Theoretische Informatik 2

Berechenbarkeits- und Komplexitätstheorie

Vorlesungsnotizen

16. Juli 2017

Sebastian Muskalla

Roland Meyer

TU Braunschweig

Sommersemester 2017

Inhaltsverzeichnis

Einführung 4

Motivation 5

Literatur 7

Fixpunkttheorie & Datenflussanalyse 9

I. Entscheidbarkeit & Berechenbarkeit 10

1. Turing-Maschinen & Entscheidbarkeit 11

2. Berechenbarkeit 33

3. Unentscheidbarkeit, universelle Turing-Maschine, Halteproblem & Reduktionen 44

4. Das Postsche Korrespondenzproblem & der Satz von Rice 57

5. Unentscheidbare Probleme kontextfreier Sprachen 66

II. Komplexitätstheorie 70

6. Zeit- und Platzkomplexität 71

7. Eine Landkarte der Komplexitätstheorie 80

8. L und NL 82

9. coNL & der Satz von Immerman und Szelepcsényi 95

10. P 107

11. NP 113

12. PSPACE und der Satz von Savitch 123

13. Der Satz von Savitch 124

14. Hierarchie-Sätze 127

Referenzen 128

Anhang 129

A. Landau-Notation 129

Einführung

Vorwort

Dies sind Notizen zur Vorlesung „Theoretische Informatik 2“, die von uns im Sommersemester 2017 an der TU Braunschweig gehalten wurde.

Hierbei handelt es sich größtenteils um eine geTeXte Version der von Roland Meyer erstellten handschriftlichen Vorlesungsausarbeitung, die jedoch um zusätzliche Anmerkungen ergänzt wurde.

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

Wir bedanken uns bei Peter Chini für das Beitragen der Kapitel 10 und 11.

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

Sebastian Muskalla, Roland Meyer

Braunschweig, 10. Juni 2018

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, ðder 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 $\mathrm{P} \stackrel{?}{=} \mathrm{NP}$, 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 $\mathrm{P} \stackrel{?}{=} \mathrm{NP}$ 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 I“:

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 für Studierende geschrieben sind. 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.

Diese Vorlesung basiert zu großen Teilen auf Vorlesungen, die Roland Meyer in der Vergangenheit ausgearbeitet und gehalten hat.

- Der erste Teil der Vorlesung zu Entscheidbarkeit basiert auf der Vorlesung „Formale Grundlagen der Programmierung“, die von Roland Meyer im Sommer 2016 an der TU Kaiserslautern gehalten wurde.

  Die handschriftliche Vorlesungsausarbeitung und weitere Materialien sind auf der folgenden Website zu finden:
  tcs.cs.tu-bs.de/teaching/FGDP_SS_2016.html

- Der zweite Teil der Vorlesung zur Komplexitätstheorie basiert auf der Vorlesung „Komplexitätstheorie“, die z.B. von Roland Meyer und Prakash Saivasan im Winter 2016/17 an der TU Braunschweig gehalten wurde.

  Die handschriftliche Vorlesungsausarbeitung und weitere Materialien sind auf der folgenden Website zu finden:
  tcs.cs.tu-bs.de/teaching/ComplexityTheory_WS_20162017.html

  Ein Teil der Vorlesungsnotizen wurden von Roland Meyer, Prakash Saivasan, Peter Chini, Judith Stengel und Sebastian Muskalla in ein geTeXtes Skript überführt:
  tcs.cs.tu-bs.de/documents/lecturenotes/complexity.pdf

Des Weiteren verweisen wir als alternative Quelle auf die Vorlesungsnotizen von Prof. Adámek zu „Theoretische Informatik I & II“:
tu-braunschweig.de/Medien-DB/iti/ti-17_02_2016.pdf

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

Fixpunkttheorie & Datenflussanalyse

In einer früheren Version dieser Notizen fanden sich an dieser Stelle die Vorlesungsinhalte zu Fixpunkttheorie und Datenflussanalyse. Mittlerweile wurden diese Themen in die Vorlesung „Theoretische Informatik 1" verschoben, dementsprechend finden sie sich nun auch in den Vorlesungsnotizen zu dieser Vorlesung:

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

Teil I.
Entscheidbarkeit & Berechenbarkeit

TODO: Einführender Text

1. Turing-Maschinen & Entscheidbarkeit

A) Entscheidungsprobleme

Wie in der Einführung erwähnt wurde, wollen wir nun untersuchen, welche Probleme algorithmisch lösbar sind. Zunächst beschränken wir uns auf **Entscheidungsprobleme**, deren Lösung Ja oder Nein ist.

Formal ist ein Entscheidungsproblem eine (typischerweise unendlich große) Menge von Instanzen $\mathcal{I}$, die in die Ja-Instanzen und die Nein-Instanzen unterteilt ist, also $\mathcal{I} = \mathcal{I}_{yes} \uplus \mathcal{I}_{no}$.

Ein Entscheidungsproblem algorithmisch zu lösen bedeutet, mit Hilfe eines Algorithmus für Instanzen $i \in \mathcal{I}$ zu **entscheiden**, ob $i \in \mathcal{I}_{yes}$ gilt.

Entscheidungsprobleme werden oft wie folgt dargestellt.

Entscheidungsproblem zu $\mathcal{I} = \mathcal{I}_{yes} \uplus \mathcal{I}_{no}$	
Gegeben:	Instanz $i \in \mathcal{I}$
Entscheide:	Gilt $i \in \mathcal{I}_{yes}$?

Wort-Probleme sind besondere Entscheidungsprobleme. Sei Σ ein (endliches, nicht-leeres) Alphabet und $\mathcal{L} \subseteq \Sigma^*$ eine (formale) Sprache. Das Wortproblem zur Sprache $\mathcal{L}$ ist wie folgt definiert.

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

Das Wortproblem ist also ein Entscheidungsproblem mit

$$\mathcal{I} = \Sigma^* = \underbrace{\mathcal{L}}_{\mathcal{I}_{yes}} \uplus \underbrace{\left(\Sigma^* \setminus \mathcal{L}\right)}_{\mathcal{I}_{no}},$$

die Wörter in $\mathcal{L}$ sind also die Ja-Instanzen, die Wörter im Komplement von $\mathcal{L}$ die Nein-Instanzen.

Bei den meisten Problemen, für die wir uns interessieren, sind die Instanzen Texte, Zahlen, Formeln, Quellcode von Programmen, oder Ähnliches. All diese Objekte lassen sich als Strings (Wörter) über einem geeigneten Alphabet Σ auffassen, oft z.B. über $\Sigma = \{0, 1\}$.

Dies bedeutet, dass sich (fast) jedes Entscheidungsproblem als Wortproblem für eine bestimmte Sprache $\mathcal{L} \subseteq \Sigma^*$ auffassen lässt. Wir werden uns daher im Folgenden auf das Lösen

von Wortproblemen fokussieren. Wir werden formale Sprachen daher im Folgenden auch als Probleme bezeichnen.

Aus „Theoretische Informatik I“ sind für bestimmte Sprachklassen bereits Entscheidungsalgorithmen für das Wortproblem bekannt:

- Simulation des Automaten für reguläre Sprachen, gegeben durch deterministische endliche Automaten.
- On-the-fly-Determinisierung des Automaten für reguläre Sprachen, gegeben durch nicht-deterministische endliche Automaten.
- CYK-Algorithmus für kontextfreie Sprachen, gegeben durch kontextfreie Grammatiken in CNF.

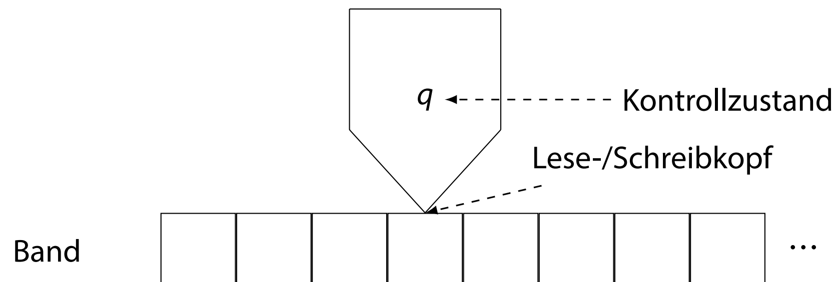
Diese Sprachklassen sind allerdings sehr eingeschränkt. Nun wollen wir herausfinden, was es im Allgemeinen bedeutet, das Wortproblem für eine Sprache algorithmisch zu lösen.

Hierbei wollen wir ein geeignetes Automatenmodell für allgemeine Algorithmen finden, nämlich Turing-Maschinen.

B) 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.

1.1 Definition
Eine **(deterministische) Turing-Maschine, kurz DTM oder nur TM**, M ist ein Tupel

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

mit

- Q ist eine endliche Menge von **Kontrollzuständen**,
  - $q_0 \in Q$ ist der **Start- oder Initialzustand**,
  - $q_{acc} \in Q$ ist der **akzeptiertende Zustand**,

  - $q_{rej} \in Q$ ist der **abweisende Zustand**,
  - Zusammen nennt man q_{acc}, q_{rej} die **Haltezustände**, es gilt $q_{acc} \neq q_{rej}$,
- Σ ist das endliche, nicht-leere **Eingabealphabet**,
- Γ ist das endliche, nicht-leere **Bandalphabet**,
  - $\Sigma \subseteq \Gamma$,
  - $\$ \in \Gamma$ ist der **(linke) Endmarker**,
  - $\sqcup \in \Gamma$ ist das **Leerzeichen** oder **Blank**-Symbol,
  - es gilt $\$ \neq \sqcup, \$ \notin \Sigma, \sqcup \notin \Sigma$,
- δ ist die (deterministische) Transitionsfunktion

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

  welche die folgenden beiden Zusatzanforderungen erfüllt, die wir später genauer erläutern werden:

  (1)
  $$\forall q \in Q\ \exists q' \in Q\colon \delta(q, \$) = (q', \$, R) ,$$

  (2)
  $$\forall a \in \Gamma\ \exists d, d' \in \{L, R\}\colon \delta(q_{acc}, a) = (q_{acc}, a, d) \text{ und } \delta(q_{rej}, a) = (q_{rej}, a, d') .$$

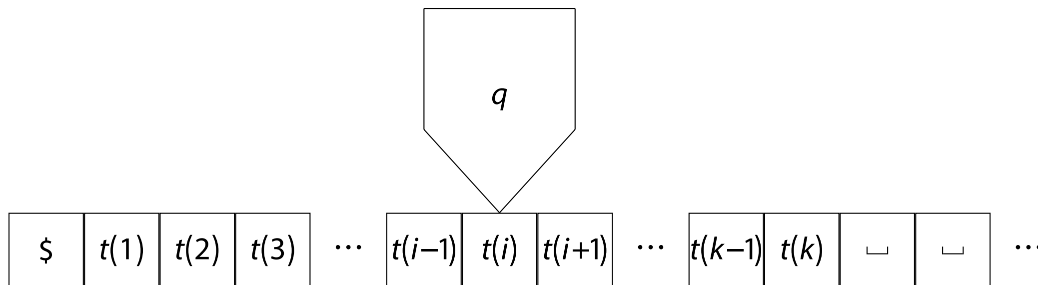
Man beachte, dass wir die Zustände q_{acc} und q_{rej} und die Symbole $\$$ und $\sqcup$ fixiert haben. Der Startzustand ist im Tupel angegeben, sein Name kann somit variiert werden.

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 die erste Zelle immer mit dem Endmarker \$ gefüllt ist, d.h. $t(0) = \$$. Des Weiteren ist nur ein endliches Anfangsstück des Bandes beschrieben, der Rest des Bandes ist mit Blanks gefüllt: $\exists k\colon \forall n > k : t(n) = \sqcup$.



Sei nun

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

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

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

im nächsten Schritt die folgenden drei Operationen ausführt:

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

Mit dieser Intuition können wir nun die beiden Zusatzanforderungen an δ aus der Definition von Turing-Maschinen verstehen:

(1) bedeutet, dass der Endmarker \$ nie durch ein anderes Symbol überschrieben wird. Zudem darf die Maschine, wenn sie den Endmarker sieht, nur nach rechts gehen.

(2) bedeutet, dass die Maschine, sobald sie in einem haltenden Zustand ankommt, den Zustand und den Bandinhalt nicht mehr ändert.

Wir formalisieren nun die intuitive Definition von Konfigurationen und Berechnungen.

1.2 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 Bandinhalt links vom Kopf ist, q der aktuelle Kontrollzustand und v der 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 $v = v\sqcup = v\sqcup\sqcup = \ldots = v\sqcup^{\omega}$.

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

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

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

Man sieht, dass die Transitionsrelation $\rightarrow$ genau die zuvor beschriebenen Schritte umsetzt. Da δ eine deterministische Transitionsfunktion ist, ist auch die Transitionsrelation auf Konfigurationen deterministisch, d.h. jede Konfiguration hat einen eindeutigen Nachfolger bezüglich $\rightarrow$.

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

Dies bedeutet, dass sich die Maschine im Startzustand befindet, das Band mit dem Endmarker, der Eingabe, und danach mit Blanks gefüllt ist, und der Kopf auf den Endmarker zeigt.

d) Eine Konfiguration $u\, q\, v$ heißt

- **akzeptierend**, falls $q = q_{acc}$,
- **abweisend**, falls $q = q_{rej}$,
- **haltend**, falls $q \in \{q_{acc}, q_{rej}\}$.

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

$$c_0 = \varepsilon\, q_0\, \$w \rightarrow c_1 \rightarrow c_2 \rightarrow c_3 \rightarrow \ldots$$

die sich von der Startkonfiguration zu w aus ergibt, in dem man in jedem Schritt den (eindeutigen) Nachfolger bezüglich der Transitionsrelation $\rightarrow$ nimmt.

Oftmals interessiert man sich nur für einen endlichen Präfix $c_0 \to \ldots \to c_k$ einer Berechnung.

f) Eine Berechnung heißt **akzeptierend**, falls sie nach endlich vielen Schritten eine akzeptierende Konfiguration erreicht, d.h. $\exists i \in \mathbb{N}$: $c_i = u\, q_{acc}\, v$.

Analog nennt man die Berechnung **abweisend** bzw. **haltend**, falls sie nach endlich vielen Schritten eine abweisende bzw. haltende Konfiguration erreicht.

Man beachte, dass gemäß Zusatzanforderung (2) die Maschine Kontrollzustand und Bandinhalt nicht mehr ändern darf, sobald sie in einer haltenden Konfiguration ankommt. Daher kann man den Rest einer haltende Berechnung abschneiden, sobald ein Haltezustand erreicht ist, ohne Informationen zu verlieren. Man interessiert sich also nur für das kleinste Präfix $c_0 \to \ldots \to c_i$ der Berechnung, so dass c_i haltend ist.

g) Maschine M **akzeptiert Eingabe** $w \in \Sigma^*$, wenn die Berechnung von M zu w akzeptierend ist.

h) Die Sprache $\mathcal{L}(M)$ der Maschine M ist die Menge aller akzeptierten Wörter,

$$\mathcal{L}(M) = \left\{w \in \Sigma^* \;\middle|\; M \text{ akzeptiert } w\right\} = \left\{w \in \Sigma^* \;\middle|\; \varepsilon\, q_0\, \$w \to^* u\, q_{acc}\, v\right\}.$$

1.3 Definition
Eine Sprache $\mathcal{L} \subseteq \Sigma^*$ heißt **semi-entscheidbar** oder **rekursiv aufzählbar** oder **(Turing) erkennbar**, wenn es eine Turing-Maschine $M = (Q, \Sigma, \Gamma, \delta, q_0)$ mit $\mathcal{L} = \mathcal{L}(M)$ gibt.

Es mag verwirrend sein, dass wir hier drei verschiedene Begriffe einführen, die alle das Selbe bezeichnen. Dies liegt daran, dass im Laufe der Zeit das Problem aus unterschiedlichen Blickwinkeln betrachtet wurde. Da alle Begriffe geläufig sind und in der Literatur verwendet werden, möchten wir sie auch alle hier nennen.

Wir werden hauptsächlich *semi-entscheidbar* verwenden. Im nächsten Kapitel werden wir sehen, dass *rekursiv aufzählbar* tatsächlich bedeutet, dass man die Sprache mit einem Algorithmus aufzählen kann.

Man sollte sich nun (mindestens) drei Fragen stellen:

1. Sind alle Sprachen semi-entscheidbar? Wenn nein, welche sind es nicht?
2. Warum heißt es bloß *semi-entscheidbar*, also halb-entscheidbar? Was ist der Unterschied zu *entscheidbar*?
3. Wir haben bei der Definition von Turing-Maschinen diverse, willkürlich anmutende Entscheidungen getroffen:

- Wir haben nur ein Band,
- dieses Band ist nur nach Rechts unendlich, und
- Turing-Maschinen sind deterministisch.

In wie fern ändert sich die Klasse der semi-entscheidbaren Sprachen, wenn wir die Definition verändern?

Zur 1. Frage:

Es gibt nicht-semi-entscheidbare Sprachen. Um genau zu sein, zeigen wir im Beweis des folgenden Satzes, dass die Klasse der semi-entscheidbaren Sprachen im Vergleich zu allen Sprachen winzig klein ist.

1.4 Theorem
Sei Σ ein Alphabet. Es gibt Sprachen über Σ, die nicht semi-entscheidbar sind.

Beweis:
Um die Aussage zu beweisen, wollen wir wie folgt verfahren: Wir zählen alle Sprachen über Σ und wir zählen die semi-entscheidbaren Sprachen über Σ und vergleichen.

Behauptung: Es gibt überabzählbar viele Sprachen.

Zunächst beweisen wir, dass es überabzählbar viele Sprachen über Σ gibt. Hierzu wollen wir ausnutzen, dass die Potenzmenge der natürlichen Zahlen $\mathcal{P}(\mathbb{N}) = \{M \mid M \subseteq \mathbb{N}\}$ als überabzählbar bekannt ist.

