Theoretische Informatik 2

Berechenbarkeits- und Komplexitätstheorie

Vorlesungsnotizen

13. Juli 2022

Sebastian Muskalla
Roland Meyer
Peter Chini
Elisabeth Neumann
Thomas Haas

TU Braunschweig

Sommersemester 2020

Inhaltsverzeichnis

Vorwort

Dies sind Notizen zur Vorlesung *Theoretische Informatik 2*, die von uns in dem Sommersemestern 2017, 2018 und 2020 an der TU Braunschweig gehalten wurde.

Die Vorlesung basiert auf den von Roland Meyer erstellten handschriftlichen Vorlesungsausarbeitungen zu den Vorlesungen *Formale Grundlagen der Programmierung* (siehe tcs.cs.tu-bs.de/teaching/FGDP_SS_2016.html) und *Komplexitätstheorie* (siehe tcs.cs.tu-bs.de/teaching/ComplexityTheory_WS_20162017.html). Die Vorlesung wurde im Laufe der letzten Jahre angepasst und erweitert – diese Notizen spiegeln den aktuellen Stand wider.

Wir geben keine Garantie auf Vollständigkeit oder Korrektheit der Notizen und bitten darum, uns Fehler per E-Mail zu melden: `s.muskalla@tu-bs.de` .

Wir bedanken uns bei Jan Merten, Pascal Gebhardt, Maximilian von Unwerth, Jakob Keller, Yannic Lieder, Moritz Pfister, Morris Kunz, Janosch Reppnow, Rene Kudlek, Nellika Krummrich und allen anderen Personen, die geholfen haben, Fehler in diesen Notizen zu verbessern.

Sebastian Muskalla, Roland Meyer, Peter Chini, Elisabeth Neumann, Thomas Haas

Braunschweig, 13. Juli 2022

Motivation

In *Theoretische Informatik 1* haben wir **endliche Automaten** und **Pushdown-Automaten** kennen gelernt. Wir haben uns angesehen, welche Sprachen von diesen Automaten akzeptiert werden können, und wir haben Algorithmen kennengelernt, um diese Automaten zu analysieren, z.B. Algorithmen, um zu entscheiden ob die Sprache eines endlichen Automaten leer ist.

Man kann diese Automatenmodelle als in ihrem Funktionsumfang **beschränkte Computer bzw. Programme** sehen. Mit dieser Sichtweise ist die Untersuchung der akzeptierten Klasse von Sprachen eine Untersuchung der **Berechnungsmächtigkeit** dieser Computermodelle, d.h. der Frage, welche Probleme von diesen eingeschränkten Programmen gelöst werden können. Die betrachteten Algorithmen sind in diesem Sinne Verifikationsalgorithmen, Verfahren, die für ein gegebenes Programm entscheiden, ob es eine bestimmte Eigenschaft hat.

In dieser Vorlesung werden wir uns mit **Turing-Maschinen**, einem Automatenmodell für „richtige“ Computer bzw. Programme, beschäftigen. Eine Turing-Maschine hat im Gegensatz zu endlichen Automaten einen unendlich großen Speicher (genau wie auch ein moderner Computer quasi unbegrenzten Speicher hat), und im Gegensatz zu Pushdown-Automaten darf sie diesen Speicher ohne Einschränkung verwenden.

Im ersten Teil der Vorlesung zu **Entscheidbarkeit und Berechenbarkeit** wollen wir uns – genau wie wir es in *Theoretische Informatik 1* getan haben – mit der Mächtigkeit von diesen Automaten befassen, wir wollen also herausfinden, welche Klasse von Problemen von Turing-Maschinen gelöst werden kann. Man kann sich insbesondere die Frage stellen, ob es mit Turing-Maschinen möglich ist, alle wohl-spezifizierten Berechnungsprobleme zu lösen. Wir werden feststellen, dass dies nicht der Fall ist: Es gibt **nicht-entscheidbare** Probleme und **nicht-berechenbare** Funktionen. Mehr noch: Ausgerechnet die Verifikationsprobleme, die in der Theoretischer Informatik von großem Interesse sind, gehören zu diesen unentscheidbaren Problemen. Selbst das Leerheitsproblem, bei dem zu einem gegebenen Automaten entschieden werden soll, ob seine Sprache leer ist, er also kein einziges Wort akzeptiert, ist für Turing-Maschinen nicht algorithmisch lösbar.

Turing-Maschinen sind also zu kompliziert, als dass sie sich selbst analysieren könnten, oder negativ formuliert, nicht mächtig genug, um sich selbst analysieren zu können. Eine offensichtliche Frage ist nun, ob Turing-Maschinen bloß eine schlechte Wahl waren, um richtige Computer zu modellieren. Dies ist nicht der Fall: Turing-Maschinen sind eine Formalisierung des Begriffs des „Algorithmus“, sie fassen also

die Mächtigkeit von Rechenverfahren, Programmen und Computern. Die **Church-Turing-These** sagt, dass sich alles, was sich berechnen lässt, auch mittels einer Turing-Maschine berechnen lässt. Diese Aussage kann man nicht beweisen, es wurde allerdings exemplarisch gezeigt, dass viele andere Berechnungsmodelle (z.B. reale Programme in Assembler, C, Java, …) höchstens genauso mächtig sind wie Turing-Maschinen.

Im zweiten Teil der Vorlesung möchten wir für Probleme, die algorithmisch lösbar sind, untersuchen, wie **effizient** sie sich lösen lassen. Damit dringen wir in das Gebiet der **Komplexitätstheorie** vor. In diesem Feld befasst man sich damit, wie viel Zeit und Speicherplatz Algorithmen brauchen, um bestimmte Probleme zu lösen. Hierbei interessieren wir uns für obere und untere Schranken. Eine untere Schranke für ein Berechnungsproblem sagt, dass jeder Algorithmus, der das Problem lösen kann, mindestens eine bestimmte Zeit oder eine bestimmte Menge Speicherplatz braucht. Eine obere Schranken hingegen wird dadurch bewiesen, dass man konkret einen Algorithmus angibt, der das Problem mit einem bestimmten Zeit- und Speicherverbrauch löst. Im Optimalfall passen die beiden Schranken zusammen. Für das Sortieren von Listen beispielsweise ist zum Einen bekannt, dass sich Listen der Länge n im Worst-Case nicht mit weniger als $n \cdot \log n$ Schritten sortieren lassen, anderseits gibt es aber auch Algorithmen wie z.B. *Mergesort*, die diese gewünschte Komplexität aufweisen.

Es gibt viele offene Probleme in der Komplexitätstheorie, die letztlich darin bestehen, dass eine große Lücke zwischen den bislang bekannten oberen und unteren Schranken klafft. Von dieser Form ist unter anderem $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$, das wohl berühmteste offene Problem der Informatik. Es gibt viele Probleme – insbesondere auch solche, die von praktischer Relevanz sind, wie z.B. viele Optimierungsprobleme – für die bislang kein effizienter deterministischer Algorithmus bekannt ist, für die man allerdings noch nicht beweisen konnte, dass ein solcher Algorithmus nicht existieren kann. In den vielen Jahren, in denen $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$ und ähnliche Probleme nun offen sind, hat man statt unterer Schranken, die die absolute Härte eines Problems beweisen würden, Definitionen eingeführt, mit denen sich die relative Härte von Problemen charakterisieren lässt: Man zeigt, dass ein gegebenes Problem mindestens so schwer ist wie eine ganze Klasse von anderen bislang ungelösten Problemen. Wir werden die entsprechenden Konzepte einführen und das Handwerkszeug erlernen, mit dem man ein Problem als schwer nachweisen kann.

Literatur

Zunächst einmal sei gesagt, dass für diese Vorlesung vorausgesetzt wird, dass die Leserin / der Leser die Grundlagen der Automatentheorie (endliche Automaten, kontextfreie Grammatiken, Pushdown-Automaten) beherrscht. Dieser Stoff wird in der Vorlesung „Theoretische Informatik 1“ vermittelt. Zur gegebenenfalls nötigen Auffrischung dieses Stoffes verweisen wir auf die Fachliteratur sowie auf die Vorlesungsnotizen zu „Theoretische Informatik 1“:

tcs.cs.tu-bs.de/documents/lecturenotes/theoinf1.pdf .

Die in „Theoretische Informatik 2“ vorgestellten Themen bilden zusammen mit den oben genannten Themen aus der Automatentheorie die Grundlagen der Theoretischen Informatik. Dementsprechend gibt es eine Vielzahl an Büchern zu diesen Inhalten, insbesondere auch solche, die sich an Studierende richten. Die Bücher, die zur Vorbereitung der Vorlesung genutzt worden sind, sind weiter unten zu finden. Es lohnt sich, zusätzlich zu den Vorlesungsnotizen einen Blick in ein Buch zu werfen. Oft bieten die verschiedenen Quellen unterschiedliche Blickwinkel auf die Themen, und je nach persönlicher Vorliebe mag die eine oder die andere Sichtweise verständlicher sein. Es ist zu beachten, dass sich die Notationen und Definitionen in den Büchern geringfügig voneinander unterscheiden, letztlich sind die dahinterstehenden Konzepte jedoch die Gleichen.

Der Ausarbeitung der Vorlesung liegen die folgenden Bücher zugrunde. Die Vorlesung folgt allerdings keiner der angegebenen Quellen streng.

[Sch08] U. Schöning
Theoretische Informatik – kurz gefasst
Springer Spektrum, 2008

[HMU02] J. E. Hopcroft, R. Motwani, J. D. Ullman
Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie
Addison-Wesley Longman, 2002

[Neb12] M. Nebel
Formale Grundlagen der Programmierung
Springer Vieweg, 2012

[Sip96] M. Sipser
Introduction to the Theory of Computation
International Thomson Publishing, 1996

[Weg05] I. Wegener
Complexity Theory
Springer, 2005

[Koz97] D. Kozen
Automata and Computability
Springer, 1977

[Gol08] O. Goldreich
Computational Complexity
Cambridge University Press, 2008

Teil I.
Berechenbarkeit & Entscheidbarkeit

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 $|\alpha_1| \leq |\alpha_2|$. Intuitiv bedeutet diese Einschränkung dass die Produktionen Länge-erhaltend 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

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

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 $w \in \Sigma^*$
Frage: 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 $\leq |w|$.

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

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. □

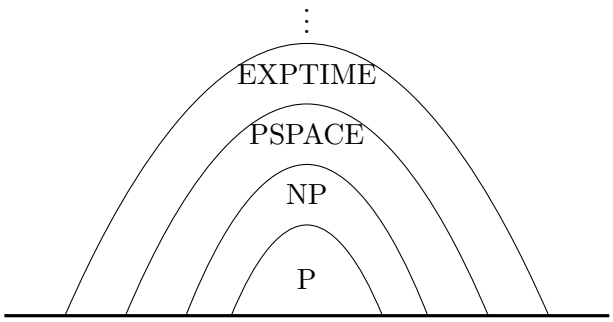
Das Wortproblem für kontextsensitive Sprachen ist schwer:

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

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

  also, dass die kontextsensitiven Sprachen genau den Problemen entsprechen, welche in Polynomialplatz 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:

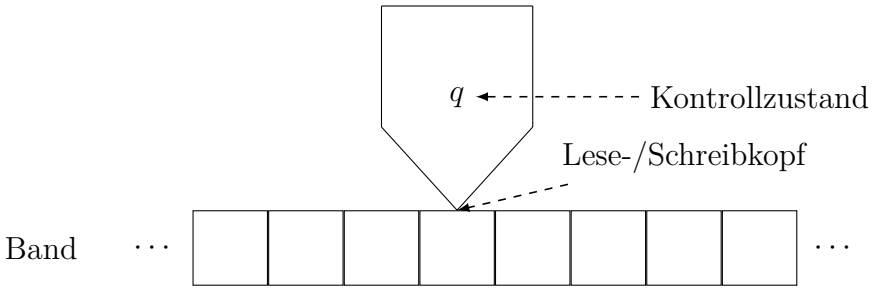
- Erreichbarkeit in Boolschen While-Programmen
- Erreichbarkeit in Multithreaded-Programmen

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

1.A Turing-Maschinen

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

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

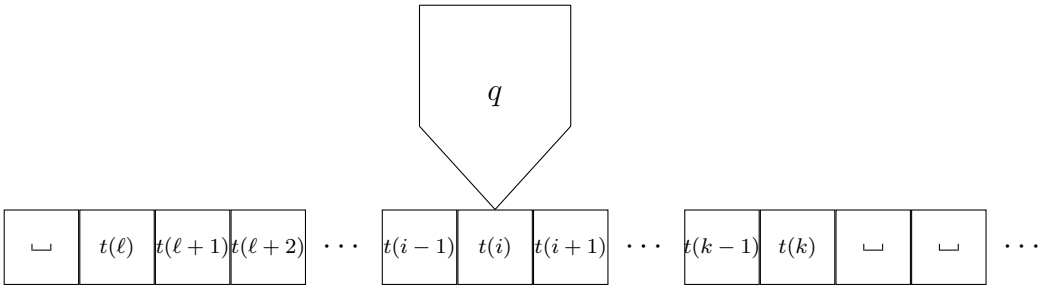
  ist, reden wir von einer **nicht-deterministischen Turingmaschine** (**NTM**).

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: $\exists \ell, k \colon \forall n < \ell, \forall n > k \colon t(n) = \sqcup$.



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

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

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 $c_0 \to \ldots \to c_k$ 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.

$$\mathcal{L}(M) = \{w \in \Sigma^* \mid M \text{ akzeptiert } w\} = \{w \in \Sigma^* \mid q_0 w \to^* uq'v \in \Gamma^* Q_F \Gamma^*\} \ .$$

1.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

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

für deterministische Mehr-Band-Turing-Maschinen bzw.

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

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 $\ldots \sqcup\sqcup\sqcup \ldots$

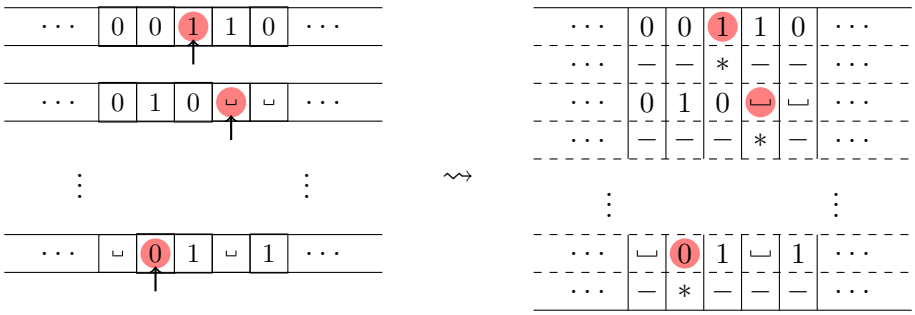
Auch wenn Mehr-Band-Turing-Maschinen auf den ersten Blick mächtiger als die Ein-Band Variante erscheinen, lassen sich, wie wir hier zeigen, mehrere Bänder mithilfe von nur einem simulieren.

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

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

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 der Leserin / 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.

1.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 $M = (Q, \Gamma, \Sigma \uplus \{\$_L, \$_R\}, q_0, \delta, Q_F)$.

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.

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

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 kontextsensitiven 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 auf 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 ($|\beta| \geq |\alpha|$), 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 $(\mathsf{act}_1, a_1), (\mathsf{act}_2, a_2), \ldots, (\mathsf{act}_n, a_n)$, wobei

$\mathsf{act}_1, \mathsf{act}_2 \ldots, \mathsf{act}_n$ 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
- $(\$_R, q, a)$, 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

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

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.

1.D Determinismus

Aus der Vorlesung „Theoretische Informatik 1“ wissen wir bereits, dass deterministische und nicht-deterministische endliche Automaten die gleiche Sprachen akzeptie-

ren (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 $\delta(q, a) = \emptyset$, 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.

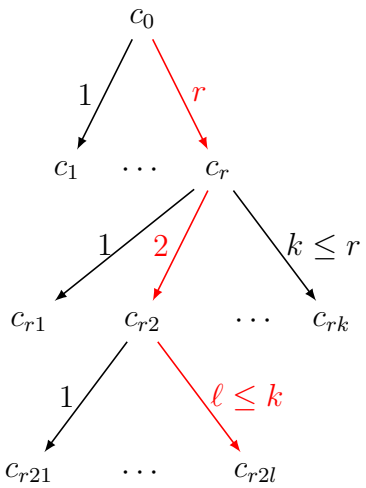
Beweis des Theorems:
Wie bereits erwähnt ist die Richtung von rechts nach links trivial. Für die andere Richtung bemerken wir, dass es zu jeder Konfiguration von M_1 zwar eventuell mehrere, aber in jedem Fall nur endlich viele Nachfolger gibt. Die Zahl der Nachfolger ist beschränkt durch

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

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 Immerman & Szelepcsényi

Mit dem Satz von Immerman & 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 Immerman, 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: Immerman & 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 $\mathsf{Graph}_{|w|}$, welcher wie folgt definiert ist.

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

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

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

Es gibt **keine** Ableitung $S \Rightarrow_G^* w$ gdw. es keinen Pfad von S zu w in $\mathsf{Graph}_{|w|}$ 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 $\mathsf{Graph}_{|w|}$) 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 $\mathsf{Graph}_{|w|}$ $(|N| + |\Sigma| + 1)^{|w|} = c^{|w|}$ 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, ist 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

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

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

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

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

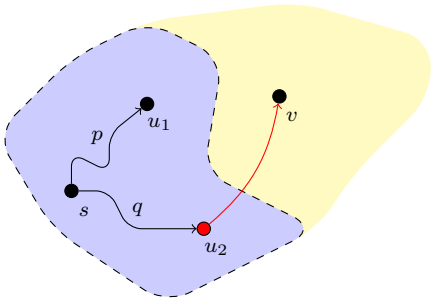
Man beachte:

- Es gilt $R(n) = N$, da jeder erreichbare Knoten auch mit einem einfachen Pfad (der insbesondere Länge $\leq n$ 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 $\leq i-1$ 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 $\leq i-1$ 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 nichtdeterministische 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. Berechen- und Entscheidbarkeit

3.A Berechenbarkeit

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

Das zugehörige Problem, das **Berechnungsproblem**, ist gegeben durch eine Funktion $f\colon M_1 \to M_2$. 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).

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:

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

Man könnte vermuten, dass diese Funktion nicht berechenbar ist, da unbekannt ist, ob $\mathsf{P} = \mathsf{NP}$ 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

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

   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

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

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

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

wobei $q_f \in Q_F$ und $\mathsf{bin}(n)$ 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

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

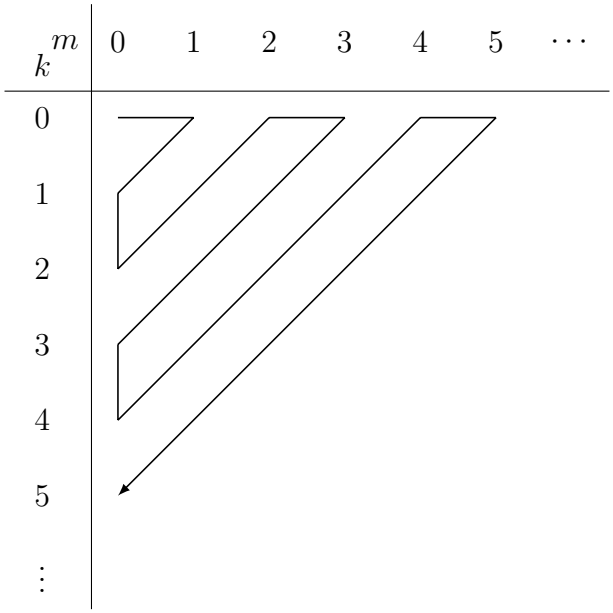
und

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

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 $((k+1)(|\Sigma|+m+3)\cdot 3)^{(k+1)(|\Sigma|+m+3)}$ 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.

3.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

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

berechenbar ist.

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

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

berechenbar ist.

3.15 Bemerkung
In der Literatur werden Sprachen $A \subseteq \Sigma^*$ oft mit Entscheidungsproblemen identifiziert. Dies liegt daran, dass eine solche Sprache ein Entscheidungsproblem, das sogenannte **Wortproblem von** A definiert.

Wortproblem zu A	
Gegeben:	Wort $w \in \Sigma^*$
Berechne:	Ist w Element von A?

Wenn die Sprache von der Form

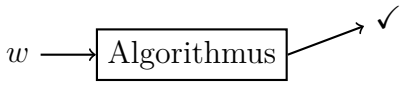
$$A = \{w \in \Sigma^* \mid w \text{ erfüllt die Bedingung für } A\}$$

ist, dann ist das Wortproblem zu A das Problem, zu entscheiden ob ein gegebenes Wort die definierende Eigenschaft der Wörter in A erfüllt.

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 Turing-Maschinen als Semi-Entscheider für Mengen wollen wir nun auch Turing-Maschinen als Entscheider für Mengen betrachten. Da diese Maschinen auf jeder Eingabe in endlich vielen Schritten halten sollen, statten wir sie mit einem speziellen abweisenden Zustand aus.

Statt Turing-Maschinen der Form $(Q, \Sigma, \Gamma, q_0, \delta, Q_F)$ betrachten wir im folgenden Turing-Maschinen der Form

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

Die Komponenten Q, Σ, Γ, q_0 und δ sind wie in Kapitel 1 definiert. Der Zustand $q_{acc} \in Q$ ist der eindeutig bestimmte **akzeptierende Zustand**. Gewissermaßen betrachten wir also eine Turing-Maschine, bei der $Q_F = \{q_{acc}\}$ eine einelementige Menge ist, die nur aus q_{acc} besteht. Dies stellt jedoch keine Einschränkung der Mächtigkeit dieser Maschinen dar. Der Zustand q_{rej} ist der eindeutig bestimmte **abweisende Zustand** von M.

Wir verlangen, dass die Transitionrelation von δ so beschaffen ist, dass die Zustände q_{acc} und q_{rej}, sobald sie einmal betreten wurden nicht wieder verlassen werden (siehe Bemerkung 3.9):

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

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

Analog 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.

Die Konfigurationen und die Transitionsrelation (deterministisch/nichtdeterministisch) eines Entscheiders sind wie inKapitel 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**.

Konfigurationen die akzeptierend oder abweisend sind, nennen wir **haltend**.

Die Sprache einer Turing-Maschine M ist die Menge der Eingaben, zu denen es eine Berechnung gibt, die nach endlich vielen Schritten eine akzeptierende Konfiguration erreicht,

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

Im Gegensatz zu den Turing-Maschinen aus Kapitel 1 können die Turing-Maschinen eine Eingabe nicht nur nach endlich vielen Schritten akzeptierend, sondern sie auch nach endlich vielen Schritten abweisen. Prinzipiell gibt es bei einer deterministischen Turing-Maschine wie hier definiert zu jeder Eingabe x drei Möglichkeiten:

1. Die Maschine akzeptiert nach endlich vielen Schritten. Dies ist der einzige Fall, in dem die Maschine die Eingabe x akzeptiert.
2. Die Maschine weist nach endlich vielen Schritten ab.
3. Die Maschine läuft unendlich lange, ohne eine Haltekonfiguration zu erreichen – Sie loopt.

Bei einer nicht-deterministischen Maschine gibt es außerdem noch die Möglichkeit, dass eine Berechnung in einer nicht-haltenden Konfiguration stecken bleibt, falls keine passende Transition zur Verfügung steht. In diesem Fall ist die Berechnung ebenfalls nicht akzeptierend. Man beachte außerdem, dass bei einer nicht-determinischen Maschine die Existenz einer akzeptierenden Berechnung genügt, damit die Eingabe in der Sprache der Maschine ist. Damit ein Wort abgewiesen wird, müssen alle Berechnungen zur Eingabe abweisend oder unendlich lange nicht-haltend sein oder stecken bleiben.

Wir erhalten folgendes Resultat.

3.16 Proposition
Eine Menge $A \subseteq \Sigma^*$ ist **semi-entscheidbar**, genau dann wenn es eine Turing-Maschine M mit $A = \mathcal{L}(M)$ gibt.

Es ist leicht zu sehen, dass sich eine Turing-Maschine, die die partielle charakteristischen Funktion von A berechnet (siehe Definition 3.8), sich in eine Maschine M mit $A = \mathcal{L}(M)$ transformieren lässt und umgekehrt.

Für das eben genannte Resultat wäre es nicht nötig gewesen, die Definition von Turing-Maschinen anzupassen. Dies ist erst dann nötig, wenn wir Probleme entscheiden wollen, also einen Algorithmus brauchen, der immer nach endlich vielen Schritten anhält.

3.17 Definition
Wir nennen eine Turing-Maschine $M = (Q, \Sigma, \Gamma, q_0, \delta, q_{acc}, q_{rej})$ **total** oder einen **Entscheider**, wenn jede Berechnung von M zu jeder Eingabe x nach endlich vielen Schritten hält.

Wie der Name suggeriert, gilt das folgende Resultat.

3.18 Proposition
Eine Menge $A \subseteq \Sigma^*$ ist **entscheidbar**, genau dann wenn es einen Entscheider M mit $A = \mathcal{L}(M)$ gibt.

Analog zum obigen Beweis lässt sich eine Turing-Maschine, die die totale charakteristische Funktion von A berechnet in einen Entscheider für A umwandeln und umgekehrt.

Es stellt sich nun natürlich die Frage, ob man für eine gegebene Turing-Maschine algorithmisch feststellen kann, ob sie ein Entscheider ist, also auf allen Eingaben hält. Wir werden später sehen, dass dieses Problem – genau wie viele andere Probleme, deren Eingabe eine Turing-Maschine ist – unentscheidbar ist, also nicht in endlicher Zeit von einem Computer gelöst werden kann.

3.19 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 $\mathcal{L}(M_{\leftrightarrow}) = \mathcal{L}(M)$.

3.20 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 $\leq |w|$ 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. □

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

Beweis:

"$\Rightarrow$" Klar.

"$\Leftarrow$" 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.

3.22 Definition
Eine Sprache $A \subseteq \Sigma^*$ ist **rekursiv aufzählbar**, wenn $A = \emptyset$ 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.

3.23 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 $A = \emptyset$ 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.

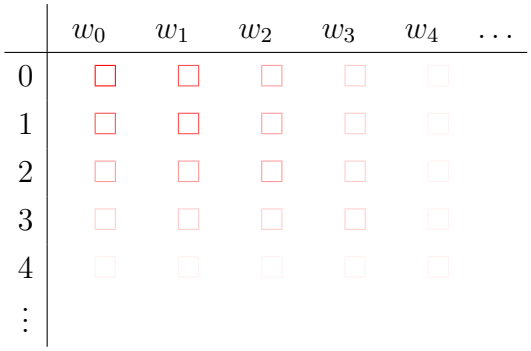
for $i = 0, 1, 2, \ldots$ **do**
  for $j = 0, 1, 2, \ldots, i$ **do**
    Berechne j-tes Wort w
    if M Eingabe w in höchstens i Schritten akzeptiert **then**
      Gebe w **aus**
    end if
  end for
end for

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: $n \in \mathbb{N}$

```
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
```

□

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

3.25 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 $g\colon \Sigma^* \to_p \Sigma_2^*$.

3.26 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:

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

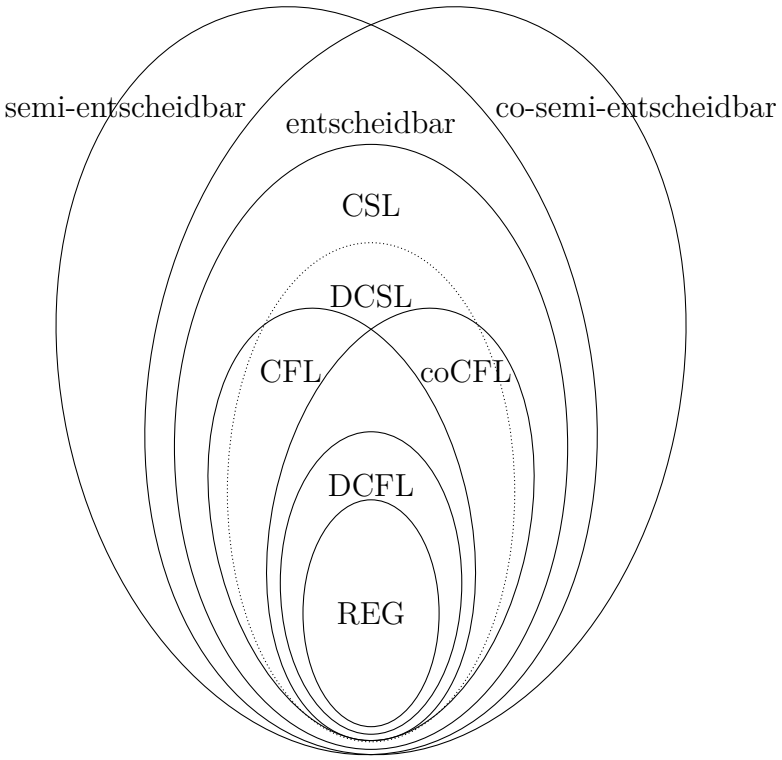
Durch Kombination der beiden Aussagen erhalten wir, dass A abzählbar ist, in dem wir alle Wörter in Σ^* abzählen und dabei die Wörter, die nicht in A liegen, überspringen. Beachte, dass dies kein Widerspruch dazu ist, dass die *Menge aller Sprachen* nicht abzählbar ist. (Analog für die natürlichen Zahlen: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

Übersicht: Inzwischen haben wir eine Reihe unterschiedlicher Sprachklassen kennengelernt. Diese werden definiert durch verschiedene Einschränkungen an formale Grammatiken, von Typ 0 (semi-entscheidbar) und Typ 1 (kontextsensitiv) über Typ 2 (kontextfrei) bis Typ 3 (regulär). Wir kennen alternative Modelle für diese Sprachklassen: Turing-Maschinen, linear-beschränkte Turing-Maschinen, Pushdown-Automaten und Endliche Automaten.

Jede dieser Automaten-Klassen besitzt eine deterministische Variante. Interessant ist hier, dass die Sprachklasse deterministischer LBAs heute noch nicht vollständig eingeordnet werden konnte, während die deterministischen PDAs eine echt kleinere Sprachklasse beschreiben (mehr dazu in Kapitel 6) und DTMs und DFAs nicht weniger mächtig sind als ihre nicht-deterministischen Varianten.



Diese Abbildung verdeutlicht nicht nur die Ordnung dieser Sprachklassen, sondern auch ihre Abschluss-Eigenschaften über dem Komplement. Uns ist bereits bekannt, dass sich endliche Automaten komplementieren lassen, also sind co-reguläre Sprachen genau die regulären Sprachen. Gleichzeitig besagt Theorem 2.1, dass eine Sprache genau dann co-kontextsensitiv ist, wenn sie kontextsensitiv ist.

4. Unentscheidbarkeit

Das Ziel dieses Kapitels ist es zu beweisen, dass es Probleme gibt, die **semi-entscheidbar** aber **unentscheidbar** sind.

In Kapitel 3 haben wir nicht-konstruktiv bewiesen, dass es nicht-berechenbare Funktionen gibt. Analog ließe sich beweisen, dass es Entscheidungsprobleme gibt, die nicht entscheidbar sind. Der Beweis ist analog zum Beweis von Theorem 3.12 und sei der Leserin / dem Leser als Übungsaufgabe überlassen.

Dieser Beweis hat allerdings zwei entscheidende Mängel. Zum einen zeigt er nur, dass es nicht-entscheidbare Probleme gibt. Es könnte allerdings sein, dass alle semi-entscheidbaren Probleme auch entscheidbar sind; man also jede beliebige Turing-Maschine in einen Entscheider umwandeln kann. Dies ist allerdings nicht der Fall. Zum anderen liefert der Beweis kein konkretes Problem, das unentscheidbar ist, und damit insbesondere auch keine Methode, um konkrete Probleme darauf zu prüfen, ob sie entscheidbar sind.

In diesem Kapitel wollen wir eine Liste konkret definierter Probleme betrachten und ihre Unentscheidbarkeit beweisen. Wir betrachten zunächst das sogenannte Akzeptanzproblem.

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

Wir werden zunächst beweisen, dass dieses Problem semi-entscheidbar, aber nicht entscheidbar ist. Es ist interessant, dass es sich hierbei um ein Verifikationsproblem handelt, also ein Problem, bei dem die Eingabe selbst ein Programm (eine Turing-Maschine) ist. Danach werden wir eine Technik kennen lernen, um ausgehend von der Unentscheidbarkeit des Akzeptanzproblems weitere Probleme als unentscheidbar nachzuweisen.

1. Gemäß unserer Definition sind Entscheidungsprobleme Menge von Wörtern. Die Eingabe des Akzeptanzproblems ist aber kein Wort, sondern besteht (unter anderem) aus einer Turing-Maschine. Um das Akzeptanzproblem als Sprache aufzufassen, müssen wir Turing-Maschinen als Wort **kodieren**.

2. Wir definieren eine **universellen Turing-Maschine**, deren Sprache das Akzeptanzproblems ist; sie semi-entscheidet es also. Dazu simuliert die universelle Turing-Maschine eine gegebene Turing-Maschine auf einer gegebenen Eingabe.

3. Nun beweisen wir, dass das Akzeptanzproblem unentscheidbar ist. Für den Beweis benötigen wir sowohl die universelle Turing-Maschine als auch das bereits bekannte Prinzip der Diagonalisierung.

4. 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 des Akzeptanzproblem die Unentscheidbarkeit vieler anderer Probleme zu schlussfolgern.

4.A Kodierungen von Turing-Maschinen

Um das Akzeptanzproblem als Sprache, also seine Ja-Instanzen als Menge von Wörtern, aufzufassen, müssen wir Turing-Maschinen als Wörter kodieren. Wir werden zunächst zu einer Turing-Maschine M die Kodierung $\langle M \rangle \in \{0,1\}^*$ definieren. Anschließend betrachten wir die umgekehrte Richtung und definieren zu jedem Binärstring $w \in \{0,1\}*$ eine Turing-Maschine M_w. Hierbei soll natürlich $M_{\langle M \rangle} = M$ gelten.

Um eine Turing-Maschine M zu kodieren, gehen wir in zwei Schritten vor. Wir definieren zunächst eine Kodierung $v_m \in \{0,1,\#\}^*$, die zusätzlich zu 0 und 1 auch # als Trennsymbol enthält, und anschließend definieren wir die gesuchte Kodierung $\langle M \rangle \in \{0,1\}^*$. Sei $M = (Q, \Sigma, \Gamma, q_0, \delta, q_{acc}, q_{rej})$ eine gegebene Turing-Maschinen. Wir machen einige Annahmen, die uns die Kodierung von M erleichtern:

- M ist deterministisch,
- M ist eine 1-Band-Maschine
- $Q = \{q_0, q_1, q_2, \ldots, q_n \mid \quad\}$ mit

  q_0 Startzustand,

  $q_1 = q_{acc}$,

  $q_2 = q_{rej}$,
- $\Sigma = \{0,1\}$
- $\Gamma = \{a_0, a_1, a_2, \ldots, a_m \mid \quad\}$ mit

  $a_0 = \sqcup$,

  $a_1 = 1$ und

  $a_2 = 0$.

Wir gehen also davon aus, dass die Eingaben der Maschine Binärstrings sind, und dass die Kontrollzustände und Bandsymbole durchnummeriert sind. Die erste Bedingung lässt sich durch das Betrachten einer Binärkodierung erzwingen. Die zweite Bedingung erzwingt man, in dem man die Kontrollzustände und Bandsymbole entsprechend umbenennt. Man beachte, dass diese Umbenennung keine Auswirkungen auf das Verhalten der Maschine hat, also darauf, welche Eingaben sie nach welcher Anzahl Schritte akzeptiert.

Nicht-deterministische Maschinen kann man analog zur hier vorgestellten Konstruktion kodieren. Für Mehrband-Maschinen kann man entweder die Konstruktion anpassen, oder sie zunächst zu einer 1-Band-Maschine konvertieren.

