Skriptum

Diskrete Mathematik 2

Ein Skriptum zur Vorlesung im Sommersemester 2010

Georg Moser

Sommersemester 2010

Dieses Dokument wurde mit der Hilfe von KOMA-Script und LaTeX erstellt. Für die intensive Betreuung in LaTeX-Fragen bin ich meinem Kollegen Christian Sternagel sehr dankbar.

Inhaltsverzeichnis

1 Einleitung und Motivation **1**

1.1 Endliche Automaten 1

1.2 Endliche Automaten und Sprachen 2

1.3 Strukturelle Repräsentationen 3

1.3.1 Grammatiken 3

1.3.2 Reguläre Ausdrücke 5

1.4 Geschichte 6

2 Endliche Automaten und reguläre Sprachen **9**

2.1 Deterministische endliche Automaten 9

2.2 Nichtdeterministische endliche Automaten 12

2.2.1 Teilmengenkonstruktion 15

2.2.2 Endliche Automaten mit Epsilon-Übergängen 19

2.3 Endliche Automaten und reguläre Ausdrücke 24

2.3.1 Endliche Automaten in regulären Ausdrücken umwandeln 25

2.3.2 Reguläre Ausdrücke in Automaten umwandeln 27

2.4 Algebraische Gesetze für reguläre Ausdrücke 29

2.5 Das “Pumping Lemma” 32

2.6 Abgeschlossenheit regulärer Sprachen 34

2.7 Minimierung von Automaten 38

2.7.1 Zustandsäquivalenz 38

2.7.2 Minimierungsalgorithmus 40

3 Turing Maschinen und Berechenbarkeit **43**

3.1 Einführung in die Berechenbarkeitstheorie 43

3.2 Turingmaschinen 46

3.3 Äquivalente Formulierungen 51

3.3.1 Mehrere Bänder 51

3.3.2 Zweiseitig Unbeschränkte Bänder 52
3.3.3 Nichtdeterminismus 53
3.3.4 Zwei Keller 55
3.3.5 Registermaschinen 56
3.3.6 Aufzählmaschinen 57
3.4 Universelle Maschinen und Diagonalisierung 57
3.4.1 Universelle Turingmaschinen 58
3.4.2 Unentscheidbarkeit des Halteproblems 59

4 Komplexitätstheorie 63
4.1 Einführung in die Komplexitätstheorie 63
4.2 Laufzeitkomplexität 66
4.3 Die Klassen P und NP 67
4.4 Many-One Reduktionen 69
4.5 Logarithmisch Platzbeschränkte Reduktionen 71
4.6 Speicherplatzkomplexität 73

Vorwort

Absolventinnen und Absolventen des Moduls “Diskrete Mathematik” sollen die Inhalte der Vorlesung verstehen sowie diese wiedergeben und anwenden können. Sie sollen die Fähigkeit erworben haben, sich ähnliche Inhalte selbständig zu erarbeiten. Weiters sollen sie ein Grundverständnis für die Methoden der diskreten Mathematik erlangt haben.

In der Vorlesung “Diskrete Mathematik” werden die folgenden Themen behandelt, die im Proseminar in weiterführenden Übungen vertieft werden.

- Wohlfundierte Induktion
- Graphen und Bäume
- Grundlagen des Abzählens
- Elementare Zahlentheorie
- *Formale Sprachen und endliche Automaten*
- *Turing Maschinen*
- *Die Komplexitätsklassen* LOGSPACE, NLOGSPACE, P, NP *und* PSPACE

Die nichtkursiv gedruckten Inhalte werden im Skriptum “Diskrete Mathematik 1” von Arne Dür behandelt, siehe [2]. In der Folge wird der Stoff jenes Skriptums vorausgesetzt. Die Inhalte der Lehrveranstaltung “Diskrete Mathematik 2” sind in der obigen Aufzählung kursiv gedruckt.

Das vorliegende Skriptum ersetzt nicht den Besuch der Vorlesung, sondern ist als vorlesungsbegleitend konzipiert.