Wir fixieren ein beliebiges Symbol $a \in \Sigma$ und definieren eine Abbildung c, die einer natürlichen Zahl eine sogenannte unäre Codierung als Wort in Σ^* zuordnet.

$$\begin{array}{rcl} c : \mathbb{N} & \to & \Sigma^* \\ n & \mapsto & a^n = \underbrace{a \ldots a}_{n \text{ Mal}} \end{array}$$

Diese Abbildung ist injektiv, d.h. aus $n \neq m$ folgt $c(n) \neq c(m)$, womit bewiesen ist, dass Σ^* mindestens so groß ist wie $\mathbb{N}$. Dementsprechend ist auch die Menge aller Sprachen über Σ, also die Potenzmenge $\mathcal{P}(\Sigma^*) = \{\mathcal{L} \mid \mathcal{L} \subseteq \Sigma^*\}$ mindestens genauso groß wie die Potenzmenge von $\mathbb{N}$. Wir können dies explizit beweisen, in dem wir die Abbildung c wie folgt liften:

$$\begin{array}{rcl} c' : \mathcal{P}(\mathbb{N}) & \to & \mathcal{P}(\Sigma^*) \\ M & \mapsto & \{c(n) \mid n \in M\} . \end{array}$$

Da c injektiv war, ist auch c' injektiv, dementsprechend gilt $|\mathcal{P}(\mathbb{N})| \leqslant \left|\mathcal{P}(\Sigma^*)\right|$. Da die Potenzmenge der natürlichen Zahlen bereits überabzählbar ist, ist auch die Menge aller Sprachen überabzählbar. (Man könnte sogar $|\mathcal{P}(\mathbb{N})| = \left|\mathcal{P}(\Sigma^*)\right| = |\mathbb{R}|$ beweisen.)

Behauptung: Es gibt nur abzählbar viele semi-entscheidbare Sprachen.

Hierzu wollen wir zunächst die Menge aller Turing-Maschinen abzählen, da sich jede semi-entscheidbare Sprache durch eine Turing-Maschine definieren lässt.

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

$$Q = \{q_0, q_1, \ldots, q_k, q_{acc}, q_{rej}\}$$

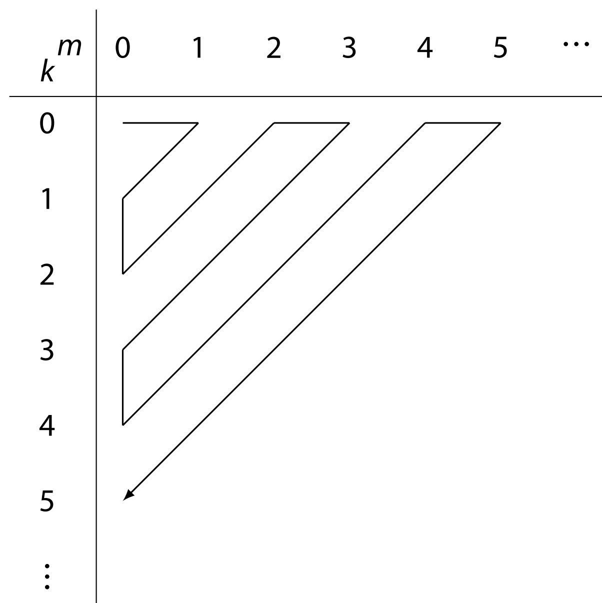
und

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

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

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

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



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

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

Wir haben nun bewiesen, dass die Menge der Turing-Maschinen abzählbar ist, es gibt nun also eine Abzählung $M_0, M_1, M_2, \ldots$, in der jede Turing-Maschine vorkommt. Betrachte nun die Abzählung $\mathcal{L}(M_0), \mathcal{L}(M_1), \mathcal{L}(M_2), \ldots$ der entsprechenden Sprachen. Diese Abzählung beinhaltet alle semi-entscheidbaren Sprachen, da jede solche Sprache die Sprache einer Turing-Maschine ist. Eventuell kann es sein, dass für Indizes $i \neq j$ die Sprachen $\mathcal{L}(M_i) = \mathcal{L}(M_j)$ gleich sind, dies stört uns allerdings nicht weiter.

Wir haben nun bewiesen, dass die Menge der semi-entscheidbaren Sprachen abzählbar ist.

Konklusion: Die Menge der Sprachen ist überabzählbar, die Menge der semi-entscheidbaren Sprachen nur abzählbar. Damit ist die Menge der nicht-semi-entscheidbaren Sprachen auch überabzählbar, insbesondere gibt es also nicht-semi-entscheidbare Sprachen. □

Dieser Beweis stellt uns nicht wirklich zufrieden, denn er ist nicht konstruktiv. Wir haben zwar bewiesen, dass es nicht-semi-entscheidbare Sprachen geben muss, allerdings immer noch kein konkretes Beispiel zur Hand. Tatsächlich sind die meisten Sprachen, die einem als Mensch in den Sinn kommen, semi-entscheidbar. Wir werden uns noch etwas gedulden müssen, bis wir für eine konkrete Sprache beweisen werden, dass sie nicht semi-entscheidbar ist.

Zur 2. Frage:

Die zweite Frage führt zum Begriff der Entscheidbarkeit.

C) Entscheidbarkeit

Wenn wir eine allgemeine Turing-Maschine M – im Folgenden nennen wir solche Turing-Maschinen auch **Semi-Entscheider** – auf einer Eingabe w simulieren, gibt es drei Möglichkeiten:

1. M akzeptiert w, d.h. die Berechnung von M auf Eingabe w erreicht nach endlich vielen Schritten eine akzeptierende Konfiguration.

2. M weist w ab, d.h. die Berechnung von M auf Eingabe w erreicht nach endlich vielen Schritten eine abweisende Konfiguration.

3. M hält nicht für Eingabe w, d.h. die unendliche Berechnung von M auf w erreicht niemals eine haltende Konfiguration.

Der letzte Fall wird umgangssprachlich als Endlosschleife bezeichnet. Nur im ersten Fall gilt $w \in \mathcal{L}(M)$, in den anderen beiden Fällen $w \notin \mathcal{L}(M)$.

Wenn wir eine Eingabe w haben, für die wir bereits wissen, dass $w \in \mathcal{L}(M)$ gilt, können wir dies verifizieren, indem wir M auf Eingabe w simulieren: Die Berechnung wird nach endlich vielen Schritten akzeptieren. Wenn wir eine Eingabe $w \notin \mathcal{L}(M)$ haben und M simulieren, kann M entweder abweisen oder unendlich lange laufen.

Für eine Eingabe $w \in \Sigma^*$, über die wir noch nichts wissen, lässt sich $w \in \mathcal{L}(M)$ nicht durch Simulation entscheiden: Falls M nach endlich vielen Schritten anhält, wissen wir ob $w \in \mathcal{L}(M)$ 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.

Das Problem am dritten Fall von oben ist also, dass er *schwer* von den anderen beiden zu unterscheiden ist.

Eine Turing-Maschine ist also zunächst bloß ein Semi-Entscheider, also ein Algorithmus, mit dem man die Ja-Instanzen in endlicher Zeit verifizieren kann, allerdings für beliebige Instanzen eventuell unendlich lange simulieren müsste.

Um dieses Problem zu lösen, schließen wir den dritten Fall oben per Definition aus.

1.5 Definition
Eine Turing-Maschine heißt **haltend**, **total** oder ein **Entscheider**, wenn sie auf jeder Eingabe nach endlich vielen Schritten hält.

1.6 Definition
Eine Sprache $\mathcal{L} \subseteq \Sigma^*$ heißt **entscheidbar** oder **rekursiv**, wenn es einen Entscheider M mit $\mathcal{L} = \mathcal{L}(M)$ gibt.

1.7 Bemerkung
Auch hier haben wir wieder verschiedene Begriffe, die das Selbe bedeuten. Wir werden *entscheidbar* verwenden, der Begriff *rekursiv* kommt von einer anderen Herangehensweise an den Vorlesungsinhalt.

Wir erklären nun noch, warum man einen Entscheider auch eine **totale** Turing-Maschine nennt. Eine Turing-Maschine M definiert eine partielle Funktion, also eine Funktion, die nicht zu jedem Wert auch einen Funktionswert hat.

$$\begin{aligned} \text{sim}_M \ : \ \Sigma^* \ &\to_p \ \{0,1\} \mathrel{\widehat{=}} \mathbb{B} = \{\mathit{false}, \mathit{true}\} \\ w \ &\mapsto \begin{cases} 0, & \text{falls } M \text{ Eingabe } w \text{ abweist,} \\ 1, & \text{falls } M \text{ Eingabe } w \text{ akzeptiert,} \\ \text{undefiniert}, & \text{falls } M \text{ auf Eingabe } w \text{ nicht hält.} \end{cases} \end{aligned}$$

Wenn M ein Entscheider ist, tritt der letzte Fall nicht auf, daher ist sim_M in diesem Fall eine **totale Funktion**. Um genau zu sein, ist sim_M dann die totale charakteristische Funktion $\chi_{\mathcal{L}(M)}$ von $\mathcal{L}(M)$,

$$\begin{aligned} \chi_{\mathcal{L}(M)} \ : \ \Sigma^* \ &\to \ \{0,1\} \mathrel{\widehat{=}} \mathbb{B} = \{\mathit{false}, \mathit{true}\} \\ w \ &\mapsto \begin{cases} 0, & \text{falls } w \notin \mathcal{L}(M), \\ 1, & \text{falls } w \in \mathcal{L}(M). \end{cases} \end{aligned}$$

Der Begriff der Entscheidbarkeit wirft neue Fragen auf:

- Ist jede semi-entscheidbare Sprache auch entscheidbar?

  Dies ist nicht der Fall, aber auch hier müssen wir uns noch gedulden, um Beispiele zu sehen.

- Wie kann man für eine gegebene Turing-Maschine feststellen, ob sie ein Entscheider ist?

Letzteres Problem ist das sogenannte **Totalitätsproblem**.

Totalitätsproblem

Gegeben: Turing-Maschine M

Entscheide: Ist M ein Entscheider?

Damit wir dieses Problem mit unseren Methoden untersuchen können, müssen wir es zunächst als Wortproblem auffassen, dass heißt wir müssen eine Turing-Maschine als Wort in Σ^*, für ein geeignetes Alphabet Σ, auffassen.

Später werden wir dies tun und feststellen, dass das Totalitätsproblem nicht einmal semi-entscheidbar ist. Teilweise wird das Totalitätsproblem in der Literatur auch *Halteproblem* genannt; Wir werden mit *Halteproblem* jedoch ein anderes Problem bezeichnen.

Zur 3. Frage:

Die dritte Frage von oben fragt danach, ob die Klasse der (semi-)entscheidbaren Sprachen **robust** gegenüber Veränderungen an der Definition von Turing-Maschinen ist. Wir haben bereits die Church-Turing-These genannt, welche vermutet, dass alle Berechnungsmodelle höchstens so mächtig wie Turing-Maschinen sind. Daher erwarten wir, dass sich alle Varianten von Turing-Maschinen durch Turing-Maschinen, wie sie oben definiert sind, simulieren lassen.

D) Varianten von Turing-Maschinen

Turing-Maschinen mit beidseitig unendlichem Band

1.8 Definition
Turing-Maschinen mit **beidseitig unendlichem Band** sind analog zu Turing-Maschinen definiert, allerdings entfällt der Endmarker \$ und auch die Zusatzanforderung (1). Wir nehmen an, dass zu einer Eingabe *w* die Initialkonfiguration von der Form

$$\dots \sqcup\sqcup\sqcup\sqcup w_0 \dots w_k \sqcup\sqcup\sqcup\sqcup \dots$$

ist, wobei der Kopf auf dem ersten Symbol der Eingabe ist, und die Maschine auf dem Band sowohl nach rechts als auch nach links unbegrenzt laufen kann.

1.9 Theorem
Zu jeder TM $M_\leftrightarrow$ mit beidseitig unendlichem Band gibt es eine TM M, die $M_\leftrightarrow$ effizient simuliert. Insbesondere gilt $\mathcal{L}(M_\leftrightarrow) = \mathcal{L}(M)$. Wenn $M_\leftrightarrow$ auf jeder Eingabe nach endlich vielen Schritten hält, dann ist auch M ein Entscheider.

Beweis: Übungsaufgabe. □

1.10 Korollar
Die Klasse der von Turing-Maschinen (semi-)entscheidbaren Sprachen ist gleich der Klasse der von Turing-Maschinen mit beidseitig unendlichem Band (semi-)entscheidbaren Sprachen.

1.11 Bemerkung
Im obigen Theorem haben wir gesagt, dass M die Maschine $M_\leftrightarrow$ **effizient** simuliert. Effizient bedeutet hierbei, dass M auf jeder Eingabe höchstens polynomiell mehr Platz und Zeit als $M_\leftrightarrow$ braucht, bis sie akzeptiert oder abweist. Derzeit ist dies nicht relevant, wir werden hierauf zurückkommen und die Definition von *effizient* präzisieren, wenn wir uns mit Komplexitätstheorie beschäftigen.

Mehr-Band-Turing-Maschinen

1.12 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

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

d.h. 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. Wenn die letzte Komponente $d = S$ ist, bedeutet dies, dass wir den entsprechenden Kopf nicht bewegen (S wie *stay*).

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

Wir nehmen an, dass die beiden Zusatzanforderungen so angepasst werden, dass der Endmarker auf allen Bändern respektiert wird, und dass kein Band mehr verändert wird, sobald ein Haltezustand erreicht wurde.

1.13 Theorem: Bandreduktion

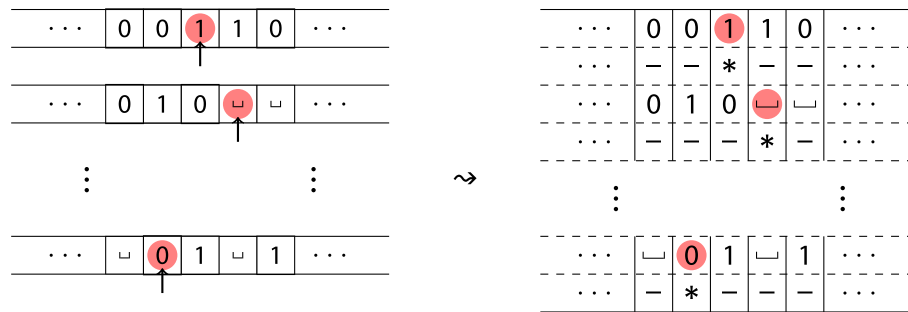
Zu jeder k-Band-Turing-Maschine M_k gibt es eine Turing-Maschine M, die M_k effizient simuliert. Insbesondere gilt $\mathcal{L}(M_k) = \mathcal{L}(M)$. Wenn M_k auf jeder Eingabe nach endlich vielen Schritten hält, dann ist auch M ein Entscheider.

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 Endmarker \$. M bewegt ihren Kopf nach rechts über das Band, bis sie das erste Symbol ␣ findet. Hiermit ist wirklich das Symbol $␣ \in \Gamma'$ gemeint, nicht ein Vektor, der als Einträge ␣ hat.

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 beim Endmarker angekommen ist, wechselt M in den entsprechenden Kontrollzustand von M_k. □

1.14 Korollar
Die Klasse der von Turing-Maschinen (semi-)entscheidbaren Sprachen ist gleich der Klasse der von Turing-Maschinen mit $k > 0$ Bändern (semi-)entscheidbaren Sprachen.

Alphabetsreduktion

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

1.15 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 $M_{\text{bin}} = (Q', \{0, 1\}, \{0, 1, \$, ␣\}, \delta', q_0')$ 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.

□

E) Nichtdeterminismus

Nichtdeterminismus ist deutlich komplizierter als die anderen Variationen. In „Theoretische Informatik I" haben wir mit Hilfe der Potenzmengenkonstruktion gesehen, dass deterministische endliche Automaten und nicht-deterministische endliche Automaten gleich mächtig sind. Hierbei gab es allerdings einen *Blowup*: Im Worst-Case hat der durch die Potenzmengenkonstruktion konstruierte DFA exponentiell mehr Zustände als der zugrunde liegende NFA.

Für Pushdown-Automaten kann man zeigen, dass Nichtdeterminismus echt mächtiger ist als Determinismus. Beispielsweise lässt sich die Sprache der Palindrome gerade Länge

$$\mathcal{L}_{Palin\ even} = \left\{ w.w^{reverse} \mid w \in \{a, b\}^* \right\}$$

nicht von einem deterministischen Pushdown-Automaten akzeptierten, obwohl sie kontextfrei ist.

Im folgenden werden wir sehen, dass sich nicht-deterministische Turing-Maschinen zwar durch DTMs simulieren lassen, allerdings führt diese Simulation zu einem Blowup. Während sich bei endlichen Automaten der Blowup nur auf die Anzahl der Zustände bezog, aber der

DFA trotzdem nur einen Schritt pro Buchstaben des Eingabewortes benötigt, führt der Blowup bei Turing-Maschinen zu einem exponentiellen Anstieg der Berechnungsschritte.

1.16 Definition

Nicht-deterministische Turing-Maschinen (NTM) sind analog zu DTMs definiert, allerdings ist δ nun eine nicht-deterministische Transitionsfunktion

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

die einem Zustand q und einem Symbol a eine Menge $\delta(q, a)$ von möglichen Aktionen (p, b, d) zuordnet.

Dementsprechend hat eine Konfiguration $u\ q\ a.v$ nun nicht mehr einen eindeutigen Nachfolger gemäß der Transitionsrelation auf Konfigurationen, sondern eine Menge von Nachfolgern, einen pro Element von $\delta(q, a)$.

Es gibt nun zu einer Eingabe w nicht mehr eine eindeutige Berechnung, sondern potentiell unendlich viele unterschiedliche Berechnungen.

Da wir $\delta(q, a) = \varnothing$ erlauben, kann es vorkommen, dass eine Konfiguration keinen Nachfolger in der Transitionsrelation hat. Eine Berechnung, die eine solche Konfiguration beinhaltet, bleibt nach endlich vielen Schritten stecken. Wir fassen diesen Fall als eine Form von abweisen auf, d.h. wir nennen jetzt eine Berechnung abweisend, wenn sie in einer Konfiguration mit Kontrollzustand q_{rej} ankommt, oder wenn sie in einer Konfiguration ohne Nachfolger ankommt.

Die Sprache einer nicht-deterministischen Maschine ist die Menge aller Wörter, zu denen es eine akzeptierende Berechnung gibt,

$$\mathcal{L}(M_{NTM}) = \left\{ w \in \Sigma^* \;\middle|\; \varepsilon\ q_0\ \$w \to^* u\ q_{acc}\ v \right\} .$$

Beachte, das dies dieselbe Definition wie zuvor ist, bloß haben Konfigurationen nun gemäß $\to$ keinen eindeutigen Nachfolger mehr.

