{M}$ simulieren und akzeptieren. Es gilt also $x \in \mathcal{L}(\mathcal{V})$.
- Wenn $x \in \mathcal{L}(\mathcal{V})$ gilt, dann gibt es ein polynomiell langes Zertifikat y, so dass $\mathcal{V}$, gestartet auf (x, y), akzeptiert. Gemäß der Konstruktion simuliert $\mathcal{V}$ allerdings $\mathcal{M}$, wobei y zum Auflösen des Nichtdeterminismus verwendet wird. Wenn $\mathcal{V}$ akzeptiert, bedeutet dies insbesondere, dass es eine akzeptierende Berechnung von $\mathcal{M}$ zu Eingabe x gibt; damit gilt $x \in \mathcal{L}(\mathcal{M})$.

□