Um die Lesbarkeit zu erleichtern, wird in der direkten Anrede des Lesers, der Leserin prinzipiell die weibliche Form gewählt. In der Erstellung des Skriptums habe ich mich auf die folgende Literatur gestützt (in der Reihenfolge der Wichtigkeit für dieses Skriptum) [4, 5, 8, 1].

1

Einleitung und Motivation

1.1 Endliche Automaten

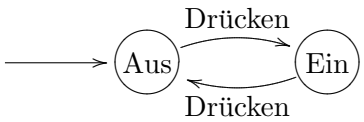
Endliche Automaten (kurz EA) finden vor allem in den folgenden Bereichen ihre Anwendung.

- Software zum Entwurf und Testen von *digitalen Schaltkreisen*.
- Softwarebausteine eines Compilers. Der *lexikalische Scanner* ("*Lexer*") eines Compilers wird üblicherweise mit Hilfe von endlichen Automaten implementiert. Hierbei versteht man unter einem Lexer die Aufteilung des Eingabetextes in logische Einheiten, wie Bezeichner, Schlüsselwörter und Satzzeichen.
- Software zum *Durchsuchen* umfangreicher Texte, wie Sammlungen von Webseiten, um Vorkommen von Wörtern, Ausdrücken oder anderer Muster zu finden.
- Software zur Verifizierung aller Arten von Systemen, die eine endliche Anzahl verschiedener Zustände besitzen, wie Kommunikationsprotokolle oder *Protokolle* zum sicheren Datenaustausch.
- Softwarebausteine eines Computerspiels. Die Logik bei der *Kontrolle von Spielfiguren* kann mit Hilfe eines endlichen Automaten implementiert werden. Dies erlaubte eine bessere Modularisierung des Codes.

Es gibt viele Systeme oder Komponenten, wie die oben aufgezählten, die so betrachtet werden können, dass sie sich zu jedem gegebenen Zeitpunkt in einem von einer endlichen Anzahl von *Zuständen* befinden. Zweck eines Zustands ist es, den relevanten Teil des Verlaufs eines Systems festzuhalten. Da es lediglich eine endliche Anzahl von Zuständen gibt, kann im Allgemeinen nicht der gesamte Verlauf beschrieben werden. Das System muss sorgfältig

entworfen werden, sodass nur Unwichtiges ignoriert wird. Da wir nur mit endlich vielen Zuständen arbeiten, können wir das System mit einer fixen Menge von Ressourcen implementieren, etwa als Schaltkreis in Hardware oder als einfaches Programm. Wir geben ein Beispiel, des wohl einfachsten endlichen Automaten, einer Abstraktion eines Ein-Aus Schalters.

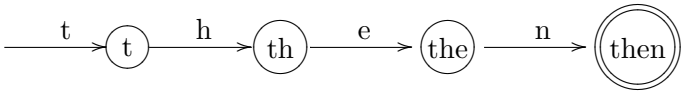
Beispiel 1.1 (Ein-Aus Schalter)**.**



Einer der Zustände wird als *Startzustand* ausgewählt, dh. als der Zustand, in dem sich das System ursprünglich befindet. In unserem Beispiel ist *Aus* der Startzustand.

Konvention. Wir führen die Konvention ein, dass der Startzustand durch einen Pfeil, der auf den betreffenden Zustand zeigt, bezeichnet wird.

Beispiel 1.2. Lexical Analyser



Der dargestellte Automat kann als Teil eines Lexers, wie sie im Compilerbau verwendet werden, verstanden werden. Der Automat erkennt das Schlüsselwort `then`. Der *akzeptierende Zustand* des Automaten ist der mit dem Schlüsselwort markierte Zustand. Prägnant wird dies durch den Doppelkreis ausgedrückt.

Konvention. Wie in Beispiel 1.2 ist es oft notwendig, einen (oder mehrere) Zustände als *akzeptierenden* Zustand zu kennzeichnen. Gemäß Konvention werden akzeptierende Zustände durch einen doppelten Kreis bezeichnet.

1.2 Endliche Automaten und Sprachen

Endliche Automaten sind eng mit der Generierung von Sprachen verbunden. Unter einer Sprache verstehen wir in diesem Zusammenhang eine Menge von Wörtern über einem vorgegebenen Alphabet Σ und ein Wort ist eine Kette von Zeichen aus Σ. (Siehe [2] für die formale Definition eines Wortes.)