Eine Maschine der obigen Form ist eindeutig bestimmt durch die Anzahl der Zustände $n+1$, die Anzahl der Bandsymbole $m+1$ und ihre Transitionsrelation. Wir betrachten nun, wie sich die Transitionsrelation kodieren lässt, beginnend mit den einzelnen Transitionen.

Es sei

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

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

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

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 ganz δ zu kodieren, schreiben wir alle Abbildungsvorschriften von δ hintereinander.

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

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

Wir behaupten, dass v_M die Turing-Maschine M eindeutig beschreibt: Die Transitionsfunktion ist in v_M kodiert, und die Anzahl der Zustände und der Bandsymbole lassen sich als die größten Werte für bin(i) und bin(j) extrahieren.

Nun haben eine Kodierung von M als Wort $v \in \{0,1,\#\}^*$. Um das Symbol $\#$, welches die Transitionen und deren Bestandteile voneinander trennt, loszu-

werden, wenden wir einen Homomorphismus an. Betrachte den Homomorphismus $h\colon \{0,1,\#\}^* \to \{0,1\}$, der auf den Buchstaben wie folgt definiert ist.

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

4.1 Definition
Zu jeder Turing-Maschine M sei $\langle M\rangle \in \{0,1\}^*$ das Wort $h(v_m)$, das sich ergibt, indem wir M wie oben beschrieben als Wort $v \in \{0,1,\#\}^*$ kodieren und dann Homomorphismus h 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
Durch die Umbenennung der Zustände und Symbole kann es passieren, dass zwei unterschiedliche Turing-Maschinen M, M' die gleiche Kodierung $\langle M\rangle = \langle M'\rangle$ haben. Es gilt dann jedoch, dass das Verhalten der beiden Maschinen gleich ist; insbesondere gilt $\mathcal{L}(M) = \mathcal{L}(M')$.

Nun können wir jede Turing-Maschine als Wort über $\{0,1\}$ kodieren. Für die andere Richtung möchten wir zu jedem Binärstring $w \in \{0,1\}^*$ eine Turing-Maschine M_w definieren.

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

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

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

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

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

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

Darüber hinaus ist M_w berechenbar: Gegeben eine Kodierung w ist es möglich zu entscheiden, ob w die valide Kodierung einer Turing-Maschine ist. In diesem Fall kann man den Homomorphismus h invertieren, um das Wort $h^{-1}(w) = v_{M_w}$ zu erhalten, welches die Transitionsfunktion von M_w kodiert. Aus diesem extrahiert man dann die Beschreibung von M_w. Wir werden im nächsten Unterkapitel näher hierauf eingehen.

Unter Zuhilfenahme der Kodierung von Turing-Maschinen lässt sich das Akzeptanzproblem als Wortproblem auffassen. Die bisherige Fassung erwartet als Eingabe eine Turing-Maschine.

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

Wir können nun Maschine M in der Eingabe durch ihre Kodierung ersetzen und erhalten die folgende Variante.

Akzeptanzproblem (ACCEPT,ACC)	
Gegeben:	Zwei Binärstrings $w, x \in \{0,1\}^*$
Frage:	Akzeptiert die durch w kodierte Maschine M_w Eingabe x?

Wenn wir die zwei Wörter mit einem Trennsymbol verknüpfen, erhalten wir die gewünschte Charakterisierung des Akzeptanzproblems als Sprache, die wir im Rest des Kapitels nutzen werden.

$$\mathsf{ACCEPT} = \{w\#x \in \{0,1\}^*\#\{0,1\}^* \mid x \in \mathcal{L}(M_w)\} \ .$$

4.5 Bemerkung
Die Kodierung $\langle M \rangle$ einer Turing-Maschine M wird auch als ihre **Gödelisierung** bezeichnet. Dieser Begriff ist nach dem Mathematiker **Kurt Gödel** (1906-1972) benannt. In seinem Beweis des ersten Unvollständigkeitssatzes 1931, ein wichtiges Resultat aus der Prädikatenlogik, hat er eine Funktion definiert, welche Formeln natürlichen Zahlen zuordnet. Zu einer Formel F erhält man also den Funktionswert $\varphi(F) \in \mathbb{N}$ die Gödelnummer oder Gödelisierung von F. Gödel hatte hiermit entdeckt, wie man komplizierte Objekte (wie z.B. Formeln oder Turing-Maschinen) zu einfachen Objekten (Zahlen oder Wörtern) kodiert. Analog zu Gödels Funktion lassen sich auch Turing-Maschinen zu Zahlen kodieren. Diese Kodierung ist allerdings aufwändiger und bringt gegenüber der hier definierten Kodierung zu Binärstrings keine für uns relevanten Vorteile.

4.B Die universelle Turing-Maschine

In diesem Kapitel werden wir zeigen, dass das Akzeptanzproblem semi-entscheidbar ist. Dazu definieren wir eine Turing-Maschine U mit $\mathcal{L}(U) = \mathsf{ACCEPT} = \{w\#x \mid x \in \mathcal{L}(M_w)\}$. Die Bedeutung dieser sogenannten **universellen Turing-Maschine (UTM)** reicht allerdings weit über die Tatsache, dass sie das Akzeptanzproblem semi-entscheidet, hinaus.

Um das Akzeptanzproblem zu semi-entscheiden, definieren wir U so, dass sie bei Eingabe von $w\#x$ die durch ihre Kodierung gegebene Maschine M_w auf Eingabe x **simuliert**. Die Schwierigkeit hierbei ist, dass Zustände und Transitionen von U 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.

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

$$\mathcal{L}(U) = \mathsf{ACCEPT} \ .$$

Beweisskizze für Theorem 4.6:
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.

Betrachte eine Eingabe $w\#x$. U überprüft zunächst, ob w die valide Kodierung einer Turing-Maschine ist. Wenn dies nicht der Fall ist, ist $M_w = M_\emptyset$ die Maschine mit leerer Sprache. Es gilt $x \notin \mathcal{L}(M_\emptyset) = \emptyset$; dementsprechend weist U ab. Falls w die valide Kodierung einer Maschine M_w ist, simuliert M_w auf Eingabe x Schritt-für-Schritt, bis M_w hält.

Eine Berechnung von U läuft wie folgt ab:

1. Überprüfe, ob w das Bild eines Wortes $v \in \{0, 1, \#\}^*$ gemäß des Homomorphismus ist. Wenn ja, speichere v auf dem zweiten Band, dem **Programmband**. Wenn nein, weise ab.

2. Überprüfe, ob v von der Form

   $$v = \prod_{i=0}^{n} \prod_{j=0}^{m} w_{i,j,i',j',d}$$

   ist, also eine Konkatenation korrekt kodierter Transitionen. Außerdem muss es zu jeder Kombination aus Zustand und Symbol eine entsprechende Transition geben. Wenn dies nicht der Fall ist, weise ab.

3. Schreibe 0 aufs Zustandsband, denn $0 = \text{bin}(0)$ ist die Binärkodierung des Index' des Startzustands q_0 von M_w.

4. Schreibe x' aufs Datenband, wobei

   $$x' = x'_1 \# x'_2 \# \ldots \# x'_{|x|}$$

   und $x'_i = 1$, wenn $x_i = 1$ und $x'_i = 10 = \text{bin}(2)$ wenn $x_i = 0$. (Beachte, dass wir in der Kodierung einer Turing-Maschine $a_2 = 0$ angenommen hatten.)

5. Simuliere M_w Schritt-für-Schritt

   - Lese Zustand q_i vom Zustandsband (kodiert durch seinen Index).
   - Lese das aktuelle Bandsymbol a_j vom Datenband.
   - Suche im Programmband nach der Kodierung

     $$v_{i,j,i',j',d} \ ,$$

     welche die Transitionsregel

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

kodiert. Hierbei sind i und j bekannt, i', j' und d werden gesucht.

- Ersetze den Inhalt des Zustandsbands durch $\text{bin}(i')$.
- Ersetze die Kodierung des aktuellen Bandsymbols $\text{bin}(j)$ auf dem Datenband durch $\text{bin}(j')$.
- Bewege den Kopf auf dem Datenband in die durch $d \in \{L, R, N\}$ spezifizierte Richtung.

6. Überprüfe den Inhalt des Zustandsbands $\text{bin}(i)$. Wenn $i = 1$, dann akzeptiere (denn $q_1 = q_{acc}$ in M_w). Wenn $i = 2$, dann weise ab (denn $q_2 = q_{rej}$ in M_w.) Ansonsten gehe zurück zu Schritt 5.

Man beachte, dass die tatsächliche Umsetzung etwas komplizierter ist, als hier angedeutet. Beispielsweise können der Index des Zustands und das aktuelle Bandsymbol aus unbeschränkt vielen Symbolen bestehen, da die Anzahl der Zustände und Bandsymbole in M_w nicht beschränkt sind. Da Maschine U nur über eine feste Anzahl Kontrollzustände verfügt, müssen Zustandsband und Datenband on-the-fly mit dem Programmband abgeglichen werden, um die richtige Transition zu finden.

Die Berechnung von U hält bzw. akzeptiert genau dann, wenn die simulierte Berechnung von M_w auf Eingabe x hält bzw. akzeptiert. Es gilt also wie gewünscht $\mathcal{L}(U) = \mathsf{ACCEPT}$. □

4.7 Bemerkung: Einfluss der universellen Turing-Maschine auf die Entwicklung des Computers

Wir argumentieren, dass die universelle Turing-Maschine das theoretische Äquivalent zum Konzept des modernen Computers ist.

Bereits seit dem 17. Jahrhundert gab es Rechenmaschinen im kaufmännischen und militärischem Bereich, um mathematische Operationen (Additionen, Multiplikationen) durchzuführen. Diese Maschinen sind allerdings keine Computer im engeren Sinne – sie arbeiten ein festgelegtes Programm ab. Analog dazu sind viele Turing-Maschinen keine Computer.

Der Unterschied dazu sind moderne Computer **programmierbar**: Bei diesen können nicht nur die Eingabedaten, sondern auch das auszuführende Programm frei gewählt werden. Bei frühen Computern war dies nur mit Tricks (Konrad Zuses Z3 ab 1941) oder durch Änderung der Verkabelung (ENIAC ab 1946) möglich. EDVAC war 1949 der erste amerikanische Computer, in dem auszuführende Programm genau wie die Eingabedaten binär codiert elektronisch gespeichert wurde („stored-program computer“). Diese Idee wurde bereits von Turing – mehr als 10 Jahre vor

der praktischen Umsetzung vorweggenommen. 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. Turings Idee einer universellen Turing-Maschine war also von fundamentaler Bedeutung für die Erfindung von dem, was wir heute als **Computer** verstehen. Die beim EDVAC verwendete **von-Neumann-Architektur** ist bis heute die Grundlage für den Aufbau von Computern.

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**.

4.C Unentscheidbarkeit des Akzeptanzproblems

Mithilfe der universellen Turingmaschine und einem Diagonalisierungsverfahren, werden wir nun die zweite Hälfte von Turings berühmtem Resultat nachweisen: Das Akzeptanzproblem ist semi-entscheidbar, aber nicht entscheidbar.

4.8 Bemerkung
Turing betrachtete eigentlich nicht das Akzeptanzproblem, sondern das mit ihm eng verwandte **Halteproblem**. Darauf werden wir im nächsten Unterkapitel zurückkommen.

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

Beweis:
Wir haben bereits gezeigt, dass ACCEPT die Sprache der universellen Turing-Maschine und damit semi-entscheidbar ist. Der andere Teil der Aussage ist das wichtigere Resultat: Es zeigt, dass die Klasse der entscheidbaren Sprachen eine echte Teilklasse der semi-entscheidbaren Sprachen ist.

Wir nehmen an, ACCEPT sei entscheidbar, und leiten einen Widerspruch her. Sei E ein Entscheider mit $\mathcal{L}(E) = \mathsf{ACCEPT}$. Eine Eingabe $w\#x$ wird von E 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 E konstruieren. Dafür simuliert D zunächst E auf der Eingabe $w\#w$ und invertiert am Ende die Antwort von E, also

- falls E abweist, akzeptiert D,
- falls E 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 E die Eingabe $\langle D \rangle \# \langle D \rangle$ ab. Da $\mathcal{L}(E) = \mathsf{ACCEPT}$, bedeutet dies, dass $\langle D \rangle \notin \mathcal{L}(D)$, D also ihre Kodierung nicht akzeptiert – ein Widerspruch.

Fall 2: D weist $\langle D \rangle$ ab:
Nach Konstruktion von D akzeptiert E also die Eingabe $\langle M \rangle \# \langle M \rangle$. Da $\mathcal{L}(E) = \mathsf{ACCEPT}$, bedeutet dies, dass $\langle D \rangle \in \mathcal{L}(D)$, D ihr Kodierung also akzeptiert – ein Widerspruch.

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

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

	M_1	M_2	M_3	...	D	...
$\langle M_1 \rangle$	accept	accept	reject		reject	
$\langle M_2 \rangle$	reject	reject	reject		accept	
$\langle M_3 \rangle$	reject	accept	accept		reject	
⋮						
$\langle D \rangle$	accept	accept	reject		?	
⋮						

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 Einträge entlang der Diagonale dieser Tabelle: Wenn Maschine M_i ihre eigene Kodierung $\langle M_i \rangle$ akzeptiert (der Eintrag auf der Diagonale also „accept“ ist), dann weist Maschine D die Eingabe $\langle D \rangle$ ab. Der Widerspruch wird bei dem Eintrag „?“ auf der Diagonale hergeleitet: Gemäß der Konstruktion von D muss der Eintrag für Maschine D und Eingabe $\langle D \rangle$ nun sein eigenes Gegenteil sein.

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 (SELF-ACCEPT,S-ACCEPT,S-ACC)	
Gegeben:	Eine Turing-Maschine M
Frage:	Akzeptiert M die Eingabe $\langle M \rangle$?

Oder als Sprache:

$$\mathsf{SELF\text{-}ACCEPT} = \{w \in \{0,1\}^* \mid w \in \mathcal{L}(M_w)\} \ .$$

4.11 Korollar
Das spezielle Akzeptanzproblem SELF-ACCEPT ist semi-entscheidbar, aber nicht entscheidbar.

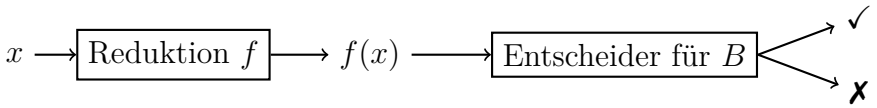
Damit ist gezeigt, dass die Schwierigkeit (Unentscheidbarkeit) des Halteproblems nicht daher rührt, dass wir eine beliebige Eingabe x erlauben.

4.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 Akzeptanzproblems 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. Angenommen B wäre entscheidbar, und f wäre eine Reduktion von A auf B. Dann wäre es möglich, einen Entscheider für B mit f zu einem Entscheider für A zu kombinieren; ein Widerspruch zur Unentscheidbarkeit von A:



Gegeben eine Instanz x von Problem A, für die entschieden werden soll, ob $x \in A$ gilt, wenden wir zunächst die Reduktion f an, um eine Instanz $f(x)$ von B zu berechnen. Nun verwenden wir den Entscheider für B (von welchem wir annehmen, dass er existiert), um zu entscheiden, ob $f(x) \in B$ gilt. Unsere Reduktion soll die Eigenschaft haben, dass dies äquivalent dazu ist, dass $x \in A$ gilt.

Um diese Idee zu formalisieren, führen wir Many-One-Reduktionen ein.

4.12 Definition
Es seien $A \subseteq \Sigma_1^*, B \subseteq \Sigma_2^*$ Sprachen. Eine Funktion $f\colon \Sigma_1^* \to \Sigma_2^*$ heißt **(Many-One-)Reduktion von A auf B**, wenn sie total und berechenbar ist und für alle $x \in \Sigma_1^*$ die Äquivalenz

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

gilt.

Wenn es eine solche Reduktion gibt, sagen wir, dass Problem A **(many-one-)reduzierbar** auf Problem B ist und schreiben $A \leq B$.

Intuitiv besagt $A \leq B$, dass Problem A leichter als Problem B ist. Genauer gesagt ist Problem B *mindestens so schwer wie* A. Dies wird durch das folgende Lemma formalisiert. Sein Beweis formalisiert die oben genannte und als Graphik dargestellte Konstruktion.

4.13 Lemma
Wenn $A \leq B$ 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 M_f die totale Funktion f berechnet, terminiert die Berechnung von $f(x)$ nach endlichen vielen Schritten. Da M_B ein Entscheider ist, terminiert auch die simulation von M_B nach endlich vielen Schritten. Die konstruierte Maschine M_A hält also immer. Da außerdem $x \in A$ 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. □

Das Lemma kann verwendet werden, um (Semi-)Entscheidbarkeit zu zeigen, in dem wir ein neues Problem auf ein bereits als (semi-)entscheidbar bekanntes Problem reduzieren. Häufiger verwendet man jedoch die Kontraposition des obigen Lemmas, um die Unentscheidbarkeit von Problemen zu zeigen.

4.14 Korollar
Wenn $A \leq B$ 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. □

4.15 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.

4.16 Bemerkung
In der Literatur werden zum Teil andere Sorten Reduktionen betrachtet, z.B. sogenannte Turing-Reduktionen. Wenn wir $A \leq B$ mittels Turing-Reduktionen definieren würden, würde nur eine schwächere Variante von Lemma 4.13 gelten. Turing-Reduktionen sind zu grob, um Probleme auf Semi- und Co-semi-entscheidbarkeit zu untersuchen. Außerdem wollen wir diese Sorte Reduktion im nächsten Teil der Vorlesung zu Komplexitätstheorie erneut verwenden.

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

Akzeptanz der leeren Eingabe ($\mathsf{ACCEPT}_\varepsilon$)	
Gegeben:	Eine Turing-Maschine M
Frage:	Akzeptiert Maschine M die leere Eingabe ε?

Als Sprache lässt sich dieses Problem wie folgt beschreiben:

$$\mathsf{ACCEPT}_\varepsilon = \{w \in \{0,1\}^* \mid \varepsilon \in \mathcal{L}(M_w)\} \ .$$

$\mathsf{ACCEPT}_\varepsilon$ ist unentscheidbar, dies beweisen wir, in dem wir das allgemeine Halteproblem HP auf HP_ε reduzieren und dann das obige Lemma in Kontraposition anwenden.

4.17 Lemma
$\mathsf{ACCEPT} \leq \mathsf{ACCEPT}_\varepsilon$.

Beweis:
Wir definieren eine Reduktion f, die ACCEPT auf $\mathsf{ACCEPT}_\varepsilon$ reduziert. Die Reduktion ist von der Form $f\colon w\#x \mapsto w'$. Sei $w\#x$ eine Eingabe für das allgemeine Akzeptanzproblem. Dann ist w' die Kodierung einer Turing-Maschine, die sich wie folgt verhält:

- Lösche die Eingabe (d.h. überschreibe den Inhalt des Eingabeband mit $\sqcup$),
- Schreibe x auf das Eingabeband, und dann
- simuliere M_w und akzeptiere, bzw. weise ab, wenn M_w akzeptiert bzw. abweist.

Man beachte, dass w und x hierbei fixiert sind.

Die Funktion f ist offensichtlich total. Man kann nachweisen, dass f tatsächlich berechenbar ist. Wir müssen dazu zeigen, dass wir die Kodierung von Turing-Maschinen berechnen können, die die oben genannten Schritte umsetzen. Anschließend kann man die drei Teile zur Kodierung einer einzigen TM verknüpfen. Um die Eingabe zu löschen, wird eine konstante Anzahl Kontrollzustände und Transitionsregeln benötigt. Das selbst gilt für den zweiten Schritt, da wir annehmen, dass x fest ist. Für die Simulation von M_w können wir die universelle Turing-Maschine U, genauer gesagt ihre Kodierung $\langle U \rangle$ angewandt auf die Eingabe $w = \langle M_w \rangle$, verwenden.

Es verbleibt zu zeigen, dass $w\#x \in \mathsf{ACCEPT}$ genau dann wenn $w' \in \mathsf{ACCEPT}_\varepsilon$. Wenn $w\#x \in \mathsf{ACCEPT}$, dann akzeptiert M_w Eingabe x. Nach Konstruktion von w' akzeptiert dann $M_{w'}$ **alle** Eingaben, insbesondere also ε und es gilt $w' \in \mathsf{ACCEPT}_\varepsilon$. Für die andere Richtung nehmen wir $w\#x \notin \mathsf{ACCEPT}$ an. Dann weist $M_{w'}$ alle Eingaben ab, insbesondere auch ε und es gilt wie gewünscht $w' \notin \mathsf{ACCEPT}_\varepsilon$. □

Als weiteres Beispiel betrachten wir das **Halteproblem** und seine Variationen. Genau wie das Akzeptanzproblem ist dieses unentscheidbar.

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

a) Das **(allgemeine) Halteproblem**

$$\mathrm{HP} = \{w\#x \in \{0,1\}^*\#\{0,1\}^* \mid M_w \text{ hält auf Eingabe } x\} .$$

b) Das **spezielle Halteproblem**

$$\text{SELF-HP} = \{w \in \{0,1\}^* \mid M_w \text{ hält auf Eingabe } w\} .$$

c) Das **Halteproblem auf der leeren Eingabe**

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

Beweis:
Wir beweisen exemplarisch die Semi-Entscheidbarkeit von HP und die Unentscheidbarkeit von HP_ε. Die anderen Beweise sind analog. Für beide Richtungen verwenden wir Reduktionen und Lemma 4.13 bzw. seine Kontraposition, Korollar 4.14.

- Wir zeigen $\mathsf{HP} \leq \mathsf{ACCEPT}$.

  Wir definieren eine Reduktion $f\colon w\#x \mapsto w'\#x'$. Hierbei gilt $x = x'$, wir verändern den zweiten Teil der Eingabe also nicht. w' ist hierbei die Kodierung einer Turing-Maschine $M_{w'}$, die sich auf einer Eingabe zunächst wie M_w verhält und dann akzeptiert, wenn M_w hält (egal ob M_w akzeptiert oder nach endlich vielen Schritten abweist).

  Die Funktion ist total und berechenbar. Letzteres folgt daraus, dass man lediglich die gegebene Kodierung w von M_w so anpassen muss, dass alle Transitionen die in den abweisenden Zustand q_{rej} führen würden, stattdessen in den akzeptierende Zuststand q_{acc} führen.

  Wir argumentieren, dass die Äquivalenz $w\#x \in \mathsf{HP}$ gdw. $w'\#x' \in \mathsf{ACCEPT}$ gilt. Wenn $w\#x \in \mathsf{HP}$, dann hält M_w auf Eingabe x. Gemäß unserer Konstruktion akzeptiert dann $M_{w'}$ Eingabe x, und es gilt $w'\#x \in \mathsf{ACCEPT}$. Die eben genannten Implikation gelten auch in die andere Richtung, sind also Äquivalenzen. Damit ist die Äquivalenz bewiesen.

- Wir zeigen $\mathsf{ACCEPT}_\varepsilon \leq \mathsf{HP}_\varepsilon$.

  Wir definieren eine Reduktion $f\colon w \mapsto w'$. Hierbei ist w' die Kodierung einer Turing-Maschine, die sich zunächst auf ihrer Eingabe wie M_w verhält. Wenn M_w akzeptiert, dann akzeptiert $M_{w'}$ ebenfalls. Wenn M_w nach endlich vielen Schritten abweisen würde, dann betritt $M_{w'}$ eine Endlosschleife und hält nicht. Wenn M_w nicht hält, ergibt sich automatisch, dass auch $M_{w'}$ nicht hält.

  Diese Funktion ist total und berechenbar. Letzteres folgt daraus, dass man w' erhält indem man zu w einen neuen Kontrollzustand hinzufügt, der nicht verlassen werden kann – also eine Endlosschleife darstellt – und alle Transitionen in w, die zu q_{rej} führen würden in diesen Zustand umleitet.

  Wenn $w \in \mathsf{ACCEPT}_\varepsilon$, dann akzeptiert M_w Eingabe ε. Nach Konstruktion akzeptiert dann auch $M_{w'}$ Eingabe ε, was insbesondere bedeutet, dass $M_{w'}$ hält. Es gilt $w' \in \mathsf{HP}_\varepsilon$. Wenn $w \notin \mathsf{ACCEPT}_\varepsilon$, dann akzeptiert M_w Eingabe ε nicht. Entweder weist M_w nach endlich vielen Schritten ab, oder M_w hält nicht. In beiden Fällen hält $M_{w'}$ auf Eingabe ε nicht und es gilt $w' \notin \mathsf{HP}_\varepsilon$ wie gewünscht.

□

4.19 Bemerkung

Turing hatte ursprünglich in seinem berühmten Resultat von 1936 nicht das Akzeptanzproblem, sondern das Halteproblem betrachtet. Die Beweise der Semi-Entscheidbarkeit (unter Verwendung der universellen TM) und der Unentscheidbarkeit (mittels Diagonalisierung) sind in beiden Fällen sehr ähnlich. Im Falle des Akzeptanzproblems sind die Beweise minimal einfacher, weshalb wir diese hier präsentiert haben. Das Halteproblem stellt allerdings besser heraus, wo die Unentscheidbarkeit von Eigenschaften von Turing-Maschinen herrührt: Es ist im Allgemeinen nicht möglich, den Fall einer Endlosschleife (also des Nicht-Haltens) vom Falle des Haltens zu unterscheiden. Es gibt also keinen Algorithmus, der nach dem endlichen Präfix einer Berechnung beurteilt, ob die Berechnung irgendwann halten wird, oder ob sie unendlich lange läuft.

Viele weitere Probleme lassen sich mit den hier vorgestellten Techniken als unentscheidbar nachweisen. Dies gilt insbesondere für die folgenden beiden Probleme, die nicht nur Unentscheidbar, sondern sogar weder semi-entscheidbar, noch co-semi-entscheidbar sind.

Totalitätsproblem
Gegeben: Eine Turing-Maschine M.
Frage: Hält M auf allen Eingaben?

Universalitätsproblem
Gegeben: Eine Turing-Maschine M.
Frage: Akzeptiert M alle Eingaben, d.h. gilt $\mathcal{L}(M) = \{0,1\}^*$?

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.

5.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 $(x_1, y_1), \ldots, (x_k, y_k)$
Frage:	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 $(x_1, y_1), \ldots, (x_k, y_k)$
Frage:	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 $\leq$ 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 $w = a_1 \ldots a_m \in \Sigma^*$ definieren wir drei Varianten:

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

Zur gegeben Instanz K konstruieren wir nun die Instanz

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

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.

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

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

„$\Leftarrow$“ Wenn Instanz $f(K)$ eine Lösung hat, dann hat sie auch eine Lösung mit minimaler Länge. Sei $i_1 \ldots i_n \in \{1, \ldots, k+2\}^*$ 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 $i_1, \ldots, i_n$ 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 $i_2 \ldots i_{n-1} \in \{2, \ldots, k+1\}^*$. Die Sequenz

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

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 $\leq$ 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 $x = a_1 \ldots a_n \in \Sigma^*$ die Eingabe für diese Maschine ist.

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

$$c_0 \to \ldots \to c_t$$

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

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

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 $\Gamma \cup Q \cup \{\#\}$. 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.

$$
\begin{array}{ll}
x\text{-Sequenz}: & \overbrace{\#}^{1}\,\overbrace{q_0a_1}^{2}\,\overbrace{a_2}^{3}\dots\overbrace{a_n}^{n+1}\,\overbrace{\#}^{n+2} \\
y\text{-Sequenz}: & \underbrace{\#\;\; q_0 \;\; a_1 \;\; a_2 \;\; \dots \;\; a_n \;\; \#}_{1} \;\; \underbrace{q_1b}_{2} \;\; \underbrace{a_2}_{3} \dots \underbrace{a_n}_{n+1} \;\; \underbrace{\#}_{n+2}
\end{array}
$$

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.

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

- 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_5 unentscheidbar ist und dass PCP_2 entscheidbar ist. Dementsprechend ist auch PCP_k für $k > 5$ unentscheidbar, und PCP_1 ist trivial. Die Entscheidbarkeit von PCP_3 und PCP_4 ist ein offenes Probleme.

5.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 $\mathrm{RE}(\Sigma)$ die Menge aller rekursiv-aufzählbaren (semi-entscheidbaren) 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

$$P\colon \mathrm{RE}(\Sigma) \to \{0,1\} \mathrel{\widehat{=}} \mathbb{B} = \{\mathrm{false}, \mathrm{true}\} \ .$$

Wir sagen, dass eine Sprache $\mathcal{L} \in \mathrm{RE}(\Sigma)$ 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 $\mathcal{L} \in \mathrm{RE}(\Sigma)$ oder $P(L) = 1$ für alle $\mathcal{L} \in \mathrm{RE}(\Sigma)$, 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

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

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 $P(\emptyset) = 0$ gilt, der Beweis für den anderen Fall funktioniert analog. (Beachte, dass $\emptyset$ 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

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

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 $\emptyset$ 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 $y \in \mathcal{L}(M^K_{w,x})$ 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 $\mathcal{L}(M^K_{w,x})$ keine Eingabe y, unabhängig von der konkreten Eingabe.

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

Zusammenfassend erhalten wir:

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

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

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 $P(\mathcal{L}) \leq P(\mathcal{L}')$. 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 Probleme zu den aus „Theoretische Informatik 1“ bekannten Sprachklassen auf ihre Entscheidbarkeit untersuchen.

Bemerkung
Um diese Probleme mit den hier vorgestellten Methoden untersuchen zu können, muss man sie als Sprachen auffassen. Hierzu wäre es nötig, endliche Automaten bzw. kontextfreie Grammatiken zu kodieren, wie wir dies in Kapitel 4 für Turing-Maschinen getan haben.

Für die regulären Sprachen haben wir bereits in Theoretische Informatik 1 festgestellt, dass alle interessanten Probleme entscheidbar sind, darunter das **Wortproblem** („Gegeben NFA A, Wort w; gilt $w \in \mathcal{L}(A)$?“), das **Leerheitsproblem** („Gegeben A; gilt $\mathcal{L}(A) = \emptyset$?“), das **Universalitätsproblem** („Gegeben A; gilt $\mathcal{L}(A) = \Sigma^*$?“), das **Inklusionsproblem** („Gegeben A, B; gilt $\mathcal{L}(A) \subseteq \mathcal{L}(B)$?“) und das **Äquivalenzproblem** („Gegeben A, B; gilt $\mathcal{L}(A) = \mathcal{L}(B)$?“). Für die Details verweisen wir auf Kapitel 5 der Vorlesungsnotizen zu Theoretische Informatik 1. Wir werden diese Probleme teilweise später im zweiten Teil von Theoretische Informatik 2 im Hinblick auf ihre Komplexität untersuchen.

Wenden wir uns nun den kontextfreien Sprachen, also der Klasse CFL (*context-free languages*), zu. Wir haben bereits in Theoretische Informatik 1 (siehe Kapitel 14 der Vorlesungsnotizen) festgestellt, dass die folgenden drei Probleme entscheidbar sind.

Wortproblem für kontextfreie Sprachen (WORD-CFL)
Gegeben: Kontextfreie Grammatik G, Wort w.
Frage: Gilt $w \in \mathcal{L}(G)$?

Leerheit kontextfreier Sprachen (EMPTINESS-CFL)
Gegeben: Kontextfreie Grammatik G.
Frage: Gilt $\mathcal{L}(G) = \emptyset$?

Unendlichkeit kontextfreier Sprachen
Gegeben: Kontextfreie Grammatik G.
Frage: Gilt $\mathcal{L}(G)$ unendlich?

Das erste Problem kann durch den CYK-Algorithmus gelöst werden, das zweite mit einer Fixpunktkonstruktion. Außerdem ist bekannt, dass das folgende Probleme entscheidbar ist.

Reguläre Inklusion kontextfreier Sprachen (REG-INCLUSION-CFL)
Gegeben: Kontextfreie Grammatik G, NFA A.
Frage: Gilt $\mathcal{L}(G) \subseteq \mathcal{L}(A)$?

Hierzu konstruieren wir zunächst einen NFA $\overline{A}$ mit $\mathcal{L}(\overline{A}) = \Sigma^* \setminus \mathcal{L}(A)$, dann eine Grammatik G' mit $\mathcal{L}(G') = \mathcal{L}(G) \cap \mathcal{L}(\overline{A})$. Für letzteres verwenden wir, dass kontextfreie Sprachen effektiv unter regulärem Schnitt abgeschlossen sind. Die Inklusion $\mathcal{L}(G) \subseteq \mathcal{L}(A)$ gilt, genau dann wenn $\mathcal{L}(G') = \emptyset$. Diese Eigenschaft lässt sich überprüfen, in dem man einen Algorithmus für das Leerheitsproblem auf G' anwendet.

Viele andere Probleme für kontextfreie Sprachen sind jedoch unentscheidbar. Einige dieser Probleme wollen wir im folgenden kennen lernen.

Wir beginnen mit den folgenden drei Problemen, die sich mit Eigenschaften des Schnitts von zwei kontextfreien Sprachen befassen.

Schnittleerheit kontextfreier Sprachen (INTERSECTION-EMPTINESS-CFL)
Gegeben: Kontextfreie Grammatiken G_1, G_2.
Frage: Gilt $\mathcal{L}(G_1) \cap \mathcal{L}(G_2) = \emptyset$?

Unendlichkeit des Schnitts kontextfreier Sprachen
Gegeben: Kontextfreie Grammatiken G_1, G_2.
Frage: Beinhaltet $\mathcal{L}(G_1) \cap \mathcal{L}(G_2)$ unendlich viele Wörter?

Kontextfreiheit des Schnitts kontextfreier Sprachen
Gegeben: Kontextfreie Grammatiken G_1, G_2.
Frage: Ist $\mathcal{L}(G_1) \cap \mathcal{L}(G_2)$ kontextfrei?

6.1 Theorem
Alle nicht-trivialen Eigenschaften des Schnitts kontextfreier Sprachen sind unentscheidbar. Insbesondere sind die folgenden Probleme unentscheidbar:

a) Schnittleerheit kontextfreier Sprachen (INTERSECTION-EMPTINESS-CFL),

b) Unendlichkeit des Schnitts kontextfreier Sprachen und

c) Kontextfreiheit des Schnitts kontextfreier Sprachen.

Wir beweisen die Unentscheidbarkeit der in a), b) und c) genannten Probleme formal und argumentieren anschließend, dass auch alle anderen Eigenschaften des Schnitts kontextfreier Sprachen (z.B. seine Regularität) unentscheidbar sind. Um Teil a) zu beweisen, reduzieren wir das PCP. Die Reduktion, die wir hierzu angeben, können wir später im Beweis von b) und c) wiederverwenden.

Beweis:

a) Wir reduzieren das $0,1$-PCP auf die Nichtleerheit des Schnittes zweier kontextfreier Sprachen, das Komplementproblem zu INTERSECTION-EMPTINESS-CFL. Wenn ein Problem unentscheidbar ist, dann auch sein Komplementproblem. Wir können also so zeigen, dass auch die Leerheit des Schnittes zweier kontextfreier Sprachen unentscheidbar ist.

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 so dass der Schnitt der beiden Sprachen $\mathcal{L}(G_1)$ und $\mathcal{L}(G_2)$ die Lösungen der PCP-Instanz K repräsentiert. Die beiden Grammatiken verwenden das Terminalalphabet $\{0, 1, \#, z_1, \ldots, z_k\}$, welches aus $0, 1$, dem Trennsymbol $\#$ und einem Symbol z_i pro Paar Paar (x_i, y_i) in K besteht.