Wir nennen eine nicht-deterministischen Turing-Maschine total, haltend oder einen Entscheider, wenn zu jeder Eingabe *alle* Berechnungen anhalten.

Selbst wenn M_{NTM} ein Entscheider ist, genügt es nun nicht mehr, M_{NTM} auf einer Eingabe w für endlich viele Schritte zu simulieren, um zu entscheiden, ob $w \in \mathcal{L}(M_{NTM})$ gilt. Eventuell erhalten wir eine abweisende Berechnung aufgrund der Entscheidungen, die wir beim Simulieren getroffen haben um den Nichtdeterminismus aufzulösen, obwohl auch eine akzeptierende Berechnung existiert hätte. Wir müssen also alle Berechnungen von M_{NTM} zur Eingabe simulieren.

Hierzu definieren wir Berechnungsbäume.

1.17 Definition
Sei M_{NTM} eine NTM und $w \in \Sigma^*$ eine Eingabe. Der **Berechnungsbaum** von M_{NTM} 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 $u\; q\; a.v$, die nicht-haltend ist (d.h. $q \notin \{q_{acc}, q_{rej}\}$), markiert ist, hat der Baum einen Kindknoten pro Element von $\delta(q, a)$, der mit der entsprechenden resultierenden Konfiguration markiert ist.

Die von der Wurzel ausgehenden Pfade im Berechnungsbaum von M_{NTM} zu w entsprechen den Berechnungen von M_{NTM} zu Eingabe w. Wenn die Berechnung nach endlich vielen Schritten hält (akzeptiert oder abweist, wobei letzteres auch bedeuten kann, dass sie stecken bleibt), ist der entsprechende Pfad endlich, unendliche Pfade entsprechen nicht-haltenden Berechnungen. Wenn M_{NTM} ein Entscheider ist, ist ihr Berechnungsbaum zu jeder Eingabe w endlich.

1.18 Theorem
Zu jeder NTM M_{NTM} gibt es eine DTM M mit $\mathcal{L}(M_{NTM}) = \mathcal{L}(M)$. Wenn M_{NTM} ein Entscheider ist, dann ist auch M ein Entscheider.

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_{NTM} 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_{NTM}, dies ist allerdings nicht der Fall.

Ein besserer Ansatz ist es, M so zu konstruieren, dass sie zu einer Eingabe w den Berechnungsbaum von M_{NTM} zu w durchsucht. Hier gibt es nun (mindestens) zwei Möglichkeiten: Tiefensuche oder Breitensuche. Tiefensuche wird nicht das gewünschte Resultat liefern, falls M_{NTM} kein Entscheider ist. In diesem Fall gibt es nämlich zu einer Eingabe w eventuell nicht-haltende Berechnungen, und damit unendliche Pfade im Baum, in denen sich die Tiefensuche verliert. Falls es gleichzeitig auch eine endliche akzeptierende Berechnung zu w gäbe, würde wir diese mit Tiefensuche also eventuell nicht finden.

Unsere Simulation wird also eine Breitensuche im Berechnungsbaum implementieren.

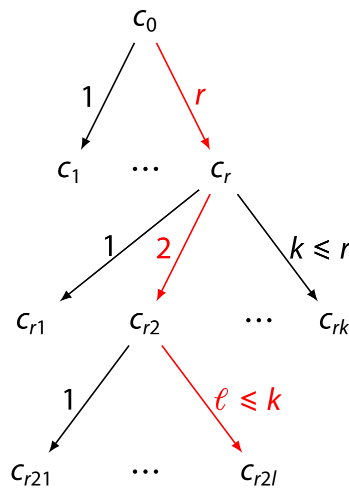
Beweis des Theorems:
Zu jeder Konfiguration von M_{NTM} gibt es zwar eventuell mehrere, aber in jedem Fall nur endlich viele Nachfolger. Die Zahl der Nachfolger ist beschränkt durch

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

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. M verwendet 3 Bänder, Theorem 1.13 kann zur Reduktion auf ein Band verwendet werden.

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

   - Sequenzen werden mit aufsteigender Länge generiert.

- Innerhalb der selben 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 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_{NTM} 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 im i-ten Schritt den s_i-ten Nachfolger auswählen.
- Falls bei der Simulation eine akzeptierende Konfiguration von M_{NTM} angetroffen wird, akzeptiere.
- Falls
  (1) eine abweisende Konfiguration, oder
  (2) eine Konfiguration ohne Nachfolger angetroffen wird, oder
  (3) sich herausstellt, dass der Sequenz s keine valide Berechnung entspricht, oder
  (4) falls die Berechnung nicht innerhalb von $|s|$ Schritten hält, breche die Simulation ab und gehe zur nächsten Sequenz.

Zusätzlich speichern wir im Kontrollzustand ein Bit c, das wir nutzen, um zu überprüfen, ob die Fälle (1) - (3) (abweisen, stecken bleiben, nicht-valide Berechnung) für alle Sequenzen einer bestimmten Länge eingetreten sind. Formal starten wir für jede Länge $|s|$ von Sequenzen mit dem Bit $c = \mathit{false}$, und setzen c auf true, sobald wir mindestens eine Berechnung gefunden haben, die nicht in obigem Szenario endet. Wenn nach der Simulation für alle Sequenzen der Länge $|s|$ immer noch $c = \mathit{false}$ gilt, weist M die Eingabe ab.

Wir argumentieren nun, dass $\mathcal{L}(M_{NTM}) = \mathcal{L}(M)$ gilt.

Sei $w \in \mathcal{L}(M_{NTM})$ eine von M_{NTM} akzeptierte Eingabe. Dann gibt es eine Berechnung von M_{NTM} zu w, die nach endlich vielen Schritten akzeptiert. Sei s die Sequenz, die den entsprechenden

Pfad im Berechnungsbaum kodiert. Wenn die DTM *M* mit dieser Sequenz *s* simuliert, wird *M* akzeptieren. Eventuell akzeptiert *M* bereits früher, wenn es noch eine kürzere akzeptierende Berechnung gibt. Der Fall, dass *M* abbricht, wird nie eintreten, da zu jeder Länge *k* der entsprechend lange Präfix der Sequenz *s* nicht in einen der Fehlerfälle (1) - (3) führt.

Sei $w \notin \mathcal{L}(M_{NTM})$ ein von M_{NTM} nicht-akzeptierte Eingabe. Dann wird *M* nicht akzeptieren, egal welche Sequenz *s* wir zur Simulation verwenden. (Angenommen es gäbe eine Sequenz *s* für die *M* akzeptiert, dann würde $w \in \mathcal{L}(M_{NTM})$ gelten, ein Widerspruch.)

Um den Beweis abzuschließen, erinnern wir uns daran, dass falls M_{NTM} ein Entscheider ist, der Berechnungsbaum zu jeder Eingabe endlich ist. In diesem Fall wird *M* entweder nach endlich vielen Schritten akzeptieren, oder die Simulation abbrechen: Wenn der Berechnungsbaum Höhe *k* hat, dann wird *M* für Sequenzen der Länge $k + 1$ feststellen, dass alle Berechnungen bereits vorher abweisen, stecken bleiben oder ungültig sind, und damit die Eingabe abweisen. □

1.19 Korollar
Die Klasse der von deterministischen Turing-Maschinen (semi-)entscheidbaren Sprachen ist gleich der Klasse der von nicht-deterministischen Turing-Maschinen (semi-)entscheidbaren Sprachen.

1.20 Bemerkung
Im Gegensatz zu den vorherigen Konstruktionen für Mehrband-TMs etc. ist die Simulation von M_{NTM} durch *M* im Allgemeinen nicht effizient: Angenommen jede Berechnung von M_{NTM} zu einer Eingabe der Länge *n* hält nach höchstens $f(n)$ Schritten, wobei $f\colon \mathbb{N} \to \mathbb{N}$ eine beliebige Funktion ist.

Dann hat der Berechnungsbaum zu einer solchen Eingabe Höhe $f(n)$ und Ausgangsgrad (maximale Nachfolger pro Knoten) *r*, und damit $r^{f(n)}$ Knoten. Wenn die Maschine wirklich nicht-deterministisch ist, also $r \geqslant 2$ gilt, hat der Berechnungsbaum also exponentiell viele Knoten.

Dementsprechend braucht *M* also eventuell exponentiell viele Schritte, um diesen Baum zu durchsuchen.

Die Frage, ob es auch eine Umwandlung von NTMs in effiziente DTMS gibt, ist der Kern des berühmten offenen Problems $P \stackrel{?}{=} NP$.

1.21 Bemerkung
Zusätzlich zu den hier betrachteten Varianten gibt es noch viele weitere Variationen von Turing-Maschinen, die man betrachten kann, z.B. könnte man TMs mit mehreren Startzuständen oder mehreren Finalzuständen betrachten. Es sei der Leserin/dem Leser überlassen, zu beweisen, dass sich auch all diese Varianten durch TMs simulieren lassen.

Des Weiteren kann man natürlich die verschiedenen Varianten miteinander kombinieren. Man könnte also zum Beispiel nicht-deterministische Maschinen mit mehreren, jeweils in beide Richtungen unendlichen Bändern betrachten. Durch leichte Modifikation der Beweise ließe sich zeigen, dass auch solche Kombinationen nicht mächtiger als TMs sind.

2. Berechenbarkeit

Bislang haben wir uns – wie am Anfang des vorherigen Kapitels erwähnt – auf Entscheidungsprobleme beschränkt. Nun wollen wir uns allgemeineren **Berechnungsproblemen** widmen.

Ein Berechnungsproblem ist gegeben durch eine Funktion $f\colon M_1 \to M_2$. Ziel ist, zu jedem Wert $m \in M_1$ den zugehörigen Funktionswert $f(m) \in M_2$ durch einen Algorithmus zu berechnen.

Berechnungsproblem zu Funktion *f*

Gegeben: Wert $m \in M_1$

Berechne: Funktionswert $f(m)$

Auch hier beschränken wir uns auf Probleme, die mit Hilfe von Wörtern ausgedrückt werden können. Das bedeutet, dass wir annehmen, dass $M_1 = \Sigma_1^*$ und $M_2 = \Sigma_2^*$ gilt für endliche Alphabete Σ_1, Σ_2. In diesem Setting ist f eine Funktion $f\colon \Sigma_1^* \to \Sigma_2^*$, die Wörter über Σ_1 auf Wörter über Σ_2 abbildet.

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.

Auch hier gibt es wieder viele alternative Berechnungsmodelle, z.B.

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

Man nimmt an, dass sich jede intuitiv berechenbare Funktion auch mit einer Turing-Maschine berechnen lässt. Insbesondere lassen sich die obigen Modelle alle durch Turing-Maschinen simulieren.

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.

Jeder (deterministische) Algorithmus berechnet eine Funktion in diesem Sinne.

2.1 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
```

2.2 Beispiel
Sei $f\colon \{0,1\}^* \to \{0,1\}^*$ die (totale) Funktion, die wie folgt definiert ist:

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

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

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

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

Wir wollen nun den Begriff der Berechenbarkeit mit Hilfe von Turing-Maschinen formalisieren.

2.3 Definition
Sei $f\colon \Sigma_1^* \to_p \Sigma_2^*$ eine partielle Funktion. Wir nennen f **(Turing-)berechenbar**, wenn es eine deterministische Turing-Maschine $M = (Q, \Sigma_1, \Gamma, \delta, q_0)$ mit mindestens $k \geqslant 2$ Bändern gibt, die auf jeder Eingabe $w \in \Sigma_1^*$

- nicht hält oder abweist, falls $f(w)$ undefiniert ist,
- ansonsten nach endlich vielen Schritten akzeptiert und dabei auf dem letzten Band, genannt **Ausgabeband**, $f(w)$ steht, d.h. der Bandinhalt ist $\$f(w)\sqcup\sqcup\sqcup\ldots$.

Hierbei nehmen wir an, dass $\Sigma_2 \subseteq \Gamma$ im Bandalphabet ist, und die besonderen Symbole $\$, \sqcup \notin \Sigma_2$ nicht in Σ_2 vorkommen.

Wie zuvor nehmen wir an, dass alle Bänder nur nach rechts unendlich sind, d.h. der Bandinhalt ist nach links durch einen Marker $ begrenzt, der nicht überschrieben wird. Zudem verändert die Maschine, sobald sie im akzeptierenden oder abweisenden Zustand ist, ihren Bandinhalt und Kontrollzustand nicht mehr.

Wir treffen nun zwei zusätzliche Annahmen:

- Das Eingabeband ist **read-only**, es wird von der Maschine nicht verändert, d.h. δ hat die Eigenschaft

  $$\delta(q, (a, a_2, \ldots, a_k)) = \delta(q', (a, b_2, \ldots, b_k), (d_1, \ldots, d_k))$$

  für alle $q \in Q$, $a, a_2, \ldots, a_k \in \Gamma$ und geeignete $q' \in Q$, $b_2, \ldots, b_k \in \Gamma$, $d_1, \ldots, d_k \in \{L, S, R\}$.

- Das Ausgabeband ist **write-only**: Wenn immer die Maschine eine Zelle des Ausgabebandes schreibt, bewegt sie sich anschließend nach rechts. Die Maschine bewegt sich niemals auf dem Ausgabeband nach links. Dies bedeutet, dass eine Zelle, die einmal beschrieben wurde, im Folgenden weder gelesen noch verändert wird.

  Formal gilt für alle $q \in Q, a_1, \ldots, a_{k-1} \in \Gamma$:

  $$\delta(q, (a_1, \ldots, a_{k-1}, \sqcup)) = \delta(q', (b_1, \ldots, b_{k-1}, b), (d_1, \ldots, d_k))$$

  für geeignete $q' \in Q, b_1, \ldots, b_{k-1}, b \in \Gamma$, $d_1, \ldots, d_k \in \{L, S, R\}$, wobei entweder $b = \sqcup$ und $d_k = S$ oder $b \neq \sqcup$ und $d_k = R$ gilt.

2.4 Bemerkung
Die beiden zusätzlichen Annahmen sind irrelevant, solange wir uns nur für Berechenbarkeit interessieren: Man kann zunächst die Eingabe auf ein zusätzliches Band kopieren, dass frei verwendet werden darf. Analog kann man die Ausgabe zunächst auf ein weiteres Band schreiben, und erst am Schluss auf das Ausgabeband kopieren.

Im Teil der Vorlesung zu Komplexitätstheorie werden wir den Platzverbrauch einer Turing-Maschine messen. Hierbei wollen wir den Platz auf dem Eingabe- und Ausgabeband nicht mitzählen, müssen dann allerdings verhindern, dass die Maschine diese Bänder zum Rechnen nutzen kann.

Wir nennen eine Funktion $f\colon M_1 \to_p M_2$ für beliebige M_1, M_2 (d.h. eventuell nicht von der Gestalt Σ_1^*, Σ_2^*) **berechenbar**, wenn es eine Kodierung der Werte gibt und die entsprechende Funktion, die auf diesen Kodierungen arbeitet, berechenbar ist.

Beispielsweise nennen wir eine Funktion $f\colon \mathbb{N} \to_p \mathbb{N}$ berechenbar, falls die Funktion f_{bin} die auf Binärdarstellungen der Zahlen arbeitet, d.h.

$$\begin{array}{rcll} f_{\text{bin}} & : & \{0,1\}^* & \to_p \quad \{0,1\}^* \\ & & \text{bin}(n) & \mapsto \begin{cases} \text{undefiniert} & \text{, falls } f(n) \text{ undefiniert ist,} \\ \text{bin}(f(n)) & \text{, sonst.} \end{cases} \end{array}$$

berechenbar ist. Hierbei ist $\text{bin}\colon \mathbb{N} \to \{0,1\}$ eine Funktion, die jeder Zahl auf eine Binärdarstellung abbildet.

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

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

Es gilt z.B. $f(314) = 1, f(5) = 0$. 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.

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 nur abzählbar viele berechenbare Algorithmen, und dementsprechend nicht für jede reelle Zahl einen Approximationsalgorithmus.

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

Beweis:
Der Beweis funktioniert prinzipiell analog zum Resultat für Entscheidbarkeit. Wir wollen ihn hier nun etwas ausführlicher machen. 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.

Um einen Widerspruch zu erhalten nehmen wir an, dass jede Funktion $f\colon \mathbb{N} \to_p \mathbb{N}$ berechenbar ist.

Analog zum Resultat für Entscheidbarkeit müssen wir nur abzählbar viele Turing-Maschinen in Betracht ziehen. 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.

2.7 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.

Wir wollen nun die Brücke zu den Begriffen schlagen, die wir im Kapitel über Entscheidbarkeit kennen gelernt haben.

2.8 Theorem
Eine Sprache $\mathcal{L} \subseteq \Sigma^*$ ist **rekursiv aufzählbar** (aka semi-entscheidbar), wenn $\mathcal{L} = \emptyset$ gilt, oder es eine totale, berechenbare Funktion $f\colon \mathbb{N} \to \Sigma^*$ gibt mit

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

Wir sagen, dass eine solche Funktion f die Sprache $\mathcal{L}$ **aufzählt**. Beachte, dass $f(i) = f(j)$ für $i \neq j$ erlaubt ist.

Beweis:

„$\Leftarrow$" Die leere Menge ist durch eine Turing-Maschine, die im abweisenden Zustand q_{rej} startet, sogar entscheidbar. Nehmen wir nun an, dass $\mathcal{L}$ 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 $\mathcal{L}$ 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 $w \in \mathcal{L}$, dann gibt es einen Index n mit $w = f(n)$, und der Semi-Entscheider akzeptiert, sobald er diesen Index erreicht. Falls $w \notin \mathcal{L}$, dann gibt es keinen solchen Index. Der Semi-Entscheider hält in diesem Fall nicht an.

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

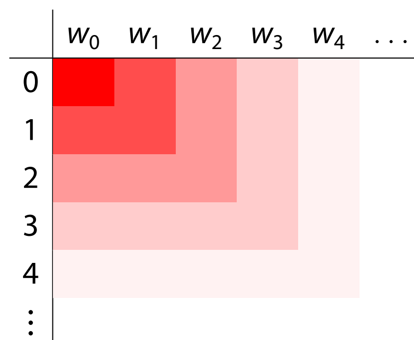
Der folgende Algorithmus $\mathcal{A}$ implementiert dieses Verfahren.