Der Zusammenhang zwischen endlichen Automaten und Sprachen wird im unten stehenden Bild 1.1 skizziert. Im folgenden werden wir auch davon sprechen, dass ein Automat eine Sprache *akzeptiert*, wenn der Automat genau bei Eingabe der Wörter der Sprache in einen *akzeptierenden* Zustand gelangt. Wie aus Skizze 1.1 erkenntlich, kann ein Automat mehrere *akzeptierende* Zustände haben und von diese können auch Kanten wegführen.

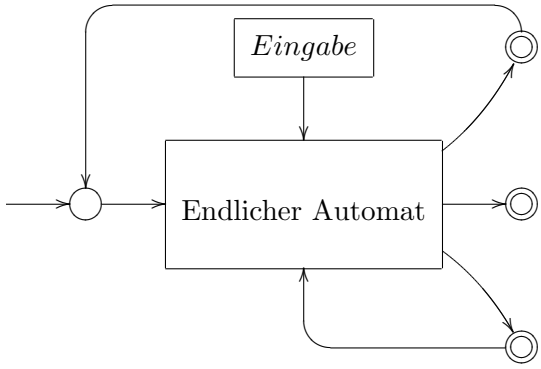


Abbildung 1.1: Schema eines endlichen Automaten

1.3 Strukturelle Repräsentationen

Obwohl endliche Automaten ein sehr anschauliches Bild eines Berechnungsmodells mit nur endlich vielen Zuständen liefern, ist es manchmal hilfreich Automaten in anderer Form zu repräsentieren. Genauer gesagt wird nicht der Automat selbst dargestellt, sondern die von dem Automaten akzeptierte Sprache. Die wichtigsten Repräsentanten sind *Grammatiken* und *reguläre Ausdrücke*.

- *Grammatiken* sind nützliche Modelle zum Entwurf von Software, die Daten mit einer rekursiven Struktur verarbeiten. Das bekannteste Beispiel ist ein *Parser*, also die Komponente eines Compilers, die mit den rekursiv verschachtelten Elementen einer typischen Programmiersprache umgeht, wie arithmetische Ausdrücke, Bedingungsausdrücke usw.

- Auch *reguläre Ausdrücke* beschreiben Sprachen, sind aber in ihrer Ausdrucksstärke gegenüber Grammatiken eingeschränkt.[1] Reguläre Ausdrücke werden bevorzugt zur Textsuche und insbesondere zum *pattern matching* verwendet.

1.3.1 Grammatiken

Ganz allgemein ausgedrückt dienen Grammatiken[2] als Regelwerk zur Bildung von *Sätzen* einer Sprache. Die Grammatik, der deutschen Sprache etwa, ist ein—meist als höchst kompliziert empfundenes—Regelwerk zur richtigen Bildung deutscher Sätze, die wiederum die

[1] Wir werden später sehen, dass die Ausdrucksstärke von regulären Ausdrücken genau der von endlichen Automaten entspricht, wohingegen Grammatiken im Allgemeinen mächtiger sein können.

[2] Formale Grammatiken werden in dieser Einleitung der Vollständigkeit halber erwähnt, werden aber in der Folge nicht weiter vertieft.

deutsche Sprache ausmachen. Vereinfacht können wir Sätze als Sequenzen von *Wörtern* verstehen. Wörter sind wiederum Sequenzen von *Buchstaben*. Also beschreiben Grammatiken, abstrakt gesprochen das "richtige" Bilden von Mengen von Buchstabensequenzen. In der Theorie der formalen Sprachen nennt man, der Einfachheit halber, Mengen von Wörtern *Sprachen*. Das folgende Beispiel verdeutlicht den Zusammenhang zu "echten" Grammatiken.