Grammatik G_1 hat die folgenden Produktionsregeln:

$$\begin{aligned}
&S \to A\#B \ , \\
&A \to z_1 A x_1 \mid \ldots \mid z_k A x_k \ , \\
&A \to z_1 x_1 \mid \ldots \mid z_k x_k \ , \\
&B \to y_1^{\text{reverse}} B z_1 \mid \ldots \mid y_k^{\text{reverse}} B z_k \ , \\
&B \to y_1^{\text{reverse}} z_1 \mid \ldots \mid y_k^{\text{reverse}} z_k \ .
\end{aligned}$$

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

Die von G_1 erzeugte Sprache ist

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

Grammatik G_2 hat die folgenden Produktionsregeln:

$$S \to z_1 S z_1 \mid \dots \mid z_k S z_k \mid T \ ,$$
$$T \to 0T0 \mid 1T1 \mid \# \ .$$

Die von G_2 erzeugte Sprache ist also

$$\mathcal{L}(G_2) = \{ u \ v \ \# v^{\text{reverse}} u^{\text{reverse}} \mid v \in \{0,1\}^*, u \in \{z_1, \dots, z_k\}^* \} \ .$$

Es gilt, dass K eine nicht-leere Lösung $\ell_1 \dots \ell_n$ hat, genau dann, wenn der Schnitt $\mathcal{L}(G_1) \cap \mathcal{L}(G_2)$ nicht-leer ist, nämlich das Wort

$$w = z_{\ell_n} \dots z_{\ell_1} x_{\ell_1} \dots x_{\ell_n} \# y_{\ell_n}^{\text{reverse}} \dots y_{\ell_1}^{\text{reverse}} z_{\ell_n} \dots z_{\ell_1}$$

enthält. Dieses Wort erzeugen wir in G_1, indem wir sowohl beim Ersetzen von A als auch vom Ersetzen von B die Produktionsregeln auswählen, die zur Indexsequenz $\ell_n \dots \ell_1$ korrespondieren. Dieses Wort ist außerdem in G_2, da es ein Palindrom ist (es gilt $w = w^{\text{reverse}}$).

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

b) Falls eine PCP-Instanz eine Lösung hat, dann hat sie unendlich viele Lösungen: Zu jeder Lösung $s = \ell_1 \dots \ell_n$ sind auch $s^2 = s.s = \ell_1 \dots \ell_n \ell_1 \dots \ell_n$, $s^3 = s.s.s$, s^4, ... Lösungen. Wir haben in Teil a) gesehen, dass es zu jeder Lösung der PCP-Instanz K ein entsprechendes Wort in $\mathcal{L}(G_1) \cap \mathcal{L}(G_2)$ gibt.

Die Reduktion f mit $f(K) = (G_1, G_2)$ aus Teil a) liefert also entweder zwei Grammatiken mit $\mathcal{L}(G_1) \cap \mathcal{L}(G_2) = \emptyset$ (wenn K eine Nein-Instanz des PCPs ist), oder zwei Grammatiken deren Schnitt unendlich viele Wörter beinhaltet. Es ist also auch eine Reduktion auf die Unendlichkeit des Schnittes.

c) Wir betrachten wieder die Reduktion f aus Teil a). Betrachte $f(K) = (G_1, G_2)$. Wenn K eine Nein-Instanz ist, gilt $\mathcal{L}(G_1) \cap \mathcal{L}(G_2) = \emptyset$ und diese Sprache ist kontextfrei. Andernfalls lässt sich mit dem aus „Theoretische Informatik 1“ bekannten Pumping-Lemma für kontextfreie Sprachen zeigen, dass $\mathcal{L}(G_1) \cap \mathcal{L}(G_2)$ nicht-kontextfrei ist.

Die Reduktion aus Teil a) zeigt also auch, dass die Nicht-Kontextfreiheit des Schnitts unentscheidbar ist, und damit auch ihr Komplementproblem, die Kontextfreiheit des Schnitts.

□

6.2 Bemerkung

Der obige Beweis zeigt formal, dass die Schnittleerheit kontextfreier Sprachen unentscheidbar ist. Wir liefern eine intuitive Begründung für dieses Resultat, und argumentieren, dass auch alle anderen nicht-trivialen Eigenschaften des Schnitts kontextfreier Sprachen unentscheidbar sind.

Betrachte kontextfreie Sprachen $\mathcal{L}_1, \mathcal{L}_2$ und Pushdown-Automaten P_1, P_2, die diese Sprachen generieren, $\mathcal{L}(P_i) = \mathcal{L}_i$. Der Schnitt $\mathcal{L}_1 \cap \mathcal{L}_2$ der beiden Sprachen entspricht dem Produkt $P_1 \times P_2$ der beiden Automaten. Das Resultat dieser Produktoperation ist jedoch kein Pushdown-Automat, sondern ein **Multipushdown-Automat**, der zwei Stacks zur Verfügung hat, die er unabhängig voneinander verwenden kann. Dieses Modell ist gleichmächtig wie Turing-Maschinen: Die Konfiguration $v\ q\ w$ einer TM kann durch die Konfiguration eines Multipushdown-Automaten repräsentiert werden, in der v auf dem ersten und w^{reverse} auf dem zweiten Stack gespeichert wird. Um die Bewegungen des Kopfes der TM auf dem Band zu simulieren, werden Symbole von einem Stack auf den anderen Stack umgeschichtet. Multipushdowns sind also ein Turing-vollständiges Modell – Zu jeder Turing-Maschine M lässt sich ein Multipushdown-Automat P mit $\mathcal{L}(M) = \mathcal{L}(P)$ konstruieren.

Ein beliebiger Multipushdown-Automat wiederum lässt sich in zwei Pushdown-Automaten aufteilen, wobei jeder Automat einen der beiden Stacks verwaltet und die beiden Automaten sich über die Eingabesymbole synchronisieren. Es gibt also zu jedem Multipushdown-Automaten P mit zwei Stacks zwei Pushdown-Automaten P_1, P_2 mit $\mathcal{L}(P) = \mathcal{L}(P_1) \cap \mathcal{L}(P_2)$.

Wir kombinieren die beiden Resultate und erhalten, dass es zu jeder Turing-Maschine M zwei kontextfreie Sprachen $\mathcal{L}(P_1)$ und $\mathcal{L}(P_2)$ gibt mit $\mathcal{L}(M) = \mathcal{L}(P_1) \cap \mathcal{L}(P_2)$. Da mit dem Satz von Rice alle nicht-trivialen Eigenschaften der Sprachen von Turing-Maschinen unentscheidbar sind, sind also auch alle nicht-trivialen Eigenschaften der Schnitte von kontext-freien Sprachen entscheidbar.

Man könnte diese Beweisidee formal umsetzen, also z.B. eine Reduktion des Akzeptanzproblems für TMs auf die Nichtleerheit des Schnittes kontextfreier Sprachen angeben. Dies wäre allerdings deutlich komplexer als die oben angegebene Reduktion des PCPs.

Deterministische kontextfreie Sprachen

Um weitere Unentscheidbarkeitsresultate zu zeigen müssen wir zunächst eine Teilklasse der kontextfreien Sprachen (CFL) einführen. Die Klasse DCFL der **deterministisch-kontextfreien Sprachen** ist die Klasse der Sprachen $\mathcal{L}(P)$, wobei P ein deterministischer Pushdown-Automat ist,

$$\mathsf{DCFL} = \{\mathcal{L}(P) \mid P \text{ deterministischer Pushdown-Automata}\} .$$

Ein **deterministischer Pushdown-Automat** ist ein Pushdown-Automat ohne ε-Transitionen, in dem jede Transition genau einen Buchstaben des Stacks liest und bei dem es zu jeder Kombination (q, a) aus Kontrollzustand und oberstem Stacksymbol jeweils höchstens eine Transition gibt, die angewandt werden kann (die also von q ausgeht und a liest). Eine auf Grammatiken basierende Definition für DCFL anzugeben ist möglich, aber komplex.

Die Klasse DCFL liegt echt zwischen den regulären Sprachen REG und den kontextfreien Sprachen CFL. Beispielsweise ist die nicht-reguläre Sprache $\{a^n b^n \mid n \in \mathbb{N}\}$ in DCFL, die kontextfreie Sprache aller Palindrome gerader Länge $\{w.w^{\text{reverse}} \mid w \in \{0,1\}^*\}$ aber nicht in DCFL. Intuitiv müsste ein Pushdown-Automat für die letztgenannte Sprache die Mitte des Wortes raten; einem deterministischen Pushdown ist dies nicht möglich.

Die Klasse DCFL ist im Gegensatz zu CFL unter Komplement abgeschlossen: Wenn $\mathcal{L}(P)$ die Sprache eines deterministischen Pushdowns ist, dann gilt $\overline{\mathcal{L}(P)} = \Sigma^* \setminus \mathcal{L}(P) = \mathcal{L}(\overline{P})$, wobei $\overline{P}$ aus P durch das Umdrehen von End- und Nichtendzuständen entsteht. Im Gegensatz zu CFL ist DCFL nicht unter Vereinigung abgeschlossen Genau wie CFL ist DCFL nicht unter Schnitt abgeschlossen.

Die im Beweis von Theorem 6.1 definierten Sprachen $\mathcal{L}(G_1)$ und $\mathcal{L}(G_2)$ sind Sprachen deterministischer Pushdown-Automaten. Dies mag im Fall von $\mathcal{L}(G_2)$ verwunderlich sein, da es sich hier ebenfalls um eine Sprache von Palindromen handelt. Der intuitive Grund für das Enthaltensein in DCFL ist, dass in Wörtern aus $\mathcal{L}(G_2)$ die Wortmitte durch das Trennsymbol # markiert ist. Den formalen Beweis sei der Leserin / dem Leser als Übung überlassen.

Daraus können wir schlussfolgern, dass die in Theorem 6.1 auch dann unentscheidbar sind, wenn wir die Eingabe auf deterministische kontextfreier Sprachen beschränken. Alle Eigenschaften des Schnittes deterministisch-kontextfreier Sprachen sind unentscheidbar. Insbesondere ist das folgende Problem unentscheidbar.

Schnittleerheit det.-kontextfreier Sprachen (INTERSECTION-EMPTINESS-DCFL)
Gegeben: Deterministische Pushdown-Automaten P_1, P_2.
Frage: Gilt $\mathcal{L}(P_1) \cap \mathcal{L}(P_2) = \emptyset$?

6.3 Korollar
Die Leerheit des Schnitts von deterministisch-kontextfreien Sprachen (INTERSECTION-EMPTINESS-DCFL) ist unentscheidbar.

Dieses Unentscheidbarkeitsresultat für die Klasse DCFL können wir jetzt verwenden, um weitere Unentscheidbarkeitsresultate für die allgemeinere Klases CFL zu zeigen. Wir betrachten die folgenden Probleme.

Inklusion kontextfreier Sprachen (INCLUSION-CFL)
Gegeben: Kontextfreie Grammatiken G_1, G_2.
Frage: Gilt $\mathcal{L}(G_1) \subseteq \mathcal{L}(G_2)$?

Sprachgleichheit kontextfreier Sprachen (EQUIVALENCE-CFL)
Gegeben: Kontextfreie Grammatiken G_1, G_2.
Frage: Gilt $\mathcal{L}(G_1) = \mathcal{L}(G_2)$?

Universalität kontextfreier Sprachen (UNIVERSALITY-CFL)
Gegeben: Kontextfreie Grammatik G mit Terminalalphabet Σ.
Frage: Gilt $\mathcal{L}(G) = \Sigma^*$?

6.4 Theorem
Die folgenden Probleme sind unentscheidbar:

a) Inklusion kontextfreier Sprachen (INCLUSION-CFL),

b) Sprachgleichheit/-äquivalenz kontextfreier Sprachen (EQUIVALENCE-CFL).

c) Universalität kontextfreier Sprachen (UNIVERSALITY-CFL).

Die Unentscheidbarkeit des Universalitätsproblems impliziert, dass auch die folgenden beiden Probleme unentscheidbar sind, da Universalität jeweils ein Spezialfall ist. Gegeben eine kontextfreie Grammatik G und ein NFA A, dann ist es unentscheidbar zu prüfen ob

- $\mathcal{L}(A) \subseteq \mathcal{L}(G)$ bzw.
- $\mathcal{L}(A) = \mathcal{L}(G)$ gilt.

Beweis:

a) Wir reduzieren von der Schnittleerheit deterministisch-kontextfreier Sprachen. Es sei (P_1, P_2) eine Instanz für INTERSECTION-EMPTINESS-DCFL.

Es gilt $\mathcal{L}(P_1) \cap \mathcal{L}(P_2) = \emptyset$ genau dann, wenn die Inklusion $\mathcal{L}(P_1) \subseteq \overline{\mathcal{L}(P_2)}$ gilt. Da DCFL unter Komplementierung abgeschlossen ist, kann man einen Pushdown-Automaten $\overline{P_2}$ konstruieren mit $\mathcal{L}(\overline{P_2}) = \overline{\mathcal{L}(P_2)}$. Anschließend verwenden wir eine Implementierung der aus „Theoretische Informatik 1“ bekannten Techniken um Grammatiken $G_1, \overline{G_2}$ mit $\mathcal{L}(G_1) = \mathcal{L}(P_1)$ und $\mathcal{L}(\overline{G_2}) = \mathcal{L}(\overline{P_2})$ zu konstruieren.

Die Funktion $f(P_1, P_2) = (G_1, \overline{G_2})$ ist die gesuchte Reduktion.

b) Wir reduzieren das Inklusionsproblem auf das Äquivalenzproblem. Betrachte zwei gegebene Grammatiken G_1, G_2. Da kontextfreie Sprachen effektiv unter Vereinigung abgeschlossen sind, kann man eine Grammatik G_3 konstruieren mit $\mathcal{L}(G_3) = \mathcal{L}(G_1) \cup \mathcal{L}(G_2)$.

Es gilt

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

Die Funktion $g(G_1, G_2) = (G_3, G_2)$ reduziert also Inklusion auf Sprachgleichheit und zeigt damit, dass das Äquivalenzproblem unentscheidbar ist.

c) Wir reduzieren die Schnittleerheit deterministisch-kontextfreier Sprachen auf das Universalitätsproblem. Für zwei deterministische Pushdowns P_1, P_2 gilt $\mathcal{L}(P_1) \cap \mathcal{L}(P_2) = \emptyset$ genau dann, wenn $\overline{\mathcal{L}(P_1)} \cup \overline{\mathcal{L}(P_2)} = \Sigma^*$. Wir konstruieren zunächst Pushdowns $\overline{P_1}, \overline{P_2}$ für die Komplementsprachen von $\mathcal{L}(P_1)$ und $\mathcal{L}(P_2)$, dann kontextfreie Grammatiken $\overline{G_1}, \overline{G_2}$ mit den selben Sprachen und schließlich eine kontextfreie Grammatik G mit $\mathcal{L}(G) = \mathcal{L}(\overline{G_1}) \cup \mathcal{L}(\overline{G_2})$. Die Funktion $h(P_1, P_2) = G$ ist die gesuchte Reduktion.

□

6.5 Bemerkung

Teil a) des obigen Beweises verwendet nur Sprachen aus DCFL. Das Inklusionsproblem ist also auch für deterministische kontextfreie Sprachen unentscheidbar.

Die Teile b) und c) des Beweises funktioniert nicht für Sprachen aus DCFL: DCFL ist nicht unter Vereinigung abgeschlossen. Selbst wenn die initial gegebenen Sprachen aus DCFL sind, so ist die neu konstruierte Sprache $\mathcal{L}(G_3)$ bzw. $\mathcal{L}(G)$ eventuell keine deterministische kontextfreie Sprache. Tatsächlich wurde 2001 von Géraud Sénizergues bewiesen, dass die Sprachgleichheit deterministischer kontextfreier Sprachen (und damit auch Universalität) entscheidbar ist [Sén01; Sén02]. Hierfür wurde ihm 2002 der Gödel-Preis verliehen.

Unter Verwendung der Techniken, die wir in diesem Kapitel kennen gelernt haben, lässt sich die Unentscheidbarkeit weiterer Probleme kontextfreier Sprachen zeigen. Wir nennen einige dieser Probleme, führen die Beweise jedoch nicht.

Regularität kontextfreier Sprachen (REGULARITY-CFL)
Gegeben: Kontextfreie Grammatik G.
Frage: Ist $\mathcal{L}(G)$ regulär?

Kontextfreiheit des Komplements einer kontextfreien Sprache
Gegeben: Kontextfreie Grammatik G.
Frage: Ist $\overline{\mathcal{L}(G)}$ kontextfrei?

Enthalten-sein in DCFL
Gegeben: Kontextfreie Grammatik G.
Frage: Ist $\mathcal{L}(G)$ deterministisch-kontextfrei?

6.6 Theorem
Regularität, Kontextfreiheit des Komplements und das Enthalten-sein in DCFL sind unentscheidbar für kontextfreie Sprachen.

Die Unentscheidbarkeit des Enthalten-seins in DCFL bedeutet, dass es keinen Algorithmus gegeben kann, der als Eingabe einen Pushdown-Automaten nimmt und einen deterministischen Pushdown-Automaten ausgibt, falls ein solcher existiert.

Teil II.
Komplexitätstheorie

7. Grundlagen der Komplexitätstheorie

Mit den Techniken aus dem ersten Teil der Vorlesung lassen sich Probleme auf ihre Entscheidbarkeit untersuchen. Für Probleme, die als entscheidbar bekannt sind, stellt sich nun die Frage, wie *effizient* sie von einem Computer gelöst werden können, das heißt, mit welchem *Ressourcenverbrauch*.

Wir betrachten hier als Modell für Computer erneut deterministische und nichtdeterministische Turing-Maschinen. Die beiden Arten von Ressourcen die wir untersuchen werden sind Zeit (Zeitverbrauch einer Turing-Maschine, gemessen in Rechenschritten) und Platz (Platzverbrauch einer Turing-Maschine, gemessen in belegten Bandzellen).

In diesem Kapitel gehen wir wie folgt vor.

1. Wir wiederholen die **Landau-Notation**, insbesondere die Definition der Klasse $\mathcal{O}(f)$ von Funktionen.
2. Wir definieren die grundlegenden **Zeit-** und **Platzkomplexitätsklassen** und die von ihnen abgeleiteten **robusten Komplexitätsklassen** wie z.B. $\mathsf{P}, \mathsf{NP}, \mathsf{PSPACE}$.
3. Wir beweisen einige grundlegende Relationen zwischen diesen Klassen (z.B. $\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE}$).
4. Wir führen Komplementklassen wie z.B. coNP ein.

Kapitel 8 beinhaltet eine kurze Zusammenfassung der hier gezeigten Resultate in Form eines Diagramms.

7.A Landau-Notation

Im Folgenden werden wir den Ressourcenverbrauch einer Turing-Maschine als eine Funktion der Form $f\colon \mathbb{N} \to \mathbb{N}$ repräsentieren. Das Argument der Funktion ist eine Zahl n, welche die Größe der Eingabe, d.h. die Länge des Eingabewortes der Turing-Maschine, repräsentiert. Der zugehörige Funktionswert $f(n)$ ist der Ressourcenverbrauch der Turing-Maschine auf Eingaben der Länge n.

Unsere Analysen in diesem Teil der Vorlesung sind **asymptotische Analysen**. Wir interessieren uns also nicht für die Funktionswerte an bestimmten Stellen, sondern

für das asymptotische Verhalten der Funktion, wenn die Eingabewerte wachsen. Außerdem wollen wir auch multiplikative Konstanten ignorieren. Um dies zu formalisieren, führen wir die folgende **Landau-Notation** ein.

Zu einer Funktion $f\colon \mathbb{N} \to \mathbb{N}$ ist $\mathcal{O}(f)$ die Klasse der Funktionen, die **asymptotisch kleiner-gleich** f sind,

$$\mathcal{O}(f) = \{g\colon \mathbb{N} \to \mathbb{N} \mid \exists n_0, m \in \mathbb{N} \forall n \geq n_0\colon g(n) \leq m \cdot f(n)\} \ .$$

Durch geeignete Wahl der Schranke n_0 und der Konstante m erlaubt es uns die Definition von $\mathcal{O}(f)$, das Verhalten von f auf einem beschränkten Intervall $[0, \ldots, n_0 - 1]$ sowie multiplikative Konstanten zu ignorieren.

7.1 Bemerkung
Im folgenden verwenden wir die folgende Kurznotation für Funktionen mit Signatur $\mathbb{N} \to \mathbb{N}$. Eine natürliche Zahl wie z.B. 5000 steht für die konstante Funktion mit Abbildungsvorschrift $n \mapsto 5000$. Ein Ausdruck wie z.B. n^2 steht für die Funktion mit Abbildungsvorschrift $n \mapsto n^2$, analog für $\log n, n^3, 2^n$ usw. Insbesondere repräsentiert n die Identitätsfunktion $n \mapsto n$.

7.2 Beispiel

a) Es gilt $\mathcal{O}(1) = \mathcal{O}(5.000) = \mathcal{O}(k)$ für alle Konstanten $k \in \mathbb{N}, k > 0$. $\mathcal{O}(1)$ ist die Klasse aller Funktionen, deren Funktionswerte durch eine Konstante c beschränkt sind. (Wähle $n_0 = 1, m = c$ in der Definition von $\mathcal{O}(f)$.)

b) $\mathcal{O}(\log n)$ ist die Klasse der logarithmischen Funktionen. (Streng genommen: Die Klasse der asymptotisch durch den Logarithmus beschränkten Funktionen.) Es gilt $1 \in \mathcal{O}(\log n)$, aber $\log n \notin \mathcal{O}(1)$. Es spielt keine Rolle, welche Basis wir verwenden; beispielsweise gilt $\mathcal{O}(\log_{10} n) = \mathcal{O}(\log_2 n) = \mathcal{O}(\ln n)$, denn die verschiedenen Logarithmen unterscheiden sich bloß um multiplikative Konstanten.

c) $\mathcal{O}(n)$ ist die Klasse der linearen Funktionen. Analog sind $\mathcal{O}(n^2), \mathcal{O}(n^3)$ die Klasse der quadratischen bzw. kubischen Funktionen.

   Zu jeder polynomiellen Funktionen p mit Abbildungsvorschrift

   $$n \mapsto a_k n^k + a_{k-1} n^{k_1} + \ldots + a_1 n + a_0$$

   gilt $p \in \mathcal{O}(n^k)$. Beispielweise gilt $2n^3 + n^2 + 300 \in \mathcal{O}(n^3)$, wie man durch Wahl von $m = 4$ und $n_0 = 7$ sieht (denn $7 > \sqrt[3]{300}$).

Die $\mathcal{O}$-Notation erlaubt es uns also, alle Terme bis auf den dominierenden Term – den mit dem höchsten Exponenten – wegfallen zu lassen.

d) Wir werden später sehen, dass die Klassen der Form $\mathcal{O}(n^k)$ zu feingranular sind. Wir betrachten stattdessen die Vereinigung $\bigcup_{k\in\mathbb{N}} \mathcal{O}(n^k)$, die Klasse aller polynomiellen Funktionen. Zu jedem Polynom p gilt $p \in \bigcup_{k\in\mathbb{N}} \mathcal{O}(n^k)$. Es gilt $2^n \notin \bigcup_{k\in\mathbb{N}} \mathcal{O}(n^k)$, da hier der Exponent keine Konstante ist, sondern vom Eingabewert abhängt.

e) Die Klasse $\{n \mapsto 2^{f(n)} \mid f \in \mathcal{O}(n)\}$ ist die Klasse aller exponentiellen Funktionen mit linearem Exponenten. Wir schreiben diese kurz als $2^{\mathcal{O}(n)}$. Die Vereinigung $\bigcup_{k\in\mathbb{N}} 2^{\mathcal{O}(n^k)}$ ist die Klasse aller Exponentialfunktionen.

7.3 Bemerkung
In der Literatur ist es üblich, dass man statt $g \in \mathcal{O}(f)$ auch $g = \mathcal{O}(f)$ schreibt. Dies impliziert jedoch nicht $\mathcal{O}(g) = \mathcal{O}(f)$: Beispielsweise gilt $n \in \mathcal{O}(n^2)$, aber $\mathcal{O}(n) \neq \mathcal{O}(n^2)$, da $n^2 \notin \mathcal{O}(n)$.

Es gibt noch einige weitere nützliche Klassen von Funktionen wie z.B. $o(f)$, die Klasse der Funktionen, die asymptotisch echt kleiner als f sind. Wir werden diese Klassen bei Bedarf an der jeweiligen Stelle einführen.

7.B Die Komplexitätsklassen

Wir definieren die grundlegenden Zeit- und Platzkomplexitätsklassen und dann die davon abgeleiteten robusten Komplexitätsklassen.

Zeitkomplexität

Zunächst wollen wir den **Zeitverbrauch** einer gegebenen Turing-Maschine definieren. Dann definieren wir die Klasse von Problemen, die von Turing-Maschinen mit einem vorgegebenen Zeitverbrauch gelöst werden können.

Wir messen den Zeitverbrauch von Turing-Maschinen in Rechenschritten. Wenn M eine deterministische Turing-Maschine und x eine Eingabe für diese Maschine ist, dann ist $\mathsf{Time}_M(x) = i$, wenn i der kleinste Index ist, so dass die Konfiguration c_i in der Berechnung $c_0 \to c_1 \to \ldots$ von M zu Eingabe x eine Haltekonfiguration ist. Wir messen also die Schrittzahl, die M benötigt, um zum ersten Mal zu halten. (Gemäß unserer Definition von Turing-Maschinen bleibt M in einer Haltekonfiguration,

sobald sie einmal eine betreten hat.) Wenn M auf Eingabe x nicht hält, setzen wir $\mathsf{Time}_M(x) = \infty$.

Wir verallgemeinern die Definition auf den Fall von nicht-deterministischen Maschinen, dadurch dass wir das Maximum über alle Berechnungen nehmen.

7.4 Definition: Zeitverbrauch
Sei M eine Turing-Maschine (potentiell nicht-deterministisch, potentiell mit mehreren Bändern). Sei $x \in \Sigma^*$ eine Eingabe für M. Der **Zeitverbrauch** (oder die Rechenzeit) von M für Eingabe x ist

$$\mathsf{Time}_M(x) = \max\{\min\{i \mid c_i \text{ Haltekonfiguration}\} \mid c_0 \to c_1 \to \dots \text{ Berechnung von } M \text{ zu } x\} .$$

Auch hier definieren wir $\mathsf{Time}_M(x) = \infty$, wenn M eine nicht-haltende Berechnung zur Eingabe x hat. In diesem Teil der Vorlesung werden wir jedoch – 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. Wenn M ein Entscheider ist, dann liefert Time_M für alle Eingaben eine natürliche Zahl, ist also ein Funktion der Form $\Sigma^* \to \mathbb{N}$.

Um eine Funktion mit Signatur der Form $\mathbb{N} \to \mathbb{N}$ zu erhalten, betrachten wir den **Worst-Case**, d.h. die Eingabe einer vorgegebenen Länge, zu der M am meisten Rechenschritte benötigt. Man beachte, dass es zu jeder Zahl nur endlich viele Eingaben der entsprechenden Länge gibt.

7.5 Definition: Zeitkomplexität
Zu einer Turing-Maschine M ist die **Zeitkomplexität** $\mathsf{Time}_M \colon \mathbb{N} \to \mathbb{N}(\cup\{\infty\})$ die Funktion mit

$$\mathsf{Time}_M(n) = \max\{\mathsf{Time}_M(x) \mid x \in \Sigma^*, |x| = n\} .$$

Nun wollen wir zu einer vorgegebenen Zeitschranke die Klasse der Probleme betrachten, die sich innerhalb der gegebenen Zeit lösen lassen. Zunächst benötigen wir folgende Hilfsdefinition.

7.6 Definition: Zeitschranke
Sei $f \colon \mathbb{N} \to \mathbb{N}$ eine Funktion. Wir sagen, dass M **f-zeitbeschränkt** ist, wenn $\mathsf{Time}_M \leq f$, also $\mathsf{Time}_M(n) \leq f(n)$ für alle $n \in \mathbb{N}$.

7.7 Bemerkung
Wenn M f-zeitbeschränkt für irgend eine Funktion $f\colon \mathbb{N} \to \mathbb{N}$ ist, dann muss M ein Entscheider sein.

Nun können wir die grundlegenden Zeitkomplexitätsklassen definieren. Zu jeder Zeitschranke betrachtet man die Klasse der Probleme, die sich innerhalb dieser Zeitschranke lösen lassen. Wir führen dabei verschiedene Klassen ein, abhängig von der Bandanzahl und davon, ob wir deterministische oder nicht-deterministische Maschinen betrachten.

7.8 Definition: Grundlegende Zeitkomplexitätsklassen
Sei $f\colon \mathbb{N} \to \mathbb{N}$ eine Zeitschranke und $m \in \mathbb{N}, m > 0$. Wir definieren die grundlegenden **Zeitkomplexitätsklassen**

$$\mathrm{DTIME}_m(f) = \{\mathcal{L}(M) \mid M \text{ ist eine } m\text{-Band DTM und } f\text{-zeitbeschränkt}\} ,$$
$$\mathrm{NTIME}_m(f) = \{\mathcal{L}(M) \mid M \text{ ist eine } m\text{-Band NTM und } f\text{-zeitbeschränkt}\} .$$

7.9 Bemerkung
Wir schreiben im 1-Band-Fall nur $\mathrm{DTIME}(f)$ bzw. $\mathrm{NTIME}(f)$ anstatt $\mathrm{DTIME}_1(f)$ bzw. $\mathrm{NTIME}_1(f)$.

Sei $F \subseteq \mathbb{N} \to \mathbb{N}$ eine Menge von Funktionen. Dann schreiben wir $\mathrm{DTIME}(F)$ für $\bigcup_{f \in F} \mathrm{DTIME}(f)$ für die Klasse aller Probleme, die sich mit Zeit beschränkt durch eine Funktion aus F lösen lassen (analog für NTIME und Mehrbandmaschinen). Beispielsweise ist $\mathrm{DTIME}_2(\mathcal{O}(n^2))$ die Klasse der Probleme, die sich von 2-Band DTMs mit asymptotisch höchstens quadratischen Zeitverbrauch lösen lassen.

Mit der oben genannten Beobachtung sind alle Probleme in $\mathrm{DTIME}_k(f)$ und $\mathrm{NTIME}_k(f)$ entscheidbar.

Platzkomplexität

Wir möchten den Platzverbrauch einer Maschine definieren sowie die Klasse von Problemen, die sich mit einem vorgegebenen Platzverbrauch lösen lassen. Der Platzverbrauch wird hierbei als die Anzahl Zellen, die auf dem Band der TM belegt sind, gemessen.

Hierbei betrachten wir jedoch spezielle Maschinen, da wir den Platzverbrauch der Eingabe nicht mitzählen wollen. Es ergibt keinen Sinn, Turing-Maschinen mit sub-

linearem Zeitverbrauch (z.B. $\mathsf{Time}_M(n) = \log n$) zu betrachten, da eine Maschine mindestens n Schritte braucht, um eine Eingabe der Länge n zu lesen. Im Gegensatz dazu ist es sinnvoll, Probleme zu betrachten, bei denen zusätzlich zur Eingabe nur logarithmisch viel Platz verbraucht wird. (Wenn wir den Inhalt der Eingabe mitzählen, verbraucht jede Turing-Maschine auf einer Eingabe der Länge n mindestens $\mathcal{O}(n)$ viel Platz.)

7.10 Definition: Read-Only Eingabe
Eine **Turing-Maschine mit read-only Eingabe und m Arbeitsbändern** ist eine $(m+1)$-Band-Turing-Maschine, bei der die Eingabe nicht verändert werden kann.

Formal ist jede Transitionsregel der Form

$$\delta(q, a, b_1, \ldots, b_m) \ni (q', a, c_1, \ldots, c_m, d, d_1, \ldots d_m) \ .$$

Hierbei ist $a \in \Gamma$ das Bandsymbol auf dem Eingabeband, das nicht verändert werden darf. Die Bandsymbole $b_1, \ldots, b_m \in \Gamma$ auf den m **Arbeitsbändern** dürfen durch beliebige Symbole $c_1, \ldots, c_m \in \Gamma$ überschrieben werden, und der Kopf darf auf allen $m+1$ Bändern in beliebige Richtungen $d, d_1, \ldots, d_m \in \{L, N, R\}$ bewegt werden.

Nun können wir eine Definition des Platzverbrauchs einer solchen TM angeben, die das Eingabeband ignoriert.

7.11 Definition: Platzverbrauch
Sei M eine Turing-Maschine mit read-only Eingabeband und m Arbeitsbändern (potentiell nicht-deterministisch). Sei $x \in \Sigma^*$ eine Eingabe für M.

a) Die Länge $|w|$ des Inhalts eines Arbeitsbandes in einer Konfiguration von M ist die Länge des belegten Bandinhaltes, wobei wir links und rechts vom belegten Bandinhalt die unendlich vielen $\sqcup$-Symbole entfernen und nicht mitzählen.

b) Sei c eine Konfiguration von M. Der **Platzverbrauch von M in Konfiguration c** ist

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

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

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

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

Genau wie für den Zeitverbrauch verallgemeinern wir Space_M zu einer Funktion mit Signatur $\mathbb{N} \to \mathbb{N}$ durch Betrachten des Worst-Case.

7.12 Definition: Platzkomplexität
Zu einer Turing-Maschine M ist die **Platzkomplexität** $\mathsf{Space}_M \colon \mathbb{N} \to \mathbb{N}(\cup\{\infty\})$ die Funktion mit

$$\mathsf{Space}_M(n) = \max\{\mathsf{Space}_M(x) \mid x \in \Sigma^*, |x| = n\} \ .$$

7.13 Bemerkung
Wenn wir nicht an sublinearer Platzkomplexitä interessiert sind, können wir auch *normale* Turing-Maschinen ohne read-only Eingabe betrachten: Eine solche Maschine kann in eine Maschine mit read-only Eingabe konvertiert werden, die zunächst ihre Eingabe auf ein zusätzliches Arbeitsband kopiert. (Dies verursacht Zeit- und Platzverbrauch $\mathcal{O}(n)$.)

Wir wollen nun zu einer vorgegeben Platzschranke die Klasse der Probleme betrachten, die sich unter Verwenden des gegebenen Platzes lösen lassen..

7.14 Definition: Platzschranke
Sei $f \colon \mathbb{N} \to \mathbb{N}$ eine Funktion. Wir sagen, dass M **f-platzbeschränkt** ist, wenn $\mathsf{Space}_M \leq f$, also $\mathsf{Space}_M(n) \leq f(n)$ für alle $n \in \mathbb{N}$.

7.15 Bemerkung
Wenn M ein Entscheider ist, dann ist Space_M eine Funktion der Form $\mathbb{N} \to \mathbb{N}$, liefert also nie ∞ als Funktionswert. Eine Turing-Maschine kann in jedem Schritt maximal eine neue Zelle auf dem Band belegen. Wenn die Turing-Maschine nach beschränkt vielen Schritten terminiert, dann belegt sie also auch nur beschränkt viel Platz auf dem Band.

Wenn M f-platzbeschränkt ist, dann muss M jedoch kein Entscheider sein. Eine Turing-Maschine kann auf einer Eingabe nicht halten, selbst wenn nur beschränkt viel Platz auf dem Band belegt wird.

7.16 Definition: Grundlegende Platzkomplexitätsklassen
Sei $f\colon \mathbb{N} \to \mathbb{N}$ eine Platzschranke und $m \in \mathbb{N}$. Wir definieren die grundlegenden **Platzkomplexitätsklassen**.

$$\mathrm{DSPACE}_m(f) = \left\{ \mathcal{L}(M) \;\middle|\; \begin{array}{l} M \text{ ist eine DTM mit read-only Eingabe, } m \text{ Arbeitsbändern,} \\ \text{ein Entscheider und } f\text{-platzbeschränkt} \end{array} \right\},$$
$$\mathrm{NSPACE}_m(f) = \left\{ \mathcal{L}(M) \;\middle|\; \begin{array}{l} M \text{ ist eine NTM mit read-only Eingabe, } m \text{ Arbeitsbändern,} \\ \text{ein Entscheider und } f\text{-platzbeschränkt} \end{array} \right\}.$$

Wir verwenden die selbe vereinfachte Notation wie bei den Zeitkomplexitätsklassen. Da eine Platzschranke $f\colon \mathbb{N} \to \mathbb{N}$ für M nicht impliziert, dass M ein Entscheider ist, verlangen wir dies hier explizit. Alle Probleme in $\mathrm{DSPACE}_k(f)$ und $\mathrm{NSPACE}_k(f)$ sind damit entscheidbar.