```
for i = 0, 1, 2, . . . do
    for j = 0, 1, 2, . . . , 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 $\mathcal{L}$ 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 $\mathcal{L}$ 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 $\mathcal{A}$, in dem er die vorherigen verwirft, und nach der n-ten Ausgabe anhält.

```
Eingabe: n ∈ ℕ
  m ← 0
  for i = 0, 1, 2, ... do
    for j = 0, 1, 2, ..., i do
      Berechne j-tes Wort w
      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
```

□

2.9 Definition

Sei $\mathcal{L} \subseteq \Sigma^*$ eine Sprache. Die totale charakteristische Funktion $\chi_\mathcal{L}$ von $\mathcal{L}$ ist die totale Funktion

$$\begin{aligned} \chi_\mathcal{L} : \Sigma^* &\to \{0, 1\} \\ w &\mapsto \begin{cases} 1 & \text{, falls } w \in \mathcal{L}, \\ 0 & \text{, sonst.} \end{cases} \end{aligned}$$

Die partielle charakteristische Funktion $\chi'_\mathcal{L}$ von $\mathcal{L}$ ist die partielle Funktion

$$\begin{aligned} \chi'_\mathcal{L} : \Sigma^* &\to_p \{1\} \\ w &\mapsto \begin{cases} 1 & \text{, falls } w \in \mathcal{L}, \\ \text{undefiniert} & \text{, sonst.} \end{cases} \end{aligned}$$

2.10 Theorem

a) Eine Sprache $\mathcal{L}$ ist semi-entscheidbar gdw. ihre partielle charakteristische Funktion $\chi'_\mathcal{L}$ berechenbar ist.

b) Eine Sprache $\mathcal{L}$ ist entscheidbar gdw. ihre totale charakteristische Funktion $\chi_\mathcal{L}$ berechenbar ist.

Beweis:

Es ist leicht, aus einem (Semi-)Entscheidungsalgorithmus für $\mathcal{L}$ einen Berechnungsalgorithmus für $\chi'_\mathcal{L}$ bzw. $\chi_\mathcal{L}$ zu konstruieren und umgekehrt. □

2.11 Korollar
Sei $\mathcal{L} \subseteq \Sigma^*$ eine Sprache. Die folgenden Aussagen sind äquivalent:

1. $\mathcal{L}$ ist semi-entscheidbar / rekursiv aufzählbar / Turing-erkennbar.

2. Es gibt eine TM M mit $\mathcal{L}(M) = \mathcal{L}$.

3. Es gibt einen Aufzählungsalgorithmus für $\mathcal{L}$, d.h. $\mathcal{L}$ ist der Wertebereich einer totalen berechenbaren Funktion $f\colon \mathbb{N} \to \mathcal{L}$.

4. $\chi'_{\mathcal{L}}$ ist berechenbar.

5. $\mathcal{L}$ ist der Definitionsbereich einer partiellen berechenbaren Funktion $g\colon \Sigma^* \to_p \Sigma_2^*$.

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

Eine Menge M heißt **abzählbar**, wenn M leer ist oder wenn es eine surjektive Funktion $f\colon \mathbb{N} \to M$ gibt. Erinnerung: Surjektiv bedeutet, dass es zu jedem $m \in M$ 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 M, wir sehen $f(0)$ als das erste, $f(1)$ als das zweite, usw. Element von M.

Wir haben bereits gesehen oder angemerkt, dass die Menge der reellen Zahlen, die Menge der Sprachen $\mathcal{L} \subseteq \Sigma^*$ 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 $\mathcal{L} \subseteq \Sigma^*$ 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 M ist auch jede Teilmenge $M' \subseteq M$ abzählbar – man überspringt beim Abzählen die Elemente, die nicht in M' liegen.

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

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

Durch Kombination der beiden Aussagen erhalten wir, dass $\mathcal{L}$ abzählbar ist, in dem wir alle Wörter in Σ^* abzählen und dabei die Wörter, die nicht in $\mathcal{L}$ 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 $\mathcal{L} \subseteq \Sigma^*$, die nicht semi-entscheidbar, und damit nicht rekursiv aufzählbar sind.

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

In Kapitel 1 haben wir nicht-konstruktiv bewiesen, dass es nicht semi-entscheidbare Probleme geben muss. Nun wollen wir endlich konkrete Beispiele für nicht-(semi-)entscheidbare Probleme kennen lernen. Wie bereits angedeutet, handelt es sich hierbei um Verfikationsprobleme, also Probleme, bei denen entschieden werden soll, ob ein gegebenes Programm eine bestimmte Eigenschaft hat. Dies bedeutet, dass die Eingabe des Problems selbst eine Turing-Maschine ist.

In diesem Kapitel werden wir viele verschiedene Konzepte kennen lernen.

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

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

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

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

A) Kodierungen von Turing-Maschinen

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

3.1 Definition: Kodierung von Turing-Maschinen
Sei $M = (Q, \Sigma, \Gamma, \delta, q_0)$ eine TM.

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

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

Hierbei gehen wir davon aus, dass q_0 der Startzustand, $q_1 = q_{acc}$, $q_2 = q_{rej}$, $a_0 = \$$, und $a_1 = \sqcup$ ist. Wir gehen davon aus, dass dann zunächst das Eingabealphabet Sigma kodiert ist, d.h. $\Sigma = \{a_2, \dots a_k\}$ für eine Zahl $k > 2$.

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

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

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

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

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

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

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

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

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

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

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

3.2 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Alternativ können wir uns dank Theorem 1.15 darauf beschränken, Turing-Maschinen zu untersuchen, deren Eingabealphabet $\{0, 1\}$ und deren Bandalphabet $\{0, 1, \$, \sqcup\}$ ist.

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

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

B) Die universelle Turing-Maschine

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

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

Wir können dieses Problem nun als das folgende Wortproblem auffassen:

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

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

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

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

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

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

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

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

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

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

Beweisskizze für Theorem 3.9:
U verwendet zusätzlich zum Eingabeband drei weitere Bänder, nämlich Programm-, Daten- und Zustandsband. Mittels Theorem 1.13 ließe sich eine äquivalente 1-Band-TM konstruieren.

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

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

Nun ist zu überprüfen, ob $w' = \prod_{i=0}^{n} \prod_{j=0}^{m} w_{i,j,i',j',d}$ gilt, das heißt ob es zu jeder Kombination aus Zustand und Symbol eine korrekt kodierte Transition gibt. Schlussendlich muss kontrolliert werden, dass die beiden Zusatzbedingungen (1) und (2) aus der Definition von TM erfüllt sind.

Falls eine dieser Überprüfungen fehlschlägt, weist U ab, denn dann gilt $M_w = M_\varnothing$, und somit $\mathcal{L}(M_w) = \varnothing$. Wenn alle Überprüfungen erfolgreich sind, kopiert U den hinteren Teil der Eingabe, also x, auf das dritte Band, das **Datenband**. Um die Kodierung der Eingabe wollen wir uns hierbei keine Gedanken machen, wir gehen also davon aus, dass $\{0, 1\}$ das Eingabealphabet und $\{0, 1, \$, \llcorner\!\lrcorner\}$ das Bandalphabet von M_w ist, siehe Bemerkung 3.7.

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

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

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

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

3.11 Bemerkung
Theorem 3.9 beweist insbesondere, dass das Akzeptanzproblem semi-entscheidbar ist.

C) Das Halteproblem

Wir betrachten nun das berühmte **Halteproblem**.

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

Wir identifizieren das Halteproblem mit der folgenden Sprache

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

3.12 Theorem: Turing 1936
Das Halteproblem HP ist semi-entscheidbar, allerdings nicht entscheidbar.

Eine Teil der Aussage ist mit Hilfe der universellen Turing-Maschine sehr leicht zu beweisen.

3.13 Lemma
Das Halteproblem HP ist semi-entscheidbar.

Beweis:
Man kann analog zur Konstruktion der universellen Turing-Maschine U eine Turing-Maschine U' konstruieren mit $\mathcal{L}(U') = \mathrm{HP}$. Für eine Eingabe $w\#x$ simuliert U' die Maschine M_w auf der Eingabe x. Falls M_w hält (also akzeptiert oder abweist), akzeptiert U'. □

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

3.14 Proposition
Das Halteproblem HP ist nicht entscheidbar.

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

Wir konstruieren eine neue Maschine D, die sich auf einer Eingabe $w\#x$ zunächst wie H verhält, dann jedoch

- falls H abweisen würde, akzeptiert,
- falls H akzeptieren würde, nicht hält.

Die Maschine D kehrt also gewissermaßen die Ausgabe von H um.

Betrachten wir nun die Eingabe $\langle D\rangle\#\langle D\rangle$ für Maschine D, wir fragen uns also, ob die Maschine D auf ihrer eigenen Kodierung hält.

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

Fall 2: D hält nicht auf $\langle D\rangle\#\langle D\rangle$:
Nach Konstruktion von D akzeptiert H also die Eingabe $\langle D\rangle\#\langle D\rangle$. Da $\mathcal{L}(H) = \text{HP}$, bedeutet dies, dass D auf $\langle D\rangle\#\langle D\rangle$ hält - ein Widerspruch.

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

3.15 Bemerkung
Im obigen Beweis haben wir Diagonalisierung verwendet, siehe den Beweis von Theorem 2.6. Dies ist allerdings nicht offensichtlich.

Statt einer Abzählung aller Turing-Maschinen, betrachten wir eine Abzählung aller Wörter $w_0, w_1, w_2, \ldots$ über $\{0,1\}^*$. Jedes solche Wort w_i kodiert eine Turing-Maschine M_{w_i}. Wenn es eine Maschine D wie im Beweis gäbe, dann gäbe es eine Zahl $i \in \mathbb{N}$ mit $D = M_{w_i}$.

Im Beweis haben wir gesehen, dass wir allerdings auf der Diagonale, also für die Turing-Maschine $D = M_{w_i}$ und die Eingabe $w_i = \langle D\rangle$, einen Widerspruch erhalten.

3.16 Bemerkung
Statt des Halteproblems könnte man auch das Akzeptanzproblem aus Teil B) des Kapitels betrachten. Dieses ist ebenso semi-entscheidbar, allerdings nicht entscheidbar.

Um seine Unentscheidbarkeit zu beweisen, genügt es, eine kleine Modifikation im Beweis der Unentscheidbarkeit des Halteproblems vorzunehmen. Dies nachzuvollziehen sei der Leserin/dem Leser als Übungsaufgabe überlassen. Alternativ kann man Reduktionen verwenden, um den Beweis zu führen, hierauf gehen wir im nächsten Teilkapitel ein.

Wir haben uns aus zwei Gründen dazu entschieden, hier zunächst das Halteproblem als unentscheidbar nachzuweisen. Zum Einen war das Halteproblem das Problem, von dem Turing initial bewies, dass es unentscheidbar ist. Zum Anderen präzisiert die Unentscheidbarkeit des Halteproblems die Bemerkung, die wir gemacht hatten, als wir Entscheidbarkeit eingeführt haben: Es ist nicht bloß schwer, Nicht-Halten von Halten (also Akzeptieren oder Abweisen) zu unterscheiden, es ist unmöglich.

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 Halteproblem** unentscheidbar ist.

Speziellles Halteproblem (SHP)	
Gegeben:	Eine Turing-Maschine M
Entscheide:	Hält die Berechnung von M auf $\langle M\rangle$ nach endlich vielen Schritten?

$$\mathrm{SHP} = \left\{w \in \{0,1\}^* \;\middle|\; M_w \text{ hält auf } w\right\} .$$

3.17 Korollar
Das spezielle Halteproblem SHP 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.

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 Halteproblems – einen Beweis genau inspizieren. Stattdessen wollen wir bereits gezeigte Resultate verwenden, um aus der Unentscheidbarkeit z.B. des Halteproblems die Unentscheidbarkeit weiterer Probleme abzuleiten.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Üblicherweise wird die Kontraposition des obigen Lemmas verwendet.

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

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

3.21 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, siehe Theorem 2.10. 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

a) Das **allgemeine Akzeptanzproblem**

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

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

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

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

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

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

Um zu zeigen, dass SELF-ACCEPT und $\text{ACCEPT}_\varepsilon$ semi-entscheidbar sind, können wir Reduktionen in Kombination mit Lemma 3.19 verwenden.

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

Um die Unentscheidbarkeit der Probleme zu beweisen, können wir die Kontraposition von Lemma 3.19, also Korollar 3.20, verwenden, und für jede Variante des Akzeptanzproblems eine Reduktion des jeweiligen Halteproblems angeben.

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

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

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

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

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

- sich zunächst wie M_w verhält,
- falls M_w akzeptiert, akzeptiert und
- falls M_w abweist, ebenfalls akzeptiert.

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

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

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

4. Das Postsche Korrespondenzproblem & der Satz von Rice

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

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

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

A) Das Postsche Korrespondenzproblem (PCP)

4.1 Definition

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

<u>**Postsches Korrespondenzproblem**</u> (PCP)

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

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

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

4.2 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.

4.3 Theorem: Post 1946
Das PCP ist unentscheidbar.

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

Modifiziertes Postsches Korrespondenzproblem (MPCP)

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

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

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

4.4 Lemma
MPCP $\leqslant$ PCP.

TODO: Bessere Erklärung, was hier passiert

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:

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

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.

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

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

„⇐“ 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 Halteproblem auf sie reduzieren.

4.5 Proposition
HP $\leqslant$ MPCP.

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

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

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

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

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

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

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

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

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

Das Alphabet der PCP-Instanz ist $\Gamma \uplus Q \uplus \{\#\}$. Das erste Paar ist $(\#, \#q_0\$x\#)$, 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.

$$\begin{array}{ll} x\text{-Sequenz}: & \overbrace{\#}^{1}\overbrace{q_0 a_1}^{2}\overbrace{a_2}^{3}\dots\overbrace{a_n}^{n+1}\overbrace{\#}^{n+2} \\ y\text{-Sequenz}: & \underbrace{\#\ q_0\ a_1\ a_2\ \dots\ a_n\ \#}_{1}\ \underbrace{q_1 b}_{2}\underbrace{a_2}_{3}\dots\underbrace{a_n}_{n+1}\underbrace{\#}_{n+2} \end{array}$$

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

- Kopierregeln:

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

- Transitionsregeln:

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

- 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.

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

- 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.

4.6 Bemerkung

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

B) Der Satz von Rice

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

4.7 Definition

Sei Σ ein Alphabet. Wir bezeichnen mit $\text{RE}(\Sigma)$ 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

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

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

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

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

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

Wir sehen eine Eigenschaft P also als Sprache

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

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

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

4.8 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.

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

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

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

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

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

woraus folgt, dass P unentscheidbar sein muss.

Sei $w\#x$ eine Eingabe für das Halteproblem, bestehend aus der Kodierung der Maschine M_w und der Eingabe x. Wir konstruieren eine Maschine $M^K_{w,x}$, deren Sprache $\varnothing$ ist, falls M_w auf x nicht hält, und deren Sprache $\mathcal{L}$ ist, falls M_w auf x hält. 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 codieren.
3. Simuliere M_w auf Eingabe x.

   Dieser Schritt terminiert eventuell nicht, falls M_w auf Eingabe x nicht hält.

4. Falls M_w auf Eingabe x hält, und damit die obige Simulation hält, simuliere K auf Eingabe y.

   Akzeptierte genau dann, wenn K Eingabe y akzeptiert.

1. Fall: M_w hält auf Eingabe x.

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

2. Fall: M_w hält auf Eingabe x nicht.

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

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

Zusammenfassend erhalten wir:

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

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

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

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

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

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

5. Unentscheidbare Probleme kontextfreier Sprachen

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

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

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

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

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

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

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

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

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

Beweis:

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

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

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

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

Grammatik G_1 hat die folgenden Produktionsregeln:

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

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

Die von G_1 erzeugte Sprache ist

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

Grammatik G_2 hat die folgenden Produktionsregeln:

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

Die von G_2 erzeugte Sprache ist also

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

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

$$a_{i_n} \ldots a_{i_1}x_{i_1} \ldots x_{i_n}\$y_{i_n}^{\text{reverse}} \ldots y_{l_1}^{\text{reverse}}a_{i_n} \ldots a_{i_1}$$

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

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

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

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

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

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

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

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

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

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

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

Es gilt

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

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

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

Es gilt

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

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

□

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

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

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

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

Teil II.
Komplexitätstheorie

TODO: Einführender Text

6. Zeit- und Platzkomplexität

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

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

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

A) Zeitkomplexität

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

B) Platzkomplexität

Wir möchten analog zu DTIME und NTIME Platzkomplexitätsklassen einführen.

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

Daher betrachten wir hier Maschinen mit einem speziellen Eingabeband. Das Eingabeband ist **read-only**, die Eingabe darf also nicht verändert werden. Formal ist read-only wie in Definition 2.3 definiert.

Die anderen Bänder bezeichnen wir als **Arbeitsbänder**. Wir messen nur den Platzverbrauch der Maschine auf ihren Arbeitsbändern.

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

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

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

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

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

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

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

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

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

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

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

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

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

Wir verwenden die selbe vereinfachte Notation wie bei den Zeitkomplexitätsklassen.

Man beachte, dass der Zeitverbrauch den Platzverbrauch beschränkt: In jedem Schritt kann eine Maschine höchstens eine neue Zelle beschreiben. Daraus folgt zum einen, dass

$\mathrm{Space}_M(x) = \infty$ nicht auftreten kann, wenn M ein Entscheider ist. Zum anderen beweist es das folgende Lemma:

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

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

Insbesondere gilt also

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

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

6.9 Beispiel
Betrachte die Sprache der Wörter mit gleich vielen *a*s und *b*s, also

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

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

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

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

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

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

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

C) Die robusten Komplexitätsklassen

6.10 Definition

Wir definieren nun die **robusten Komplexitätsklassen**.

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

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

6.11 Bemerkung

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

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

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

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

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

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

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

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

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

Beweis:
Zu einer Maschine M mit k-Bändern lässt sich eine äquivalente TM M' mit nur einem Band konstruieren, siehe Theorem 1.13.