Beispiel 1.3.

$$\begin{aligned}
\mathsf{S} &\to \mathsf{Pronomen\ Nomen\ Verb\ Adjektiv} \\
\mathsf{Nomen} &\to \text{Lehrveranstaltungsleiter} \\
\mathsf{Nomen} &\to \text{Vortragende} \\
\mathsf{Pronomen} &\to \text{Unsere} \mid \text{Meine} \\
\mathsf{Verb} &\to \text{sind} \\
\mathsf{Adjektiv} &\to \text{lästig} \mid \text{nett} \mid \text{streng} \mid \text{zu schnell} \mid \text{anspruchsvoll}
\end{aligned}$$

Es gilt $\mathsf{S} \overset{*}{\Rightarrow}$ "Unsere Lehrveranstaltungsleiter sind anspruchsvoll", wobei die binäre Relation $\mathsf{S} \overset{*}{\Rightarrow}$ "Satz" informell bedeutet, dass der "Satz" aus dem *Startsymbol* S korrekt hergeleitet werden kann (in der Grammatik).

Wir sagen, der Satz

"Unsere Lehrveranstaltungsleiter sind anspruchsvoll" ,

ist aus S *ableitbar*. Im Allgemeinen ist unser Interesse aber nicht auf die syntaktische Simulierung von real existierenden Sprachen bezogen, sondern vielmehr auf die Analyse rekursiver Strukturen, wie etwa Programmiersprachen, gerichtet.

Beispiel 1.4. Wenn wir von der Bedeutung in der obigen Grammatik abstrahieren, erhalten wir die folgende Grammatik. (Um die Verwechslung mit dem Startsymbol auszuschließen, wurde die Abkürzung "H" für Nomen gewählt.)

$$\begin{aligned}
\mathsf{S} &\to \mathsf{PHVA} & \mathsf{V} &\to \mathsf{s} \\
\mathsf{H} &\to \mathsf{l} \mid \mathsf{v} & \mathsf{A} &\to \text{lä} \mid \mathsf{n} \mid \mathsf{str} \mid \mathsf{zs} \mid \text{a} \\
\mathsf{P} &\to \mathsf{u} \mid \mathsf{m}
\end{aligned}$$

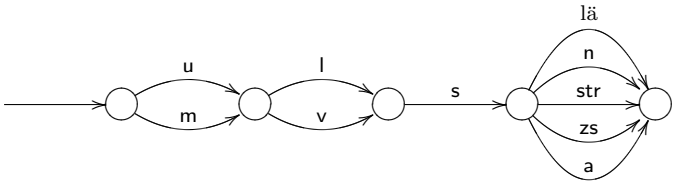
Offensichtlich gilt $\mathsf{S} \overset{*}{\Rightarrow} \mathsf{ulsa}$.

Wie kann nun die von der gerade eingeführten Grammatik beschrieben Sprache durch eine endlichen Automaten beschrieben werden? Sehr einfach, indem die Übergänge des Automaten mit der Eingabe verknüpft werden und bei jedem Übergang ein Buchstabe des eingegebenen Wortes verarbeitet wird.

Symbol	Bedeutung
.	bezeichnet jeden Charakter
\s	steht für das Sonderzeichen s
^, $	bezeichnen Zeilenanfang, bzw. Zeilenende
$[a_1 a_2 \cdots a_k]$	bezeichnet $(a_1 + a_2 + \cdots + a_k)$
$[\hat{}\, a_1 a_2 \cdots a_k]$	alle Zeichen *außer* $a_1, a_2, \ldots, a_k$
$[x - y]$	alle ASCII Zeichen zwischen x und y.
	Beispiel: $[A - Z]$ beschreibt alle Großbuchstaben.
?	heißt "keines oder eines"
+	bezeichnet "eines oder mehrere". Bitte beachten Sie den Unterschied zwischen dem Unix-Symbol + und dem Symbol für Vereinigung + in der formalen Definition von regulären Ausdrücken.
*	bezeichnet Kleene Stern
$R\{n\}$	"genau n Kopien" von R.

Abbildung 1.2: Reguläre Ausdrücke in Unix

Beispiel 1.5. Der folgende Automat akzeptiert genau die von der obigen Grammatik beschriebene Sprache.



1.3.2 Reguläre Ausdrücke

Reguläre Ausdrücke (kurz RA) werden intensiv in Betriebssystemen, wie etwa MS-DOS oder Unix verwendet. Die folgenden Beispiele zeigen für diese Betriebssysteme typische Ausdrücke:

Beispiel 1.6.