Wir geben ein Beispiel für ein Problem an, dass sich mit sublinearem Platzverbrauch entscheiden lässt.

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

$$\mathcal{L} = \{x \in \{a,b\}^* \mid \text{ Anzahl von } a \text{ und } b \text{ in } x \text{ ist gleich } \}\ .$$

Es gilt $\mathcal{L} \in \mathrm{DSPACE}(\mathcal{O}(\log n))$.

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 den Wert 0 hat. Damit entscheidet M tatsächlich $\mathcal{L}$.

Wenn der Zähler als Binärzahl gespeichert wird, hat M nur logarithmischen Platzverbrauch: Der Wert des Zähler ist bei Eingabelänge n durch $-n$ nach unten und n nach oben beschränkt. Es werden also $\lceil \log n \rceil + 1$ viele Bits (Zellen) benötigt, um eine Binärkodierung des Zählerwerts (inklusive Vorzeichen) zu speichern.

Die robusten Komplexitätsklassen

Es würde nun naheliegen, die Klassen wie z.B. DTIME($\mathcal{O}(n)$) zu untersuchen. Die bislang definierten Klassen sind jedoch nicht robust unter Veränderungen des Berechnungsmodels. Dies verdeutlichen wir anhand des folgenden Beispiels.

7.18 Beispiel
Betrachte die Sprache

$$\mathsf{COPY} = \{w\#w \mid w \in \{0,1\}^*\}$$

welche das Entscheidungsproblem repräsentiert, bei dem zwei Strings $w, w' \in \{0,1\}^*$ gegeben sind, deren Gleichheit entschieden werden soll.

Für dieses Problem gilt

a) $\mathsf{COPY} \in \mathrm{DTIME}_1(\mathcal{O}(n^2))$,

b) $\mathsf{COPY} \in \mathrm{DTIME}_2(\mathcal{O}(n))$,

c) $\mathsf{COPY} \notin \mathrm{DTIME}_1(\mathcal{O}(n))$.

Denn:

a) Eine 1-Band-DTM kann COPY mit quadratischem Zeitaufwand entscheiden. Dazu vergleicht sie das erste Eingabesymbol mit dem ersten Symbol nach #, das zweite Eingabesymbol mit dem zweitem Symbol nach # usw. Das Eingabewort $w\#w'$ ist genau dann in COPY, wenn alle Vergleiche erfolgreich sind. Da pro Symbol der Eingabe ein kompletter Durchlauf durch das Band nötig ist, hat der resultierende Algorithmus quadratischen Zeitverbrauch.

b) Eine 2-Band-DTM kann COPY mit linearem Zeitaufwand entscheiden. Dazu kopiert sie zunächst den zweiten Teil der Eingabe auf das erste Arbeitsband. Nun ist es möglich, die beiden Teile der Eingabe in nur einem Durchlauf zu vergleichen. Der resultierende Algorithmus benötigt 3 Durchläufe durch das Band, hat also Zeitverbrauch $3n \in \mathcal{O}(n)$.

c) Man kann zeigen, dass eine 1-Band-DTM das Problem COPY nicht mit echt weniger als quadratischem Zeitaufwand entscheiden kann. Hierzu betrachten man sogenannte Crossing-Sequenzen. Der komplexe formale Beweis wäre dem Ziel dieser Vorlesung nicht dienlich.

Wir wir am Beispiel sehen ändert ein kleiner Wechsel des Berechnungsmodell – von 1-Band zu 2-Band DTMs – die Klasse der in Linearzeit lösbaren Probleme. Analog sind auch die Klassen $\mathrm{DTIME}_m\big(\mathcal{O}(n^k)\big), \mathrm{NTIME}_m\big(\mathcal{O}(n^k)\big), \mathrm{DSPACE}_m\big(\mathcal{O}(n^k)\big), \mathrm{NSPACE}_m\big(\mathcal{O}(n^k)\big)$ für beliebige Konstanten k, m nicht robust unter Veränderungen des Berechnungsmodels.

Dies stellt ein großes Problem für die Theoretische Informatik dar: Die in der Theoretischen Informatik gezeigten Resultate sollen natürlich auf *reale* Programme und Computer anwendbar sein. Wenn nun bereits eine minimale Veränderung des Berechnungsmodels die zugehörigen Komplexitätsklassen verändert, so ist klar, dass dies auch für einen Wechsel vom theoretischen Modell der Turing-Maschine zu einem realen Computersystem (z.B. Java-Programmen) gilt. Die theoretischen Resultate, die wir unter Verwendung der grundlegenden Zeit- und Platzkomplexitätsklassen zeigen können, können also zunächst in der Praxis **nicht angewandt werden**.

Um dieses Problem zu lösen, haben sich zwei verschiedene Ansätze herausgebildet.

- Im sogenannten **Track A** der Theoretischen Informatik, der **Algorithmik**, trifft man spezifische Annahmen an das betrachtete Berechnungsmodell, und zeigt dann genaue Resultate. Die Resultate können in der Praxis angewandt werden, wenn die spezifischen Annahmen erfüllt sind, besitzen jedoch keine universelle Gültigkeit.
- Im **Track B** der Theoretischen Informatik, der **Verifikation**, zeigt man Resultate, die universell – also für (fast) jedes Berechnungsmodell – gültige Resultate, die allerdings ungenauer sind.

Wir verfolgen hier den zweiten Weg. Um das oben genannte Problem der Robustheit zu lösen, macht man die folgende Beobachtung. Die Implementierung eines Algorithmus in einem Berechnungsmodell mit Zeitverbrauch $f(n)$ hat nach dem Übertragen in ein anderes Berechnungsmodell üblicherweise Zeitverbrauch $f(n^k)$. Hierbei ist k eine Konstante, die von den involvierten Berechnungsmodellen abhängt, allerdings nicht vom Algorithmus und dessen Eingaben.

Damit man eine robuste Komplexitätsklassen erhält, muss man ihre Definitionen also so wählen, dass sie unter **polynomiellen Blowup** der Form $n \mapsto n^k$ abgeschlossen sind.

7.19 Definition
Wir definieren nun die **robusten Komplexitätsklassen**.

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

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 k gibt, so dass es sich in $\mathrm{DTIME}\big(\mathcal{O}(n^k)\big)$ lösen lässt. Im Allgemeinen sieht man P als die Klasse der **effizient lösbaren** Probleme an.

7.20 Bemerkung
Man kann auch noch größere Klassen betrachten, zum Beispiel zu jeder Zahl $m \in \mathbb{N}$ die Klassen mEXP und m$\mathsf{EXPSPACE}$, 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 eine Konstante k gibt, so dass $\mathcal{L}$ in $\mathrm{DTIME}\Big(2^{2^{\mathcal{O}(n^k)}}\Big)$ ist.

7.C Grundlegende Relationen

Wir wollen nun einige Relationen zwischen den grundlegenden Zeit- und Komplexitätsklassen und die daraus resultierenden Relationen zwischen den robusten Komplexitätsklassen beweisen.

- Insbesondere wollen wir beweisen, dass die robusten Komplexitätsklassen in der Reihenfolge, in der sie in Definition 7.19 gelistet sind, eine aufsteigende Kette bilden. Es gilt $\mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{P} \subseteq \mathsf{P} \subseteq \mathsf{PSPACE} \subseteq \ldots$
- Wir wollen zeigen, dass die robusten Komplexitätsklassen tatsächlich robust sind gegen Veränderungen des Berechnungsmodells, z.B. gegen Veränderung der Bandanzahl.

Direkt aus den Definitionen ergeben sich Inklusionen zwischen den deterministischen und den nicht-deterministischen Komplexitätsklassen.

7.21 Lemma
Für jede Funktion $f\colon \mathbb{N} \to \mathbb{N}$ und Bandanzahl $m \in \mathbb{N}$ gilt

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

Beweis:
Jede DTM kann auch als NTM mit dem gleichen Platz- und Zeitverbrauch gesehen werden. □

7.22 Korollar
Es gilt $\mathsf{L} \subseteq \mathsf{NL}$, $\mathsf{P} \subseteq \mathsf{NP}$, $\mathsf{PSPACE} \subseteq \mathsf{NPSPACE}$, $\mathsf{EXP} \subseteq \mathsf{NEXP}$ usw.

Beweis:
Wir beweisen exemplarisch $\mathsf{P} \subseteq \mathsf{NP}$. Es gilt $\mathsf{P} = \bigcup_{k\in\mathbb{N}} \mathrm{DTIME}\big(\mathcal{O}(n^k)\big)$, $\mathsf{NP} = \bigcup_{k\in\mathbb{N}} \mathrm{NTIME}\big(\mathcal{O}(n^k)\big)$. Für jedes k gilt mit dem obigen Lemma und der Definition von NP: $\mathrm{DTIME}\big(\mathcal{O}(n^k)\big) \subseteq \mathrm{NTIME}\big(\mathcal{O}(n^k)\big) \subseteq \mathsf{NP}$. Also gilt auch $\mathsf{P} \subseteq \mathsf{NP}$. □

Als nächstes stellen wir fest, dass der Zeitverbrauch einer Maschine den Platzverbrauch beschränkt: In jedem Schritt kann eine Maschine höchstens eine neue Zelle beschreiben.

7.23 Lemma
Für jede Funktion $f\colon \mathbb{N} \to \mathbb{N}$ und jede Bandanzahl $m > 0$ gilt

$$\mathrm{DTIME}_m(f) \subseteq \mathrm{DSPACE}_m(f),$$
$$\mathrm{NTIME}_m(f) \subseteq \mathrm{NSPACE}_m(f) \ .$$

Beweis:
Betrachte $\mathcal{L}(M) \in \mathrm{DTIME}_m(f)$, wobei M eine f-zeitbeschränkte m-Band DTM ist. Wir konstruieren eine DTM M' mit read-only Eingabe und m Arbeitsbändern.

Da die Eingabe von M' read-only ist, nutzt M' das erste Arbeitsband, um die Veränderungen, die M am Eingabeband vornimmt, zu speichern. Formal bewegt M' die Kopfpositionen auf dem Eingabeband und dem ersten Arbeitsband synchron. Wenn M das Symbol a auf dem Eingabeband durch b ersetzt, dann lässt M' die Eingabe unverändert und schreibt b an der gleichen Stelle auf das erste Arbeitsband. Beim Auswählen einer Transition richtet sich M' nach dem Symbol auf dem ersten Arbeitsband, wenn dieses nicht $\sqcup$ ist, ansonsten nach dem Symbol auf dem Eingabeband. Die restlichen $m-1$ Arbeitsbänder entsprechenden den restlichen $m-1$ Bändern von M.

Die beiden Maschinen entscheiden die selbe Sprache, $\mathcal{L}(M) \subseteq \mathcal{L}(M')$. Jeder Berechnungschritt von M entspricht einem Schritt von M' und umgekehrt. Also belegt M' pro Schritt von M höchstens eine neue Zelle auf dem Band. Da M f-zeitbeschränkt war, ist M' f-platzbeschränkt. Also ist M' ein f-platzbeschränkter Entscheider mit read-only Eingabe und m Arbeitsbändern; es gilt $\mathcal{L}(M) = \mathcal{L}(M') \in \mathrm{DSPACE}_m(f)$.

Für nicht-deterministische Maschinen M ist der Beweis analog. □

Wie bei obigen Korollar lassen sich daraus Inklusionen zwischen den robusten Komplexitätsklassen folgern.

7.24 Korollar
Es gilt $\mathsf{P} \subseteq \mathsf{PSPACE}$, $\mathsf{NP} \subseteq \mathsf{NPSPACE}$, $\mathsf{EXP} \subseteq \mathsf{EXPSPACE}$ und $\mathsf{NEXP} \subseteq \mathsf{NEXPSPACE}$.

Man kann tatsächlich sogar eine stärkere Variante des obigen Lemma beweisen: Eine Zeitschranke für nicht-deterministische Maschinen ist eine Platzschranke für deterministische Maschinen. Wenn wir also eine Zeitschranke in eine Platzschranke konvertieren, können wir ohne zusätzliche Kosten determinisieren.

7.25 Lemma
Für jede Funktion $f\colon \mathbb{N} \to \mathbb{N}$ und jede Bandanzahl $m > 0$ gilt

$$\mathrm{NTIME}_m(f) \subseteq \mathrm{DSPACE}_{m+1}(f) \ .$$

Beweis:
In Theorem 1.17 haben wir gesehen, dass man zu einer nicht-deterministischen Turing-Maschine M eine deterministische Turing-Maschine M' konstruieren kann. Es gilt $\mathcal{L}(M) = \mathcal{L}(M')$, und wenn M ein Entscheider ist, dann auch M'.

Wir wiederholen kurz die Konstruktion von M': Eine Berechnung von M' zu Eingabe x führt einen Breitensuche im Berechnungsbaum von M zu Eingabe x durch. M' akzeptiert, wenn in einem Breitendurchlauf durch den Baum eine akzeptierende Konfiguration gefunden wird. Wenn M eine m-Band-Maschine ist, dann ist M' eine $(m+2)$-Band-Maschine:

- Das erste Band speichert die Eingabe x und ist read-only.
- Das zweite Band speichert den Knoten im Berechnungsbaum, bei dem der Breitendurchlauf angekommen ist. Hierzu speichern wir eine Sequenz von Richtungen, die angeben, welcher Weg im Berechnungsbaum genommen werden muss, um zum aktuellen Knoten zu kommen.
- Die restlichen m Bänder werden verwendet, um die Konfiguration von M zu speichern, die dem aktuellen Knoten im Berechnungsbaum entspricht.

Wenn M f-zeitbeschränkt ist, dann ist die Länge jeder Berechnung zu einer Eingabe der Länge n – also die Höhe des Berechnungsbaums – durch $f(n)$ beschränkt. Wir analysieren den Platzverbrauch von M':

- Die Eingabe ist read-only; ihr Platzverbrauch wird nicht mitgezählt.
- Da der Berechnungsbaum höchstens Höhe $f(n)$ hat, hat auch die Sequenz der Richtungen, die einen beliebigen Knoten im Baum beschreibt, höchstens Länge $f(n)$.
- Wie in Lemma 7.23 ist eine Zeitschranke für M auch eine Platzschranke auf den restlichen m Bändern von M'.

M' braucht also auf allen $m+1$ Arbeitsbändern höchstens $f(n)$ Platz Also gilt $\mathcal{L}(M) = \mathcal{L}(M') \in \mathrm{DSPACE}_{m+1}(f)$. □

Die obige Konstruktion führt ein neues Arbeitsband ein. Wir werden gleich sehen, dass man dieses eliminieren kann, ohne den Platzverbrauch zu erhöhen. Damit gelten die folgenden Inklusionen.

7.26 Korollar
Es gelten $\mathsf{NP} \subseteq \mathsf{PSPACE}$ und $\mathsf{NEXP} \subseteq \mathsf{EXPSPACE}$.

Nun beweisen wir, dass sich Mehrband-Maschinen zu Einband-Maschinen umwandeln lassen, wobei sich der Zeitverbrauch quadriert und der Platzverbrauch gleich bleibt.

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

$$\begin{aligned} &\mathrm{DTIME}_m(f) \subseteq \mathrm{DTIME}_1(\mathcal{O}(f \cdot f)), \quad &&\mathrm{DSPACE}_m(f) \subseteq \mathrm{DSPACE}_1(\mathcal{O}(f)), \\ &\mathrm{NTIME}_m(f) \subseteq \mathrm{NTIME}_1(\mathcal{O}(f \cdot f)), \quad &&\mathrm{NSPACE}_m(f) \subseteq \mathrm{NSPACE}_1(\mathcal{O}(f)) \ . \end{aligned}$$

Beweis:
In Lemma 1.8 haben wir gesehen, dass sich eine m-Band-Maschine M in eine Einband-Maschine M' umwandeln lässt. Es gilt $\mathcal{L}(M) = \mathcal{L}(M')$ und wenn M ein Entscheider ist, dann auch M'.

Der Platzverbrauch von M' richtet sich nach dem längsten Band von M. Damit ist eine Platzschranke für M auch eine für M' und $\mathrm{DSPACE}_m(f) \subseteq \mathrm{DSPACE}_1(\mathcal{O}(f))$ und $\mathrm{NSPACE}_m(f) \subseteq \mathrm{NSPACE}_1(\mathcal{O}(f))$ sind bewiesen.

Der Zeitverbrauch von M' ist höher als der von M: Um eine einzige Transition von M zu simulieren, muss M' mehrfach (allerdings konstant oft) den kompletten Bandinhalt durchlaufen, um die Symbole an den Kopfpositionen zu lesen, die korrekte Transition auszuwählen und den Bandinhalt und die Kopfpositionen anzupassen. Wie in Lemma 7.23 argumentiert ist die Zeitschranke f auch eine Platzschranke. Bei einer Eingabe der Länge n ist der Bandinhalt von M (und damit auch der Bandinhalt von M') also höchstens $f(n)$ lang. Um einen Schritt von M zu simulieren, benötigt M' also höchstens $\mathcal{O}(f(n))$ viele Schritte. Um alle $f(n)$ vielen Schritte zu simulieren, benötigt M' also höchstens $\mathcal{O}(f(n) \cdot f(n))$ viele Schritte. Damit sind $\mathrm{DTIME}_m(f) \subseteq \mathrm{DTIME}_1(\mathcal{O}(f \cdot f))$ und $\mathrm{NTIME}_m(f) \subseteq \mathrm{NTIME}_1(\mathcal{O}(f \cdot f))$ bewiesen. □

7.28 Bemerkung
Wenn wir im obigen Beweisen sagen, dass M' höchstens höchstens $\mathcal{O}(f(n))$ viele Schritte benötigt, meinen wir eigentlich, dass es eine Funktion $g\colon \mathbb{N} \to \mathbb{N}$ gibt, so dass

- zu jeder Eingabe der Länge n der Funktionswert $g(n)$ die Schrittzahl von M' beschränkt und
- $g \in \mathcal{O}(f)$ gilt.

Wir verwenden also die $\mathcal{O}$-Notation für Funktionswerte und meinen damit, dass es eine Funktion aus der entsprechenden Klasse gibt, die die passenden Funktionswerte liefert.

Mit dem obigen Lemma kann man zeigen, dass die robusten Komplexitätsklassen unter Veränderung der Bandanzahl abgeschlossen sind. Wir haben die Klassen in Definition 7.19 unter Verwendung der 1-Band-Versionen der grundlegenden Zeit- und Platzkomplexitätsklassen definiert. Man könnte jedoch zur Definition jedoch auch m-Band-Maschinen für ein beliebiges $m > 0$ verwenden und würde immer noch die selbe Definition erhalten. Wir zeigen dies exemplarisch im Fall von P.

7.29 Lemma
Für jedes $m > 0$ gilt $\mathsf{P} = \bigcup_{k\in\mathbb{N}} \mathrm{DTIME}_m\big(\mathcal{O}(n^k)\big)$.

Beweis:
Offensichtlich gilt $\mathrm{DTIME}_1\big(\mathcal{O}(n^k)\big) \subseteq \mathrm{DTIME}_m\big(\mathcal{O}(n^k)\big)$, da jede 1-Band-DTM auch als m-Band-DTM aufgefasst werden kann. Damit gilt $\mathsf{P} \subseteq \bigcup_{k\in\mathbb{N}} \mathrm{DTIME}_m\big(\mathcal{O}(n^k)\big)$.

Für die andere Richtung betrachten wir $\mathrm{DTIME}_m\big(\mathcal{O}(n^k)\big) \subseteq \mathrm{DTIME}_1\big(\mathcal{O}(n^k \cdot n^k)\big) = \mathrm{DTIME}_1\big(\mathcal{O}(n^{2k}$ unter Verwendung von Lemma 7.27 und den Potenzgesetzen. Die Klasse $\mathrm{DTIME}_1\big(\mathcal{O}(n^{2k})\big)$ ist allerdings für jedes k in P enthalten, also gilt

$$\bigcup_{k\in\mathbb{N}} \mathrm{DTIME}_m\big(\mathcal{O}\big(n^k\big)\big) \subseteq \bigcup_{k\in\mathbb{N}} \mathrm{DTIME}_1\big(\mathcal{O}\big(n^{2k}\big)\big) \subseteq \bigcup_{k\in\mathbb{N}} \mathsf{P} = \mathsf{P}\ .$$

□

Unser letztes Lemma zeigt eine Inklusion zwischen Platz- und Zeitkomplexitätsklassen. Eine f-platzbeschränkte Turing-Maschine ist auch eine $2^{\mathcal{O}(f)}$-zeitbeschränkte Turing-Maschine. Zudem lässt sich beim Übergang von der Platzschranke f zur exponentiellen Zeitschranke $2^{\mathcal{O}(f)}$ ohne zusätzliche Kosten determinisieren. Da uns Lemma 7.27 die Elimination zusätzlicher Bänder erlaubt, betrachten wir hier nur die 1-Band-Version.

7.30 Proposition
Für jede Funktion $f\colon \mathbb{N} \to \mathbb{N}$ mit $f(n) \geq \log n \forall n$ gilt

$$\mathrm{NSPACE}(f) \subseteq \mathrm{DTIME}\big(2^{\mathcal{O}(f)}\big)\ .$$

Beweis:
Es sei $\mathcal{L}(M) \in \text{NSPACE}(f)$, wobei M ein f-platzbeschränkter nichtdeterministischer Entscheider mit read-only Eingabe und einem Arbeitsband ist. Betrachte eine Eingabe $x \in \Sigma^*$ der Länge $|x| = n$.

Wir rechnen zunächst aus, wie viele verschiedene Konfigurationen von M in Berechnungen zur Eingabe x auftreten können. Eine Konfiguration von M lässt sich durch den Inhalt der beiden Bänder und durch die Kopfposition in den Bändern beschreiben. Da das Eingabeband read-only ist, müssen wir tatsächlich sogar nur den Inhalt des Arbeitsbandes beschreiben. Eine Konfiguration ist also von der Form $\{1, \ldots, n\} \times \Gamma^k \times \{1, \ldots, k\}$, wobei k die aktuelle Länge des Arbeitsbandes ist. Da M f-platzbeschränkt ist, gilt $k \leq f(n)$ und wir erhalten

$$\left|\{1, \ldots, n\} \times \Gamma^{f(n)} \times \{1, \ldots, f(n)\}\right| = n \cdot |\Gamma|^{f(n)} \cdot f(n).$$

Wir behaupten, dass dieser Wert in $2^{\mathcal{O}}(f(n))$ liegt.

- $n \in 2^{\mathcal{O}}(f(n))$ gilt, da wir $f(n) \geq \log n$ angenommen hatten.
- $|\Gamma|^{f(n)} = 2^{\log|\Gamma| \cdot f(n)}$ gilt unter Verwendung der Potenzgesetze. Hierbei ist $\log|\Gamma|$ eine Konstante, die nur von der Maschine M, aber nicht von der Eingabe x abhängt. Also gilt $2^{\log|\Gamma| \cdot f(n)} \in 2^{\mathcal{O}(f(n))}$.
- $f(n) \in 2^{\mathcal{O}(f(n))}$ gilt, da $2^{f(n)} \geq f(n)$.

Also ist die Anzahl der Konfiguration in

$$2^{\mathcal{O}(f(n))} \cdot 2^{\mathcal{O}(f(n))} \cdot 2^{\mathcal{O}(f(n))} = 2^{3 \cdot \mathcal{O}(f(n))} = 2^{\mathcal{O}(f(n))} .$$

Als nächstes rechnen wir aus, wie lange eine Berechnung von M zu Eingabe x maximal zum Halten brauchen kann. Wir behaupten, dass dies ebenfalls der oben ausgerechnete Wert ist: Die Platzschranke f induziert also die Zeitschranke $2^{\mathcal{O}(f(n))}$.

Angenommen dies wäre nicht der Fall, und M hätte zu Eingabe x eine Berechnung

$$c_0 \to c_1 \to \ldots c_j$$

mit $j > 2^{\mathcal{O}(f(n))}$, so dass keine der c_i eine Haltekonfiguration ist. Da die Anzahl der Konfigurationen in der Berechnung die Anzahl der verschiedenen Konfigurationen von M zu Eingabe x übersteigt, muss die Sequenz eine Wiederholung beinhaltet. Es gibt also Indizes $i < i'$ mit $c_i = c_{i'}$. Durch unendliche Wiederholung des Infixes

$c_i \to \ldots \to c_{i'}$ konstruieren wir eine nicht-haltende Berechnung von M zu Eingabe x, nämlich

$$c_0 \to c_1 \to \ldots \to c_i \to \ldots \to c_{i'} = c_i \to \ldots \to c_{i'} = c_i \to \ldots \ .$$

Dies ist ein Widerspruch zur Annahme, dass M ein Entscheider ist. Also ist M $2^{\mathcal{O}(f)}$-zeitbeschränkt.

Wir haben damit bereits $\mathrm{DSPACE}(f) \subseteq \mathrm{DTIME}\big(2^{\mathcal{O}(f)}\big)$ und $\mathrm{NSPACE}(f) \subseteq \mathrm{NTIME}\big(2^{\mathcal{O}(f)}\big)$ gezeigt. Um die gewünschte stärkere Aussage zu zeigen, betrachten wir den Konfigurationsgraph von M.

Der **Konfigurationsgraph** von M zu einer Eingabe x ist ein Graph, dessen Knoten die Konfigurationen von M zu dieser Eingabe repräsentieren und dessen Kanten Berechnungsschritte von M. Im Gegensatz zum Berechnungsbaum verschmelzen wir hier allerdings Knoten, die die selbe Konfiguration repräsentieren, die auf verschiedene Arten erreicht werden kann. Das Resultat ist ein azyklischer (kreisfreier) gerichteter Graph (**DAG** - *directed acyclic graph*). (Ein Kreis im Graphen würde wie oben argumentiert die Existenz einer nicht-haltenden Berechnung implizieren.)

Wir konstruieren nun eine DTM M', die zu einer gegebenen Eingabe x den Konfigurationsgraphen von M zu x konstruiert und ihn nach einer akzeptierenden Berechnung durchsucht. Der Konfigurationsgraph besteht mit der obigen Abschätzung aus maximal $2^{\mathcal{O}(f(|x|))}$ vielen Knoten. Damit kann M' so konstruiert werden, dass ihr Zeitverbrauch maximal $2^{\mathcal{O}(f(|x|))}$ ist, womit die gewünschte Aussage folgt. Die formalen Details der Konstruktion lassen wir aus. □

Aus dem Lemma folgen die fehlenden Inklusionen zwischen den Klassen.

7.31 Korollar
Es gilt $\mathsf{NL} \subseteq \mathsf{P}$ und $\mathsf{NPSPACE} \subseteq \mathsf{EXP}$.

Beweis:
Wir beweisen $\mathsf{NL} \subseteq \mathsf{P}$. Es gilt $\mathsf{NL} = \mathrm{NSPACE}(\mathcal{O}(\log n)) \subseteq \mathrm{DTIME}\big(2^{\mathcal{O}(\log n)}\big) = \mathrm{DTIME}(\mathcal{O}(n)) \subseteq \mathsf{P}$ unter Verwendung des obigen Lemmas und der Definitionen von NL und P. □

Die Korollare, die wir in diesem Unterkapitel gezeigt haben, beweisen insgesamt das folgende Theorem.

7.32 Theorem
Die robusten Komplexitätsklassen bilden eine aufsteigende Kette,

$$\mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE} \subseteq \mathsf{NPSPACE} \subseteq \mathsf{EXP} \subseteq \mathsf{NEXP} \subseteq \mathsf{EXPSPACE} \subseteq \ldots$$

Zum Abschluss des Unterkapitels beweisen wir noch, dass die robusten Komplexitätsklassen unter polynomiellen Blowup abgeschlossen sind. Wenn Beispielsweise $\mathrm{DTIME}(f(n)) \subseteq \mathsf{P}$, dann gilt auch für jedes $k \in \mathbb{N}$ dass $\mathrm{DTIME}\big(f(n^k)\big) \subseteq \mathsf{P}$.

7.33 Lemma
Die robusten Komplexitätsklassen sind unter polynomiellem Blowup abgeschlossen.

Beweis:
Für alle Klassen außer L und NL folgt dies direkt aus der Definition. Formal argumentiert man wie im Beweis von Lemma 7.29.

Betrachte $\mathsf{L} = \mathrm{DSPACE}(\mathcal{O}(\log n))$. Wir wenden die polynomielle Transformation $n \mapsto n^k$ an und erhalten $\mathrm{DSPACE}\big(\mathcal{O}(\log(n^k))\big)$. Unter Verwendung der Logarithmusgesetze gilt jedoch

$$\log(n^k) = \log(n \cdot n \cdot \ldots \cdot n) = \log n + \log n + \ldots + \log n = k \cdot \log n \in \mathcal{O}(\log n) \ ,$$

also gilt $\mathrm{DSPACE}\big(\mathcal{O}(\log(n^k))\big) = \mathrm{DSPACE}(\mathcal{O}(\log n)) = \mathsf{L}$. Der Fall von NL ist analog. □

7.D Komplementklassen

Weitere wichtige Komplexitätsklassen sind die sogenannten **Komplementklassen**.

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.

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

$$\mathsf{co}\mathcal{C} = \big\{\overline{\mathcal{L}} \;\big|\; \mathcal{L} \in \mathcal{C}\big\}$$

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

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

Mit der obigen Definition sind nun insbesondere die Komplementklassen $\mathsf{coDTIME}(f)$, $\mathsf{coNTIME}(f)$, $\mathsf{coDSPACE}(f)$, $\mathsf{coNSPACE}(f)$ sowie coL, coNL, coP, coNP usw. definiert.

Man stellt allerdings leicht fest, dass die deterministischen Komplexitätsklassen gleich ihrer Komplementklassen sind, da sie unter Komplementbildung abgeschlossen sind.

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

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

Beweis:
Wir beweisen exemplarisch $\mathsf{coDSPACE}(f) \subseteq \mathrm{DSPACE}(f)$. Betrachte ein Problem aus $\mathsf{coDSPACE}(f)$. Dieses ist von der Form $\overline{\mathcal{L}}$ mit $\mathcal{L} \in \mathrm{DSPACE}(f)$. Es sei $\mathcal{L} = \mathcal{L}(M)$, wobei M ein f-platzbeschränkter Entscheider ist. Wir konstruieren die Turing-Maschine $\overline{M}$, die aus M durch das Vertauschen der Rollen der Haltezustände von q_{acc} und q_{rej} entsteht. Eine akzeptierende Berechnung von M entspricht also einer abweisenden Berechnung von $\overline{M}$ und umgekehrt; es gilt $\mathcal{L}(\overline{M}) = \overline{\mathcal{L}(M)}$. Zudem hat $\overline{M}$ den selben Zeit- und Platzverbrauch wie M.

Also ist $\overline{\mathcal{L}} = \mathcal{L}(\overline{M}) \in \mathrm{DSPACE}(f)$. □

Aus dieser Eigenschaft von DSPACE und DTIME folgt, dass auch alle robusten Komplexitätsklassen die unter Verwendung von DSPACE oder DTIME definiert sind unter Komplementbildung abgeschlossen sind.

7.36 Korollar
Es gilt

$$\mathsf{L} = \mathsf{coL}\ , \qquad \mathsf{PSPACE} = \mathsf{coPSPACE}\ ,$$
$$\mathsf{P} = \mathsf{coP}\ , \qquad \mathsf{EXP} = \mathsf{coEXP}\ .$$

Der Beweis des obigen Lemmas kann nicht verwendet werden, um zu zeigen, dass auch NTIME(f) und NSPACE(f) ihren Komplementklassen entsprechen. Hierzu rufen wir uns in Erinnerung, dass wir bei NTMs **existentiellen Nichtdeterminismus** verwenden,

$$\mathcal{L}(M) = \{x \mid \textbf{ es gibt } \text{eine akzeptierende Berechnung von } M \text{ zu } x\} .$$

Wenn wir nun die TM $\overline{M}$ durch Umdrehen der Haltezustände konstruieren, erhalten wir die Sprache

$$\begin{aligned}\mathcal{L}\big(\overline{M}\big) &= \big\{x \mid \textbf{ es gibt } \text{eine akzeptierende Berechnung von } \overline{M} \text{ zu } x\big\} \\ &= \{x \mid \textbf{ es gibt } \text{eine abweisende Berechnung von } M \text{ zu } x\} .\end{aligned}$$

Diese Sprache ist nicht die gesuchte Komplementsprache von $\mathcal{L}(M)$,

$$\overline{\mathcal{L}(M)} = \big\{x \mid \textbf{ alle } \text{Berechnungen von } \overline{M} \text{ zu } x \text{ sind abweisend}\big\} .$$

Ein ähnliches Problem zur Komplementierung endlicher Automaten ist aus „Theoretische Informatik 1“ bekannt. Um diese Probleme zu lösen, könnte man z.B. *universellen Nichtdeterminismus* betrachten.

Ob die nichtdeterministische Komplexitätsklassen abgeschlossen unter Komplement sind, also z.B. $\mathsf{NL} \stackrel{?}{=} \mathsf{coNL}$, $\mathrm{NPSPACE} \stackrel{?}{=} \mathsf{coNPSPACE}$, $\mathsf{NP} \stackrel{?}{=} \mathsf{coNP}$ 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 $\mathsf{NP} \stackrel{?}{=} \mathsf{coNP}$ ist offen, man glaubt allerdings, dass die beiden Klassen unterschiedlich sind.

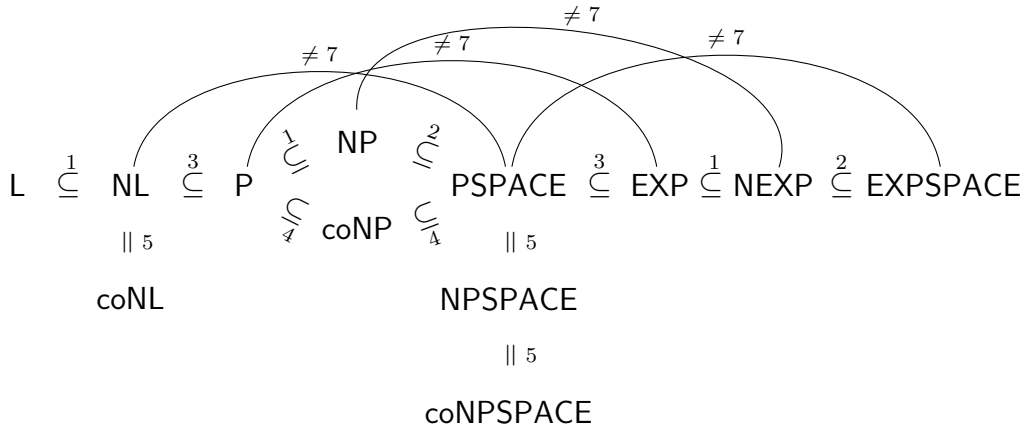
Man kann zeigen, dass coNP die selben Inklusionen erfüllt, die auch für NP gelten.

7.37 Lemma
Es gilt $\mathsf{P} \subseteq \mathsf{coNP} \subseteq \mathsf{PSPACE}$.

Beweis:
Wir wissen, dass $\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE}$ gilt. Der Operator co$-$ ist monoton, also gilt $\mathsf{coP} \subseteq \mathsf{coNP} \subseteq \mathsf{coPSPACE}$. Nun verwenden wir Korollar 7.36 und setzen $\mathsf{P} = \mathsf{coP}$ und $\mathsf{PSPACE} = \mathsf{coPSPACE}$ ein, um die gewünschte Aussage zu erhalten. □

8. Eine Landkarte der Komplexitätstheorie

Wir zeichnen ein Diagramm der Beziehungen zwischen den robusten Komplexitätsklassen. Wir geben sowohl die bereits gezeigten Relationen, als auch die Relationen die wir im Rest der Vorlesungen zeigen werden an.