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

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

In Theorem 1.13 haben wir deterministische 1-Band-TMs betrachtet, mit analogen Beweisen kann man allerdings auch Varianten des Theorems für nicht-deterministische Maschinen oder Maschinen mit Arbeitsband zeigen. □

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

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

D) Komplementklassen

Wir wollen auch Komplementklassen kennen lernen.

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

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

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

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

$$\mathrm{co}\mathcal{C} = \left\{\overline{\mathcal{L}} \,\middle|\, \mathcal{L} \in \mathcal{C}\right\}$$

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

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

6.15 Beispiel
Sei

$$\mathrm{COPY} = \left\{w\#w \,\middle|\, w \in \{0,1\}^*\right\}$$

das bekannte Kopierproblem. Wir haben in den Übungen gesehen, dass sich COPY in $\mathrm{DTIME}_1\left(\mathcal{O}\left(n^2\right)\right) \subseteq \mathrm{P}$ und in $\mathrm{DSPACE}_1(\mathcal{O}(\log n)) \subseteq \mathrm{L}$ lösen lässt.

Daraus ergibt sich, dass

$$\overline{\mathrm{COPY}} = \{0,1\}^* \cup \{0,1,\#\}^*\#\{0,1,\#\}^*\#\{0,1,\#\}^* \cup \left\{w\#w' \,\middle|\, w, w' \in \{0,1\}^*, w \neq w'\right\}$$

in $\mathrm{coDTIME}_1\left(\mathcal{O}\left(n^2\right)\right) \subseteq \mathrm{coP}$ und $\mathrm{coDSPACE}_1(\mathcal{O}(\log n)) \subseteq \mathrm{coL}$ ist.

Die deterministischen Klassen stimmen mit ihren Komplementklassen überein.

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

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

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

6.17 Korollar
Die deterministischen Komplexitätsklassen sind abgeschlossen unter Komplement, es gilt insbesondere

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

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

E) Grundlegende Relationen zwischen den Komplexitätsklassen

Direkt aus den Definitionen ergeben sich einige Inklusionen innerhalb der Komplexitätsklassen.

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

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

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

Als Korollar von Lemma 6.8 erhalten wir die folgende Aussage.

6.19 Korollar
Es gilt

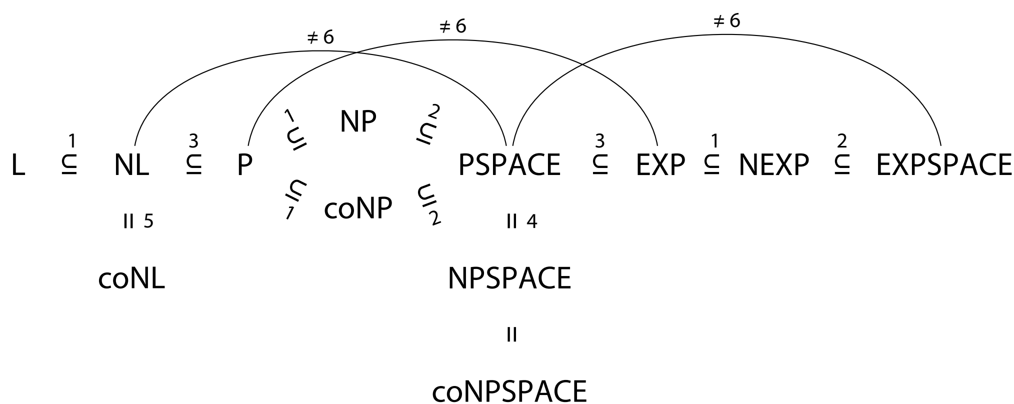
$$\mathrm{P} \subseteq \mathrm{PSPACE}\,, \qquad \mathrm{NP} \subseteq \mathrm{NPSPACE}\,.$$

7. Eine Landkarte der Komplexitätstheorie

Im Rest der Vorlesung wollen wir

- zum Einen die robusten Komplexitätsklassen untersuchen: Wir werden typische Probleme in den Klassen betrachten und wir werden untersuchen, welche Arten von Algorithmen sich mit einem vorgegeben Zeit und Platzverbrauch umsetzen lassen.
- zum Anderen die Beziehungen der Klassen untereinander untersuchen.

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



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

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

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

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

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

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

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

8. L und NL

Nun wollen wir die robusten Komplexitätsklassen im Detail untersuchen, beginnend mit L und NL. Wir wollen Probleme kennen lernen, die sich mit logarithmisch viel Platz (nichtdeterministisch bzw. deterministisch) lösen lassen. Ob L = NL gilt, ist offen. Wir versuchen die beiden Klassen voneinander abzugrenzen. Hierzu werden wir Probleme in NL identifizieren, von denen man glaubt, dass sie nicht in L liegen.

A) 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
Entscheide: Gilt $i + j = \ell$?

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

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

$$\begin{aligned} \text{ADD} &= \{x\#y\#z \mid \exists i, j, \ell : x = \text{bin}(i), y = \text{bin}(j), z = \text{bin}(\ell), x + y = z\}, \\ \text{MULT} &= \{x\#y\#z \mid \exists i, j, \ell : x = \text{bin}(i), y = \text{bin}(j), z = \text{bin}(\ell), x \cdot y = z\}, \end{aligned}$$

8.1 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.

B) Das Pfadproblem in NL

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.

<u>**Pfadexistenz**</u> (PATH)	
Gegeben:	Gerichteter Graph $G = (V, R)$, Quellknoten $s \in V$, Zielknoten $t \in V$
Entscheide:	Gibt es einen Pfad von s nach t in G?

Um dieses Problem als Wortproblem aufzufassen, müssen wir festlegen, wie ein gerichteter Graph G kodiert werden soll. Wir können davon ausgehen, dass die Knoten $V = \{1, \dots, n\}$ durchnummeriert sind. Dies erlaubt es uns, einen Knoten i durch die Binärdarstellung $\mathrm{bin}(i)$ 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

$$\mathrm{adj}_i = i \to j_1, j_2, \dots, j_{k_i}$$

wobei $j_1, \dots j_{k_i}$ die Knoten sind, zu denen i eine ausgehende Kante hat. Die Zahlen $i, j_1, \dots j_{k_i}$ können wir wieder wie üblich binär kodieren.

Insgesamt kodieren wir dann einen Graphen G mit n Knoten als Wort

$$\mathrm{adj}_1; \dots; \mathrm{adj}_n \,.$$

Im Folgenden werden wir einen Graphen mit seiner Kodierung identifizieren. Wir können PATH damit als Wortproblem

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

auffassen.

8.2 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. □

C) Reduktionen und Vollständigkeit

Wir möchten nun L gegen NL abgrenzen. Leider ist es bislang nicht gelungen, zu zeigen, dass L ≠ NL gilt. Daher gehen wir wie folgt vor: Wir identifizieren innerhalb von NL die schwersten Probleme. Man geht davon aus, dass diese Probleme nicht in L liegen.

Formal nennen wir ein Problem NL-**hart** oder NL-**schwer**, wenn es mindestens so schwer ist wie jedes andere Problem in NL. Ein Problem heißt NL-**vollständig**, wenn es NL-hart ist und in NL liegt.

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

Im Folgenden lernen wir Techniken kennen, um Probleme als schwer nachzuweisen. Was bedeutet es, zu zeigen, dass ein Problem mindestens so schwer ist, wie jedes andere Problem 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.

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

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

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

8.4 Definition
Sei $\mathcal{C}$ eine Klasse von Problemen, R eine Menge von Funktionen und B ein Problem.

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

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

Intuitiv bedeutet dies, dass B mindestens so schwer wie jedes Problem 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 des Problems, welches 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.

TODO: Problem fixen

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

Insbesondere sollte, wenn A ein $\mathcal{C}$-schweres Problem ist und $A \leqslant_m^R B$ gilt, auch Problem B $\mathcal{C}$-schwer sein.

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

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

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

8.6 Definition
Sei M ein totaler Berechner, also eine Turing-Maschine mit speziellem Ein- und Ausgabeband, die auf jeder Eingabe nach endlich vielen Schritten akzeptiert. Wir erinnern uns daran, dass das Eingabeband read-only und das Ausgabeband write-only ist.

Wie bei Entscheidern sagen wir, dass der Zeitverbrauch von M durch eine Funktion $t\colon \mathbb{N} \to \mathbb{N}$ beschränkt ist, wenn für jede Zahl $n \in \mathbb{N}$ und jede Eingabe x mit Länge $|x| = n$ der Berechner M nach höchstens $t(n)$ Schritten hält.

Analog ist der Platzverbrauch von M durch eine Funktion $s\colon \mathbb{N} \to \mathbb{N}$ beschränkt, wenn für jede Zahl $n \in \mathbb{N}$ und jede Eingabe x mit Länge $|x| = n$ der Berechner M in jeder Konfiguration auf jedem Arbeitsband höchstens $s(n)$ Zellen benutzt werden. 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 M diese Bänder nicht zum rechnen zweckentfremden kann.

8.7 Definition
Eine Funktion $f\colon \Sigma_1^* \to \Sigma_2^*$ ist **logspace-berechenbar**, wenn es einen deterministischen Berechner mit speziellem Ein- und Ausgabeband gibt,

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

Eine Funktion $f\colon \Sigma_1^* \to \Sigma_2^*$ ist **in Polynomialzeit berechenbar**, wenn es einen deterministischen Berechner mit speziellem Ein- und Ausgabeband gibt,

- der total ist,
- zu jeder Eingabe $x \in \Sigma_1^*$ nach endlich vielen Schritten hält und die Ausgabe $f(x) \in \Sigma_2^*$ geschrieben hat und
- dessen Zeitverbrauch durch $\mathcal{O}\left(n^k\right)$ für eine von der Eingabegröße unabhängige Konstante k beschränkt ist.

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

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

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

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

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

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

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.

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

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

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

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

Problem: Wir haben bei logspace-berechenbaren Funktionen nur den Platzverbrauch auf den Arbeitsbändern beschränkt. Die Ausgabe $f(x)$ von M_f kann mehr als logarithmisch viel Platz brauchen, und ist damit zu groß, um auf einem Arbeitsband zwischengespeichert werden zu können.

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

Wichtig hierfür ist,

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

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

Eine Konfiguration von M_f zu Eingabe x ist gegeben durch

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

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

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

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

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

viele Möglichkeiten für den Bandinhalt.

Insgesamt gibt es höchstens

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

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

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

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

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

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

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

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

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

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

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

Beweis der eigentlichen Aussage

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

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

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

Gemäß unserer obigen Diskussion ist der Wert des Zählers i durch $\mathcal{O}\left(n^d\right)$ beschränkt, wenn wir ihn Binär speichern, benötigen wir also nur $\mathcal{O}\left(\log(n^d)\right) = \mathcal{O}(d \cdot \log n) = \mathcal{O}(\log n)$ viele Zellen. Die Werte $(f(x))_i$ können mit logarithmischem Platz berechnet werden, und da sie nur eine einzige Zelle belegen, können wir sie auch jeweils speichern. Wichtig ist, dass wir den Speicherplatz für die aktuelle Zelle und die entsprechende Berechnung wiederverwenden konnen. Insgesamt erhalten wir, dass $M_{g\circ f}$ nur logarithmisch viel Platz braucht. □

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

8.11 Lemma
Sei Σ ein endliches Alphabet.

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

b) Jede Sprache $\mathcal{L} \subseteq \Sigma^*$ in L mit $\mathcal{L} \neq \varnothing$ und $\mathcal{L} \neq \Sigma^*$ ist bereits L-vollständig (bezüglich logspace-many-one-Reduktionen).

Beweis: Übungsaufgabe. □

Es ist also nur sinnvoll, Reduktion zu betrachten, die schwächer sind als die betrachtete Klasse.

Viele Klassen sind abgeschlossen unter logspace-Reduktionen

8.12 Lemma
Sei $A \leqslant_m^{log} B$. Wenn B in L bzw. NL bzw. P ist, dann ist auch A in L bzw. NL bzw. P.

Das folgende Lemma zeigt, dass sich schwere Probleme tatsächlich dazu eignen, die Klassen voneinander abzugrenzen. Sollte es möglich sein, ein hartes Problem in einer Klasse mit weniger Aufwand zu lösen, dann fallen die entsprechenden Klassen zusammen.

8.13 Lemma
Sei A eine Sprache.

a) Falls A NL-schwer bezüglich logspace-many-one-Reductions ist und $A \in$ L gilt, dann folgt NL = L.

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

D) PATH ist NL-vollständig

Wir haben bereits bewiesen, dass PATH in NL lösbar ist. Nun wollen wir als PATH als erstes Problem als NL-vollständig 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

8.14 Theorem
PATH ist NL-schwer (bezüglich logspace-many-one-Reduktionen).

Beweis:
Wir müssen jedes andere Problem in NL auf PATH reduzieren. Sei $\mathcal{L} \in$ NL ein solches Problem. Es gibt eine NTM M mit durch $\mathcal{O}(\log n)$-beschränkten Platzverbrauch, die $\mathcal{L}$ entscheidet, also $\mathcal{L}(M) = \mathcal{L}$.

Wir zeigen, dass es eine Funktion f_M gibt, die mit logarithmischem Platz berechenbar ist, mit: Für eine Eingabe x und den zugehörigen Funktionswert $f_M(x) = G\#s\#t$ gilt:

$$M \text{ akzeptiert } x \quad \text{gdw.} \quad \text{es gibt in } G \text{ einen Pfad von } s \text{ nach } t\,.$$

Hierbei ist G ein gerichtet Graph und s und t sind Knoten in G.

Der Graph G ist der Konfigurationsgraph von M zu Eingabe x. Seine Knoten sind Konfigurationen mit Eingabe x und einem Bandinhalt, der durch $\mathcal{O}(\log n)$ beschränkt ist. Für zwei Konfigurationen c_1, c_2 beinhaltet der Graph eine Kante (c_1, c_2) genau dann, wenn c_2 ein mögliche Nachfolgekonfiguration von c_1 ist. Der Quellknoten s ist die Startkonfiguration von M, d.h. initialer Kontrollzustand, Eingabe x und leeres Arbeitsband.

Wir gehen o.B.d.A. davon aus, dass M eine eindeutige akzeptierende Konfiguration hat: Kontrollzustand q_{acc}, Eingabe x, leeres Arbeitsband und Kopfposition in beiden Bändern auf dem Endmarker \$. 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 vorm akzeptieren den Bandinhalt löscht und die Köpfe zum Endmarker 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.

Der Berechner 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. □

8.15 Korollar
PATH ist NL-vollständig (bezüglich logspace-many-one-Reduktionen).

Man kann das Pfadproblem sogar auf sogenannte **azyklische** Graphen, also auf Graphen, in denen es keine Kreise $c \to c_0 \to \ldots \to c_k \to c$ gibt, einschränken. Es bleibt NL-vollständig.

Pfadexistenz in azyklischen Graphen (ACYCLICPATH)
Gegeben: Gerichteter azyklischer Graph $G = (V, R)$, Knoten $s, t \in V$
Entscheide: Gibt es einen Pfad von s nach t in G?

8.16 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. coNL & der Satz von Immerman und Szelepcsényi

Wir möchten nun die Klasse coNL untersuchen, also die Komplemente von Problemen in NL. Wir werden das Problem 2SAT einführen und als coNL-vollständig nachweisen. Später beweisen wir den Satz von Immerman und Szelepcsényi, der besagt, das coNL = NL. Damit erhalten wir dann, dass 2SAT auch NL-vollständig ist.

Zunächst halten wir ein allgemeines Resultat fest.

9.1 Lemma
Sei $\mathcal{C}$ eine Komplexitätsklasse, R eine Menge von Funktionen und $\mathcal{L} \in \mathcal{C}$ ein Problem. Wenn $\mathcal{L}$ $\mathcal{C}$-schwer/vollständig ist, dann ist $\overline{\mathcal{L}}$ co$\mathcal{C}$-schwer/vollständig (jeweils bezüglich R-many-one-Reduktionen).

Beweis: Übungsaufgabe. □

Wir können dieses Lemma nun auf die bereits betrachteten, NL-vollständigen Probleme anwenden.

<u>**Nicht-Existenz von Pfaden**</u> ($\overline{\text{PATH}}$)	
Gegeben:	Gerichteter Graph $G = (V, R)$, Quellknoten $s \in V$, Zielknoten $t \in V$
Entscheide:	Gibt es keinen Pfad von s nach t in G?

Analog lässt sich die Nicht-Existenz von Pfaden in azyklischen Graphen, $\overline{\text{ACYCLICPATH}}$, definieren.

9.2 Korollar
$\overline{\text{PATH}}$ und $\overline{\text{ACYCLICPATH}}$ sind coNL-vollständig.

9.3 Bemerkung
Wenn man PATH als Wortproblem auffasst, ist sein Komplementproblem eigentlich leicht anders: Gegeben eine Eingabe, entscheide ob entweder die Eingabe keine korrekt kodierte Eingabe $G\#s\#t$ für PATH ist, oder ob sie es ist, es aber keinen Pfad von s nach t in G gibt.

Ob die Eingabe korrekt formatiert ist, lässt sich jedoch leicht (deterministisch mit logarithmisch viel Platz, also in L) feststellen. Wir beschränken uns hier daher darauf, korrekt kodierte Eingaben zu untersuchen.

A) 2SAT

Wir interessieren uns insbesondere für die Komplexität von Problemen, deren Eingabe die „Theoretische Informatik I“ bekannten endlichen Automaten bzw. die aus „Einführung in die Logik“ bekannten aussagenlogischen Formeln sind. Hier wollen wir die Erfüllbarkeit von aussagenlogischen Formeln in konjunktiver Normalform betrachten.

Sei $x_0, x_1, x_2, \ldots$ eine abzählbar unendliche Menge von Aussagenvariablen. Ein **Literal** L ist von der Form x_i (positiv) oder $\neg x_i$ (negativ). Eine **Klausel** ist eine Disjunktion von Literalen,

$$C = L_1 \vee \ldots L_k .$$

Eine **aussagenlogische Formel in konjunktiver Normalform (CNF)** ist eine Konjunktion von Klauseln

$$F = C_1 \wedge \ldots \wedge C_n .$$

Belegungen und die Auswertung von Formeln sind dabei wie üblich definiert. Wir identifizieren $0 \mathrel{\widehat{=}} \mathit{false}$, $1 \mathrel{\widehat{=}} \mathit{true}$.