MS-DOS	`dir␣a*.exe`
Unix	`^[0-9]+\.[0-9]*(E[+-]?[0-9]+)?$`

Traditionellerweise haben reguläre Ausdrücke im Besonderen im Betriebssystem Unix eine ausgewählte Rolle gespielt (siehe auch Sektion 1.4). Im Beispiel 1.6 wird etwa ein regulärer Ausdruck dargestellt, der eine Gleitkommazahl repräsentiert. Hier werden die Ausdruckselemente aus Tafel 1.2 verwendet, die unter Unix zur Verfügung stehen. Wie aus Beispiel 1.6 ersichtlich, werden auch in MS-DOS reguläre Ausdrücke verwendet.

Unabhängig von den Ausdrücken in Tafel 1.2 geben wir eine formale induktive Definition von regulären Ausdrücken an. Hier können wir uns auf die Operationen der *Verkettung*, der *Vereinigung* $+$ und den *Kleene Stern* * beschränken. Vorest genügt es reguläre Ausdrücke E zu definieren und die Definition der *Sprache* $\mathrm{L}(E)$ des RA E informell zu lassen. In Kapitel 2.3 wird die Definition vervollständigt.

Definition 1.1. Sei Σ ein endliches Alphabet.

1. Basis
   a) Die Konstanten $\varnothing$ und ϵ sind reguläre Ausdrücke, die die Sprachen $\{\epsilon\}$ bzw. $\varnothing$ beschreiben.
   b) Für jedes $e \in \Sigma$ ist $\mathbf{e}$ ein regulärer Ausdruck (RA), der die Sprache $\{e\}$ bezeichnet.
2. Schritt
   a) Für jeden RA E ist auch E^* ein RA, der die beliebige Aneinanderkettung von Wörtern in der Sprache von E bezeichnet.
   b) Für Ausdrücke E und F ist auch EF ein RA, der die Aneinanderkettung der von E und F beschrieben Wörter bezeichnet.
   c) Für RA E und F ist auch $E + F$ ein RA, der die Vereinigung der Sprachen von E und F angibt.
   d) Wenn E ein RA ist, dann ist auch (E), der geklammerte Ausdruck E ein RA.

Nur die durch die obigen Regeln erstellbaren Zeichenketten werden als *reguläre Ausdrücke* bezeichnet, nichts sonst.

Beispiel 1.7. Mit Hilfe von regulären Ausdrücken können wir die in Beispiel 1.3 betrachtete Sprache beschreiben:

$$\mathsf{ulsa} \in \mathrm{L}((\mathsf{u} + \mathsf{m})(\mathsf{l} + \mathsf{v})(\mathsf{s})(\mathsf{lä} + \mathsf{n} + \mathsf{str} + \mathsf{zs} + \mathsf{a}) \ ,$$

das heißt, das Wort ulsa ist eine Element der Sprache des regulären Ausdrucks:

$$(\mathsf{u} + \mathsf{m})(\mathsf{l} + \mathsf{v})(\mathsf{s})(\mathsf{lä} + \mathsf{n} + \mathsf{str} + \mathsf{zs} + \mathsf{a} \ .$$

1.4 Geschichte

Mit dem Begriff *Automatentheorie* ist das Studium abstrakter Rechengeräte oder *Maschinen* gemeint. In den 1930er Jahren studierte *Alan Turing* eine abstrakte Maschine, die über sämtliche Fähigkeiten heutiger Computer verfügt, zumindest was deren prinzipielle Rechenleistung betrifft.

Turing hatte zum Ziel, die Grenze zwischen dem, was ein Computer berechnen kann, und dem, was er nicht berechnen kann, genau zu beschreiben. Seine Schlussfolgerungen treffen nicht nur auf seine abstrakten *Turing-Maschinen* zu, sondern auch auf die heutigen realen Maschine, zum Beispiel dass es Grenzen der Berechenbarkeit gibt; dh. Entscheidungsprobleme, die rein prinzipiell von einer Maschine nicht gelöst werden können.