Bereits gezeigte Inklusionen:

1. Nichtdeterminismus ist potentiell mächtiger als Determinismus. Korollar von Lemma 7.21.
2. Jede Zeitschranke für nicht-deterministische Maschinen ist auch eine Platzschranke für deterministische Maschinen. Korollar von Lemma 7.25.
3. Eine Platzschranke für nicht-deterministische Maschinen liefert eine exponentiell größere Zeitschranke für deterministische Maschinen. Korollar von Proposition 7.30.
4. coNP erfüllt die selben Inklusionsbeziehungen wie NP, Lemma 7.37.

Weitere Relationen:

5. Der **Satz von Immerman und Szelepcsényi**, Theorem 2.1, kann verwendet werden, um zu zeigen, dass $\mathsf{coNSPACE}(f) = \mathrm{NSPACE}(f)$ gilt. Daraus folgen $\mathsf{NL} = \mathsf{coNL}$ und $\mathsf{NPSPACE} = \mathsf{coNPSPACE}$.
6. Der **Satz von Savitch**, Theorem 12.5, zeigt, dass man TMs determinisieren kann und dabei ihren Platzverbrauch quadriert. Daraus folgt insbesondere $\mathsf{NPSPACE} = \mathsf{PSPACE}$. Daraus folgt jedoch **nicht** $\mathsf{L} = \mathsf{NL}$.

7. Die **Hierarchiesätze für Zeit und Platz**, wie z.B. Theorem 13.5, zeigen, dass man mit echt mehr Zeit bzw. Platz echt mehr Probleme lösen kann. Daher gilt $\mathsf{NL} \subsetneq \mathsf{PSPACE} \subsetneq \mathsf{EXPSPACE}$, $\mathsf{P} \subsetneq \mathsf{EXP}$ und $\mathsf{NP} \subsetneq \mathsf{NEXP}$.

Aus den Hierarchiesätzen folgt z.B. dass mindestens eine der Inklusionen $\mathsf{NL} \subseteq \mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE}$ strikt sein muss. Wenn alle diese Inklusionen Gleichheiten wären, dann würde auch $\mathsf{NL} = \mathsf{PSPACE}$ gelten.

Man glaubt, dass alle Inklusionen im obigen Diagramm strikte Inklusionen sind und damit insbesondere $\mathsf{P} \neq \mathsf{NP}$ gilt. Zusätzlich glaubt man, dass $\mathsf{NP} \neq \mathsf{coNP}$ gilt. Diese Aussagen sind jedoch nicht bewiesen und damit seit über 40 Jahren ungelöste Probleme.

Diese Probleme, insbesondere die Frage nach der Gleichheit von P und NP, werden im Allgemeinen als die **wichtigsten offenen Probleme der Informatik** angesehen: **Wie ist das Verhältnis von deterministischer Zeit- zu nichtdeterministischer Zeitkomplexität? Wie ist das Verhältnis von Zeit- zu Platzkomplexität?**

Wir werden später die Komplexitätsklassen P und NP detailliert betrachten und begründen, warum die Frage nach ihrer Gleichheit ein so wichtiges Problem darstellt.

Zusätzlich zu den Relationen zwischen den Komplexitätsklassen werden wir auch **charakteristischen Probleme** zu jeder Komplexitätsklasse kennen lernen. Diese beinhalten

- für L: Grundlegende Probleme der Arithmetik, wie z.B. gegeben Zahlen x, y, z in Binärkodierung; gilt $x + y = z$?
- für NL: Suchprobleme in gerichteten Graphen, wie z.B. gegeben Graph G und Knoten s, t; gibt es einen Pfad von s nach t?
- für P: Auswertungsprobleme, wie z.B. CVP, das Auswertungsproblem für Schaltkreise
- für NP: Suchprobleme wie z.B. das berühmte Problem SAT, das Erfüllbarkeitsproblem der Aussagenlogik
- für PSPACE: QBF, die Auswertung quantifizierter Boolescher Formeln sowie das Universalitätsproblem für NFAs.

Zu all diesen Problemen glaubt man, dass man die kleinste Komplexitätsklasse gefunden hat, in der sie enthalten sind. Beispielsweise weiß man, dass $\mathsf{SAT} \in \mathsf{NP}$ gilt, und man glaubt, dass $\mathsf{SAT} \notin \mathsf{P}$ gilt. Da man jedoch $\mathsf{NP} \neq \mathsf{P}$ bislang nicht beweisen

konnte, behilft man sich wie folgt: Man zeigt für SAT (und die anderen oben genannten Probleme), dass sie zu den **schwersten Problemen in ihrer jeweiligen Klasse gehören**. Dies bedeutet:

- Angenommen $\mathsf{P} \neq \mathsf{NP}$. Dann folgt direkt $\mathsf{SAT} \notin \mathsf{P}$. Wenn es Probleme in NP gibt, die nicht in P sind, dann ist SAT eines von ihnen.
- Angenommen $\mathsf{SAT} \in \mathsf{P}$. Dann gilt $\mathsf{P} = \mathsf{NP}$. Wenn man SAT effizient lösen kann, dann auch jedes andere Problem in NP.

In den nächsten Kapiteln werden wir neben den weiteren Inklusionen zwischen den Komplexitätsklassen und den charakteristischen Problemen auch die Techniken kennen lernen, um diese Aussagen zu formalisieren.

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 $\mathsf{L} = \mathsf{NL}$ 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.

9.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 $\mathsf{L} = \mathsf{NL}$ 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 $A \leq_m^R B$.

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:

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

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 $A \leq_m^R B$ 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 ($\leq_m^{\text{poly}}$) und
- die Reduktionen, die mit logarithmischem Platz berechenbar sind ($\leq_m^{\log}$).

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 $\mathcal{O}(n^k)$-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 $A \leq_m^{\log} B$.

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 $A \leq_m^{\text{poly}} B$.

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 $\mathsf{L} \subseteq \mathsf{P}$ und sogar $\mathsf{NL} \subseteq \mathsf{P}$ 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 $A \leq_m^{\log} B$, dann auch $A \leq_m^{\text{poly}} B$.

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 $g \circ f \colon \Sigma_1^* \to \Sigma_3^*$ eine logspace-berechenbare Funktion.

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

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 $\mathcal{O}(n^d)$ 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 $\mathcal{O}(n^d)$ beschränkt, wenn wir ihn Binär speichern, benötigen wir also nur $\mathcal{O}(\log(n^d)) = \mathcal{O}(d \cdot \log n) = \mathcal{O}(\log n)$ 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 $\mathcal{L} \leq_m^{log} \{1\}$.
Hier bezeichnet $\{1\}$ die Sprache $\{1\} \subseteq \{0,1\}^*$.

b) Jede Sprache $\mathcal{L} \subseteq \Sigma^*$ in L mit $\mathcal{L} \neq \emptyset$ 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 $A \leq_m^{\log} B$. 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 $A \in \mathsf{L}$ gilt, dann folgt $\mathsf{NL} = \mathsf{L}$.

b) Falls A P-schwer bezüglich logspace-many-one-Reductions ist und $A \in \mathsf{NL}$ gilt, dann folgt $\mathsf{NL} = \mathsf{P}$.

9.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 $t \in V$
Frage:	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 $\text{bin}(i)$ 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

$$\mathsf{PATH} = \{G\#s\#t \mid G \text{ kodiert einen Graphen, in dem es einen Pfad von } s \text{ nach } t \text{ gibt}\}$$

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 $\mathcal{L} \in \mathsf{NL}$ 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 $s, t \in V$
Frage:	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.

9.C coNL und der Satz von der Satz von Immerman & Szelepcsényi

Analog zu NL kann man auch die Klasse coNL betrachten, welche die Komplemente der Probleme aus NL beinhaltet,

$$\mathsf{coNL} = \left\{ \overline{\mathcal{L}} \;\middle|\; \mathcal{L} \in \mathsf{NL} \right\} .$$

Beispielweise liegt $\overline{\mathsf{PATH}}$, das Komplementproblem von PATH in coNL.

Unerreichbarkeit ($\overline{\mathsf{PATH}}$)	
Gegeben:	Gerichteter Graph $G = (V, R)$, Quellknoten $s \in V$, Zielknoten $t \in V$
Frage:	Gibt es **keinen** Pfad von s nach t in G?

Mit dem Satz von Immerman & Szelepcsényi, Theorem 2.1, lässt sich zeigen, dass $\mathsf{coNL} = \mathsf{NL}$ gilt. Für jedes Problem aus NL ist also auch das Komplementproblem in NL enthalten.

9.15 Theorem: Immerman & Szelepcsényi; Komplexitätstheoretische Fassung
$\mathsf{coNL} = \mathsf{NL}$.

Um diese Version des Satzes zu beweisen, beobachtet zunächst, dass unser Beweis der Originalfassung einen Algorithmus für $\overline{\mathsf{PATH}}$ liefert. Dieser Algorithmus verwendet Nichtdeterminismus sowie zusätzlich zur Eingabe nur logarithmischen Platz. Wir haben also damals tatsächlich das Folgende gezeigt.

9.16 Proposition
$\overline{\mathsf{PATH}} \in \mathsf{NL}$.

Hieraus lässt sich leicht folgern, dass $\mathsf{coNL} = \mathsf{NL}$ gilt.

Beweis von Theorem 9.15:
Betrachten wir ein beliebiges Problem aus coNL. Dieses lässt sich schreiben als $\overline{\mathcal{L}}$ mit $\mathcal{L} \in \mathsf{NL}$.

Da PATH NL-vollständig ist, so ist $\overline{\mathsf{PATH}}$ coNL-vollständig: Wenn f eine logspace-Reduktion von $\mathcal{L}$ auf PATH ist, dann ist die selbe Funktion f auch eine logspace-Reduktion von $\overline{\mathcal{L}}$ auf $\overline{\mathsf{PATH}}$. Es gilt also $\overline{\mathcal{L}} \leq_m^{\log} \overline{\mathsf{PATH}}$. Wie oben argumentiert ist $\overline{\mathsf{PATH}}$ in NL enthalten. Damit ist nun allerdings auch $\overline{\mathcal{L}}$ in NL enthalten, da es leichter als ein Problem aus NL ist (siehe Lemma 9.10).

Wir haben also $\mathsf{coNL} \subseteq \mathsf{NL}$ gezeigt. Die andere Inklusion kann man analog beweisen. □

Tatsächlich lässt sich sogar eine verallgemeinerte Aussage zeigen: Für jede Platzklasse $\mathcal{C}$ gilt $\mathsf{co}\mathcal{C} = \mathcal{C}$.

9.17 Theorem: Immerman & Szelepcsényi; Verallgemeinerte komplexitätstheoretische Fassung
Es sei $f\colon \mathbb{N} \to \mathbb{N}$ mit $f(n) \geq \log n$ für alle n. Dann gilt

$$\mathsf{co}\mathrm{NSPACE}(\mathcal{O}(f)) = \mathrm{NSPACE}(\mathcal{O}(f)) \ .$$

Skizze:
Wir skizzieren den Beweis kurz. Sei $\mathcal{L} \in \text{NSPACE}(f)$, also $\mathcal{L} = \mathcal{L}(M)$ für eine f-platzbeschränkte NTM M. Wir müssen zeigen, dass sich auch das Komplementproblem $\overline{\mathcal{L}}$ durch eine $\mathcal{O}(f)$-platzbeschränkte Maschine lösen lässt.

Die Maschine M hat zu jeder Eingabe x eine eindeutige initiale Konfiguration c_0. Wir nehmen nun an, dass wenn M die Eingabe x akzeptiert, auch eine eindeutige akzeptierende Konfiguration c_{accept} existiert. Hierzu verändern wir M so, dass sie beim Akzeptieren den Inhalt aller Arbeitsbänder löscht (mit $\sqcup$ überschreibt) und den Kopf auf dem read-only Eingabeband zur ersten Zelle der Eingabe bewegt.

Zu M und einer Eingabe x lässt sich wie im Beweis von Proposition 7.30 ein Konfigurationsgraph erstellen. Die Knoten dieses Graphen entsprechen Konfigurationen von M, die Kanten entsprechenden Transitionen. Es gilt, dass M die Eingabe x akzeptiert, genau dann, wenn es im Konfigurationsgraphen zu x einen Pfad von c_0 zu c_{accept} gibt: Ein solcher Pfad entspricht genau einer akzeptierenden Berechnung.

Eine Maschine, die das Komplementproblem von $\mathcal{L}(M)$ löst, verfährt wie folgt:

- Konstruiere den Konfigurationsgraphen von M zur Eingabe x.
- Wende den Algorithmus für $\overline{\mathsf{PATH}}$ auf diesen Graphen und die Knoten c_0 und c_{accept} an. Akzeptiere, wenn es keinen Pfad gibt.

Die Maschine akzeptiert genau dann, wenn es keine akzeptierende Berechnung von M zur Eingabe x gibt; sie entscheidet also tatsächlich das Komplementproblem von $\mathcal{L}(M)$. Zudem handelt es sich um eine Maschine mit (existentiellem) Nichtdeterminismus.

Analysieren wir den Platzverbrauch der Maschine. Der Konfigurationsgraph zu einer Eingabe der Länge n hat bei einer f-platzbeschränkten Maschine Größe $2^{\mathcal{O}(f(n))}$. Der Algorithmus für $\overline{\mathsf{PATH}}$ hat logarithmischen Platzverbrauch, siehe Proposition 9.16. Sein Platzverbrauch ist also $\mathcal{O}(\log 2^{\mathcal{O}(f(n))}) = \mathcal{O}(f)$ wie gewünscht.

Problematisch ist jedoch, dass wir den Konfigurationsgraph nicht explizit konstruieren können, ohne die Platzschranke f zu verletzten. Daher muss man die Maschine so implementieren, dass sie beim Anwenden des Algorithmus für $\overline{\mathsf{PATH}}$ die benötigten Teile des Konfigurationsgraphen on-the-fly konstruiert. So muss zu jedem Zeitpunkt immer nur ein kleiner Teil des Konfigurationsgraphen gespeichert werden, und man kommt insgesamt mit Platzverbrauch $\mathcal{O}(f)$ aus. Für die Details verweisen wir auf die Fachliteratur. □

9.D 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
Frage:	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
Frage:	Ist F erfüllbar, d.h. gibt es eine Belegung φ mit $\varphi(F) = \mathit{true}$?

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.

9.18 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. $\text{2SAT} \in \text{NL}$ folgt dann mit dem Satz von Satz von Immerman und Szelepcsényi, welcher aussagt, dass $\text{NL} = \text{coNL}$.
- „Hardness“: 2SAT ist coNL-schwer. Wir reduzieren $\overline{\text{ACYCLICPATH}}$. Genau wie ACYCLICPATH, ist auch $\overline{\text{ACYCLICPATH}}$ NL-schwer.

9.19 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

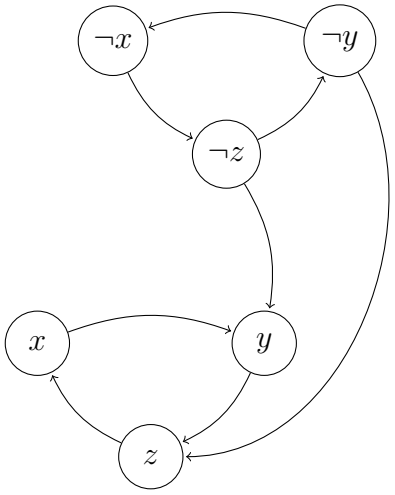
9.20 Beispiel
Sei

$$F = (\neg x \vee y) \wedge (\neg y \vee z) \wedge (x \vee \neg z) \wedge (z \vee y) \ .$$

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.

9.21 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

$$\varphi(\neg L \vee L') = \min(1 - \varphi(L), \varphi(L')) = 0.$$

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.

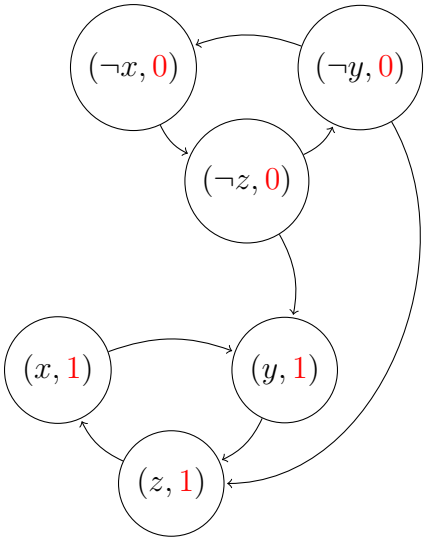
  Es gelte also $L \to^* L'$ und $L \to^* \neg L'$. Aufgrund der Symmetrie im Graphen gilt dann auch $\neg L' \to \neg L$ und $L' \to^* \neg L$. Wir erhalten $L \to^* L' \to \neg L$, ein Widerspruch zur Wahl von L im Algorithmus.

- 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.