Erfüllbarkeit von aussagenlogischen Formeln in CNF (SAT)	
Gegeben:	Eine Formel F in CNF
Entscheide:	Ist F erfüllbar, d.h. gibt es eine Belegung φ mit $\varphi(F) = \mathit{true}$?

SAT steht für *Satisfiablity*, also Erfüllbarkeit.

Man interessiert sich insbesondere für die Abhängigkeit der Härte dieses Problems von diversen Parametern, z.B. der Größe der Klauseln. Sei $k > 0$ eine natürliche Zahl. Eine Formel F ist in k-CNF, wenn sie in CNF ist, und jede Klausel aus höchstens k Literalen besteht.

Erfüllbarkeit von aussagenlogischen Formeln in k-CNF (kSAT)	
Gegeben:	Eine Formel F in k-CNF
Entscheide:	Ist F erfüllbar, d.h. gibt es eine 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.

9.4 Theorem
Das Problem 2SAT ist coNL-vollständig.

Es sind zwei Dinge zu zeigen:

- „Membership“: 2SAT ist in coNL. Wir beweisen, dass das Komplementproblem $\overline{\text{2SAT}}$ in NL liegt.
- „Hardness“: 2SAT ist coNL-schwer. Wir reduzieren $\overline{\text{ACYCLICPATH}}$.

9.5 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

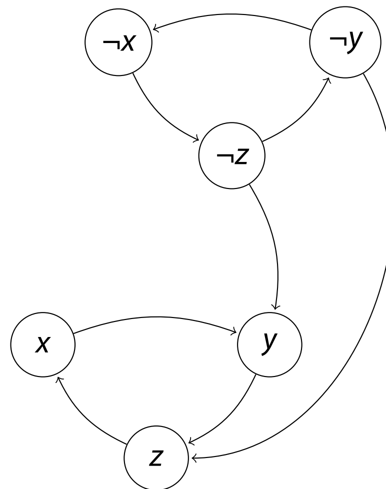
9.6 Beispiel
Sei

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

Wir konstruieren G_F und erhalten

$$\begin{aligned} G_F &= \{x, y, z, \neg x, \neg y, \neg z\} \\ E_F &= \{(x, y), (\neg y, \neg x), (y, z), (\neg z, \neg y), (z, x), (\neg x, \neg z), (\neg z, y), (\neg y, z)\} \,. \end{aligned}$$

Die Folgende Abbildung stellt den Graphen dar.



Das Folgende Lemma ist wichtig, um die Korrektheit unserer Reduktion nachzuweisen.

9.7 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

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

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 ¬L gibt
    for Literal L', das von L aus erreichbar ist, also L →* L' do
        Setze φ(L') = 1.
    end for
    for Literal L', von dem aus ¬L erreichbar ist, also L' →* ¬L do
        Setze φ(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.

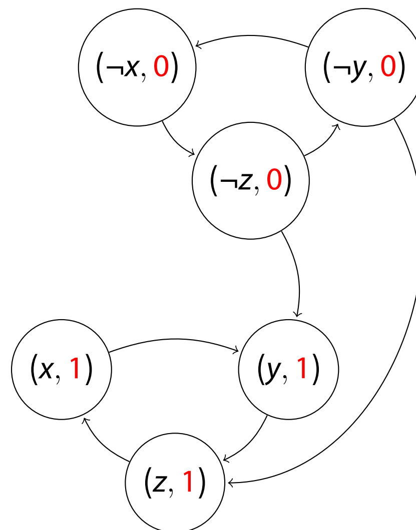
  Es gelte also $L \to^* L'$ und $L \to^* \neg L'$. Aufgrund der Symmetrie im Graphen gilt dann auch $\neg L' \to \neg L$ und $L' \to^* \neg L$. Wir erhalten $L \to^* L' \to \neg L$, ein Widerspruch zur Wahl von L im Algorithmus.

- 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.

□

9.8 Beispiel

Wir setzen das obige Beispiel fort und konstruieren eine erfüllende Belegung wie im Beweis.



Wir können das Lemma nun beweisen. Wie bereits gesagt beweisen wir, dass das Komplementproblem in NL ist.

Beweis:
Der folgende Algorithmus löst $\overline{\text{2SAT}}$ in NL.

Eingabe: 2CNF F
Ausgabe: true falls F unerfüllbar

```
Konstruierte G_F.
for Variable x in F do
    if G#x#¬x ∈ PATH und G#¬x#x ∈ PATH then
        return true
    end if
end for
return false
```

□

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

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

Sei $G\#s\#t$ eine $\overline{\text{ACYCLICPATH}}$ Instanz. Wir konstruieren eine 2-CNF formal F wie folgt: Die Variablen von F sind die Knoten des Graphen

$$Vars = \{x \mid x \text{ ist Knoten von } G\}$$

Für jede Kante $x \to y$ des Graphen G führen wir eine Klausel $\neg x \vee y$ ein. Darüber hinaus fügen wir die Klauseln s und $\neg t$ für Quell- und Zielknoten ein.

Es gilt:

F erfüllbar ist genau dann, wenn es keinen Pfad von s nach t in G gibt.

Die Beweis dieser Aussage sei der Leserin/dem Leser als Übungsaufgabe überlassen.

Die Konstruktion der Formel lässt sich mit logarithmischem Platz realisieren. □

B) Der Satz von Immerman und Szelepcsényi

Die Frage ist nun, ob $\overline{\text{PATH}}$, 2SAT, ... auch NL-vollständig sind. In diesem Fall würde coNL = NL gelten.

Die Frage war lange ungeklärt (die Komplexitätstheorie hat ihre Anfänge 1965), aber man glaubte, dass coNL ≠ NL gelten würde. Überraschenderweise konnten Neil Immerman (Professor an der University of Massachusetts Amherst) und Róbert Szelepcsényi (Student in Bratislava in der Slovakei) unabhängig voneinander 1987 zeigen, dass Gleichheit gilt. Nicht nur das Resultat war überraschend, sondern der Beweis führte auch eine neue Beweistechnik ein, das **induktive Zählen**. Sie wurden für ihr Resultat 1995 mit dem Gödel-Preis ausgezeichnet.

9.10 Theorem: Immerman and Szelepcsényi, 1987
Für eine Platzschranke s mit $s(n) \geqslant \log n \forall n \in \mathbb{N}$ gilt

$$\text{NSPACE}(s) = \text{coNSPACE}(s) .$$

9.11 Korollar
NL = coNL .

Wir zeigen zunächst das folgende Theorem.

9.12 Theorem
$\overline{\text{PATH}} \in \text{NL}$.

Unter Verwendung des Theorems lässt sich der Satz von Immerman und Szelepcsényi zeigen: Hierzu wenden wir das Theorem auf den Konfigurationsgraphen einer gegebenen Maschine anwendet (siehe den Beweis von Theorem 8.14). Eine nicht-determinische Maschine akzeptiert Eingabe x nicht, wenn es im Konfigurationsgraphen zu Eingabe x **keinen** Pfad von der initialen zu einer akzeptierenden Konfiguration gibt.

Es genügt also, Theorem 9.12 zu beweisen. Wir müssen für einen Graphen G und Knoten s, t überprüfen, ob t nicht von s aus erreichbar ist.

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, ist in NL möglich. Alle solchen Knoten zu speichern kostet allerdings mehr als logarithmisch viel Platz.

Der Beweis löst dieses Problem indirekt, in dem 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 (in NL)

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

Schritt 1: Nicht-Erreichbarkeit unter Verwendung von N

Wir gehen im folgenden davon aus, dass

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

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

9.13 Algorithm: unreach
unreach(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** $count \neq N$ **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**

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

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

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

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

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

Schritt 2: Induktives Zählen

Wir wollen die Zahl

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

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

Man beachte:

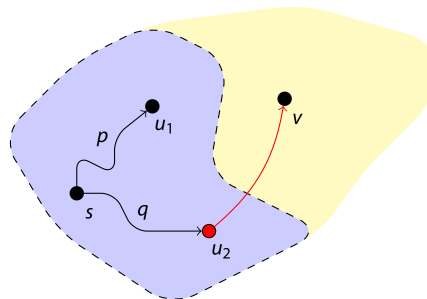
- Es gilt $R(n) = N$, da jeder erreichbare Knoten auch mit einem einfachen Pfad (der insbesondere Länge $\leqslant n$ hat) erreichbar ist.
- Es gilt $R(0) = 1$, da nur s selbst mit einem Pfad der Länge 0 erreichbar ist.

- Jeder Knoten v, der in i Schritten erreichbar ist, ist Nachfolger eines Knotens u, der in $i - 1$ Schritten erreichbar ist.

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

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

Die folgende Grafik stellt diese Idee dar.



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

9.15 Algorithm: #reach

```
# reach(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)
```

9.16 Lemma

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

Beweis:

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

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

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

Finaler Algorithmus

9.17 Algorithm: Algorithmus für $\overline{\text{PATH}}$

nopath(G,s,t)

  N = #reach(G,s)

  return unreach(G,s,t)

9.18 Lemma

nopath(G,s,t) löst $\overline{\text{PATH}}$ in NL.

Beweis:

Die Korrektheit ergibt sich direkt aus der Korrektheit von #reach und unreach.

Es bleibt zu begründen, warum beide Algorithmen mit logarithmischem Platzverbrauch implementiert werden können. Hierzu ist es zum einen wichtig, die Pfade „on-the-fly“ zu raten, also immer nur einen Zähler für die Länge und beiden aktuellsten Knoten, aber nicht den gesamten Pfad, zu speichern. Zum anderen müssen während der Berechnung von $\#reach$ nur $R(i-1)$ und $R(i)$ gespeichert werden, man kann also Speicherplatz wiederverwenden. □

9.19 Bemerkung

Im Theorem von Immerman & Szelepcsényi haben wir gefordert, dass $s(n) \geqslant \log n$ gilt. Wenn dies nicht der Fall ist, haben wir nicht genug Platz für die Zählvariablen, die im Algorithmus benötigt werden.

10. P

Unser Ziel ist es, zu verstehen, welche Probleme in polynomieller Zeit gelöst werden können. Wir beginnen mit der Klasse P. Diese enthält alle Probleme, welche durch eine deterministische Turing Maschine entschieden werden, die polynomielle Laufzeit hat. Üblicherweise lassen sich Probleme in P als Auswertungen oder Überprüfungen formulieren. Dies beinhaltet das Prüfen der Korrektheit von Beweisen oder die Evaluation einer Funktionen an einem bestimmten Wert. Konkret werden wir ein erstes P-vollständiges Problem betrachten. Der Beweis der Vollständigkeit geht auf Ladner zurück.

Das Circuit Value Problem

Unser Ziel ist es, das Circuit Value Problem (CVP) zu definieren. Wir beginnen mit der Definition von Schaltkreisen (Circuits).

10.1 Definition
Ein **Boolescher Schaltkreis (Circuit)** ist ein Programm, das aus endlich vielen Zuweisungen der folgenden Form besteht:

$$P_k = 0 \mid 1 \mid P_i \vee P_j \mid P_i \wedge P_j \mid \neg P_i,$$

mit $i, j \in \mathbb{N}$ und $i, j < k$.

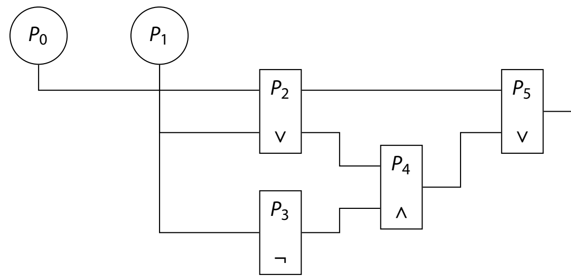
Jedes P_k darf dabei nur ein Mal definiert werden, also nur ein mal auf der linken Seite einer Zuweisung stehen.

10.2 Beispiel
Ein einfacher Schaltkreis ist gegeben durch folgende Zuweisungen:

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

Wir können Schaltkreise als gerichtete Graphen auffassen. Dabei sind die P_k, welchen ein Wert 0 oder 1 zugewiesen wird die Inputsignale. Die anderen P_k stellen Gatter dar, die bis zu zwei

Inputs haben und diese mit ¬, ∨ oder ∧ kombinieren. Dies erinnert dann an Schaltkreise aus der Elektrotechnik. Für obiges Beispiel ergibt sich der folgende Graph:

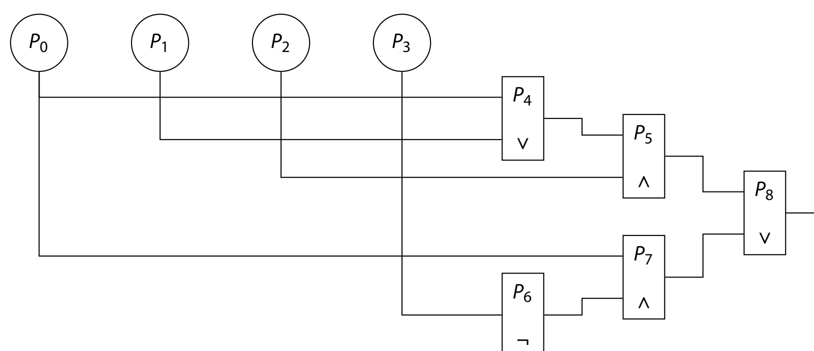


Besonders ist hierbei, dass P_2 zwei Ausgänge hat. Es kann also passieren, dass ein P_i in mehreren P_k mit $k > i$ verwendet wird. Dies ist ein wichtiger Unterschied zu Booleschen Formeln. Wenn wir Boolesche Formeln als Schaltkreise auffassen, darf jedes P_i, das nicht auf den Wert 0 oder 1 gesetzt wird, in höchstens einem P_k mit $k > i$ vorkommen. Der Schaltkreis entspricht dann einem Baum.

Wenn wir beispielsweise die Formel $F = ((a \vee b) \wedge c) \vee (\neg d \wedge a)$ als Schaltkreis modellieren, interpretieren wir die Variablen a, b, c und d als Zuweisungen mit festem Wert (obwohl wir diesen noch nicht kennen). Die logischen Operationen können dann einfach übersetzt werden:

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

Der zugehörige, baumartige Graph ist dann:



Wir können also folgern, dass boolsche Formeln spezielle Schaltkreise sind. Dies wird bestätigt durch die Tatsache, dass die Auswertung von boolschen Formeln ein Problem in L ist. Die Auswertung von allgemeinen Schaltkreisen (CVP) dagegen, ist ein Problem in P:

Circuit Value Problem (CVP)	
Gegeben:	Ein boolscher Schaltkreis C als Liste von Zuweisungen $P_0, \ldots, P_\ell$.
Entscheide:	Ist der Wert von P_ℓ gleich 1?

10.3 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 also von oben nach unten durch die Liste der Zuweisungen. Wenn man auf ein neues, unausgewertetes P_k trifft, holt man sich die Werte der P_i mit $i < k$, die in der Zuweisung von P_k verwendet werden. Da diese schon ausgewertet wurden, kann man P_k auswerten. Diese Idee werden wir im ersten Teils des folgenden Theorems verwenden, um zu zeigen, dass CVP in P liegt.

10.4 Theorem: Ladner 1975

CVP ist P-vollständig (bezüglich logspace-many-one-Reduktionen).

Wir teilen den Beweis des Theorems in zwei Schritte auf. Im ersten Schritt zeigen wir, dass CVP in P liegt („Membership"). Im zweiten Schritt, dass CVP auch P-schwer ist („Hardness"). Die Schwierigkeit im zweiten Schritt liegt dabei in der Tatsache, dass es sich um das erste P-schwere Problem handelt: Wir müssen also jedes Problem aus P auf CVP reduzieren.

10.5 Lemma

CVP liegt in P.

Beweis:

Wir konstruieren eine Turing-Maschine M, welche die Idee des obigen Algorithmus umsetzt. Angenommen, M startet auf dem gegebenen Schaltkreis C. Wir nehmen an, dass die Zuweisungen getrennt durch ein Zusatzsymbol $\#$ auf dem Band von M stehen:

\$	P_0	$\#$	P_1	$\#$	P_2	$\#$	$\cdots$

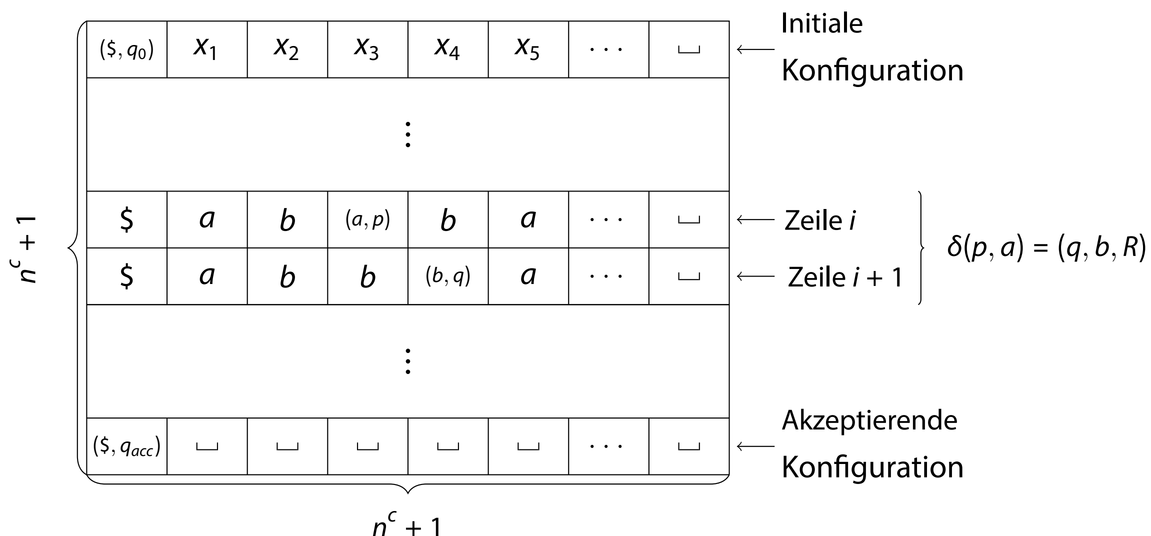
Nun wertet M die P_i von links nach rechts aus. Angenommen, wir haben $P_0, \ldots, P_i$ schon ausgewertet und wollen nun P_{i+1} auswerten. Dann bewegt M seinen Kopf zu P_{i+1}. P_{i+1} besteht aus zwei (oder einem) P_j, P_e mit $j, e \leqslant i$. Da P_j und P_e schon ausgewertet wurden, kann M seinen Kopf nach links bewegen und die Werte abfragen. Danach setzt die Maschine die Werte in P_{i+1} ein und wertet aus. Insgesamt ergibt sich ein Zeitaufwand von $\mathcal{O}(n^2)$. □

10.6 Lemma
CVP ist P-schwer (bezüglich logspace-many-one-Reduktionen).

Beweis:
Wir reduzieren ein beliebiges Problem A aus P auf CVP. Das Problem A ist die Sprache einer deterministischen Turing-Maschine M, die n^c-Zeit beschränkt ist. Hierbei ist c eine Konstante.