In den 1940er und 1950er Jahren wurden von einigen Forschern einfachere Maschinen untersucht, die heute als *endliche Automaten* bezeichnet werden. Diese Automaten, ursprünglich zur Simulation von Gehirnfunktionen von *Warren McCulloch* und *Walter Pitts* eingeführt, haben sich für verschiedene andere Zwecke als nützlich erwiesen.

In den 1950er Jahren wurden die von *Warren McCulloch* und *Walter Pitts* vorgelegten Definition von *Stephen Kleene* aufgenommen und mathematisch präzise gefasst. Auf Kleene geht die Definition von *regulären Sprachen* zurück und in seinem Namen wird der Operator * auch oft als *Kleene-Stern* bezeichnet. In den späten 1950er Jahren begann zudem der Linguist N. Chomsky, formale *Grammatiken* zu untersuchen. Diese Grammatiken sind zwar streng genommen keine Maschinen, weisen jedoch enge Verwandtschaft zu abstrakten Automaten auf und dienen heute als Grundlage einiger wichtiger Softwarekomponenten, wie etwa Compiler.

1969 führte Stephen Cook, Turings Untersuchung der Frage fort, was berechnet werden kann und was nicht, indem er untersuchte welche Probleme *tatsächlich*—also mit vertretbaren Aufwand—algorithmisch zu lösen sind. Die Klasse der gängigerweise als *nicht handhabbar* (*intractable*) betrachteten Probleme als NP-hart bezeichnet, diese Begriffsbestimmung geht auf Cook zurück. Es ist sehr unwahrscheinlich, dass die exponentielle Steigerung der Rechengeschwindigkeit, die bei der Computerhardware erzielt worden ist ("Moores Gesetz"), sich bemerkenswert auf unsere Fähigkeit auswirken wird, umfangreiche Beispiele solcher nicht handhabbaren Probleme berechnen zu können.

Anwendungen von regulären Sprachen, bzw. regulären Ausdrücken finden sich etwa auch in der Genese des Betriebssystems Unix. *Ken Thompson* baute reguläre Ausdrücke in den Texteditor `qed` ein, und später in den Editor `ed`. Reguläre Ausdrücke werden seit Beginn bei Unix verwendet. Beispiele hierfür sind `expr`, `awk`, `Emacs`, `vi`, `lex` und `Perl`. Die bessere Integration von regulären Ausdrücken ist das erklärte Ziel in der Entwicklung von `Perl 6`, siehe http://dev.perl.org/perl6.

2

Endliche Automaten und reguläre Sprachen

2.1 Deterministische endliche Automaten

Wir definieren formal, was wir in der Folge unter einem endlichen Automaten verstehen werden. Wir beginnen mit der Definition von deterministischen endlichen Automaten.

Definition 2.1. Ein *deterministischer endlicher Automat* (*DEA*) ist gegeben durch

1. eine endliche Menge Q , deren Elemente *Zustände* heißen,

2. eine endliche Menge Σ , die *Eingabealphabet* heißt und deren Elemente *Eingabezeichen* genannt werden,

3. eine Abbildung
$$\delta : Q \times \Sigma \to Q$$
die *Zustandsübergangsfunktion* heißt und angibt, wie sich der Zustand des Automaten bei einer Eingabe ändern kann. Zu beachten ist, dass δ für alle möglichen Argumente definiert sein muss.

4. einen ausgezeichneten Zustand, der *Startzustand* genannt wird,

5. eine Teilmenge $F \subseteq Q$, deren Elemente *akzeptierende Zustände* genannt werden.

Die kürzeste Repräsentation eines DEA A besteht darin das obige 5-Tupel anzugeben.

$$A = (Q, \Sigma, \delta, q_0, F) \ .$$

Beispiel 2.1. Wir definieren einen DEA, der alle aus 0en und 1en bestehenden Zeichenketten akzeptiert, die die Folge 01 enthalten und sonst nichts. Diese Sprache L wird wie folgt beschrieben

$$L = \{x\mathsf{01}y \mid x, y \text{ sind beliebige Zeichenketten aus } \mathsf{0}\text{en und } \mathsf{1}\text{en}\} \ .$$

Die Sprache L enthält etwa die Wörter $\mathsf{01}, \mathsf{11010}, \mathsf{100011}$.