□

9.22 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 $\overline{\mathsf{2SAT}}$ 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
```

□

9.23 Lemma
2SAT is coNL-hart (bezüglich logspace-Reduktionen).

Beweis:
Wir reduzieren $\overline{\mathsf{ACYCLICPATH}}$ auf 2SAT.

Sei $G\#s\#t$ eine $\overline{\mathsf{ACYCLICPATH}}$ 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. □

9.E Probleme in L: Arithmetik

In L liegen die grundlegenden Probleme der Arithmetik, insbesondere die folgenden beiden Probleme.

Addition (ADD)
Gegeben: Natürliche Zahlen i, j, ℓ.
Frage: Gilt $i + j = \ell$?

Multiplikation (MULT)
Gegeben: Natürliche Zahlen i, j, ℓ.
Frage: Gilt $i \cdot j = \ell$?

Wir fassen diese Probleme als Wortprobleme auf, in dem wir die Zahlen durch ihre Binärdarstellung repräsentieren.

$$\mathsf{ADD} = \{x\#y\#z \mid \exists i, j, \ell \colon x = \mathrm{bin}(i), y = \mathrm{bin}(j), z = \mathrm{bin}(\ell), x + y = z\},$$
$$\mathsf{MULT} = \{x\#y\#z \mid \exists i, j, \ell \colon x = \mathrm{bin}(i), y = \mathrm{bin}(j), z = \mathrm{bin}(\ell), x \cdot y = z\},$$

9.24 Lemma
ADD und MULT sind in L.

Die Aussage des Lemmas ist stärker, als man zunächst annehmen könnte. Die Probleme sind sogar in logarithmischem Platz in der Eingabegröße, das heißt der Binarkodierung der Zahlen, lösbar. Die Größe der Binarkodierung wiederum ist logartihmisch in der Größe der Zahlen. Es gibt also Algorithmen, die doppelt logarithmisch in der Größe der Zahlen sind.

Man könnte auf die Idee kommen, die Binärkodierung z' der Zahl $i + j$ (bzw. $i \cdot j$) durch binäre Addition (bzw. Multiplikation) von x und y zu berechnen, und dann z und z' zu vergleichen. Dies löst das Problem, benötigt allerdings linearen Platz.

Der Trick, um mit logarithmisch viel Platz auszukommen, ist, jedes Bit von z' einzeln zu berechnen, mit dem entsprechenden Bit von z zu vergleichen, und danach den Speicherplatz erneut zu verwenden. Es wird dann bloß eine konstante Anzahl von Bits für Überträge benötigt, sowie Zähler, die angeben, in welchem Bit der Eingabe man sich gerade befindet. Der Wert dieser Zähler ist durch die Länge der Eingabe beschränkt, wenn man sie also binär kodiert, ist ihre Größe logarithmisch in der Länge der Eingabe.

Die Details des Beweises überlassen wir der Leserin / dem Leser als Übung.

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.

Die Probleme aus P werden im Allgemeinen als die effizient lösbaren Probleme angesehen. Dies liegt daran, dass es zu jedem Problem aus $\mathcal{L}$ eine Konstante k gibt und einen Algorithmus, der Eingaben der Länge mit Zeitverbrauch $\mathcal{O}(n^k)$ löst. Wenn man die Eingabegröße des Algorithmus verdoppelt, steigt der Zeitverbrauch um den konstanten Faktor 2^k.

Wenn der Grad des Polynoms, also der konstante Exponent k sehr groß ist, kann dies dennoch dazu führen, dass der Algorithmus nur beschränkt praxistauglich ist. Betrachten wir beispielsweise das berühmte Primzahlproblem.

PRIMES	
Gegeben:	Natürliche Zahl n.
Frage:	Ist n eine Primzahl?

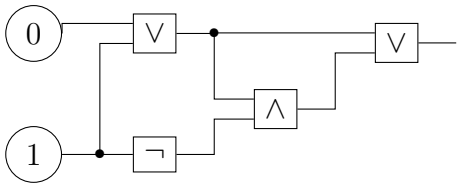
10.1 Theorem: Agrawal, Kayal, Saxena 2002
PRIMES ist in P.

Der AKS-Primzahltest, der das Theorem beweist, ist ein Algorithmus mit Laufzeit $\mathcal{O}((\log n)^{6+\epsilon})$. (Hierbei ist $\log n$ die Eingabegröße, da wir davon ausgehen, dass die Zahl n Binär kodiert ist.) Eine Verdoppelung der Eingabegröße führt also im Worst-Case zu einem Anstieg des Zeitverbrauchs um den Faktor $2^6 = 64$.

Das Circuit-Value-Problem

Unser Ziel im Rest dieses Kapitels ist es, das **Circuit-Value-Problem (CVP)** zu definieren und zu beweisen, dass dieses P-vollständig ist. Wir beginnen mit der Definition von Schaltkreisen (Circuits).

Betrachte den folgenden Schaltkreis.



Der Schaltkreis besteht aus einer Abfolge von konstanten Eingängen und logischen Gattern. Die Eingänge jeder Stufe sind dabei die Ausgänge der vorherigen Stufen.

Um einen Schaltkreis zu kodieren nummerieren wir die Ausgänge der Gatter durch. Zu jedem Ausgang assoziieren wir eine Variable P_i und eine Zuweisung an diese Variable, die die Funktion des Gatters widerspiegelt.

10.2 Beispiel

Wenn wir den oben dargestellten Schaltkreis kodieren erhalten wir

$$\begin{aligned} C:\ & P_0 = 0, \\ & P_1 = 1, \\ & P_2 = P_1 \vee P_0, \\ & P_3 = \neg P_1, \\ & P_4 = P_3 \wedge P_2, \\ & P_5 = P_2 \vee P_4. \end{aligned}$$

Wir verallgemeinern das Beispiel um die Instanzen des Circuit-Value-Problems zu definieren.

10.3 Definition

Ein **Boolescher Schaltkreis (Circuit)** ist ein Sequenz von endlich vielen Zuweisungen

$$P_0 = \ldots, P_1 = \ldots, \ \ldots\ , P_\ell = \ldots\ ,$$

wobei jede Zuweisung von einer der folgenden Formen ist:

$$P_i = 0, \ \vert\ \ P_i = 1 \ \vert\ \ P_i = \neg P_s \ \vert\ \ P_i = P_s \vee P_t \ \vert\ \ P_i = P_s \wedge P_t$$

mit $s, t < i$.

Jedes P_i darf dabei nur ein Mal definiert werden, also nur ein mal auf der linken Seite einer Zuweisung stehen. Es ist aber erlaubt, dass ein P_i auf der rechten Seite von mehreren P_j mit $j > i$ verwendet wird. Die rechte Seite der Zuweisung für P_i darf nur Variablen mit kleinerem Index, also Ausgaben von Gattern weiter links im Schaltkreis, verwenden.

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 $P_0, \ldots, P_\ell$.
Frage:	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_i trifft, sucht man die Werte der P_s mit $s < i$, die in der Zuweisung von P_i verwendet werden. Da diese schon ausgewertet wurden, kann man P_i auswerten. Diese Idee werden wir im ersten Teils des folgenden Theorems verwenden, um zu zeigen, dass CVP in P liegt.

10.5 Bemerkung

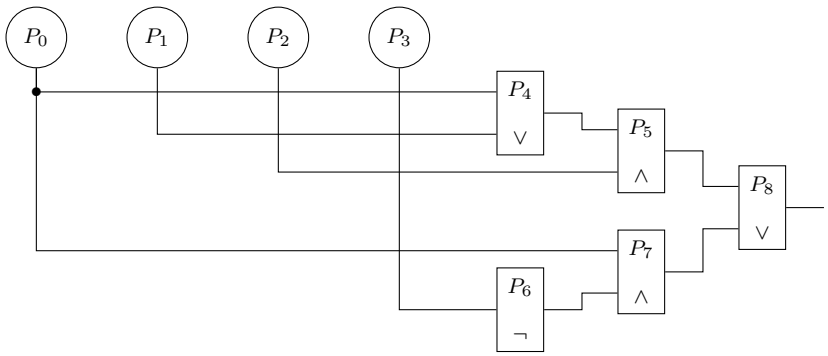
Die Tatsache, dass der Wert eines P_i mehrfach verwendet werdet darf, ist ein wichtiger Unterschied zu Booleschen Formeln.

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:

$$
\begin{aligned}
C\colon\ & P_0 = a, & & P_4 = P_0 \vee P_1,\\
& P_1 = b, & & P_5 = P_4 \wedge P_2,\\
& P_2 = c, & & P_6 = \neg P_3,\\
& P_3 = d, & & P_7 = P_6 \wedge P_0,\\
& & & P_8 = P_5 \vee P_7.
\end{aligned}
$$

Es handelt 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.

10.6 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 P-schwer ist („Hardness“).

10.7 Lemma: „Membership“
CVP liegt in P.

Beweis:
Wir nehmen an, dass eine CVP-Instanz durch eine durch #-Symbole getrennte Liste von Zuweisungen gegeben ist. In den Zuweisungen selbst sind die Indizes der P_i binär

kodiert. Mit dieser Kodierung hat eine Instanz mit ℓ Zuweisungen Größe in $\mathcal{O}(\ell \cdot \log \ell)$. Eine Instanz der Größe n besteht aus höchstens n Zuweisungen.

Wir konstruieren eine Mehrband-Turing-Maschine, welche die Werte der P_i von links nach rechts berechnet. Dazu bewegt sie die Köpfe aus dem ersten und zweiten Band synchron. An der selben Stelle, an der auf dem ersten Band das erste Symbol für die Zuweisung an Variable P_i steht, wird auf dem zweiten Band der Wert $\in \{0, 1\}$ gespeichert.

Eine Zuweisung der Form $P_i = 0$ bzw. $P_i = 1$ kann direkt ausgewertet werden.

Um eine Zuweisung der Form $P_i = \neg P_s$ auszuwerten, verfährt die Maschine wie folgt:

- Übertrage den Index $\text{bin}(s)$ vom Eingabeband auf ein weiteres Arbeitsband.
- Laufe auf dem Eingabeband nach links, um die Zuweisung für P_s zu finden (durch Abgleich mit dem gespeicherten Index). Bewege dabei den Kopf auf dem zweiten Band synchron mit.
- Lese an der entsprechenden Stelle auf dem zweiten Band den Wert von P_s und speichere seine Negation im Kontrollzustand.
- Laufe auf dem ersten und zweiten Band synchron nach rechts bis zur Zuweisung für P_i. (Hierzu kann man nutzen, dass P_i die erste Variable ist, für die auf dem zweiten Band noch kein Wert gespeichert ist.)
- Schreibe den im Kontrollzustand gespeicherten Wert an die entsprechende Stelle des zweiten Bands.

Das Auswerten von Zuweisungen der Form $P_i = P_s \mathbin{\hat{\vee}} P_t$ erfolgt ähnlich.

Im schlimmsten Fall muss die Maschine für das Berechnen eines jeden P_i einmal komplett durch die bereits berechneten Werte laufen. Dazu werden jeweils $\mathcal{O}(\ell)$ Schritte benötigt. Insgesamt ergibt sich ein Zeitaufwand von $\mathcal{O}(\ell^2)$. Da die Anzahl der Zuweisungen ℓ durch die Eingabegröße beschränkt ist, erhalten wir wie gewünscht einen polynomiellen (quadratischen) Algorithmus. □

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.

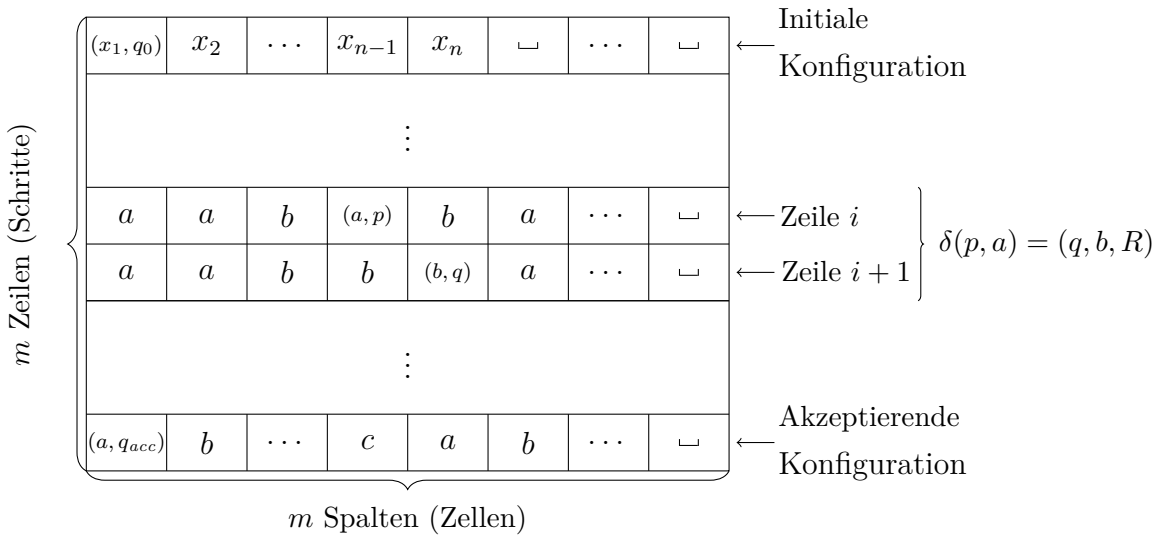
10.8 Theorem: „Hardness“
CVP ist P-schwer (bezüglich logspace-many-one-Reduktionen).

Beweis:
Wir reduzieren ein beliebiges Problem A aus P auf CVP. Das Problem $A = \mathcal{L}(M)$ ist die Sprache einer deterministischen Turing-Maschine M mit Zeitverbrauch $\mathcal{O}(n^k)$. Wir treffen drei Annahmen:

- Es gibt Konstanten $k, c, d \in \mathbb{N}$, so dass M zu einer Eingabe der Länge n nach höchstens $c \cdot n^k + d$ Schritten hält. Dies ergibt sich aus der Definition von $\mathcal{O}(n^k)$.
- Wir nehmen an, dass M die Zellen des Bands links vom ersten Symbol der Eingabe nicht besucht. M verwendet also ein Band, dass nur auf der rechten Seite unendlich ist. Man kann zeigen, dass solche Maschinen gleichmächtig zu Maschinen mit beidseitig unendlichem Band sind.
- Wir nehmen an, dass M um zu akzeptieren zunächst zur ersten Zelle des Bandes läuft (also die Zelle, auf der initial das erste Symbol der Eingabe stand), dann in den akzeptierenden Zustand q_{acc} wechselt und dann den Kopf nicht mehr bewegt. Durch das Einfügen eines zusätzlichen Kontrollzustandes lässt sich diese Eigenschaft erzwingen; der Zeitverbrauch steigt dabei nur linear.

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. Wir definieren $m = c \cdot n^k + d + 1$, und stellen fest, dass spätestens die m. Konfiguration der Berechnung von M zu x eine Haltekonfiguration ist. Zudem belegt jede der m Konfigurationen höchstens m Zellen auf dem Band von M. (Jede Zeitschranke ist auch eine Platzschranke.)

Wir schreiben Konfigurationen, die in der Berechnung von M zu x auftreten in eine $m \times m$-Matrix. Die i. Zeile der Matrix beschreibt hierbei die i. Konfiguration. Dies entspricht einem String der Länge m 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. Spalte der Matrix beschreibt den Inhalt der j. Zelle von M in jeder Konfiguration. Ein Sprung von Zeile i zu $i+1$ entspricht dem Ausführen einer Transition.



Man kann in der Berechnungsmatrix an der Zelle unten links ablesen, ob M Eingabe x akzeptiert: $x \in \mathcal{L}(M)$ genau dann, wenn die Zelle $(m-1, 0)$, welche die erste Zelle der letzten Konfiguration beschreibt, den Kopf beinhaltet und der Kontrollzustand q_{acc} ist. Dies gilt aufgrund der Anforderungen, die wir an M gestellt haben.

Die Berechnungsmatrix ist zu groß, um sie in einer logspace-Reduktion explizit konstruieren zu können. Stattdessen konstruieren wir einen Schaltkreis, der die Matrix beschreibt. Hierzu nutzen wir, dass die Berechnung von M **lokal** ist: Der Bandinhalt ändert sich pro ausgeführter Transition nur an wenigen Stellen, nämlich in der Umgebung der Kopfposition.

Konstruktion: Wir werden zuerst die Variablen des Schaltkreises und deren Idee erläutern:

- P_{ij}^a soll den Wert 1 erhalten, wenn in der i. Konfiguration in Zelle j das Symbol a steht.
- Q_{ij}^p soll den Wert 1 erhalten, wenn in der i. Konfiguration der Kopf von M auf Zelle j steht und M im Zustand p ist.

Wir definieren diese Variablen für $0 \leq i, j \leq n^c$, $a \in \Gamma$, $p \in Q$. Offensichtlich beschreiben diese Variablen die Berechnungsmatrix exakt. Es verbleibt, Zuweisungen zu definieren, durch die die Variablen die gewünschten Werte erhalten.

Wir beginnen mit der Kodierung der ersten Zeile der Matrix, der initialen Konfiguration. Um die Eingabe x zu kodieren, setzen wir für $j = 1, \ldots, n-1$: $P_{0j}^{x_j} = 1$ und $P_{0j}^b = 0$, 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 $j = n, \ldots, m-1$: $P_{0j}^{\sqcup} = 1$ und $P_{0j}^b = 0$,

wobei $b \in \Gamma \setminus \{\sqcup\}$. Kodieren wir nun die Kopfposition und den initialen Zustand von M. Wir setzen $Q_{00}^{q_0} = 1$, denn der Kopf von M steht zunächst in Zelle 0 und M ist im initialen Zustand q_0. Weiter setzen wir $Q_{00}^{q} = 0$ für $q \in Q \setminus \{q_0\}$ und $Q_{0j}^{q} = 0$ für alle $j = 1, \ldots, m-1$, $q \in Q$.

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:

$$P_{(i+1)j}^{b} = \underbrace{\bigvee_{\delta(q,a)=(p,b,d)} \left(Q_{ij}^{q} \wedge P_{ij}^{a}\right)}_{M \text{ schreibt in Zelle } j.} \vee \underbrace{\left(P_{ij}^{b} \wedge \bigwedge_{q \in Q} \neg Q_{ij}^{q}\right)}_{\text{Kopf steht nicht in Zelle } j.}.$$

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. Analog haben wir Fälle für den Fall, dass der Kopf nicht oder nach links bewegt wurde.

$$Q_{(i+1)j}^{p} = \underbrace{\bigvee_{\delta(q,a)=(p,b,N)} \left(Q_{i,j}^{q} \wedge P_{i,j}^{a}\right)}_{M \text{ bewegt den Kopf nicht.}} \vee \underbrace{\bigvee_{\delta(q,a)=(p,b,R)} \left(Q_{i,j-1}^{q} \wedge P_{i,j-1}^{a}\right)}_{M \text{ bewegt den Kopf nach rechts.}} \vee \underbrace{\bigvee_{\delta(q,a)=(p,b,L)} \left(Q_{i,j+1}^{q} \wedge P_{i,j+1}^{a}\right)}_{M \text{ bewegt den Kopf nach links.}}.$$

Man beachte, dass der Kopf sich in der 0. Zelle nicht nach links bewegen kann (und analog in Zelle $m-1$ nicht nach rechts). In den entsprechenden Zuweisungen muss man den jeweiligen Fall auslassen.

Wir konstruieren nun eine Circuit, der alle oben aufgelisteten Zuweisungen beinhaltet. 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. Da jede unserer Zuweisungen nur konstant viele Variablen auf der rechten Seiteverwendet, kann durch das Einführen von polynomiell vielen Hilfsvariablen die Circuit-Instanz in die gewünschte Form gebraucht werden.

Die Reihenfolge der Zuweisungen wählen wir so, dass alle Variablen die sich auf die i. Zeile beziehen, vor denen kommen, die sich auf die $(i+1)$. Zeile beziehen. Zudem

soll die Zuweisung an $Q^{q_{acc}}_{(m_1)\ 0}$ die letzte sein. Es sei C der resultierende Circuit. Wir behaupten, dass die Funktion $x \mapsto C$ die gesuchte Reduktion ist.

Die Zuweisungen sind so gewählt, dass wenn man die Werte der Variablen in ausrechnet, tatsächlich eine exakte Beschreibung der Berechnungsmatrix von M zu Eingabe x erhält. Insbesondere gilt die folgende Äquivalenz: Wenn x von M akzeptiert wird, genau dann ist die Kopfposition in der m. Konfiguration in Zelle 0. und der Kontrollzustand ist q_{acc}. Genau in diesem Fall hat $Q^{q_{acc}}_{(m_1)\ 0}$, die letzte Variable des Circuits C, den Wert 1. Wie gewünscht ist C genau dann eine Ja-Instanz des CVP, wenn x eine Ja-Instanz von $\mathcal{L}(M)$ ist.

Es verbleibt zu argumentieren, dass die Reduktion logspace-berechenbar ist. Hierzu beobachtet man, dass sich alle Zuweisungen von C unter Verwendung von Binärzählern berechnen lassen (für die Indizes i, j etc.) Die Anzahl der Binärzähler hängt dabei von der Anzahl Kontrollzustände, Bandsymbolen und Transitionen von M ab. Da M fixiert ist, handelt es sich hierbei um Konstanten. Die Größe der Binärzähler, als den maximal gespeicherten Wert, ist m. Um diese Zahl zu speichern, werden $\log m = \log(n^k + d + 1) \in \mathcal{O}(\log n^k) = \mathcal{O}(k \cdot \log n) = \mathcal{O}(\log n)$ viele Bits benötigt. Die Gesamtgröße des benötigten Speichers ist logarithmisch in der Eingabegröße n. □

Dadurch, dass wir das CVP als P-vollständig nachgewiesen haben, lassen sich nun weitere Probleme als P-vollständig nachweisen: Um zu zeigen, dass $\mathcal{L}$ P-schwer ist, reicht es nun $\mathsf{CVP} \leq_m^{\log} \mathcal{L}$ zu zeigen. Da sich jedes Problem $\mathcal{L}'$ aus P auf CVP reduzieren lässt ($\mathcal{L}' \leq_m^{\log} \mathsf{CVP}$), $\mathsf{CVP} \leq_m^{\log} \mathcal{L}$ gilt und logspace-Reduzierbarkeit transitiv ist (Lemma 9.8), dann gilt auch $\mathcal{L}' \leq_m^{\log} \mathcal{L}$. Wie gewünscht lässt sich jedes Problem aus P auf $\mathcal{L}'$ reduzieren.

Als konkrete Beispiele betrachten wir zwei bereits aus *Theoretische Informatik 1* bekannte Probleme.

Wortproblem für kontextfreie Sprachen (WORD-CFL)
Gegeben: Kontextfreie Grammatik G, Wort w.
Frage: Gilt $w \in \mathcal{L}(G)$?

Leerheit kontextfreier Sprachen (EMPTINESS-CFL)
Gegeben: Kontextfreie Grammatik G.
Frage: Gilt $\mathcal{L}(G) = \emptyset$?

Beide Probleme sind P-vollständig. Membership zeigt man unter Verwendung der aus Theoretische Informatik 1 bekannten Algorithmen. Hardness zeigt man durch Reduktion des CVP. Die Reduktion auszuarbeiten überlassen wir der Leserin / dem Leser als Übungsaufgabe.

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 Nichtdeterminismus ist es nun möglich, Probleme zu konstruieren, bei denen die Maschine den Beweis 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 gehen wie folgt vor:

A) Wir zeigen, dass SAT als „Master"-Problem NP-vollständig ist.

B) Es gibt viele weitere NP-vollständige Probleme in unterschiedlichen Bereichen. Hier betrachten wir exemplarisch HamiltonianCycle, das Finden Hamiltonscher Kreise in Graphen.

C) Wir liefern eine alternative Charakterisierung der Klasse NP, basierend auf Zertifikaten. Darauf aufbauen argumentieren wir, warum $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$ ein so wichtiges ungelöstes Problem ist.

11.A Die NP-Vollständigkeit von SAT

Genau wie P mit CVP und NL mit PATH hat NP mit SAT ein charakteristisches „Master"-Problem. Ziel dieses Unterkapitels ist es, zu zeigen dass SAT NP-vollständig ist. Tatsächlich betrachten wir jedoch zunächst eine Variante des CVP und zeigen die NP-Vollständigkeit dieser Variante. Dies erlaubt es uns, viele Idee aus dem Beweis der P-Schwere des CVP wiederzuverwenden (siehe Kapitel Kapitel 10).

11.1 Definition
Ein Boolscher Schaltkreis **mit variablen Eingängen** ist ein Boolscher Schaltkreis wie in Definition 10.3, in welchem Zuweisungen der Form

$$P_i = ? \ .$$

Wir nennen die P_i, deren Zuweisung von der Form $P_i = ?$ ist, die Variablen Eingänge des Circuits. Intuitiv gesehen kann man den Wert eines solchen P_i auf 0 oder 1 setzen, bevor man die Werte der anderen Variablen ausrechnet.

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.

Das zugehörige Entscheidungsproblem ist das folgende.

Erfüllbarkeitsproblem für Schaltkreise (CircuitSAT)	
Gegeben:	Ein Boolescher Schaltkreis C mit Variablen Eingängen $P_1, \ldots, P_k$.
Frage:	Gibt es Wahrheitswerte $y_1, \ldots, y_k \in \{0, 1\}$ mit $C(y_1, \ldots, y_k) = 1$?

11.2 Theorem
CircuitSAT ist NP-vollständig (bezüglich logspace-many-one-Reduktionen).

11.3 Bemerkung
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.

Wir beweisen „Membership“ und „Hardness“ wieder getrennt.

11.4 Lemma
CircuitSAT $\in$ NP.

Beweis:
Wir konstruieren einen nicht-deterministischen Algorithmus, der zu jedem variablen Eingang P_i einen Wahrheitswert $y_i \in \{0, 1\}$ rät. Die Zuweisungen der Form $P_i = ?$ werden durch $P_i = y_i$ ersetzt. Wir erhalten die CVP-Instanz $C(y_1, \ldots, y_k)$ auf die wir einen Algorithmus für das CVP anwenden, um $C(y_1, \ldots, y_k) = 1$ zu entscheiden.

Der Algorithmus akzeptiert nur dann, wenn er Wahrheitswerte rät, durch welche die entstehende CVP-Instanz zu 1 ausgewertet wird. Dies ist genau dann dann möglich, wenn der initial gegebene Circuit mit Variablen Eingängen erfüllbar war.

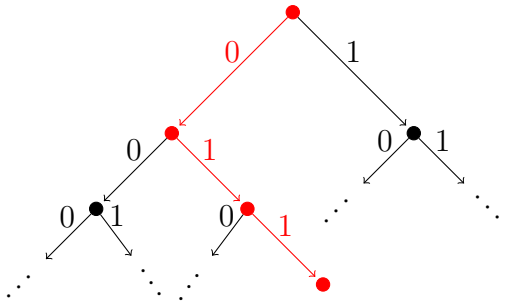


Abbildung 1: Der binäre Berechnungsbaum von M auf Input x. Der rot markierte Pfad wird durch den String $y = 011$ beschrieben.

Das Raten und Einsetzen der Wahrheitswerte benötigt linear viel Zeit und erzeugt eine CVP-Instanz, die genauso groß wie die Eingabe ist. Da $\mathsf{CVP} \in \mathsf{P}$ lässt sich diese CVP-Instanz in polynomieller Zeit auswerten. □

11.5 Proposition
CircuitSAT ist NP-schwer.

Der Beweis ist sehr ähnlich zum Beweis von Theorem 10.8. Wir heben die Unterschiede zu diesem Beweis durch **Fettdruck** hervor.

Beweis:
Sei $\mathcal{L}$ ein Problem aus NP. Dann gibt es eine nicht-deterministische Turing-Maschine M mit $\mathcal{L} = \mathcal{L}(M)$ mit polynomiellem Zeitverbrauch.

Wir treffen vier Anforderungen an M, wobei die ersten drei wie im Beweis von Theorem 10.8 sind.

- Zu einer Eingabe der Länge n braucht M weniger als $m = c \cdot n^k + d$ Schritte bis zum Halten (für geeignet gewählte Konstanten).
- M benutzt die Zellen links von der Eingabe nicht.
- Wenn M akzeptiert, befindet sich der Kopf auf der ersten Zelle der Eingabe. Anschließend wird der Kopf nicht mehr bewegt.
- M **verzweigt binär**.

  Das bedeutet, dass M sich in jeder Konfiguration immer zwischen höchstens zwei verschiedenen Transitionen entscheiden muss.

Formal verlangen wir, dass es zu jedem $q \in Q$ und $a \in \Gamma$ entsprechende $p_0, p_1 \in Q$, $b_0, b_1 \in \Gamma$, $d_0, d_1 \in \{L, N, R\}$ gibt mit

$$\delta(q,a) = \{\underbrace{(p_0, b_0, d_0)}_{\delta_0(q,a)}, \underbrace{(p_1, b_1, d_1)}_{\delta_0(q,a)}\} .$$

Wir bezeichnen die beiden Einträge mit $\delta_0(q,a)$ („linke“ Transition) und $\delta_1(q,a)$ („rechte“ Transition). Wir erlauben, dass die beiden gleich sind, jedoch nicht dass $\delta(q,a) = \emptyset$.

Mit dieser Annahme ist der Berechnungsbaum von M zu jeder Eingabe ein Binärbaum, in dem jeder (endliche) Ast mit einer Konfiguration in einem Haltezustand endet.

Um die Annahme zu erzwingen, sorgt man zunächst dafür, dass jede Konfiguration, in der M stecken bleibt ($\delta(q,a) = \emptyset$) stattdessen in den abweisenden Zustand führt. Dann fügt man Hilfszustände ein, um eine Entscheidung zwischen k Transitionen durch mehrere aufeinander folgende Entscheidungen zwischen jeweils nur 2 Transitionen kodieren. Die Anzahl Hilfszustände ist eine Konstante, die von der Transitionsrelation von M, aber nicht von der konkreten Eingabe abhängt. Dementsprechend steigt die Länge der Berechnung auch nur um einen konstanten Faktor und bleibt polynomiell.

Sei x nun ein Input von M der Länge n. Wie im Beweis von Theorem 10.8 möchten wir eine $m \times m$-Berechnungsmatrix zur Berechnung von M zu x, und anschließend einen Circuit, der diese Matrix beschreibt. Um dem Nichtdeterminismus von M Rechnung zu tragen, verwenden wir **variable Eingänge**. Je nachdem mit welchen Werten diese belegt werden, beschreiben wir eine unterschiedliche Berechnung von M zu x. Hier nutzen wir aus, dass eine Berechnung von M, also im Pfad im Berechnungsbaum, einer Sequenz über $\{0,1\}$ entspricht (siehe Abbildung 1).

Wie zuvor haben wir Variablen

- P_{ij}^a mit Wert 1, wenn in der i. Konfiguration in Zelle j das Symbol a steht,
- Q_{ij}^p mit Wert 1, wenn in der i. Konfiguration der Kopf von M auf Zelle j steht und M im Zustand p ist.

Zudem haben wir **variable Eingänge** $R_0, \ldots, R_{m-1}$. Der Wert von R_i bestimmt darüber, ob M in der i. Konfiguration die linke Transition $\delta_0(q,a)$ (wenn $R_i = 0$) oder die rechte Transition $\delta_1(q,a)$ (wenn $R_i = 1$) benutzt.

- Die **Zuweisungen für die Variablen der Form** R_i sind jeweils

  $$R_i =?$$

  für $i \in \{0, \ldots, m-1\}$.

- Die Zuweisungen für die Variablen, die die erste Zeile der Matrix beschreiben, sind wie zuvor. (Die Initialkonfiguration ist durch die Eingabe festgelegt und wird nicht vom Nichtdeterminismus beinflusst.)

  Für $j \in \{0, \ldots, n-1\}$: $P_{0j}^{x_j} = 1$ und $P_{0j}^{b} = 0$, wobei $b \in \Gamma \setminus \{x_j\}$. Für $j \geq n$: $P_{0j}^{\sqcup} = 1$ und $P_{0j}^{b} = 0$, wobei $b \in \Gamma \setminus \{\sqcup\}$. Außerdem $Q_{00}^{q_0} = 1$ und $Q_{0j}^{q} = 0$ sonst.

- Die Zuweisungen für die restlichen Variablen sind ähnlich wie zuvor. Wir stellen sicher, dass die durch R_i **spezifizierte Transition** angewandt wird.

  $$\begin{aligned} P_{(i+1)j}^{b} = \quad & \left(P_{ij}^{b} \wedge \bigwedge_{q \in Q} \neg Q_{ij}^{q} \right) \\ & \vee \left(\neg R_i \wedge \bigvee_{\delta_0(q,a)=(p,b,d)} \left(Q_{ij}^{q} \wedge P_{ij}^{a} \right) \right) \vee \left(R_i \wedge \bigvee_{\delta_1(q,a)=(p,b,d)} \left(Q_{ij}^{q} \wedge P_{ij}^{a} \right) \right) . \end{aligned}$$

  Der Teil der Formel mit $\neg R_i$ entspricht dabei der linken Transition $\delta_0(q, a)$, der Teil mit R_i entspricht $\delta_1(q, a)$.

  Die Definition der Q-Variablen modifizieren wir auf ähnliche Weist.

  $$\begin{aligned} Q_{(i+1)j}^{p} = \quad & \Big(\neg R_i \wedge \big(\bigvee_{\delta_0(q,a)=(p,b,N)} \left(Q_{i,j}^{q} \wedge P_{i,j}^{a} \right) \vee \bigvee_{\delta_0(q,a)=(p,b,R)} \left(Q_{i,j-1}^{q} \wedge P_{i,j-1}^{a} \right) \vee \ldots \big) \Big) \\ & \vee \left(\neg R_i \wedge \big(\bigvee_{\delta_1(q,a)=(p,b,N)} \left(Q_{i,j}^{q} \wedge P_{i,j}^{a} \right) \vee \ldots \big) \right) \end{aligned}$$

Es sei C der Circuit mit **variablen Eingängen**, der aus den obigen Zuweisungen besteht und dessen letzte Zuweisung an die Variable $Q_{(m_1)\ 0}^{q_{acc}}$ ist. Wie im Beweis von Theorem 10.8 kann man die Zuweisungen durch einfügen von Hilfsvariablen in die nötige Form bringen.

Wir behaupten, dass $x \mapsto C$ die gesuchte Reduktion ist.

Wenn $x\mathcal{L}(M)$ gilt, dann gibt es eine akzeptierende Berechnung von M zu x. Diese akzeptierende Berechnung ist ein Pfad im Berechnungsbaum, der wie in Abbildung 1

durch eine Sequenz über $\{0,1\}$ identifiziert wird. Wenn wir die **variablen Eingänge** mit den entsprechenden Werten belegen, dann beschreibt die entstehende CVP-Instanz genau die Berechnungsmatrix von M, die dieser Berechnung entspricht. Da es sich um eine akzeptierende Berechnung handelt, erhält $Q^{q_{acc}}_{(m_1)\ 0}$ den Wert 1. Analog dazu identifiziert eine Belegung der R_i, die zu $Q^{q_{acc}}_{(m_1)\ 0} = 1$ führt, eine akzeptierende Berechnung von M zu x und beweist damit $x \in \mathcal{L}(M)$.

Wie im Beweis von Theorem 10.8 lässt sich die Reduktion durch eine konstante Anzahl Binärzähler, deren Werte durch m beschränkt sind, implementieren. Es handelt sich also um eine logspace-Reduktion. □

SAT

Für die formale Definition von Klauseln, Formeln in KNF und SAT verweisen wir auf Unterkapitel 9.D.

11.6 Theorem: Cook 1971, Levin 1973
SAT ist NP-vollständig.

11.7 Bemerkung
Wie viele Resultate während der Zeit des kalten Krieges, wurde die NP-Vollständigkeit unabhängig voneinander von Forschern auf beiden Seiten des Eisernen Vorhangs gezeigt: Einerseits vom US-amerikanischen Forscher Stephen Cook 1971, andererseits vom russischen Forscher Leonid Levin 1973.

Tatsächlich ist bereits 3SAT, also die Einschränkung von SAT auf Formeln, in denen jede Klausel aus höchstens drei Literalen besteht, NP-vollständig.

11.8 Theorem
3SAT ist NP-vollständig.

Der Beweis gliedert sich in „Membership“ und „Hardness“. Membership ($\mathsf{SAT} \in \mathsf{NP}$) ist einfach: Man rät eine Variablenbelegung und wertet die gegebene Formel unter dieser Belegung aus. Die Details sind ähnlich zu Lemma 11.4

Ausgehend von der NP-Vollständigkeit von CircuitSAT lässt sich nun die NP-Schwere von 3SAT leicht zeigen. Wir geben einfach eine Reduktion von CircuitSAT auf 3SAT an.

11.9 Lemma
CircuitSAT $\leq_m^{\log}$ 3SAT.

Beweis:
Es sei C eine gegebene CircuitSAT-Instanz. Wir konstruieren eine 3-KNF F. Wir assoziieren zu jeder Circuit-Variable P_i eine Variable x_i in der Formel.

Wir konstruieren zu jeder Zuweisung $P_i = \ldots$ eine entsprechende Formel in 3-KNF $F(P_i = \ldots)$.

- $P_i = 0$ wird zu $x_i \leftrightarrow \mathit{false} \Leftrightarrow (\neg x_i)$.
- $P_i = 1$ wird zu $x_i \leftrightarrow \mathit{true} \Leftrightarrow (x_i)$.
- $P_i =?$ wird nicht in eine Formel übersetzt.
- $P_i = \neg P_s$ wird zu $x_i \leftrightarrow \neg x_s \Leftrightarrow (x_i \vee x_s) \wedge (\neg x_i \vee \neg x_s)$.
- $P_i = P_s \wedge P_t$ wird zu $x_i \leftrightarrow (x_s \wedge x_t) \Leftrightarrow (x_i \vee \neg x_s \vee \neg x_t) \wedge (\neg x_i \vee x_s) \wedge (\neg x_i \vee x_t)$.
- $P_i = P_s \vee P_t$ wird zu $x_i \leftrightarrow (x_s \vee x_t) \Leftrightarrow (\neg x_i \vee x_s \vee x_t) \wedge (x_i \vee \neg x_s) \wedge (x_i \vee \neg x_t)$.

Die CircuitSAT-Instanz $C = P_1 = \ldots, \ldots, P_\ell = \ldots$ übersetzen wir dann in die Formel

$$F = \bigwedge_{i=0,\ldots,\ell} F(P_i = \ldots) \wedge (x_\ell) \ .$$

Da jede der Formeln $F(P_i = \ldots)$ in 3-KNF ist, ist auch F in 3-KNF.

Die Struktur der Formel F legt die Werte aller Variablen fest, außer denen, die variablen Eingängen der Circuit-SAT Instanz entsprechen. Die letzte Klausel (x_ℓ) erzwingt, dass die Variable x_ℓ und damit die letzte Variable P_ℓ in C den Wert 1 erhält. Insgesamt gilt $C \in$ CircuitSAT genau dann, wenn F erfüllbar ist.

Den formalen Beweis, dass es sich bei $C \mapsto F$ um eine korrekte logspace-Reduktion handelt, überlassen wir der Leserin / dem Leser als Übungsaufgabe. □

11.B Hamiltonsche Kreise

Die Wichtigkeit der Klasse NP wurde von der Forschungsgemeinschaft nicht direkt erkannt, nachdem Cook 1971 seinen Beweis, dass SAT NP-vollständig ist, veröffentliche. Erst ein Jahr später wurde die Klasse durch Karps Paper *Reduzierbarkeit zwischen kombinatorischen Problemen* [Kar72] in den Mittelpunkt gerückt. In diesem Paper zeigte Karp für 21 Probleme aus unterschiedlichen Bereichen der Informatik, dass sie NP-vollständig sind, indem er eine Reihe von Reduktionen angab.

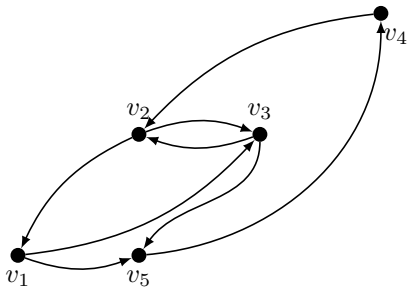
Wir betrachten hier eines dieser 21 Probleme, und zwar eines aus der Welt der Graphen: Wir suchen nach einem Hamiltonschen Kreis, einem Kreis, der alle Knoten eines Graphen genau ein mal besucht.

11.10 Definition
Sei $G = (V, E)$ ein gerichteter Graph mit $|V| = n$. Ein **Hamiltonscher Kreis** in G ist ein Pfad, der alle Knoten genau einmal besucht, also ein Pfad

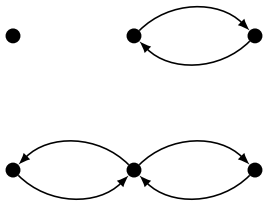
$$v_1 \to v_2 \to \ldots \to v_n \to v_1$$

so dass jeder Knoten $v \in V$ als genau eines der v_i auftritt.

11.11 Beispiel
Der folgende Graph hat den Hamiltonscher Kreis $v_1 \to v_3 \to v_5 \to v_4 \to v_2 \to v_1$.



In den folgenden beiden Graphen gibt es jeweils keinen Hamiltonschen Kreis.



Im ersten Graphen ist der linke Knoten isoliert und kann nicht besucht werden. Im zweiten Graphen besucht jeder Kreis, der die äußeren Knoten besucht, den mittleren Knoten doppelt.

Die Frage, ob es zu einem gegebenen Graphen einen Hamiltonschen Kreis gibt, formulieren wir im folgenden Problem:

Existenz Hamiltonscher Kreise (HamiltonianCycle)	
Gegeben:	Ein gerichteter Graph $G = (V, E)$.
Frage:	Gibt es einen Hamiltonschen Kreis in G?

Obwohl es in den obigen Beispielen jeweils einfach war, einen Hamiltonschen Kreis zu finden bzw. einzusehen, dass es keinen geben kann, ist das Entscheidungsproblem NP-vollständig.

11.12 Theorem: Karp [Kar72]
HamiltonianCycle ist NP-vollständig (bezüglich Polynomialzeitreduktionen).

11.13 Bemerkung
Hier und im Folgenden verwenden wir für Klassen ab NP üblicherweise die schwächeren Polynomialzeitreduktionen statt den stärkeren logspace-Reduktionen. Dies erspart es uns, argumentieren zu müssen, dass die Reduktion nur logarithmischen Platzverbrauch hat. $\mathcal{L} \leq_m^{\log} \mathcal{L}'$ ist eine stärkere Aussage als $\mathcal{L} \leq_m^{\text{poly}} \mathcal{L}'$, da jede logspace-Reduktion auch eine Polynomialzeitreduktion ist, allerdings nicht umgekehrt. Wenn $\mathsf{L} \neq \mathsf{P}$ gilt, ist es sogar eine potentiell echt stärkere Aussage.

Dennoch erfüllen Polynomialzeitreduktionen und $\leq_m^{\text{poly}}$ viele nützliche Eigenschaften, die auch für $\leq_m^{\log}$ gelten.

- Aus $\mathcal{L} \leq_m^{\text{poly}} \mathcal{L}'$ und $\mathcal{L}' \in \mathsf{NP}$ folgt $\mathcal{L} \in \mathsf{NP}$.
- Aus $\mathcal{L} \leq_m^{\text{poly}} \mathcal{L}'$ und $\mathcal{L}$ NP-schwer (bezüglich Polynomialzeitreduktionen) folgt, dass auch $\mathcal{L}'$.
- Falls $\mathsf{P} \neq \mathsf{NP}$, dann sind alle NP-schweren Probleme (bezüglich Polynomialzeitreduktionen) nicht in P enthalten.

Der Beweis gliedert sich wieder in zwei Schritte.

11.14 Lemma
HamiltonianCycle $\in$ NP.

Beweis:
Wir präsentieren einen Nicht-deterministischen Algorithmus, der HamiltonianCycle in Polynomialzeit löst.

- Rate eine Sequenz von Knoten $v_1, \ldots, v_n$ der Länge $n = |V|$.
- Überprüfe, dass für alle $i \in \{1, \ldots, n-1\}$ jeweils $v_i \to v_{i+1}$ im Graphen gilt, und überprüfe $v_n \to v_1$.
- Überprüfe, dass jeder Knoten aus V in der Sequenz der v_i vorkommt.
- Überprüfe, dass es keinen Knoten aus V gibt, der in der Sequenz doppelt vorkommt.

Wenn eine der Überprüfungen fehlschlägt, weise ab. Ansonsten akzeptiere.

Der Algorithmus akzeptiert genau dann, wenn die im ersten Schritt geratenen Knoten einen Hamiltonschen Kreis bilden. Dies kann nur dann geschehen, wenn ein solcher existiert.

Der erste Schritt benötigt Linearzeit, alle anderen jeweils höchstens quadratisch viel Zeit. □

11.15 Lemma
HamiltonianCycle ist NP-schwer.

Beweis:
Wir reduzieren SAT auf HamiltonianCycle, zeigen also SAT $\leq_m^{\text{poly}}$ HamiltonianCycle. Sei dazu F eine SAT-Instanz, also eine Formel in CNF. Seien $K_1, \ldots, K_m$ die Klauseln von F und $x_1, \ldots, x_k$ die Variablen, die in F vorkommen. Die Formel F hat dann die Form $F = K_1 \wedge \cdots \wedge K_m$.

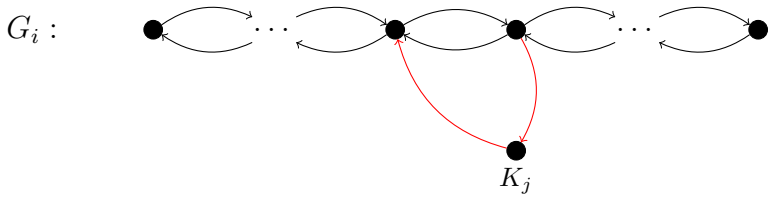
Schritt 1: Für jede Variable x_i konstruieren wir einen Graphen G_i, welcher die Belegung der Variable simuliert. Sei dazu m_i die Anzahl der Klauseln, die x_i oder $\neg x_i$ enthalten. Der Graph G_i hat $2m_i + 2$ Knoten, die wie folgt miteinander verbunden sind:

G_i : 

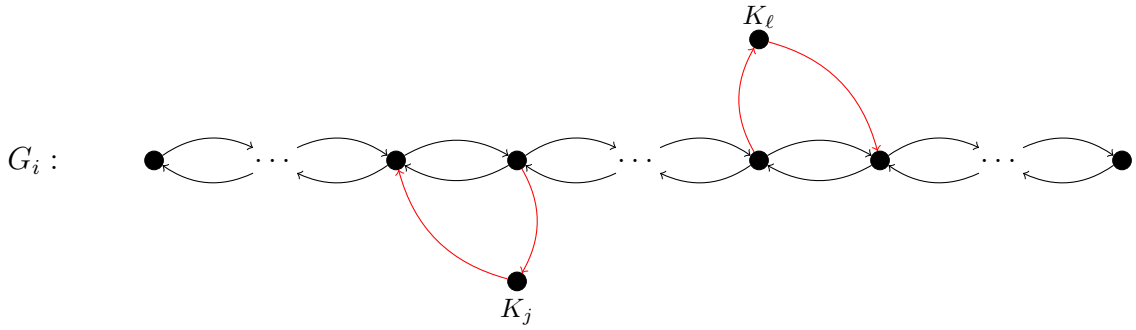
Wir haben also jeweils zwei Knoten pro Klausel, in der x_i oder $\neg x_i$ vorkommt, sowie jeweils einen zusätzlichen Knoten außen links bzw. außen rechts.

Wie wir später sehen werden, wird die Belegung von x_i durch einen Durchlauf durch G_i simuliert: Ein Durchlauf von rechts nach links entspricht der Belegung 1/true, ein Durchlauf von links nach rechts entspricht der Belegung 0/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 K_j eine Klausel, die x_i enthält. Dann fügen wir einen neuen Knoten in K_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 (bei $x_i = 1$) besucht werden kann. Diese Kanten werden wir im Folgenden rot markieren, um sie von den vorherigen Kanten unterscheiden zu können.



Sei nun K_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 ($x_i = 0$) besucht werden kann:

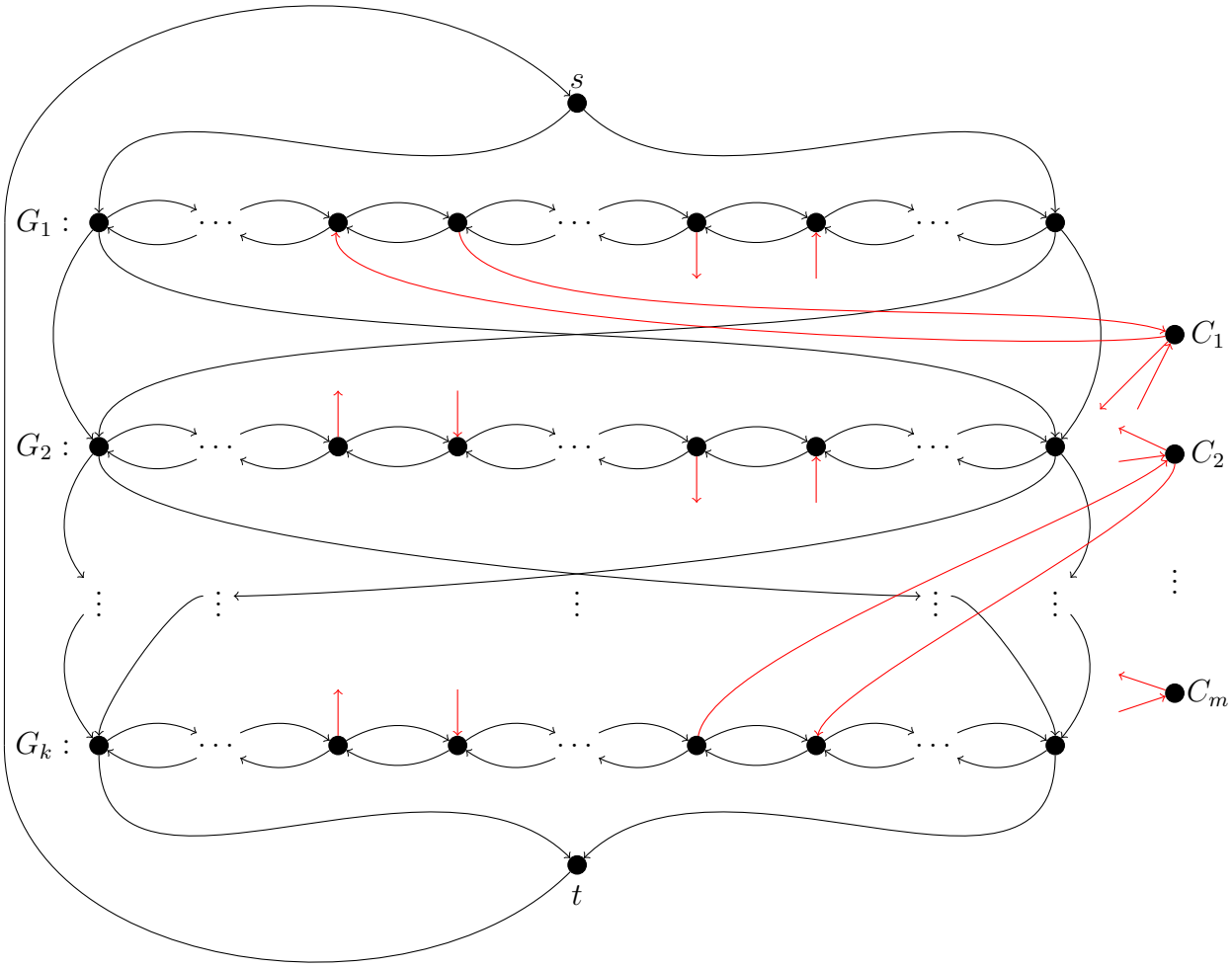


Angenommen wir durchlaufen G_i von links nach rechts ($x_i = 0$). Dann können wir K_j in einem Hamiltonschen Kreis nicht besuchen: Um K_j zu besuchen, müssen wir zunächst am linken Ankerknoten zu K_j in G_i zum rechten Ankerknoten gehen. Dann können wir die roten Kanten verwenden, um K_j zu besuchen, landen aber wieder beim linken Ankerknoten, den wir damit doppelt besucht haben. Den Knoten für die Klausel K_ℓ, in der $\neg x_i$ vorkommt, können wir jedoch besuchen: Statt vom linken

Ankerknoten direkt zum rechten zu gehen, verwenden wir die roten Kanten, um einen Umweg über K_ℓ zu machen. Der Rest unseres Durchlaufs durch G_i wird davon nicht beeinflusst.

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.

Die Funktion, die zu einer gegebenen Formel F den Graphen G ausgibt, beweist $\mathsf{SAT} \leq_m^{\text{poly}} \mathsf{HamiltonianCycle}$. Die Funktion kann leicht so implementiert werden, dass ihr Zeitverbrauch polynomiell ist.

Argumentieren wir nun kurz, dass sie auch korrekt ist: $F \in \mathsf{SAT}$ gdw. $G \in \mathsf{HamiltonianCycle}$. Nehmen wir an, dass es eine Variablenbelegung $\varphi\colon \{x_1, \ldots, x_k\} \to \{0, 1\}$ gibt, die F erfüllt. Wir konstruieren einen Hamiltonschen Kreis in G.

Wir beginnen bei s und wählen die Kante zum linken Ende von G_1, wenn $\varphi(x_1) = 0$. Nun durchlaufen wir G_1 von links nach rechts und besuchen dabei alle Knoten K_i,

die in der richtigen Richtung eingebunden sind, also die Knoten zu den Klauseln, die $\neg x_1$ beinhalten. Wenn $\varphi(x_1) = 1$ beginnen wir auf der rechten Seite von G_1 und laufen nach links. Auch hier besuchen wir entsprechenden Klauselknoten. Nach dem Durchlauf durch G_1 wählen wir nun die Kante zum linken oder rechten Ende von G_1, abhängig von $\varphi(x_2)$. Wir durchlaufen G_2 wie oben beschrieben. Alle Klauselknoten, die bereits im vorherigen Durchlauf besucht wurden, besuchen wir nun nicht. Wir verfahren analog für die restlichen G_i bzw. x_i. Am Ende gehen wir von einem der äußeren Kanten von G_k zu t und anschließend zu s um den Kreis zu schließen.

Wenn die Variablenbelegung die Formel erfüllt, erfüllt sie mindestens ein Literal pro Klausel. Dies bedeutet, dass im oben konstruierten Kreis jeder Klauselknoten besucht wird. Es handelt sich in der Tat um einen Hamiltonschen Kreis.

Analog entspricht jeder Hamiltonsche Kreis in G einer Variablenbelegung, die F erfüllt. □

Ähnliche Probleme

Analog zur Existenz Hamiltonscher Kreise in gerichteten Graphen kann man auch zeigen, dass die beiden folgenden Probleme NP-vollständig sind.

Existenz Hamiltonscher Kreise in ungerichteten Graphen	
Gegeben:	Ein **ungerichteter** Graph $G = (V, E)$.
Frage:	Gibt es einen Hamiltonschen Kreis in G?

Existenz Hamiltonscher Pfade (HamiltonianPath)	
Gegeben:	Ein gerichteter Graph $G = (V, E)$, Knoten $s, t \in V$.
Frage:	Gibt es einen Hamiltonschen Pfad von s nach t in G?

Ein Hamiltonscher Pfad ist ein Pfad, der alle Knoten genau einmal besucht.

Um zu zeigen, dass die Probleme jeweils NP-schwer sind, kann man HamiltonianCycle reduzieren. Dies sei der Leserin / dem Leser als Übungsaufgabe überlassen.

Berechnen Hamiltonscher Kreise

Wir haben bislang die Existenz Hamiltonscher Kreise untersucht. Eigentlich möchte man allerdings einen Algorithmus, der nicht nur entscheidet, ob es einen solchen Kreis gibt, sondern ihn im Ja-Fall auch ausgibt.

Allgemeiner formuliert haben wir in den letzten Kapiteln der Vorlesung ausschließlich Entscheidungsprobleme betrachtet. In der Praxis möchte man oft allerdings

Berechnungsprobleme lösen. Es ist jedoch so, dass ein effizienter Algorithmus für das Entscheidungsproblem auch zu einem effizienten Algorithmus für das Berechnungsproblem führt. Wir demonstrieren das exemplarisch für Hamiltonsche Kreise.

Hamiltonsche Kreise berechnen	
Gegeben:	Ein gerichteter Graph $G = (V, E)$.
Berechne:	Einen Hamiltonschen Kreis in G, oder *false*, wenn es keinen gibt.

Wir nehmen an, dass wir einen deterministischen Algorithmus HamiltonianCycle(G) gegeben haben, der bestimmt, ob es im Graphen G einen Hamiltonschen Kreis gibt. Diesen werden wir in unserem Algorithmus aus Subroutine aufrufen.

Sei G der Eingabegraph. Wir werden diesen jedoch im Lauf des Algorithmus verändern.