Im Folgenden beschreiben wir die Idee des Beweises. Sei Q die Zustandsmenge von M, Δ das Bandalphabet und δ die Transitionsrelation. Ferner, sei x ein Input von M der Länge n. Eine Berechnung von M auf x ist eine Folge von $n^c + 1$ Konfigurationen. Jede dieser Konfigurationen belegt höchstens $n^c + 1$ Zellen auf dem Band von M. Wir können die Konfigurationen also in eine Matrix der Größe $(n^c + 1) \times (n^c + 1)$ schreiben.

Die i-te Zeile der Matrix beschreibt dann die i-te Konfiguration. Dies entspricht einem String der Länge n^c+1 über $(\Delta \cup (\Delta \times Q))$. Man beachte, dass nur ein Eintrag pro Zeile in $(\Delta \times Q)$ liegen darf. Dieser beschreibt die Kopfposition und den aktuellen Zustand. Die j-te Spalte der Matrix beschreibt den Inhalt der j-ten Zelle von M in jeder Konfiguration. Ein Sprung von Zeile i zu $i + 1$ entspricht dem Ausführen einer Transition.



Man kann erkennen, dass sich der Bandinhalt pro ausgeführter Transition nur an wenigen Stellen ändert. Die Berechnung von M ist also in gewissem Sinne **lokal**. Diese Lokalität erlaubt es uns, die Matrix und damit die Berechnung von M als Schaltkreis zu kodieren.

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

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

Wir erhalten diese Variablen für $0 \leqslant i, j \leqslant n^c$, $a \in \Delta$, $p \in Q$.

Wir beginnen mit der Kodierung der ersten Zeile der Matrix, der initialen Konfiguration. Wir setzen $P_{00}^{\$} = 1$. Dies sichert uns, dass das Symbol \$ zu Beginn ganz links auf dem Band von M steht. Weiter setzen wir $P_{00}^{b} = 0$ für alle $b \in \Delta \setminus \{\$\}$. Hiermit stellen wir sicher, dass an der Stelle nur das \$-Symbol steht und kein anderes Symbol.

Um die Eingabe x zu kodieren, setzen wir für $j = 1, \ldots, n$: $P_{0j}^{x_j} = 1$ und $P_{0j}^{b} = 0$, wobei $b \in \Delta \setminus \{x_j\}$. Nun fehlen noch die $\sqcup$-Symbole, die den Rest des Bandes von M füllen. Dazu setzen wir für $j = n+1, \ldots, n^c$: $P_{0j}^{\sqcup} = 1$ und $P_{0j}^{b} = 0$, wobei $b \in \Delta \setminus \{\sqcup\}$.

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 $j = 1, \ldots, n^c$, $q \in Q$. Den letzten Schritt benötigen wir, um die Kopfposition eindeutig zu halten.

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:

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

Den Zustand von M können wir anhand der Transition ändern. Für die Kopfposition ergibt sich Folgendes: Der Kopf steht in Konfiguration $i+1$ in Zelle j, wenn er in Konfiguration i in Zelle $j-1$ stand und M ihn nach rechts bewegt hat, oder wenn er in Konfiguration i in Zelle $j+1$ stand und M ihn nach links bewegt hat.

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

Man beachte, dass man Zelle 0 nur von rechts erreichen kann, also nur durch Kopfbewegung nach links, und Zelle n^c nur von links, also nur durch Kopfbewegung nach rechts. Die entsprechenden Zuweisungen ändern sich dann geringfügig.

Wir können annehmen, dass Konfiguration n^c folgende Form hat: Ganz links steht das Symbol \$. Rechts davon ist das Band mit $\sqcup$-Symbolen aufgefüllt. Die Konfiguration ist akzeptierend, wenn M im Zustand q_{acc} ist. Andernfalls ist M im Zustand q_{rej} und die Konfiguration ist abweisend. Demnach ergibt sich: M akzeptiert x genau dann, wenn $Q_{n^c,0}^{q_{acc}} = 1$. Dies ist die letzte

Zuweisung im Schaltkreis. Daher akzeptiert x genau dann, wenn der Schaltkreis, der über die P_{ij}^{a} und Q_{ij}^{p} definiert wurde, eine positive CVP-Instanz ist.

Diesen Schaltkreis kann man in logarithmischem Platz konstruieren, es handelt sich hierbei also um eine logspace-Reduktion. □

In der Definition von Circuits haben wir in den Zuweisungen an Variablen P_k nur binäre Konjunktionen und Disjunktionen erlaubt. Im Beweis haben wir komplexere Formeln als rechte Seite von Zuweisungen verwendet. Durch das Einführen von Hilfsvariablen lassen sich diese erweiterten Zuweisungen in die initial geforderte Form bringen.

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, haben typische Problem in P die Aufgabe einen gegebenen Beweis zu prüfen. Durch die Hinzunahme des Nicht-Determinismus ist es dann die Aufgabe der Probleme in NP, Beweise zu finden. Intuitiv können solche Probleme in zwei Schritten gelöst werden: (1) ein Beweis wird nicht-deterministisch geraten, (2) der geratene Beweis wird überprüft. Wir werden sehen, dass sich diese Intuition mit Hilfe von Zertifikaten formulieren lässt und eine alternative Charakterisierung der Klasse NP ergibt. Wir beginnen jedoch mit einem NP-vollständigen Problem, welches auf Cook und Levin zurückgeht.

A) NP-vollständige Probleme

Wir werden zunächst zeigen, dass SAT NP-vollständig ist. Bevor wir aber mit dem Beweis beginnen, brauchen wir folgendes Hilfsmittel:

11.1 Definition
Ein boolscher Schaltkreis **mit variablem Input** ist ein boolscher Schaltkreis, in welchem wir die zusätzliche Zuweisung

$$P_k := ?$$

erlauben. Diese gibt an, dass wir den Wert für P_k nicht kennen, er kann also 0 oder 1 sein.

Sei C ein Schaltkreis mit variablen Inputs $P_1, \dots, P_k$ und $y_1, \dots, y_k \in \{0, 1\}$. Wir schreiben $C(y_1, \dots, y_k)$ für den Schaltkreis, den wir erhalten, wenn wir anstatt der Zuweisung $P_i = ?$ die Zuweisung $P_i = y_i$ für $i = 1, \dots, k$ einsetzen. Weiter schreiben wir $C(y_1, \dots, y_k) = 1$, falls der Schaltkreis $C(y_1, \dots, y_k)$ eine positive CVP-Instanz ist.

Wir haben schon festgestellt, dass das Auswerten von boolschen Formeln effizienter ist (L) als das Auswerten von Schaltkreisen (P). Das Testen der Erfüllbarkeit hat jedoch für beide Modelle dieselbe Komplexität.

11.2 Bemerkung
Sei C ein Schaltkreis mit Zuweisungen $P_1, \dots, P_\ell$ und variablen Inputs $P_1, \dots, P_k$. Wir konstruieren eine boolsche Formel F_C wie folgt: Für jede Zuweisung P_i erhalten wir eine Variable x_i. Wenn P_i eine Zuweisung der Form $P_i = E$ ist, wobei $E \neq ?$, fügen wir eine Klausel $x_i \leftrightarrow E$ zu F_C hinzu. Um den Ausdruck E in der Formel F_C zu verwenden, ersetzen wir in E alle P_i durch die

entsprechenden Variablen x_i. Für die letzte Zuweisung P_k hängen wir dann noch die Variable x_k mit einer Konjunktion an:

$$F_C = x_k \wedge \bigwedge_{P_i = E, E \neq ?} (x_i \leftrightarrow E).$$

Nun gilt, dass F_C erfüllbar ist, genau dann, wenn es $y_1, \dots, y_k \in \{0,1\}$ gibt, sodass $C(y_1, \dots, y_k) = 1$.

11.3 Beispiel
Verwendet man die obige Konstruktion für folgenden Schaltkreis C:

$$\begin{aligned}
P_0 &= ? \\
P_1 &= ? \\
P_2 &= P_0 \vee P_1 \\
P_3 &= \neg P_1 \\
P_4 &= P_2 \wedge P_3,
\end{aligned}$$

ergibt sich die Formel $F_C = (x_2 \leftrightarrow x_0 \vee x_1) \wedge (x_3 \leftrightarrow \neg x_1) \wedge (x_4 \leftrightarrow x_2 \wedge x_3) \wedge x_4$. Erfüllende Inputs $P_0 = 1$ und $P_1 = 0$ für C lassen sich nun zu einer erfüllenden Belegung von F_C erweitern: Die Variablen x_0 und x_1 werden wie die Zuweisungen P_0 und P_1 auf 1 und 0 gesetzt. Der Wert der verbleibenden Variablen kann dann errechnet werden: $x_2 \to 1, x_3 \to 1, x_4 \to 1$.

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

Wie für das CVP, teilen wir auch hier den Beweis in zwei Schritte auf. Der erste Schritt, **Membership** in NP, ist dabei schnell einzusehen: Ein Algorithmus für SAT rät zu einer gegebenen Formel F eine Belegung und prüft nach, ob diese erfüllend ist. Dazu muss man die geratene Belegung nur in die Formel einsetzen und diese auswerten. Dieses Verfahren läuft in Polynomialzeit, daher gilt SAT $\in$ NP.

Im zweiten Schritt zeigen wir, dass SAT NP-schwer ist.

11.5 Lemma
SAT ist NP-schwer.

Beweis:
Sei A ein Problem aus NP. Dann gibt es eine nicht-deterministische Turing-Maschine M mit $A = L(M)$, die n^c-Zeit beschränkt ist. Wir könne annehmen, dass sich M während einer Be-

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

rechnung immer höchstens zwischen zwei Transitionen entscheiden muss (binär verzweigt). Andernfalls könnten wir dies mit polynomieller Mehrzeit immer erzwingen.

Sei x nun ein Input von M. Eine Berechnung von M auf x ist ein Pfad im binären Berechnungsbaum. Solch ein Pfad wird durch einen String $y \in \{0,1\}^{n^c}$ beschrieben. Dabei steht die i-te Stelle y_i für die Wahl der Transition in Schritt i. Für ein Beispiel, siehe Abbildung 1.

Sei M' nun die deterministische Turing-Maschine, die als Input $x\#y$ mit $|y| = n^c$ erwartet und M auf x simuliert. Jede nicht-deterministische Entscheidung von M wird dann durch y aufgelöst: M' wählt in der Simulation von M in Schritt i die y_i-te Transition aus. Es gilt daher: M akzeptiert x genau dann, wenn es ein $y \in \{0,1\}^{n^c}$ gibt, sodass M' die Eingabe $x\#y$ akzeptiert.

Da M' deterministisch ist, können wir nun, wie im Satz von Ladner (Theorem 10.4), einen Schaltkreis C konstruieren. Dieser modelliert die Berechnung von M' auf $x\#y$. Es gilt dann: M' akzeptiert $x\#y$ genau dann, wenn C eine positive CVP-Instanz ist. Seien $P_1, \dots, P_{n^c}$ die Zuweisungen, welche y modellieren. Wenn wir diese unbestimmt machen, also durch die Zuweisungen $P_1 =?, \dots, P_{n^c} =?$ ersetzen, erhalten wir einen neuen Schaltkreis C'. Für diesen gilt dann: M akzeptiert x genau dann, wenn es ein $y \in \{0,1\}^{n^c}$ gibt, sodass $C'(y_1, \dots, y_{n^c}) = 1$.

Nun übersetzen wir den Schaltkreis C' in die Formel $F_{C'}$, wie in Bemerkung 11.2. Mit der Formel erhalten wir dann folgende Äquivalenz: M akzeptiert x genau dann, wenn $F_{C'}$ erfüllbar ist. Also haben wir ein beliebiges Problem A in NP auf die Erfüllbarkeit einer boolschen Formel zurückgeführt. □

Betrachten wir nun ein Problem aus der Welt der Graphen: Wir suchen nach einem hamiltonschen Kreis, einem Kreis, der alle Knoten eines Graphen genau ein mal besucht. Wir werden zeigen, dass auch dieses Problem NP-vollständig ist. Dazu verwenden wir eine Reduktion von SAT. Widmen wir uns aber zunächst den benötigten Definitionen:

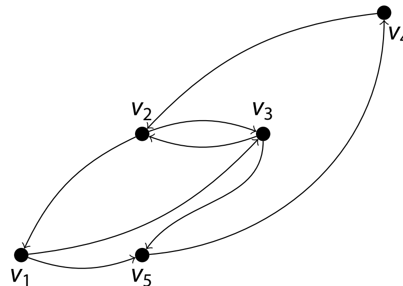
11.6 Definition

Sei $G = (V, E)$ ein gerichteter Graph mit $|V| = n$. Ein **hamiltonscher Kreis** in G ist ein Pfad

$v_{i_1}, \ldots, v_{i_n}$, sodass $v_{i_j} \neq v_{i_k}$ für $j \neq k$ und es gibt eine Kante von v_{i_n} nach v_{i_1}. In anderen Worten: Es handelt sich um einen Kreis, welcher jeden Knoten genau ein mal besucht.

11.7 Beispiel

Der folgende Graph hat den hamiltonschen Kreis: v_1, v_3, v_5, v_4, v_2.



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

<u>**Hamiltonian Cycle**</u>	
Gegeben:	Ein gerichteter Graph $G = (V, E)$.
Entscheide:	Gibt es einen hamiltonschen Kreis in G?

11.8 Theorem

Hamiltonian Cycle ist NP-vollständig.

Der Beweis gliedert sich wieder in zwei Schritte. Wie bei SAT kann man auch hier die Zugehörigkeit zu NP schnell einsehen: Sei $G = (V, E)$ der gegebene Graph. Eine nicht-deterministische Turing-Maschine für Hamiltonian Cycle rät ein Folge $v_1, \ldots, v_n$ von Knoten in V und schreibt diese auf das Band. Nun wird verifiziert, dass es sich dabei um einen hamiltonschen Kreis handelt: Zuerst wird getestet, ob alle Knoten vorkommen, also ob je zwei der geratenen Knoten verschieden sind. Dies kann in $\mathcal{O}(n^2)$ ermittelt werden. Dann wird getestet, ob es von v_i nach v_{i+1}, mit $i = 1, \ldots, n-1$ und von v_n nach v_1 je eine Kante gibt. Ist dies erfüllt, akzeptiert die Turing-Maschine. Falls nicht, weist sie ab. Die Maschine läuft in $\mathcal{O}(n^2)$ Zeit.

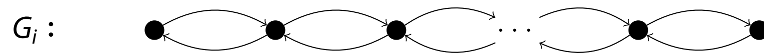
11.9 Lemma

Hamiltonian Cycle ist NP-schwer.

Beweis:

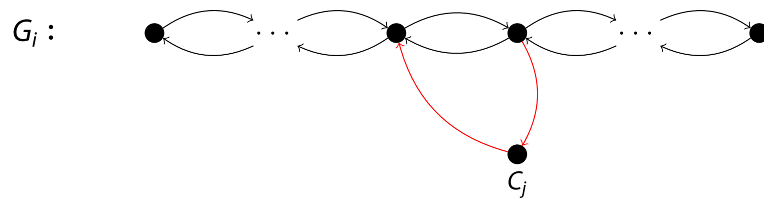
Wir reduzieren SAT auf Hamiltonian Cycle. Sei dazu F eine SAT-Instanz, also eine Formel in CNF. Seien $C_1, \ldots, C_m$ die Klauseln von F und $x_1, \ldots, x_k$ die Variablen. Die Formel F hat dann die Form $F = C_1 \wedge \cdots \wedge C_m$.

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

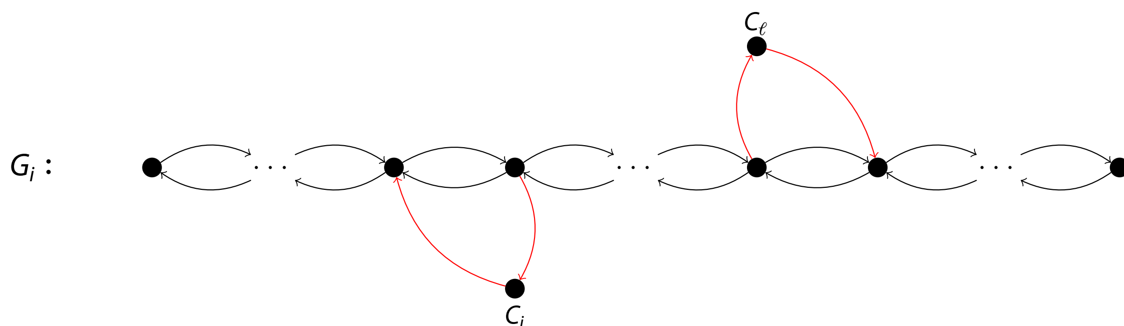


Die Belegung von x_i wird durch einen Durchlauf durch G_i simuliert: Ein Durchlauf von rechts nach links entspricht der Belegung true, ein Durchlauf von links nach rechts entspricht der Belegung false.

Schritt 2: Da man für die Erfüllbarkeit von F alle Klauseln von F erfüllen muss, werden wir die Klauseln nun in die Graphen G_i einbauen. Sei C_j eine Klausel, die x_i enthält. Dann fügen wir einen neuen Knoten in G_i ein. Das Durchlaufen durch diesen Knoten in einem Pfad soll bedeuten, dass wir die Klausel mit der aktuellen Belegung der Variablen x_i erfüllen. Also verbinden wir den Knoten so mit G_i, dass er nur in einem Durchlauf von rechts nach links (true) besucht werden kann. Diese Kanten werden wir im Folgenden rot markieren, um sie von den vorherigen Kanten unterscheiden zu können.