Um die Konstruktion eines geeigneten Automaten A zu bewerkstelligen, betrachten wir die folgenden 3 möglichen Zustände des Automaten A.

- A hat die Sequenz 01 bereits gelesen, dann akzeptiert A jede weitere Eingabenfolge.
- A hat nur die Sequenz 0 gelesen, dann gelangt A in den ersten Zustand sobald als *nächstes* 1 gelesen wird; sonst verharrt A in diesem Zustand.
- A hat entweder noch nichts gelesen, oder 01 wurde noch nicht gefunden und das letzte gelesene Symbol ist 1, dann gelangt A in den zweiten Zustand, wenn eine 0 gelesen wird, sonst verharrt er in diesem Zustand.

Diese Beobachtungen sind ausreichend, um die Übergangsfunktion zu definieren. Zunächst erkennen wir, dass der letzte der drei oberen Zustände, genau dem Startzustand q_0 entspricht. Wir definieren

$$\delta(q_0, \mathsf{1}) = q_0 \ ,$$

dies entspricht dem "Verharren" in diesem Zustand. Nun identifizieren wir den zweiten der oberen Zustände mit q_1 und setzen

$$\delta(q_0, \mathsf{0}) = q_1 \ ,$$

das heißt A geht beim Lesen von 0 in den Zustand q_1. Mit der selben Art der Übersetzung der informellen Beschreibung in die (formale) Übergangsfunktion δ erhalten wir

$$\delta(q_1, \mathsf{0}) = q_1 \qquad \delta(q_1, \mathsf{1}) = q_2 \ ,$$

und

$$\delta(q_2, \mathsf{0}) = q_2 \qquad \delta(q_2, \mathsf{1}) = q_2 \ .$$

Die vollständige Spezifikation des Automaten A, der genau die Sprache L aus dem Beispiel 2.1 beschreibt, lautet

$$A = (\{q_0, q_1, q_2\}, \{\mathsf{0}, \mathsf{1}\}, \delta, q_0, \{q_2\}) \ .$$

momentaner Zustand	Eingabe $e \in \Sigma$
$\vdots$	
$Q \ni q$	$\delta(q, e)$
$\vdots$	

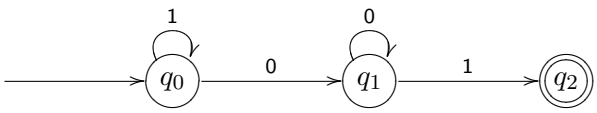
Abbildung 2.1: Die Zustandsübergangsfunktion dargestellt durch die Zustandstafel

Die Zustandsübergangsfunktion kann tabellarisch in der *Zustandstafel* angegeben werden, siehe Abbildung 2.1. Visualisiert kann der Automat durch seinen *Zustandsgraphen* werden, der als gerichteter Graph wie folgt definiert ist:

- Die Ecken sind die Zustände.
- Für Zustände $p, q \in Q$ sind die Kanten von p nach q alle Tripel

$$(p, a, q) \qquad \text{mit} \qquad a \in \Sigma \qquad \text{und} \qquad \delta(p, a) = q\ .$$

Üblicherweise schreibt man auf jede Kante (p, a, q) die Eingabe a, den Startzustand markiert man mit einem Pfeil, und die akzeptierenden Zustände werden mit einem doppelten Kreis gekennzeichnet. Der in Beispiel 2.1 gefundene Automat kann somit durch seinen Zustandsgraphen wie folgt dargestellt werden.



Gleichzeitig ist auch eine Darstellung durch die Zustandstafel denkbar.

	0	1
$\rightarrow q_0$	q_1	q_0
q_1	q_1	q_2
$*q_2$	q_2	q_2

Wir schreiben $\rightarrow q$, um darzustellen, dass q der Startzustand ist und wir schreiben $*q$, um q als akzeptierenden Zustand zu kennzeichnen. Wenn wir diese Konventionen einhalten, dann kann aus der Zustandstafel alles Wissenswerte über den beschriebenen DEA herausgelesen werden, wie Menge der Zustände, Eingabealphabet, Übergangsfunktion, Startzustand und finale Zustände. In der Folge definieren wir die *erweiterte Übergangsfunktion*.