```
if not HamiltonianCycle(G) then
    return false
end if
while G hat Knoten mit Ausgangsgrad > 1 do
    Wähle eine solchen Knoten v
    while HamiltonianCycle(G) do
        Lösche beliebige von v ausgehende Kante
    end while
    Stelle letzte gelöschte Kante wieder her
    Lösche alle anderen von v ausgehenden Kanten
end while
return G
```

Wir argumentieren, dass der Algorithmus das korrekte Ergebnis liefert und untersuchen dann seine Laufzeit.

Ein Hamiltonscher Kreis in G benutzt zu jedem Knoten v genau eine ausgehende Kante. Alle anderen Kanten, die von einem Knoten ausgehen, können gelöscht werden, ohne dass die Existenz des Hamiltonschen Kreises beeinflusst wird. Die innere Schleife löscht ausgehende Kanten, bis es keinen Hamiltonschen Kreis mehr gibt. (Dies tritt spätestens dann auf, wenn alle ausgehenden Kanten gelöscht wurden.) Sobald dies auftritt, wissen wir, dass die letzte gelöschte Kante im Hamiltonschen Kreis verwendet wird. Durch das Wiederherstellen dieser Kante stellen wir sicher, dass HamiltonianCycle(G) wahr ist. Nun können wir alle anderen ausgehenden Kanten löschen, da sie nicht benötigt werden.

Nach dem Durchlauf der äußeren Schleife erhalten wir einen Teilgraphen des Eingabegraphen, der einen Hamiltonschen Kreis hat. Da der Teilgraph jedoch zu jedem Knoten nur einen Nachfolger hat, ist der Graph selbst eine Darstellung eines Hamiltonschen Kreises im Eingabegraphen.

Nehmen wir an, dass unser Entscheider HamiltonianCycle Laufzeit f hat, und n die Größe der Eingabe G ist. Alle Graphen, für die wir HamiltonianCycle aufrufen, sind höchstens so groß wie G. Dementsprechend verursacht jeder Aufruf von HamiltonianCycle im Algorithmus einen Zeitverbrauch von höchstens $f(n)$. Die Anzahl der Aufrufe von HamiltonianCycle ist höchstens $k + 1$, wobei k die Anzahl der Kanten im Originalgraphen ist. Insgesamt ist die Laufzeit also in $\mathcal{O}((k + 1) \cdot f(n))$.

Wenn wir HamiltonianCycle deterministisch in Polynomialzeit lösen könnten, dann würde auch der obige Algorithmus in Polynomialzeit laufen. Das Berechnungsproblem ließe sich dann also auch effizient lösen.

11.C 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 polynomiell 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. Dies liefert eine alternative Definition für NP.

11.16 Bemerkung
$\mathsf{P} \stackrel{?}{=} \mathsf{NP}$ ist eines von 7 Millenium-Problemen. Diese Probleme wurden im Jahr 2000 vom Clay Mathematics Institute ausgeschrieben; ihre Lösung wird mit einem Preisgeld von einer Million US-Dollar belohnt. 6 der 7 Probleme sind derzeit noch ungelöst, darunter $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$.

In der offiziellen Problembeschreibung zu $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$, `https://www.claymath.org/sites/default/files/pvsnp.pdf`, verfasst von Stephen Cook, wird NP nicht als

$$\mathsf{NP} = \bigcup_{k \in \mathbb{N}} \mathrm{NTIME}\big(\mathcal{O}(n^k)\big)$$

definiert. Stattdessen verwendet Cook die zertifikatsbasierte Definition, die wir im folgenden präsentieren werden.

Wir beginnen zunächst damit, das Konzept der Zertifikate zu formalisieren.

11.17 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.

Wir nennen ein solches $\mathcal{V}$ auch einen Verifizierer für $\mathcal{V}$.

Im folgenden sind wir in **Polynomialzeit-Verifizierer** interessiert. Ein solcher hält auf Eingabe (x,y) in $\mathcal{O}((|x|+|y|)^k)$ vielen Schritten. Außerdem ist hier ein Wort x nur in der Sprache, wenn es ein polynomiell langes Zertifikat y gibt, also y mit $|y| \leq |x|^k$, wobei die Konstante k von x unabhängig ist.

$$\mathcal{L}(\mathcal{V}) = \left\{ x \in \Sigma^* \;\middle|\; \exists y \in \{0,1\}^*, |y| \leq |x|^k \colon \mathcal{V} \text{ akzeptiert } (x,y) \right\}.$$

Polynomialzeit-Verifizierer liefern eine alternative Definition von NP. Man beachte, dass diese Verifizierer deterministisch sind. Die Rolle des Nichtdeterminismus nimmt hier die **existentielle Quantifizierung** über y in der Definition von $\mathcal{L}(\mathcal{V})$ ein.

11.18 Theorem
Die Klasse NP sind genau die Probleme, die von Polynomialzeit-Verifizierern gelöst werden können: Es gilt $\mathcal{L} \in \mathsf{NP}$ genau dann, wenn es einen Polynomialzeit-Verifizierer $\mathcal{V}$ mit $\mathcal{L}(\mathcal{V}) = \mathcal{L}$ gibt.

Bevor wir den Satz zeigen, demonstrieren wir zunächst, dass sich mit dieser Definition leicht zeigen lässt, dass viele Probleme in NP liefen.

11.19 Beispiel

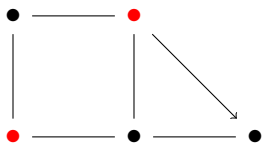
Wie bereits oben angesprochen kann man bei SAT eine erfüllende Belegung, welche für Ja-Instanzen existieren muss, als Zertifikat verwenden.

11.20 Beispiel

Independent Set	
Gegeben:	Ein gerichteter Graph $G = (V, E)$ und eine Zahl k
Frage:	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 11.18 ist also gezeigt, dass Independent Set in NP liegt. Tatsächlich ist Independent Set sogar NP-vollständig.

11.21 Beispiel

Subset Sum	
Gegeben:	Ein Menge von Zahlen $N = \{n_1, \ldots, n_k\}$ und eine Zahl S
Frage:	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 11.18 ist also gezeigt, dass Subset Sum in NP liegt. Tatsächlich ist Subset Sum sogar NP-vollständig.

11.22 Beispiel

Factorization	
Gegeben:	Zwei Zahlen $n, b \in \mathbb{N}$.
Frage:	Gibt es eine Zahl m mit $1 < m \leq b$, die n teilt?

Ein Polynomialzeit-Verifizierer für Factorization überprüft, ob auf dem Zertifikatsband die Binärkodierung einer Zahl m steht, die $1 < m \leq b$ erfüllt und n teilt. Das Zertifikat ist also der gesuchte Teiler. Mit Hilfe von Theorem 11.18 ist also gezeigt, dass Factorization in NP liegt.

Factorization liegt außerdem auch in coNP. Um dies zu beweisen, zeigen wir, dass das Komplementproblem in NP liegt.

$\overline{\text{Factorization}}$	
Gegeben:	Zwei Zahlen $n, b \in \mathbb{N}$.
Frage:	Gibt es keinen Teiler von n, der in $\{2, \ldots, b\}$ liegt?

Ein Polynomialzeit-Verifizierer für $\overline{\text{Factorization}}$ überprüft, ob auf dem Zertifikatsband die Kodierung der Primfaktorzerlegung

$$p_1^{k_1} + \ldots + p_\ell^{k_\ell}$$

von n steht. Dazu muss getestet werden, dass jedes der p_i tatsächlich eine Primzahl ist (z.B. mit dem AKS-Primzahltest), und dass $n = p_1^{k_1} + \ldots + p_\ell^{k_\ell}$ gilt. Nun akzeptiert der Verifizierer genau dann, wenn der kleinste Primfaktor p_i größer als b ist.

Factorization ist das zugehörige Entscheidungsproblem zum Berechnungsproblem, bei dem der kleinste Primfaktor einer Zahl bestimmt werden sol..

Finden des kleinsten Primfaktors	
Gegeben:	Eine Zahl $n > 1$.
Berechne:	Den kleinsten nicht-trivialen (> 1) Teiler von n.

Ein effizienter deterministischer Algorithmus für Factorization kann mit dem Konzept der Binärsuche kombiniert werden, um den kleinsten Primfaktor einer Zahl effizient zu finden. Dies würde einige asymmetrische Verschlüsselungsverfahren, wie z.B. RSA, angreifbar machen.

Da Factorization in $\mathsf{NP} \cap \mathsf{coNTIME}$ liegt, glaubt man, dass Factorization nicht NP-vollständig ist. (Dies würde dazu führen, dass die sogenannte polynomielle Hierarchie kollabiert, was man für unwahrscheinlich hält. Dies formal zu machen würde den Rahmen dieser Vorlesung sprechen.) Es ist jedoch auch kein deterministisches Verfahren bekannt, dass Factorization in Polynomialzeit löst.

Nun skizzieren wir noch den Beweis für Theorem 11.18.

Beweisskizze für Theorem 11.18:

- Sei $\mathcal{L} \in \mathsf{NP}$, also $\mathcal{L} = \mathcal{L}(M)$ für eine NTM M mit Zeitschranke n^k. Wir nehmen wir im Beweis von Proposition 11.5 an, dass M nur binär verzweigt.

  Wir konstruieren einen Polynomialzeit-Verifizierer $\mathcal{V}$, der als Zertifikat einen Binärstring der Länge n^k erwartet. Dieser verhält sich dann wie M auf der gegebenen Eingabe, benutzt aber das Zertifikat, um den Nichtdeterminismus von M aufzulösen: In der i. Transition wird die i. Zelle des Zertifikatbands benutzt, um eine deterministische Entscheidung zwischen der linken und der rechten Transition von M zu treffen.

  Wenn M eine akzeptierende Berechnung hat, dann akzeptiert auch $\mathcal{V}$, wenn wir als Zertifikat den Binärstring wählen, der die akzeptierende Berechnung kodiert. Analog entspricht ein Zertifikat, das von $\mathcal{V}$ akzeptiert wird, einer akzeptierenden Berechnung von M.

- Sei $\mathcal{V}$ ein Polynomialzeit-Verifizierer, dessen Zertifikate Länge $\leq n^k$ haben. Wir konstruieren eine NTM M, die zunächst ein Zertifikat der entsprechenden Länge rät, und sich dann verhält wie $\mathcal{V}$ auf der gegebenen Eingabe und dem geratenen Zertifikat.

□

Mit der Charakterisierung von NP aus Theorem 11.18 ergibt sich eine neue Sicht auf die Komplexitätsklassen und auf die Frage $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$.

- P ist die Klasse der Probleme, bei denen eine Lösung effizient bestimmt werden kann.
- NP ist die Klasse der Probleme, bei denen für ein gegebenen Lösungsvorschlag effizient verifiziert werden kann.

Die Frage nach der Gleichheit von P und NP hat damit eine philosophische Dimension: Existiert Kreativität? Gibt es Probleme, bei denen es schwierig ist, die Lösung zu bestimmen (oder man dafür „kreativ“ / „genial“ sein muss)? Oder ist es so, dass wenn immer das Verifizieren der Lösung leicht ist, auch das Bestimmten der Lösung leicht sein muss?

Bedeutung von $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$

$\mathsf{P} \stackrel{?}{=} \mathsf{NP}$ ist nicht bloß eine akademische Frage, sondern hat Bedeutung für die Praxis.

- Viele Optimierungsprobleme, die NP-vollständig sind, wie z.B. ganzzahlige lineare Optimierung, werden tagtäglich für große Instanzen gelöst. Moderne SAT-Solver entscheiden die Erfüllbarkeit von Formeln mit Millionen von Variablen in Sekunden oder Minuten.

  Hier nutzt man diverse Eigenschaften aus:

  - Wir machen hier nur Worst-Case-Analyse. Selbst wenn die Theorie sagt, dass sich nicht *alle* Instanzen eines Problems effizient lösen lassen, kann es dennoch sein, dass sich viele Instanz effizient lösen lassen. Mit ein bisschen Glück handelt es sich hierbei um die Instanzen, die in der Praxis auftreten.

  - Viele Instanzen, die in der Praxis auftreten, sind zwar sehr groß, haben aber eine spezielle Struktur, die von einem Algorithmus ausgenutzt werden kann. Es gibt das Gebiet der **Fine-Grained Complexity Theory**, dass sich mit der Theorie hierzu beschäftigt.

  - Bei einigen (aber nicht allen) NP-vollständigen Optimierungsproblemen lässt sich die Lösung approximieren. Das bedeutet, dass man die Optimallösung nicht deterministisch in Polynomialzeit bestimmen kann, aber eine Approximation der Lösung, die nur minimal schlechter ist.

- Andererseits nutzt man an machen Stellen auch aus, dass manche Probleme bislang nicht effizient gelöst werden kann. Asymmetrische Verschlüsselungsverfahren basieren beispielsweise darauf, dass man den privaten (geheimen) Schlüssel zum Entschlüsseln nicht auf einfache Art und Weise aus dem öffentlichen Schlüssel zum Verschlüsseln berechnen kann.

  Beim oft verwendeten Verfahren RSA (welches zum Beispiel zur sicheren Internetkommunikation via HTTPS verwendet werden kann) werden die Schlüssel

aus dem Produkt von Primzahlen errechnet. Wenn man das Faktorisierungsproblem effizient lösen könnte, dann wäre es möglich, aus dem öffentlichen Schlüssel den privaten auszurechnen, und RSA wäre damit unsicher.

Möglichkeiten für $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$

- $\mathsf{P} = \mathsf{NP}$
  - Konstruktiver Beweis, der zu einem praxistauglichen Algorithmus für ein NP-vollständiges Problem führt.

    Konsequenz: Alle NP-vollständigen Probleme können nun effizient gelöst werden.
  - Konstruktiver Beweis, der jedoch nicht zu einem praxistauglichen Algorithmus für ein NP-vollständiges führt. Beispielweise könnte die Laufzeit des Algorithmus $c \cdot n^k + d$ für sehr große c, k, d sein.

    Konsequenz: NP-vollständige Probleme können in Polynomialzeit gelöst werden, die Algorithmen sind jedoch (noch) nicht praxistauglich. Mehr Forschung wäre vonnöten.
  - Nicht-konstruktiver Beweis.

    Konsequenz: Man weiß, dass alle NP-vollständige Probleme in Polynomialzeit gelöst werden können, kennt jedoch die Algorithmen nicht.
- $\mathsf{P} \neq \mathsf{NP}$

  Konsequenz: Kein NP-vollständiges Problem lässt sich in Polynomialzeit lösen.
- $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$ ist logisch unabhängig, also auf Grundlage unseres mathematischem Axiomensystem nicht lösbar. (Einige andere Probleme, die lange Zeit ungelöst waren, wie z.B. die Kontinuumshypothese, haben diese Eigenschaft.)

  Konsequenz: Es ist zwar nicht ausgeschlossen, dass sich NP-vollständige Probleme in Polynomialzeit lösen lassen, die Algorithmen lassen sich aber auf jeden Fall nicht explizit konstruieren und sind uns daher nicht zugänglich. (Unter der Annahme, dass unser mathematisches Axiomensystem nicht falsch ist.)

Warum glaubt man, dass $\mathsf{P} \neq \mathsf{NP}$ gilt?

Die meisten Informatiker glauben, dass $\mathsf{P} \neq \mathsf{NP}$ gilt. Scott Aaronson, amerikanischer Forscher im Bereich der Komplexitätstheorie, argumentiert in seinem Blog (`https://www.scottaaronson.com/blog/?p=1720`) wie folgt: Es gibt tausende Probleme aus P. Außerdem gibt es tausende NP-vollständige Probleme, und Reduktionen von jedem solchen Problem auf jedes andere solche Problem. Es gibt also mehr als $1000 \cdot 1000 = 1.000.000$ Kombinationen $(\mathcal{L}_{\mathsf{NP}}, \mathcal{L}_{\mathsf{P}})$ aus einem NP-vollständigen und einem P-Problem. Wenn nur für eine solche Kombination $\mathcal{L}_{\mathsf{NP}} \leq_m^{\text{poly}} \mathcal{L}_{\mathsf{P}}$ gelten würde, wäre $\mathsf{P} = \mathsf{NP}$ gezeigt. In diesem Fall würde dann sogar $\mathcal{L}_{\mathsf{NP}} \leq_m^{\text{poly}} \mathcal{L}_{\mathsf{P}}$ für alle Paare von $\mathsf{NP} = \mathsf{P}$-vollständigen Problemen gelten. Die Tatsache, dass man trotz einer Million möglichen Kandidaten bislang keine solche Reduktion für kein Paar von Problemen finden konnte, lässt es als unwahrscheinlich erscheinen, dass eine solche Reduktion existieren kann.

Dies ist ein überzeugendes Argument (für die vollständige Argumentation verweisen wir auf den Blogpost), aber kein Beweis. Tatsächlich gibt es bereits andere Probleme, für die die Lösung erst nach Jahrzehnten gefunden wurde. In manchen dieser Fälle war die Lösung zudem eine andere als initial erwartet. Beispielweise zeigte Khachiyan 1979, dass lineare Optimierungsprobleme in P gelöst werden können, und auch der Satz von Immerman & Szelepcsényi, der $\mathsf{coNL} = \mathsf{NL}$ zeigt, ist ein solches Beispiel.

Was hat die Forschung die letzten 50 Jahre gemacht?

$\mathsf{P} \stackrel{?}{=} \mathsf{NP}$ ist nun seit fast 50 Jahren ein ungelöstes Problem. Einerseits schreckt dies viele Forscher ab: Niemand möchte mehrere Jahre seines Lebens in den Versuch investieren, die Frage zu lösen, nur um mit hoher Chance zu scheitern. Andererseits war die Forschung nicht untätig: Man hat für viele mögliche Beweisansätze gezeigt, dass sie nicht zum Erfolg führen können. Man kennt also keinen Beweis für $\mathsf{P} \neq \mathsf{NP}$, man weiß aber, wie ein solcher Beweis *nicht* aussehen kann. Außerdem wurde im Rahmen der Forschung zu $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$ viele Konzepte entwickelt, die zwar die Frage nicht lösen könnten, sich aber an anderer Stelle als hilfreich erwiesen haben, wie z.B. die polynomielle Hierarchie.

12. PSPACE und der Satz von Savitch

In diesem Kapitel möchten wir uns mit den Problemen befassen, welche mit polynomiellem Platz gelöst werden können. Viele Probleme in PSPACE enthalten eine Form von Alternierung. Beispiele sind das Finden von Gewinnstrategien in rundenbasierten Brettspielen oder die Handhabung von (alternierenden) Quantoren in logischen Formeln. Andererseits sind auch einige Probleme der formalen Sprachen in PSPACE enthalten. Wir werden hier Probleme beider Sorten kennenlernen. Wir beginnen, wie auch in den Kapiteln zu den anderen Komplexitätsklassen, mit einem ersten PSPACE-vollständigem Problem.

12.A QBF ist PSPACE-vollständig

Wir werden zunächst zeigen, dass das Erfüllbarkeitsproblem für quantifizierte boolsche Formeln PSPACE-vollständig ist. Der Beweis geht zurück auf Stockmeyer und Meyer.

12.1 Definition
Eine quantifizierte Boolsche Formel ist von der Form

$$F = Q_1 x_1 \dots Q_n x_n . G(x_1, \dots x_n),$$

wobei $Q_i \in \{\forall, \exists\}$ Quantoren sind und G eine Boolsche Formel mit (freien, also unquantifizierten) Variablen $x_1, \dots x_n$ ist. Den Präfix $Q_1 x_1 \dots Q_n x_n$ nennt man auch den Quantorenblock von F.

Der Wahrheitswert einer quantifizierten Boolschen Formel wird durch folgende rekursive Prozedur bestimmt. Seien dazu

$$F_{(x_1 = \mathit{true})} = Q_2 x_2 \dots Q_n x_n . G(\mathit{true}, x_2, \dots x_n) \text{ und}$$
$$F_{(x_1 = \mathit{false})} = Q_2 x_2 \dots Q_n x_n . G(\mathit{false}, x_2, \dots x_n).$$

Die Boolsche Formel $G(B, x_2, \dots x_n)$ wird aus G erhalten, indem jedes Vorkommen von x_1 durch $B \in \{\mathit{true}, \mathit{false}\}$ ersetzt wird. Falls $Q_1 = \forall$, hat F den Wahrheitswert *true*, genau dann wenn $F_{(x_1 = \mathit{true})}$ und $F_{(x_1 = \mathit{false})}$ den Wahrheitswert *true* haben. Falls $Q_1 = \exists$, hat F den Wahrheitswert *true*, genau dann wenn $F_{(x_1 = \mathit{true})}$ oder $F_{(x_1 = \mathit{false})}$ den Wahrheitswert *true* haben. Die quantifizierten Formeln $F_{(x_1 = \mathit{true})}$ und $F_{(x_1 = \mathit{false})}$ haben eine Variable weniger als F und werden rekursiv ausgewertet. Die Prozedur

wird fortgesetzt bis die erstellten Formeln keine Variablen mehr enthalten und sie auf bereits bekannte Weise ausgewertet werden können.

Intuitiv kann man sich quantifizierte Boolsche Formeln und ihre Wahrheitswerte wie folgt vorstellen. Die Variablen x in der Formel G können die Werte *true* und *false* annehmen. Die Quantoren vor den jeweiligen Variablen bestimmen, ob alle Belegungen ($\forall$) für x betrachtet werden müssen, oder ob eine passende ausgesucht werden kann ($\exists$). Ein Belegung für eine quantifizierte Variable x kann allerdings nur in Abhängigkeit der Belegungen der Variablen gewählt werden, die im Quantorenblock links von x stehen. In der obigen Prozedur erkennt man diese Prinzip daran, dass der Quantorenblock von links nach rechts abgearbeitet wird. Folgendes Beispiel verdeutlicht die Abhängigkeiten der Quantoren.

12.2 Beispiel
Betrachten wir die Formeln

$$F_1 = \forall x\ \exists y.\ (x \leftrightarrow y) \text{ und}$$
$$F_2 = \exists y\ \forall x.\ (x \leftrightarrow y).$$

Die Formel F_1 hat den Wahrheitswert *true*. Für jede Belegung von x wählen wir die gleiche Belegung für Variable y. Formel F_2 hingegen hat den Wahrheitswert *false*. Hier müssen wir zuerst eine Belegung für Variable y wählen. Es gibt aber keine Belegung für y, sodass für jede Belegung für x die Boolsche Formel $(x \leftrightarrow y)$ zu *true* ausgewertet wird.

Das Problem, ob einen gegebene quantifizierte Boolsche Formel den Wahrheitswert *true* hat, ist unser erstes PSPACE-vollständiges Problem.

Quantified Boolean Formula Problem (QBF)	
Gegeben:	Eine quantifizierte Boolsche Formel F.
Frage:	Hat F den Wahrheitswert *true*?

12.3 Bemerkung
Das aus Kapitel 11 bekannte Problem SAT ist ein Spezialfall von QBF, bei dem alle Quantoren von F Existenzquantoren ($\exists$) sind. Das coNP-vollständige Problem VALID (gegeben eine Boolsche Formel, ist sie allgemeingültig?) ist ein weiterer Spezialfall. In diesem Fall sind alle Quantoren in F Allquantoren ($\forall$).

12.4 Theorem: Stockmeyer & Meyer 1973
QBF ist PSPACE-vollständig bezüglich logspace-many-one-Reduktionen.

Beweis:

QBF $\in$ PSPACE

Wir verwenden die Prozedur aus Definition 12.1 und argumentieren, dass sie nur polynomiell viel Platz in der Länge der Eingabe F benötigt. Die Rekursionstiefe des Algorithmus ist gegeben durch die Anzahl der Variablen. Bei jedem Aufruf wird die linkeste Variable im verbleibenden Quantorenblock bearbeitet. Wir müssen uns nur die Belegung der bereits bearbeiteten Variablen merken, sowie die daraus entstehende quantifizierte Formel. Des weiteren kann beim rekursiven Aufruf für $F_{(x_1=\mathit{false})}$ der Platz, der für den Aufruf von $F_{(x_1=\mathit{true})}$ benötigt wurde, wiederverwendet werden. Somit wird nur linear viel Platz in der Länge von F benötigt.

QBF ist PSPACE-hart

Sei A ein Problem aus PSPACE, welches von einer Turingmaschine M mit n^k (n ist die Eingabegröße und $k \in \mathbb{N}$) viel Platz gelöst werden kann. Wir geben eine logspace-Reduktion an, die für eine Eingabe w eine Formel F generiert, so dass

> F hat Wahrheitswert *true* gdw. M akzeptiert w.

Um F zu konstruieren, lösen wir ein allgemeineres Problem. Gegeben zwei Variablen c_1 und c_2 (die Konfigurationen von M darstellen sollen) und eine natürliche Zahl $t > 0$, konstruieren wir die Formel $F_{c_1,c_2,t}$. Wenn wir den Variablen c_1 und c_2 Konfigurationen zuweisen, soll $F_{c_1,c_2,t}$ folgende Eigenschaft haben.

> $F_{c_1,c_2,t}$ hat Wahrheitswert *true*
> gdw.
> M kann von c_1 zu c_2 mit höchstens t Schritten gehen.

Da eine Turingmaschine mit einer Platzschranke von n^k sich durch eine Turingmaschine mit Zeitschranke 2^{dn^k} simulieren lässt (siehe Kapitel 8), ist die gesuchte Formel $F_{c_{\text{start}},c_{\text{acc}},t}$ mit $t = 2^{dn^k}$. Mit c_{start} (c_{acc}) ist die eindeutige Startkonfiguration (akzeptierende Konfiguration) gemeint. Wie im Beweis von Theorem 9.13, können wir davon ausgehen, dass eine eindeutige akzeptierende Konfiguration existiert.

Technisch gesehen kodiert die Formel den Inhalt der Zellen einer Konfiguration, wie in dem Beweis von Ladners Satz (Theorem 10.6). Jede Zelle j wird durch mehrere Variablen beschrieben:

- eine Variable P_j^a für jedes Bandsymbol a vom M und
- eine Variable Q_j^p für jeden Zustand p von M.

Da jede Konfiguration höchstens n^k Zellen verwendet, gibt es $\mathcal{O}(n^k)$ Variablen pro Konfiguration.

Für $t = 1$ können wir $F_{c_1,c_2,t}$ wie folgt konstruieren. Die Formel sagt entweder aus, dass $c_1 = c_2$ oder dass M von c_1 nach c_2 mit nur einem Schritt gehen kann. Für den ersten Fall verlangt die Formel, dass jede Variable P_j^a bzw. Q_j^p die c_1 repräsentiert, den gleichen Wahrheitswert hat wie die zugehörige Variable für c_2. Für den zweiten Fall verwenden wir die Konstruktion aus Ladners Satz.

Für $t > 1$ konstruieren wir $F_{c_1,c_2,t}$ rekursiv. Um die Idee zu verdeutlichen, fangen wir mit einer Formel-Konstruktion an, die zwar das Gewünschte aussagt, jedoch zu groß wird. Wir definieren

$$F_{c_1,c_2,t} := \exists c_j.(F_{c_1,m,t/2} \wedge F_{m,c_2,t/2}).$$

Intuitiv, sagt die Formel, dass es eine Zwischenkonfiguration c_j geben muss, so dass M von c_1 zu c_j und von c_j zu c_2 mit jeweils $t/2$ Schritte gehen kann. Die Variable c_j steht hier eigentlich für eine Sequenz von $\mathcal{O}(n^k)$ existentiell quantifizierter Variablen, nämlich die Variablen P_j^a, Q_j^p, welche die Zwischenkonfiguration c_j beschrieben

Diese Formel hat zwar den gewünschten Wahrheitswert, allerdings wird dadurch $F_{c_{\text{start}},c_{\text{acc}},t}$ zu groß. Jeder rekursive Schritt halbiert t und verdoppelt dabei die Größe der Formel. Wir haben also am Ende eine Formel der Größe $t = 2^{dn^k}$.

Um dieses Problem zu umgehen, verwenden wir zusätzlich Allquantoren um die beiden rekursiven Subformeln mit nur einer Subformel zu beschreiben. Dazu verwenden wir folgende neue Definition.

$$F_{c_1,c_2,t} := \exists m\ \forall D_1 \forall D_2.\ \big[(D_1 = c_1 \wedge D_2 = m) \vee (D_1 = m \wedge D_2 = c_2)\big] \rightarrow F_{D_1,D_2,t/2}$$

Jeder rekursive Schritt fügt nun einen Teil der Größe $\mathcal{O}(n^k)$ zu der Formel hinzu. Die Anzahl der rekursiven Schritte ist

$$\log t = \log 2^{dn^k} = \mathcal{O}\big(n^k\big)$$

und somit ist die Größe von $F_{c_{\text{start}},c_{\text{acc}},t}$ beschränkt durch $\mathcal{O}((n^k)^2)$.

Es bleibt zu argumentieren, dass die Formel auch logspace-berechenbar ist. Da wir im Grunde nur einen Zähler bis n^k brauchen, benötigen wir nur $\log n^k = k \log n$ viele Bits auf dem Arbeitsband. □

12.B Der Satz von Savitch

Das wichtigste offene Problem der Theoretischen Informatik ist $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$, oder allgemeiner formuliert, die Beziehung zwischen deterministischen und nicht-deterministischen Zeitkomplexitätsklassen. Ähnliche Fragen stellen sich auch für die Platzkomplexitätsklassen z.B. $\mathsf{PSPACE} \stackrel{?}{=} \mathsf{NPSPACE}$. Diese wurden jedoch bereits 1970 von Walter Savitch gelöst: Gleichheit gilt. In diesem Kapitel möchten wir den Beweis seines Satzes nachvollziehen.

12.5 Theorem: Satz von Savitch, 1970
Sei $s\colon \mathbb{N} \to \mathbb{N}$ eine Platzschranke mit $s(n) \geq \log n$ für alle n. Es gilt

$$\text{NSPACE}(s) \subseteq \text{DSPACE}\big(s^2\big) \ .$$

Eine nicht-deterministische Maschine lässt sich also so determinisieren, dass sich der Platzverbrauch lediglich quadriert (insbesondere also nur polynomiell erhöht).

12.6 Korollar
Es gilt

$$\mathsf{PSPACE} = \mathsf{NPSPACE} \text{ und } \mathsf{EXPSPACE} = \mathsf{NEXPSPACE} \ .$$

Das Korollar folgt aus dem Satz von Savitch, da PSPACE und $\mathsf{EXPSPACE}$ unter Quadrierung abgeschlossen sind. Als weitere Folgerung erhalten wir auch, dass $\mathsf{coNPSPACE} = \mathsf{coPSPACE} = \mathsf{PSPACE}$ gilt.

12.7 Bemerkung
Der Satz von Savitch zeigt **nicht**, dass $\mathsf{L} = \mathsf{NL}$ gilt, dieses Problem ist nach wie vor offen. Ein Problem aus NL hat einen nicht-deterministischen Entscheider mit Platzverbrauch $s \in \mathcal{O}(\log n)$. Der Satz von Savitch liefert uns einen deterministischen Entscheider mit Platzverbrauch $s^2 \in \mathcal{O}((\log n)^2)$. Es gilt jedoch nicht zwangsweise $s^2 \in \mathcal{O}(\log n)$.

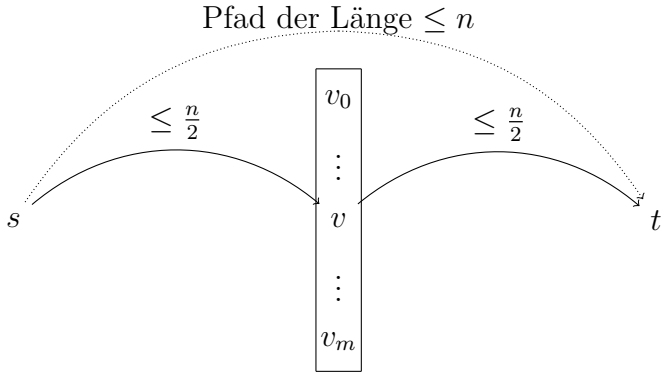
Um den Satz zu beweisen, werden wir verwenden, dass sich das Akzeptanzproblem für Turing-Maschinen als Pfadproblem im Konfigurationsgraphen auffassen lässt. Wir werden zeigen, wie sich das Pfadproblem PATH für Graphen mit n Knoten deterministisch mit Platzverbrauch $(\log n)^2$ lösen lässt. Wenn wir dies auf den Konfigurationsgraphen mit $2^{\mathcal{O}(s(n))}$ vielen Knoten anwenden, erhalten wir eine deterministische Lösung mit Platzverbrauch $s(n)^2$.

Die Idee hierzu ist die Folgende: Wenn es einen Pfad von Knoten s zu Knoten t gibt, dann gibt es auch einen einfachen Pfad, und damit insbesondere einen Pfad, der höchstens Länge n hat. Um diese Existenz zu überprüfen, suchen wir nach einem Zwischenknoten v und überprüfen rekursiv

1. ob es einen Pfad von s nach v mit Länge höchstens $\frac{n}{2}$ gibt, und
2. ob es einen Pfad von v nach t mit Lange höchstens $\frac{n}{2}$ gibt.

Dabei ist vorteilhaft, dass wir bei Schritt 2. den Platz, den wir für Schritt 1. verwendet haben, wiederverwenden können. Da wir die Zwischenkonfiguration v nicht kennen, müssen wir über alle möglichen Konfiguration iterieren.

Die folgende Abbildung illustriert die Prozedur.



Um Schritt 1. und Schritt 2. umzusetzen, rufen wir die Prozedur rekursiv auf.

12.8 Algorithm: savitch(G, s, t, k)
Eingabe: Graph G, Startknoten s, Zielknoten t, Schranke $k \in \mathbb{N}$
Ausgabe: true genau dann, wenn es in G einen Pfad von s nach t der Länge $\leq k$ gibt