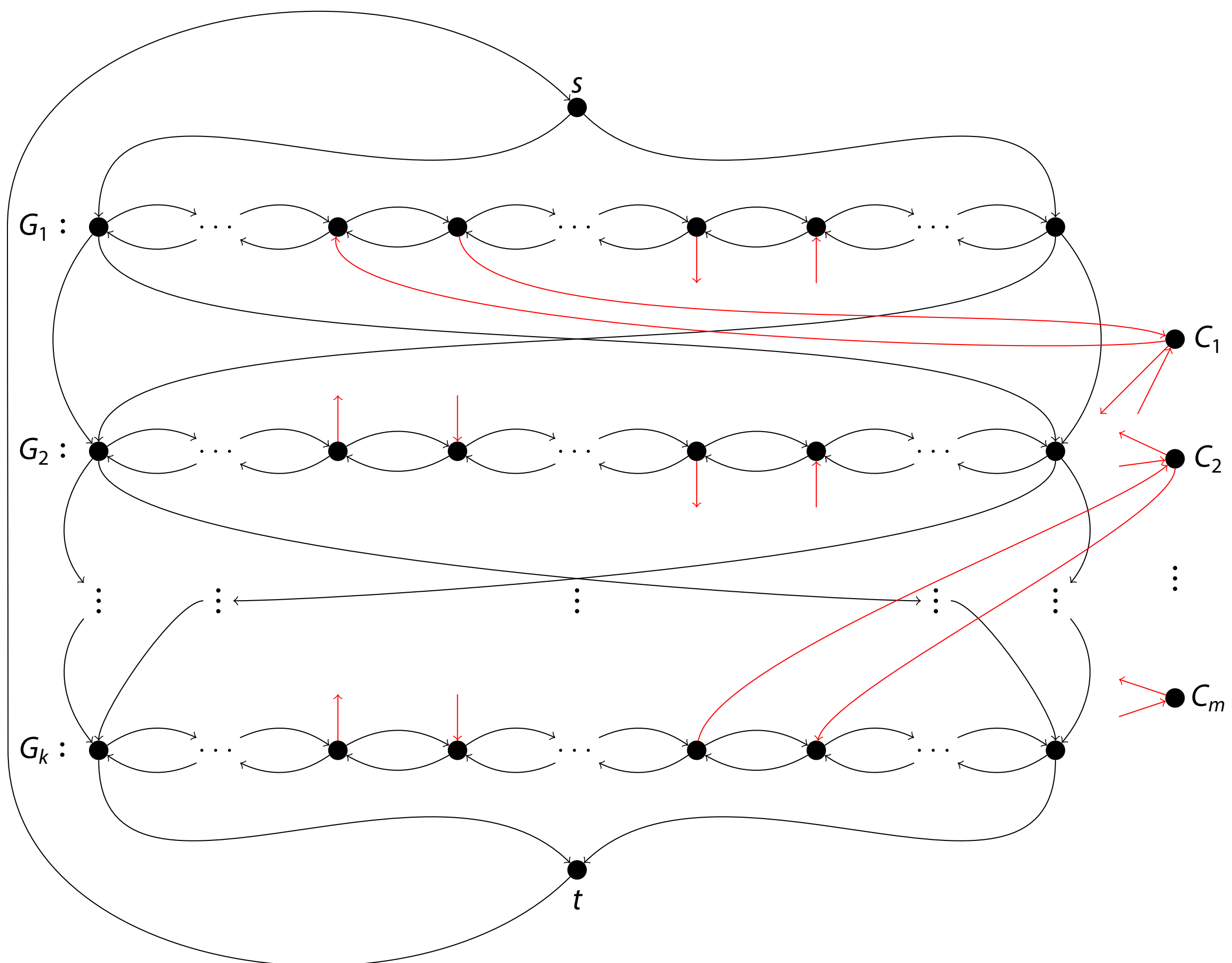
Sei nun C_ℓ eine Klausel, welche $\neg x_i$ enthält. Auch hier fügen wir einen neuen Knoten ein und verbinden diesen so mit G_i, dass er nur durch einen Durchlauf von links nach rechts (false) besucht werden kann:



Da jede Klausel, die x_i oder $\neg x_i$ beinhaltet mit dem Graphen an zwei Knoten verbunden wird und wir später zusätzlich zwei Knoten an den Rändern brauchen, benötigen wir die oben erwähnten $2m_i + 2$ Knoten.

Schritt 3: Nun setzen wir alle Graphen G_i zusammen. Das heißt wir konstruieren einen Graphen G, der aus allen G_i besteht. Dabei werden die Knoten, welche für die Klauseln eingeführt wurden aber nur ein mal für alle Graphen G_i eingeführt und wie oben verbunden. Ein Klauselknoten hat dann eine Verbindung zu mehreren G_i, nämlich genau zu jenen, sodass x_i als positives oder negatives Literal in der Klausel vorkommt.

Zudem führen wir Verbindungen von G_i zu G_{i+1} für $i = 1, \ldots, k-1$ ein. Diese ermöglichen es, nach einem Durchlauf durch G_i in beliebiger Richtung, den Durchlauf durch G_{i+1} in beliebiger Richtung fortzuführen. Intuitiv entspricht dies der Wahl der Belegung von x_i und x_{i+1}. Sobald man x_i auf true oder false gesetzt hat, setzt man x_{i+1} auf true oder false. Die Belegung von x_{i+1} ist nicht von x_i abhängig.



Als letzten Teil der Konstruktion fügen wir einen Initialknoten s und einen Finalknoten t ein und eine Kante (t, s), um einen Kreis zu schließen. Intuitiv beginnt ein hamiltonscher Kreis in G dann in s und endet in t.

Nehmen wir nun an, dass G einen hamiltonschen Kreis hat. Also ist es möglich, durch alle Knoten, auch die Knoten für die Klausen zu laufen. Dies entspricht dann einer erfüllenden Belegung von F. Analog kann jede erfüllende Belegung von F in einen hamiltonschen Kreis von G übersetzt werden. □

B) Zertifikate

Viele Probleme aus NP können mit einem nicht-deterministischen Algorithmus gelöst werden, der zunächst eine „Lösung" rät und danach deterministisch überprüft, dass richtig geraten wurde. Anders ausgedrückt ist das Verifizieren von Ja-Instanzen leicht, wenn ein 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. Im Gegensatz dazu können theoretisch bei den bisher betrachteten Algorithmen sich nicht-deterministische und deterministische Phasen der Berechnung beliebig abwechseln.

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

11.10 Definition
Ein **Verifizierer** ist eine deterministische, totale Turing-Maschine $\mathcal{V}$ mit zwei Eingabebändern, einem über Alphabet Σ und einem zweiten Zertifikatsband über $\{0, 1\}$.

Die Sprache von $\mathcal{V}$ ist die Menge aller Wörter $x \in \Sigma^*$, so dass es ein **Zertifikat** $y \in \{0, 1\}^*$ gibt, so dass $\mathcal{V}$ die Eingabe (x, y) akzeptiert (d.h. $\mathcal{V}$ akzeptiert, wenn initial das erste Eingabeband x und das zweite Eingabeband y beinhaltet),

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

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

Im folgenden sind wir in **Polynomialzeit-Verifizierer** interessiert. Im Gegensatz zu normalen Verifizierern ist hier ein Wort x nur in der Sprache, wenn es ein polynomiell langes Zertifikat y gibt, also y mit $|y| \in \mathcal{O}\left(|x|^k\right)$, wobei die Konstante k von x unabhängig ist. Desweiteren verlangen wir, dass $\mathcal{V}$ auf einer Eingabe x und einem polynomiell großen Zertifikat auch in polynomieller Zeit hält.

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

Bevor wir den Satz zeigen, möchten wir zunächst einiger Anwendung des Resultats sehen. Wie bereits oben angesprochen kann man bei SAT eine erfüllende Belegung, welche für Ja-Instanzen existieren muss, als Zertifikat verwenden.

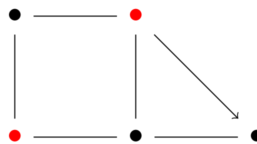
11.12 Beispiel

Independent Set

Gegeben:	Ein gerichteter Graph $G = (V, E)$ und eine k
Entscheide:	Gibt es in G ein Independent Set der Größe k?

Hierbei ist ein **Independent Set** der Größe k eine Menge $V' \subseteq V$ aus k Knoten, so dass es keine Kante zwischen zwei Knoten aus V' gibt.

Im folgenden Graphen bilden die rot markierten Knoten ein Independent Set der Größe 2.



Ein Polynomialzeit-Verifizierer für Independent Set überprüft, ob auf dem Zertifikatsband (die Kodierung von) k Knoten steht, so dass es zwischen diesen Knoten keine Kanten gibt. Das Zertifikat ist also das Independent Set selbst. Mit Hilfe von Theorem 11.11 ist also gezeigt, dass Independent Set in NP liegt.

11.13 Beispiel

Subset Sum

Gegeben:	Ein Menge von Zahlen $N = \{n_1, \ldots, n_k\}$ und eine Zahl S
Entscheide:	Gibt es eine Teilmenge $N' \subseteq N$, so dass die Summe der Zahlen in N' die Zahl S ergibt, $S = \sum_{n \in \mathbb{N}'} n$?

Ein Polynomialzeit-Verifizierer für Subset Sum überprüft, ob auf dem Zertifikatsband (die Kodierung von) Zahlen steht, die in der ursprünglichen Menge enthalten sind, berechnet ihre Summe und vergleicht das Ergebniss mit der gegebenen Zahl S. Das Zertifikat ist also die gesuchte Teilmenge. Mit Hilfe von Theorem 11.11 ist also gezeigt, dass Subset Sum in NP liegt.

11.14 Bemerkung
Tatsächlich sind beide Probleme, Independent Set und Subset Sum, sogar NP-vollständig.

Beweis von Theorem 11.11:
Wir müssen zeigen, wie sich eine polynomiell zeitbeschränkte NTM in einen Polynomialzeit-Verifizierer umwandeln lässt und umgekehrt.

Nehmen wir zunächst an, dass ein Polynomialzeit-Verifizierer $\mathcal{V}$ gegeben ist. Wir konstruieren eine NTM $\mathcal{M}$, die sich auf Eingabe x wie folgt verhält:

1. Rate auf einem Arbeitsband das Zertifikat y mit $|y| \in \mathcal{O}\left(|x|^k\right)$. Hierbei ist k die Konstante für $\mathcal{V}$.
2. Simuliere $\mathcal{V}$ auf (x, y).

Es ist leicht einzusehen, dass $\mathcal{M}$ nur polynomiell viel Zeit braucht und $\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{V})$ gilt.

Für die andere Richtung nehmen wir an, dass eine NTM $\mathcal{M}$ gegeben ist, die höchstens polynomiell viel Zeit braucht. Wir können ohne Beschränkung der Allgemeinheit annehmen, dass es in jedem Schritt höchstens zwei mögliche Transitionen gibt. (Ähnliche Annahmen haben wir bereits in anderen Beweisen verwendet und begründet.)

Wir wissen außerdem, dass es eine Konstante k gibt, so dass jede Berechnung von $\mathcal{M}$ zu einer Eingabe x in höchstens $\mathcal{O}\left(|x|^k\right)$ Schritten hält. Wir konstruieren einen Polynomialzeit-Verifizierer $\mathcal{V}$, der sich auf Eingabe x wie folgt verhält:

1. Lese in Schritt i den Wert $y_i \in \{0, 1\}$, d.h. den i-ten Eintrag des Zertifikatsbands.
2. Falls dieser Eintrag nicht existiert, weise ab.
3. Ansonsten simuliere $\mathcal{M}$ auf der aktuellen Konfiguration und verwende y_i um den Nichtdeterminismus aufzulösen: Falls $y_i = 0$, wähle die erste Transition, falls $y_i = 1$ die zweite.
4. Wenn das Ergebnis eine akzeptierende oder abweisende Konfiguration ist, akzeptiere oder weise ab.

Da $\mathcal{V}$ höchstens einen Schritt von $\mathcal{M}$ pro Eintrag auf dem Zertifikatsband simuliert und wir nur polynomiell lange Zertifikate betrachten, ist klar, dass $\mathcal{V}$ höchstens polynomiell lange läuft. Wir argumentieren, dass $\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{V})$ gilt:

- Wenn $x \in \mathcal{L}(\mathcal{M})$ gilt, dann gibt es eine akzeptierende Berechnung zu x polynomielle Länge. Wenn wir die Wahlen von Transitionen, die entlang dieser Berechnung getroffen werden, zu einer Sequenz y kodieren und als Zertifikat verwenden, wird $\mathcal{V}$, gestartet auf (x, y), genau diese Berechnung von $\mathcal{M}$ simulieren und akzeptieren. Es gilt also $x \in \mathcal{L}(\mathcal{V})$.

- Wenn $x \in \mathcal{L}(\mathcal{V})$ gilt, dann gibt es ein polynomiell langes Zertifikat y, so dass $\mathcal{V}$, gestartet auf (x, y), akzeptiert. Gemäß der Konstruktion simuliert $\mathcal{V}$ allerdings $\mathcal{M}$, wobei y zum Auflösen des Nichtdeterminismus verwendet wird. Wenn $\mathcal{V}$ akzeptiert, bedeutet dies insbesondere, dass es eine akzeptierende Berechnung von $\mathcal{M}$ zu Eingabe x gibt; damit gilt $x \in \mathcal{L}(\mathcal{M})$.

$\square$

12. PSPACE und der Satz von Savitch

13. Der Satz von Savitch

Das wichtigste offene Problem der Theoretischen Informatik ist P $\stackrel{?}{=}$ NP, oder Allgemeiner formuliert, die Beziehung zwischen deterministischen und nicht-deterministischen Zeitkomplexitätsklassen. Ähnliche Fragen stellen sich auch für die Platzkomplexitätsklassen z.B. PSPACE $\stackrel{?}{=}$ NPSPACE. Diese wurden jedoch bereits 1970 von Walter Savitch gelöst: Gleichheit gilt. In diesem Kapitel möchten wir den Beweis seines Satzes nachvollziehen.

13.1 Theorem: Satz von Savitch, 1970
Sei $s\colon \mathbb{N} \to \mathbb{N}$ eine Platzschranke mit $s(n) \geqslant \log n$ für alle n. Es gilt

$$\mathrm{NSPACE}(s) \subseteq \mathrm{DSPACE}\left(s^2\right).$$

Eine nicht-deterministische Maschine lässt sich also so determinisieren, dass sich der Platzverbrauch lediglich quadriert (insbesondere also nur polynomiell erhöht).

13.2 Korollar
Es gilt

$$\mathrm{PSPACE} = \mathrm{NPSPACE} \text{ und } \mathrm{EXPSPACE} = \mathrm{NEXPSPACE}.$$

Das Korollar folgt aus dem Satz von Savitch, da PSPACE und EXPSPACE unter Quadrierung abgeschlossen sind. Als weitere Folgerung erhalten wir auch, dass coNPSPACE = coPSPACE = PSPACE gilt.

13.3 Bemerkung
Der Satz von Savitch zeigt **nicht**, dass L = 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 $s^2 \in \mathcal{O}\left((\log n)^2\right)$. Es gilt jedoch nicht zwangsweise $s^2 \in \mathcal{O}(\log n)$.

Um den Satz zu Beweisen werden wir erneut verwenden, dass sich das Akzeptanzproblem für Turing-Maschinen als Pfadproblem im Konfigurationsgraphen auffassen lässt (vergleiche mit dem Beweis von Theorem 8.14 und Teilkapitel 9 B)). 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 $s(n)^2$.

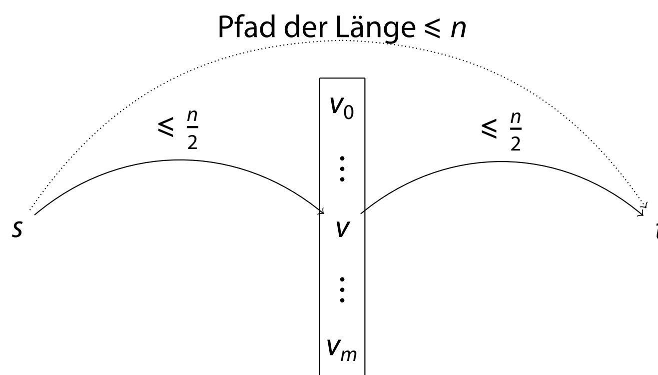
Die Idee hierzu ist die Folgende: Wenn es einen Pfad von Knoten s zu Knoten t gibt, dann gibt es auch einen einfachen Pfad, und damit insbesondere einen Pfad, der höchstens Länge n hat.

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.

13.4 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 $\leqslant k$ gibt

1: **if** $k = 0$ **then**
2:     **return** $(s = t)$
3: **end if**
4: **if** $k = 1$ **then**
5:     **return** $(s \to t)$
6: **end if**
7: **if** $k > 1$ **then**
8:     **for** v Knoten von G **do**
9:         **if** savitch$(G, s, v, \lceil \frac{k}{2} \rceil)$ und savitch$(G, v, t, \lfloor \frac{k}{2} \rfloor)$ **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 $k \leqslant n$ 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.

14. Hierarchie-Sätze

Referenzen

[Gol08] O. Goldreich. *Computational Complexity: A Conceptual Perspective*. Cambridge University Press, 2008.

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

[Koz97] D. C. Kozen. *Automata and Computability*. Springer, 1997.

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

[Neu93] J. v. Neumann. *First Draft of a Report on the EDVAC*. In: IEEE Ann. Hist. Comput. 15.4 (1993), S. 27–75.

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

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

[Sé01] G. Sénizergues. *L(A)=L(B)? decidability results from complete formal systems*. In: Theoretical Computer Science 251.1 (2001), S. 1 –166.

[Sé02] G. Sénizergues. *L(A)=L(B)? A simplified decidability proof*. In: Theoretical Computer Science 281.1 (2002). Selected Papers in honour of Maurice Nivat, S. 555 –608.

[Weg05] I. Wegener. *Complexity Theory: Exploring the Limits of Efficient Algorithms*. Springer, 2005.

Anhang

A. Landau-Notation