```
1: if k = 0 then
2: |   return (s = t)
3: end if
4: if k = 1 then
5: |   return (s → t)
6: end if
7: if k > 1 then
8: |   for v Knoten von G do
9: |   |   if savitch(G, s, v, ⌈k/2⌉) and savitch(G, v, t, ⌊k/2⌋) then
10: |  |   |   return true
11: |  |   end if
12: |  end for
13: |  return false
14: end if
```

Wir haben bereits argumentiert, dass der Algorithmus tatsächlich das Pfadproblem löst. Wir untersuchen nun noch den Platzverbrauch. Wir können den Algorithmus so implementieren, dass er einen Stack als Speicher nutzt. Der oberste Eintrag des Stacks speichert dabei die Parameter der aktuellen Prozedurinstanz, also s, t, v und k. Wenn wir annehmen, dass $s, t, v \in \{1, \ldots, n\}$ und $k \leq n$ gilt, dann benötigen wir für einen solchen Stackeintrag höchstens $4 \cdot \log n$ viel Platz. Die Tiefe der Rekursion ist höchstens $\log n$, da wir k in jedem Schritt halbieren und damit nach maximal $\log k$ Schritten in einem der Basisfälle ankommen. Insgesamt benötigt unser Algorithmus also

$$\underbrace{(4 \cdot \log n)}_{\text{pro Stackeintrag}} \cdot \underbrace{(\log n)}_{\text{Stackhöhe}} \in \mathcal{O}\big((\log n)^2\big)$$

viel Platz.

12.C PSPACE-vollständige Probleme regulärer Sprachen

Wir wenden uns nun Entscheidungsproblemen zu deren Eingabe regulärer Sprachen – repräsentiert durch endliche Automaten (NFAs) – sind . Wir zeigen, dass viele interessante Problem PSPACE-vollständig ist.

Universality
Gegeben: Ein endlicher Automat A über dem Alphabet Σ.
Frage: Gilt $\mathcal{L}(A) = \Sigma^*$?

Equivalence
Gegeben: Zwei endliche Automaten A und B.
Frage: Gilt $\mathcal{L}(A) = \mathcal{L}(B)$?

Inclusion
Gegeben: Zwei endliche Automaten A und B.
Frage: Gilt $\mathcal{L}(A) \subseteq \mathcal{L}(B)$?

12.9 Theorem
Universality ist PSPACE-vollständig bezüglich logspace-many-one-Reduktionen.

Beweis:

Universality $\in$ PSPACE

Die Idee des Algorithmus is es, ein Wort in Σ^* zu raten, welches nicht von A akzeptiert wird. Wenn es ein solches Wort gibt, dann gibt es auch ein Wort, welches höchstens exponentiell lang in der Größe von A ist und von A nicht akzeptiert wird (Beweis: Übung). Da wir das Wort aber nicht speichern können, raten wir das Wort Buchstabe für Buchstabe und verwenden einen Zähler um die Anzahl der Rateschritte zu begrenzen. Für jeden Rateschritt merken uns in welchen Zuständen der Automat sein könnte und determinisieren so den Automaten entlang dem geratenen Wort.

Genauer starten wir mit einem Marker auf dem Startzustand. Wenn ein Buchstabe $a \in \Sigma$ geraten wurde, bewegen wir alle Marker von dem Zustand, auf den sie zeigen, zu allen mit a erreichbaren Zuständen. Dabei können neue Marker enstehen (wenn es mehrere ausgehenden a-Kanten gibt) oder Marker gelöscht werden (falls es keine

ausgehenden a-Kanten gibt). Wenn der Rateprozess zu Ende ist und kein Marker auf einem Endzustand steht, akzeptiert der Algorithmus.

Der Algorithmus benötigt nur polynomiell viel Platz in der Größe von A. Zum einen muss der Algorithmus für jeden Rateschritt die Zustände speichern, in denen der Automat sein könnte, was linear viel Platz benötigt. Zum anderen braucht der Algorithmus einen Zähler bis $2^{|A|}$, für den $|A|$ Bits nötig sind. Wir haben also gezeigt, dass Universality $\in$ coNPSPACE. Mit dem Satz von Savitch folgt dann

$$\text{Universality} \in \mathsf{coNPSPACE} = \mathsf{coPSPACE} = \mathsf{PSPACE},$$

da deterministische Komplexitätsklassen unter Komplementierung abgeschlossen sind.

Universality ist PSPACE−hart

Sei A ein Problem aus PSPACE, welches von einer Turingmaschine M mit n^k (n ist die Eingabegröße und $k \in \mathbb{N}$) viel Platz gelöst werden kann. Wir nehmen wieder an, dass M eine eindeutige akzeptierende Konfiguration hat.

Die Idee der Reduktion ist die folgende. Gegeben eine Eingabe x für M, konstruieren wir einen endlichen Automaten A mit $\mathcal{O}(n^k)$ Zuständen, der jedes Wort akzeptiert, welches **keine** akzeptierende Berechnung von M auf x darstellt. Dann gilt

$$\mathcal{L}(A) = \Sigma^* \quad \text{gdw.} \quad M \text{ die Eingabe } x \text{ nicht akzeptiert.}$$

Bevor wir mit der Konstruktion des endlichen Automaten beginnen, schauen wir uns die Worte an, die eine akzeptierende Berechnung von M auf x darstellen. Sei dazu $n := |x|$. Dann haben akzeptierende Berechnung die folgende Form.

$$\#\alpha_0\#\alpha_1\#\ldots\#\alpha_m\#, \qquad (1)$$

wobei jedes α_i ein Wort der Länge n^k über einem Alphabet $\Delta := \Gamma \cup \Gamma \times Q$ ist. Hier ist Γ die Menge der Bandsymbole und Q die Menge der Zustände von M. Die α_i kodieren die Konfigurationen von M, die während der Berechnung auf x besucht werden.

Des Weiteren muss das Wort noch folgende Eigenschaften erfüllen.

(2) α_0 ist die eindeutige Startkonfiguration von M auf Eingabe x.

(3) α_m ist die eindeutige akzeptierende Konfiguration von M auf Eingabe x.

(4) α_{i+1} ist eine Nachfolger-Konfiguration von α_i, gemäß der Transitionsrelation von M.

Wenn ein Wort keine akzeptierende Berechnung ist, ist es entweder nicht von der Form (1) oder eine der Bedingungen (2),(3),(4) ist verletzt. Der endliche Automat rät zuerst, welche der Bedingungen verletzt ist, und prüft dann seine Behauptung nach.

Nachprüfen von (1):
Der Automat muss hier nachprüfen, ob das Wort **nicht** von der Form $(\#\Delta^{n^k})^*\#$ ist und dass die α_i tatsächlich Konfigurationen darstellen. Zweiteres beinhaltet zum Beispiel, dass die Konfigurationen mit dem Startmarker beginnen und dass genau ein Symbol aus $\Gamma \times Q$ vorkommt.

Für diese Überprüfung benötigt der Automat $\mathcal{O}(n^k)$ viele Zustände.

Nachprüfen von (2) und (3):
Da die initiale und akzeptierende Konfigurationen eindeutig sind, muss der Automat hier bloß nachprüfen, ob das Eingabewort mit genau diesen Worten der Länge n^k beginnt/endet.

Auch für diese Überprüfung benötigt der Automat nur $\mathcal{O}(n^k)$ viele Zustände.

Nachprüfen von (4):
Der Automat muss hier überprüfen, ob die α_i tatsächlich aufeinanderfolgende Konfigurationen darstellen. Aus Ladners Theorem, Theorem 10.6, ist uns bekannt, dass eine Transition nur lokale Änderung auf dem Band verursacht. Genauer kann sich, wenn der Kopf von M in α_i auf das $j$te Symbol gezeigt hat, nur das j. Symbol in α_{i+1} geändert haben (a) und der Kopf muss entweder auf das $(j-1)$., das j. oder das $(j+1)$. Symbol zeigen (b). An allen anderen Stellen in α_{i+1} darf sich, im Vergleich zu α_i, nichts geändert haben (c).

Um zu prüfen, ob (4) gilt, läuft der Automat A durch die Konfigurationen und rät nicht-deterministisch wo eine Verletzung der Eigenschaft auftritt. Um dies nachzuprüfen, merkt sich der Automat das nächste Symbol (um (a) oder (c) zu prüfen) oder die nächsten drei Symbole (für (b)) im Kontrollzustand. Danach überspringt er die nächsten n^k Symbole um zur gleichen Stelle auf dem Band der nächsten Konfiguration zu gelangen. Falls die dort gefundenen Symbole nicht mit den gespeicherten Symbolen und der Transitionsrelation übereinstimmen, gilt (4) nicht und der Automat akzeptiert.

Für diese Überprüfung, benötigt der Automat nur logarithmisch viel Platz (für den Zähler bis n^k).

□

Folgendes Korollar folgt leicht aus Theorem 12.9. Der Beweis ist der Leserin / dem Leser als Übung überlassen.

12.10 Korollar
Equivalence und Inclusion sind PSPACE-vollständig.

13. Hierarchiesätze

Ziel des letzten Kapitels ist es, die *Landkarte der Komplexitätstheorie* aus Kapitel 8 zu vervollständigen, indem wir $\mathsf{NL} \subsetneq \mathsf{PSPACE} \subsetneq \mathsf{EXPSPACE}$, $\mathsf{P} \subsetneq \mathsf{EXP}$ und $\mathsf{NP} \subsetneq \mathsf{NEXP}$ zeigen. Hierzu verwenden wir die **Hierarchiesätze für Zeit und Platz**.

Die Hierarchiesätze sagen, dass wenn man sich auf eine Art von Ressource festlegt (Platzverbrauch, deterministischer Zeitverbrauch oder nicht-deterministischer Zeitverbrauch), dass dann mit echt mehr Ressourcenverbrauch auch echt mehr Probleme gelöst werden können. Die Hierarchiesätze treffen jedoch keine Aussage zu den Relationen zwischen den unterschiedlichen Ressourcen.

Wir beweisen die Platzhierarchiesätze explizit. Bei den Zeithierarchiesätzen beschränken wir uns auf das Nennen des deterministischen Zeithierarchiesatzes.

Um den ersten Platzhierarchiesatz zu formalisieren, benötigen wir zwei neue Notationen. Zum einen benötigen wir die Klasse $o(f)$ der Funktionen, die asymptotisch echt kleiner als eine gegebene Funktion f sind – analog zur Klasse $\mathcal{O}(f)$ die asymptotisch kleiner-gleich f sind.

13.1 Definition
Zu einer Funktion $f\colon \mathbb{N} \to \mathbb{N}$ ist

$$o(f) = \{g\colon \mathbb{N} \to \mathbb{N} \mid \forall m \in \mathbb{N} \exists n_0 \in \mathbb{N}\colon \forall n \geq n_0\colon m \cdot g(n) < \cdot f(n)\}$$

die Klasse der Funktionen, die **asymptotisch echt kleiner als** f sind.

Die Definition von $o(f)$ ersetzt nicht einfach das $\leq$ in der Definition von $\mathcal{O}(f)$ durch $<$. Stattdessen verlangen wir, dass ab einer gewissen Schranke n_0 immer $m \cdot g(n) < f(n)$ gilt – egal mit welcher multiplikativen Konstante m wir g skalieren.

13.2 Beispiel
Wir betrachten $o(n^2)$.

- Die konstante Funktion 1 sowie die Funktionen $\log n, n, n^{1.9}$ sind asymptotisch echt kleiner als n^2; es gilt $1, \log n, n, n^{1.9} \in o(n^2)$.
- Die Funktionen n^2 sowie $\frac{1}{2}n^2$ verhalten sich asymptotisch (bis auf multiplikative Konstanten) genau wie n^2; es gilt $n^2, \frac{1}{2}n^2 \notin o(n^2)$.

- Alle Funktionen, die nicht in $\mathcal{O}(n^2)$ enthalten sind, sind auch nicht in $o(n^2)$ enthalten: $n^3, 2^n \notin o(n^2)$.

Das zweite neue Konzept ist das Konzept der Platzkonstruierbarkeit.

13.3 Definition
Eine Funktion $f\colon \mathbb{N} \to \mathbb{N}$ heißt **platzkonstruierbar**, wenn

- $f(n) \geq \log n$ für alle n und
- es eine Turing-Maschine M read-only Eingabe und write-only Ausgabe gibt, die auf einer Eingabe y der Länge $|y|$ eine Ausgabe der Länge $f(|y|)$ zurückgibt und Platzverbrauch (auf den Arbeitsbändern) in $\mathcal{O}(f)$ hat.

Eine platzkonstruierbare Funktion hat also die Eigenschaft, dass ihre Unärkodierung mit Platzverbrauch $\mathcal{O}(f)$ berechnet werden kann. Wir erinnern uns daran, dass die Unärkodierung einer Zahl n ein Wort der Länge n wie z.B. $a^n = \underbrace{a \dots a}_{n\text{-Mal}}$ ist. Die Abbildungsvorschrift für eine solche Funktion könnte also wie folgt aussehen:

$$a^n = \underbrace{a \dots a}_{n\text{-Mal}} \qquad \mapsto \qquad a^{f(n)} = \underbrace{a \dots a}_{f(n)\text{-Mal}} \ .$$

13.4 Beispiel
Die folgenden Funktionen sind platzkonstruierbar: $\log n, n, n^2, n^3, \dots, 2^n, 2^{n^2}, 2^{n^3}, \dots$

Nun können wir den Platzhierarchiesatz angeben.

13.5 Theorem: Platzhierarchiesatz (deterministische Version)
Es sei $f\colon \mathbb{N} \to \mathbb{N}$ platzkonstruierbar. Dann gilt

$$\mathrm{DSPACE}(o(f)) \subsetneq \mathrm{DSPACE}(\mathcal{O}(f)) \ .$$

Mit Platz f lassen sich also echt mehr Probleme lösen als mit Platz echt kleiner als f. Der Beweis des Resultats kombiniert viele der Techniken, die wir in dieser Vorlesung kennen gelernt haben. Neben den Techniken aus dem Bereich der Komplexitätstheorie verwenden wir auch Diagonalisierung, die Kodierung von Turing-Maschinen und die universelle Turing-Maschine.

Beweis:
Wir zeigen die Existenz einer Sprache $\mathcal{L}(M)$ so dass $\mathcal{L}(M) \in \text{DSPACE}(\mathcal{O}(f)) \setminus \text{DSPACE}(o(f))$. Hierzu konstruieren wir eine DTM M mit Platzverbrauch $\mathcal{O}(f)$, um $\mathcal{L}(M) \in \text{DSPACE}(\mathcal{O}(f))$ zu zeigen. Anschließend zeigen wir $\mathcal{L}(M) \notin \text{DSPACE}(o(f))$ mittels Diagonalisierung.

Es sei $h\colon \{0,1,\#\}^* \to \{0,1\}^*$, der wie folgt definierte Homomorphismus aus Unterkapitel 4.A:

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

Diesen hatten wir eingeführt, um das Symbol # aus Binärstrings zu entfernen.

Die Eingabe für M ist ein Binärstring $y \in \{0,1\}^*$. Die Berechnung von M zu einer solchen Eingabe läuft wie folgt ab:

1. Wir gehen davon aus, dass y von der Form $h(w\#x)$ für Binärstrings $w, x \in \{0,1\}^*$ ist. (Dies kann ohne Platzverbrauch mittels eines endlichen Automaten überprüft werden.) Ansonsten weist M ab.

2. M verwendet (mindestens) zwei Arbeitsbänder. Auf beiden markieren wir $f(|y|)$ viele Zellen, in dem sie mit einem speziellen Symbol $\sqcup'$ belegen.

   Hierzu nutzen wir die Tatsache, dass f nach Annahme platzkonstruierbar ist. Wir können die Turing-Maschine, die die Unärkodierung von $f(|y|)$ berechnet, nutzen, um auf den ersten beiden Arbeitsbändern eine Ausgabe der Länge $f(|y|)$ zu schreiben. Anschließend ersetzen wir alle Symbole dieser Ausgabe durch $\sqcup'$. Das Aufrufen dieser Turing-Maschine benötigt eventuell zusätzliche Arbeitsbändern und $\mathcal{O}(f)$ Platz.

3. Wir erinnern uns daran, dass die Eingabe von der Form $h(w\#x)$ ist. Simuliere die durch w kodierte Turingmaschine M_w auf der Eingabe y (nicht auf x!).

   Hierzu verhält sich M wie die universelle Turing-Maschine M. Als Speicher verwenden wir das erste Arbeitsband.

   Hierzu muss on-the-fly der aktuell benötigte Buchstabe von w aus der Eingabe y extrahiert werden. Dies benötigt höchstens $\log|y|$ Platz zum Speichern einer binär-kodierten Position im Eingabestring. Da $f(n) \geq \log n$ gemäß der Definition von Platzkonstruierbarkeit, benötigt dies Platz in $\mathcal{O}(f)$.

4. Auf dem ersten Arbeitsband sind $f(|y|)$ viele Zellen durch $\sqcup'$ markiert. Wenn die Simulation von M_w mehr als die markierten Zellen verwendet, **weise ab**.

Wir erzwingen so also, dass die Simulation höchstens Platzverbrauch f hat.

5. Auf dem zweiten Arbeitsband speichern wir einen Binärzähler, mit dem wir die Schritte von M_w, die wir simulieren, mitzählen.

   Initial sind $f(|y|)$ viele Zellen markiert. Wenn der Binärzähler mehr als die markierten Zellen verwendet, M_w also $2^{f(|y|)}$ oder mehr Schritte macht, dann **akzeptiere**.

6. Wenn M_w innerhalb der Zeit- und Platzschranken hält, dann hält auch M. Wenn M_w akzeptiert, dann **weise ab**. Wenn M_w abweist, dann **akzeptiere**.

Ein Wort y wird also von M akzeptiert, wenn die folgenden Bedingungen erfüllt sind:

- $y = h(w\#x)$
- Die Simulation von M_w auf y verbraucht $\leq f(|y|)$ Platz,
- M_w auf y benötigt **entweder** $\geq 2^{f(|y|)}$ Schritte,
  oder die M_w hält rechtzeitig und weist ab.

Die Konstruktion von M stellt sicher, dass M Platzverbrauch in $\mathcal{O}(f)$ hat. Damit ist $\mathcal{L}(M) \in \text{DSPACE}(\mathcal{O}(f))$ bewiesen. Um das Theorem zu beweisen, verbleibt es $\mathcal{L}(M) \notin \text{DSPACE}(o(f))$ zeigen.

Wir führen einen Widerspruchsbeweis und nehmen an, dass $\mathcal{L}(M) = \mathcal{L}(N)$, wobei N eine DTM mit einem Arbeitsband und Platzverbrauch $g \in o(f)$ ist. Es sei $w = \langle N \rangle$ die Kodierung von N. Wir betrachten die Eingabe $y = h(w\#x)$ für x beliebig, aber ausreichend lang (siehe unten).

Man beachte, dass M auf Eingabe y die durch w kodierte Turing-Maschine $M_w = N$ simuliert, da $\langle N \rangle = w$. Da $\mathcal{L}(M) = \mathcal{L}(N)$ simuliert M also eine zu sich selbst äquivalente Maschine.

Fall 1: $y \in \mathcal{L}(M)$**.**

Wenn M Eingabe y akzeptiert, dann verbraucht $M_w = N$ entweder zu viel Zeit, oder sie hält rechtzeitig und weist ab.

- Wenn N rechtzeitig hält und abweist, dann gilt $y \notin \mathcal{L}(N)$. Da $\mathcal{L}(M) = \mathcal{L}(N)$ gilt dann auch $y \notin \mathcal{L}(M)$, ein Widerspruch zur Annahme.

- Also macht N mehr als $2^{f(|y|)}$ viele Schritte. Da N nach Annahme allerdings g-platzbeschränkt ist, ist N auch $2^{\mathcal{O}(g)}$-zeitbeschränkt, wie im Beweis von Proposition 7.30. Es muss also

  $$2^{f(|y|)} \leq 2^{\mathcal{O}(g(|y|))}$$

  gelten.

  Für ausreichend große Werte von $|y|$ können wir

  $$2^{\mathcal{O}(g(|y|))} = 2^{m \cdot g(|y|)}$$

  für ein geeignet gewähltes $m \in \mathbb{N}$ schreiben (nach der Definition von $\mathcal{O}(g)$). Da jedoch $g \in o(f)$ so gilt für ausreichend große Werte von $|y|$

  $$m \cdot 2^{g(|y|)} < 2^{f(|y|)} \ ,$$

  egal wie groß m ist (nach der Definition von $o(f)$).

  Dadurch dass wir x, den zweiten Teil der Eingabe, ausreichend lang wählen, können wir $|y|$ so groß machen, dass

  $$2^{f(|y|)} < 2^{m \cdot g(|y|)} < 2^{f(|y|)}$$

  gilt. Wir erhalten $f(|y|) < f(|y|)$, ein Widerspruch.

Für ausreichend lange y führen beide Fälle zum Widerspruch; der Fall $y \in \mathcal{L}(M)$ kann nicht eintreten.

Fall 2: $y \notin \mathcal{L}(M)$**.**

Wenn M Eingabe y nicht akzeptiert, dann verbraucht die Simulation von $M_w = N$ entweder zu viel Platz, oder N hält rechtzeitig und akzeptiert.

- Wenn N rechtzeitig hält und akzeptiert, dann gilt $y \in \mathcal{L}(N)$. Da $\mathcal{L}(M) = \mathcal{L}(N)$ gilt dann auch $y \in \mathcal{L}(M)$, ein Widerspruch zur Annahme.
- Angenommen die Simulation von $M_w = N$ auf y verbraucht mehr als $f(|y|)$ viel Platz.

  Da N g-platzbeschränkt ist, benötigt N auf Eingabe y höchstens $g(|y|)$ viel Platz. Die Simulation von N unter Verwendung der universellen Turing-Maschine erhöht den Platzverbrauch auf bis zu $|w| \cdot g(|y|)$: Die Bandsymbole und Kontrollzustände von N müssen binär kodiert werden und verbrauchen

hierdurch mehr Platz in der Simulation. Der Anstieg des Platzverbrauchs ist aber durch den Faktor $|w|$ limitiert. Da N fixiert ist, ist $|w|$ eine Konstante.

Der tatsächliche Platzverbrauch der Simulation liegt also zwischen $f(|y|)$ und $|w| \cdot g(|y|)$ und es gilt

$$f(|y|) \leq |w| \cdot g(|y|) \ .$$

Da $g \in o(f)$ gilt für $|y|$ groß genug, dass $|w| \cdot g(|y|) < f(|y|)$, egal wie groß $|w|$ ist.

Dadurch dass wir x, den zweiten Teil der Eingabe, ausreichend lang wählen, können wir $|y|$ so groß machen, dass

$$f(|y|) \leq |w| \cdot g(|y|) < f(|y|)$$

ein Widerspruch.

Für ausreichend lange y führt also auch $y \notin \mathcal{L}(M)$ immer zum Widerspruch. Es gibt also keine DTM mit Platzverbrauch in $o(f)$, die $\mathcal{L}(M)$ entscheidet, $\mathcal{L}(M) \notin \mathrm{DSPACE}(o(f))$. □

Der deterministische Platzhierarchiesatz erlaubt es uns direkt, $\mathsf{L} \subseteq \mathsf{PSPACE}$ und einige andere Eigenschaften zu schlussfolgern.

13.6 Korollar

a) Es gilt $\mathsf{L} \subsetneq \mathsf{PSPACE}$.

Denn $\mathsf{L} = \mathrm{DSPACE}(\mathcal{O}(\log n)) \subseteq \mathrm{DSPACE}(o(n)) \subsetneq \mathrm{DSPACE}(\mathcal{O}(n)) \subseteq \mathsf{PSPACE}$. Hier haben wir $\log n \in o(n)$ und den Platzhierarchiesatz für die Funktion n verwendet.

b) Analog gilt $\mathsf{PSPACE} \subsetneq \mathsf{EXPSPACE}$.

Für jedes k gilt $\mathrm{DSPACE}\big(\mathcal{O}(n^k)\big) \subseteq \mathrm{DSPACE}(o(2^n))$, da $n^k \in o(2^n)$. Also $\mathsf{PSPACE} = \bigcup_{k\in\mathbb{N}} \mathrm{DSPACE}\big(\mathcal{O}(n^k)\big) \subseteq \mathrm{DSPACE}(o(2^n)) \subsetneq \mathrm{DSPACE}(\mathcal{O}(2^n)) \subseteq \mathsf{EXPSPACE}$ mit dem Platzhierarchiesatz.

c) Für jedes $k \in \mathbb{N}$ gilt $\mathrm{DSPACE}\big(\mathcal{O}(n^k)\big) \subsetneq \mathrm{DSPACE}\big(\mathcal{O}(n^{k+1})\big)$.

Es gilt $\mathrm{DSPACE}\big(\mathcal{O}(n^k)\big) \subseteq \mathrm{DSPACE}\big(o(n^{k+1})\big) \subsetneq \mathrm{DSPACE}\big(\mathcal{O}(n^{k+1})\big)$.

Es gilt also $\mathrm{DSPACE}(\mathcal{O}(n)) \subsetneq \mathrm{DSPACE}(\mathcal{O}(n^2)) \subsetneq \mathrm{DSPACE}(\mathcal{O}(n^3)) \subsetneq \ldots$ (Die Klassen der Form $\mathrm{DSPACE}\big(\mathcal{O}(n^k)\big)$ sind jedoch nicht robust unter Veränderung des Berechnungsmodels.) Analog gilt $\mathrm{DSPACE}(\mathcal{O}(2^n)) \subsetneq \mathrm{DSPACE}\Big(\mathcal{O}(2^{n^2})\Big) \subsetneq \mathrm{DSPACE}\Big(\mathcal{O}(2^{n^3})\Big) \subsetneq \ldots$

Insbesondere gibt es keine Konstante k mit $\mathsf{PSPACE} = \mathrm{DSPACE}\big(\mathcal{O}(n^k)\big)$ bzw. $\mathsf{EXPSPACE} = \mathrm{DSPACE}\big(2^{\mathcal{O}}(n^k)\big)$.

Um statt $\mathsf{L} \subsetneq \mathsf{PSPACE}$ die stärkere Aussage $\mathsf{NL} \subsetneq \mathsf{PSPACE}$ zu zeigen, benötigen wir den nicht-deterministischen Platzhierarchiesatz. Dieser ergibt sich aus dem deterministischen Hierarchiesatz und Savitch's Theorem.

13.7 Korollar: Platzhierarchiesatz (nicht-deterministische Version)
Es sei $f\colon \mathbb{N} \to \mathbb{N}$ platzkonstruierbar. Dann gilt

$$\mathrm{NSPACE}(o(f)) \subsetneq \mathrm{NSPACE}(\mathcal{O}(f \cdot f)) \ .$$

Beweis:
$\mathrm{NSPACE}(o(f)) \subseteq \mathrm{DSPACE}(o(f \cdot f))$ gilt mit dem Satz von Savitch. $\mathrm{DSPACE}(o(f \cdot f)) \subsetneq \mathrm{DSPACE}(\mathcal{O}(f \cdot f))$ gilt unter Verwendung der deterministischen Versionen des Hierarchiesatzes. $\mathrm{DSPACE}(\mathcal{O}(f \cdot f)) \subseteq \mathrm{NSPACE}(\mathcal{O}(f \cdot f))$ gilt offensichtlich. □

Analog zu $\mathsf{L} \subsetneq \mathsf{PSPACE}$ kann nun unter der Verwendung des nicht-deterministischen Hierarchiesatzes folgendes gezeigt werden.

13.8 Korollar
Es gilt $\mathsf{NL} \subsetneq \mathsf{NPSPACE} = \mathsf{PSPACE}$.

Um die analogen Resultate für die Zeitkomplexitätsklassen zu zeigen, werden die Zeithierarchiesätze benötigt. Wir beschränkten uns hierbei darauf, die formulieren des deterministischen Zeithierarchiesatzes anzugeben.

13.9 Korollar: Zeithierarchiesatz
Es sei f zeitkonstruierbar und g eine Funktion mit $g(n) \cdot \log g(n) \in o(f)$. Dann gilt

$$\mathrm{DTIME}(g) \subsetneq \mathrm{DTIME}(f) \ .$$

Hierbei ist die Definition von *zeitkonstruierbar* analog zur Definition von *platzkonstruierbar*.

Bemerkung
Der Zeithierarchiesatz ist schwächer als der Platzhierarchiesatz. Statt $g \in o(f)$ verlangen wir $g(n) \cdot \log g(n) \in o(f)$. Der Grund hierfür sind die Kosten der Bandreduktion.

Die Maschine M, die wir im Beweis von Theorem 13.5 konstruieren, verwendet mindestens vier verschiedene Arbeitsbänder. Darunter sind neben dem Band, auf dem wir die Schritte mitzählen, die drei Bänder, die die universelle Turing-Maschine verwendet (siehe unser Beweis von Theorem 4.6). Selbst wenn man die Definition optimiert, erhält man eine Maschine M, die mindestens 2 Arbeitsbänder verwendet. Genau genommen zeigen wir im Beweis von Theorem 13.5 also nicht

$$\mathcal{L}(M) \in \text{DSPACE}(\mathcal{O}(f)) \setminus \text{DSPACE}(o(f)) \ ,$$

sondern

$$\mathcal{L}(M) \in \text{DSPACE}_2(\mathcal{O}(f)) \setminus \text{DSPACE}(o(f)) \ .$$

Glücklicherweise lassen sich Mehrband-Maschinen zu 1-Band-Maschinen konvertieren, ohne den Platzverbrauch zu erhöhen – siehe Lemma 7.27. Es gilt also $\text{DSPACE}_2(\mathcal{O}(f)) = \text{DSPACE}(\mathcal{O}(f))$ und unser Beweis ist valide.

Wenn wir den Zeithierarchiesatz beweisen wollen, ist die Lage eine andere. Das Konvertieren einer Mehrband-Maschine zu einer 1-Band-Maschine quadriert den Zeitverbrauch: Um einen Schritt der Mehrband-Maschine zu simulieren, muss die 1-Band-Maschine einmal ihr komplettes Band durchlaufen. Eine Mehrband-Maschine mit Zeitverbrauch $g(n)$, deren Bandlänge automatisch ebenfalls durch $g(n)$ beschränkt ist, wird also zu einer 1-Band-Maschine mit Zeitverbrauch $g(n)^2$. Dementsprechend wurde der Zeithierarchiesatz von Stearns und Hartmanis in 1965 [HS65] mit der noch stärkeren Annahme $g(n)^2 \in o(f)$ gezeigt. Diese Annahme sorgt dafür, dass der Beweis gültig ist, auch nachdem die Bandreduktion angewandt wurde.

Im Jahr 1966 lieferten Hennie und Stearns [HS66] eine verbesserte Konstruktion der Bandreduktion. Mit dieser kann eine 2-Band-Maschine mit Zeitverbrauch $g(n)$ in eine 1-Band-Maschine mit Zeitverbrauch $g(n) \cdot \log g(n)$ umgewandelt werden. Durch Kombination dieser Konstruktion mit dem ursprünglichen Beweis von Hartmanis und Stearns erhält man die oben genannte Version des Zeithierarchiesatzes.

Wir können ähnliche Schlussfolgerungen wie im Fall der Platzkomplexitätsklassen ziehen.

13.10 Korollar

a) $\mathsf{P} \subsetneq \mathsf{EXP}$ gilt unter Verwendung des deterministischen Zeithierarchiesatzes.

b) $\mathsf{NP} \subsetneq \mathsf{NEXP}$ gilt unter Verwendung des nicht-deterministischen Zeithierarchiesatzes (den wir ausgelassen haben).

c) Es gilt $\mathrm{DTIME}(\mathcal{O}(n)) \subsetneq \mathrm{DTIME}(\mathcal{O}(n^3)) \subsetneq \mathrm{DTIME}(\mathcal{O}(n^2)) \subsetneq \ldots$, analog für NTIME und für Exponentialfunktionen.

   Es gibt keine Konstante k mit $\mathsf{P} = \mathrm{DTIME}\big(\mathcal{O}(n^k)\big)$ bzw. $\mathsf{NP} = \mathrm{NTIME}\big(\mathcal{O}(n^k)\big)$ bzw. $\mathsf{EXP} = \mathrm{DTIME}\Big(2^{\mathcal{O}(n^k)}\Big)$ bzw. $\mathsf{NEXP} = \mathrm{NTIME}\Big(2^{\mathcal{O}(n^k)}\Big)$.

Mit den Hierarchiesätzen und ihren Konsequenzen sind alle in unserer Landkarte der Komplexitätstheorie, siehe Kapitel 8, eingezeichneten Relationen bewiesen.

Da nun unter anderem $\mathsf{P} \subsetneq \mathsf{EXP}$ gilt, gibt es in EXP Probleme, die sich nicht in Polynomialzeit lösen lassen. Insbesondere ist dies der Fall für die schwersten Problem in EXP, für die EXP-vollständigen Probleme. Wir haben in dieser Vorlesung bislang kein solches Problem kennen gelernt. Ein Beispiel für ein EXP-vollständiges Problem ist eine verallgemeinerte Variante des Brettspiels Schach, bei der Statt auf einem 8×8-Brett auf einem $n\times n$-Brett (für variables n) gespielt wird. Die sonstigen Regeln von Schach, insbesondere die Arten von Figuren und ihre Züge, bleiben gleich.

CHESS	
Gegeben:	Startbelegung eines $n \times n$ Schachbretts.
Frage:	Hat die Spielerin mit den weißen Figuren eine Gewinnstragie von der gegebenen Startposition aus?

Eine Gewinnstrategie ist eine systematische Art zu spielen, die sicherstellt, dass die Spielerin gewinnt, egal was der Gegner tut. Bei einer Ja-Instanz von CHESS gewinnt die Spielerin mit den weißen Figuren, wenn beide Spieler perfekt spielen. Bei einer Nein-Instanz gewinnt die Spielerin mit den schwarzen Figuren, oder das Ergebnis ist Remis.

Aus $\mathsf{NL} \subsetneq \mathsf{PSPACE} \subsetneq \mathsf{EXPSPACE}$ folgt, dass sich Probleme, die PSPACE-vollständig (bezüglich **logspace-Reduktionen**) sind, nicht mit logarithmischem Platzverbrauch lösen lassen. Dies gilt Beispielsweise für QBF. Hier ist es wichtig, dass wir uns auf logspace-Reduktionen beschränken: Da $\mathsf{NL} \neq \mathsf{P} \neq \mathsf{PSPACE}$ nicht

bewiesen ist, wäre es denkbar, dass eine Polynomialzeitreduktion zu mächtig ist, um die Ungleichheit von NL und PSPACE zu demonstrieren.

Analog lassen sich EXPSPACE-vollständige Probleme (bezüglich logspace- oder Polynomialzeitreduktionen) nicht mit polynomiellen Platz lösen. Als Beispiel verweisen wir hier auf die Vorlesung *Concurrency Theory* (*Nebenläufigkeitstheorie*), in der wir das EXPSPACE-vollständige Überdeckbarkeitsproblem von Petri-Netzen behandeln.

Man beachte, dass die Hierarchiesätze keine Aussage zu den Relationen zwischen den Komplexitätsklassen für unterschiedliche Arten von Ressourcen treffen. Mit dem derzeitigen Stand der Forschung ist es nicht ausgeschlossen, dass sich NP-vollständige Probleme wie z.B. SAT deterministisch in Linearzeit lösen lassen.

Man glaubt, dass sich SAT nicht in Zeit echt kleiner als 2^n lösen lässt, wobei n die Anzahl der Variablen der Eingabeformel ist. Insbesondere glaubt, man, dass sich SAT nicht in Zeit $2^{o(n)}$ (beispielsweise in $2^{\sqrt{n}}$) und auch nicht in Zeit $2^{(1-\epsilon)\cdot n}$ (beispielsweise in $2^{\frac{1}{2}n}$) lösen lässt. Letzteres ist die sogenannte **SETH** (*strong exponential time hypothesis*). Genau wie die Ungleichheit der Komplexitätsklassen $\mathsf{L}, \mathsf{NL}, \mathsf{P}, \mathsf{NP}, \mathsf{PSPACE}, \mathsf{EXP}$ usw. handelt es sich hierbei um eine unbewiesene Aussage.

Das Verhältnis der Klassen von Problemen, die sich mit einem bestimmten Platzverbrauch, deterministischen Zeitverbrauch oder nicht-deterministischen Zeitverbrauch lösen lassen, bleibt bis auf Weiteres das größte ungelöste Problem der Informatik.