Definition 2.2. Sei δ eine Übergangsfunktion, dann definiere $\hat{\delta}\colon Q \times \Sigma^* \rightarrow Q$ induktiv.

1. Basis:
$$\hat{\delta}(q, \epsilon) := q \ .$$

2. Schritt:
$$\hat{\delta}(q, xa) := \delta(\hat{\delta}(q, x), a) \ .$$

Beachten Sie, dass wir hier von unserer Konvention Gebrauch machen, dass Buchstaben von Ende des Alphabets oder griechische Buchstaben (in der Definition etwa x) Strings bezeichnen, wogegen lateinische Buchstaben vom Anfang des Alphabets (etwa a) einzelne Buchstaben im Alphabet bezeichnen.

Schließlich können wir die *Sprache* eines DEA $A = (Q, \Sigma, \delta, q_0, F)$ definieren. Diese Sprache wird als $\mathrm{L}(A)$ bezeichnet und wie folgt definiert.

$$\mathrm{L}(A) := \{x \mid \hat{\delta}(q_0, x) \in F\} \ .$$

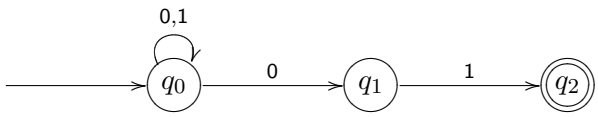
Das heißt, die Sprache von A ist die Menge der Zeichenreihen x, die vom Startzustand q_0 in einen akzeptierenden Zustand führen.

Definition 2.3. Wenn L für einen DEA A gleich $\mathrm{L}(A)$ ist, dann bezeichnen wir L als *reguläre Sprache.*

2.2 Nichtdeterministische endliche Automaten

Wir wenden uns *nichtdeterministischen endlichen Automaten* zu. Angenommen wir befinden uns in einem endlichen Automaten in einem bestimmten Zustand q. In einem nichtdeterministischen endlichen Automaten (kurz NEA) ist der Zustand p, in den wir gelangen, wenn wir den mit einem bestimmten Eingabesymbol a markierten Kanten folgen, nicht eindeutig bestimmt. Es ist zugelassen, dass es überhaupt keine solche Kante gibt, oder dass es mehrere solche Kanten gibt. Trotzdem werden wir in dieser Sektion zeigen, dass die von nichtdeterministischen endlichen Automaten akzeptierten Sprachen auch von deterministischen endlichen Automaten akzeptiert werden. Wir beginnen mit einer informellen Erklärung anhand eines Beispiels.

Beispiel 2.2. Wir betrachten den folgenden endlichen Automaten N:



In dem Beispiel ist zu beachten, dass der Automat tatsächlich nichtdeterministisch ist, da vom Zustand q_0 zwei mit 0 markierte Kanten wegführen. Der Automat muss also eine *Wahl*

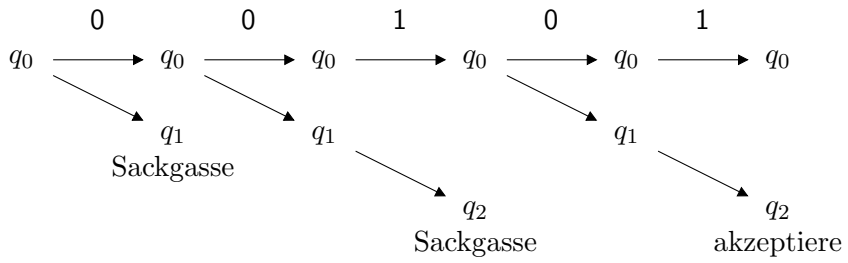


Abbildung 2.2: Akzeptieren in einem nichtdeterministischen endlichen Automaten

treffen, ob er bei Eingabe 0 in Zustand q_0 bleiben soll oder in den Zustand q_1 wechseln soll. Trotzdem ist es intuitiv verständlich, dass der Automat alle Wörter, die mit dem String 01 enden, akzeptiert. Wir erklären die Situation mit dem folgenden Bild, das die Art darlegt, wie der NEA im Beispiel das Wort 00101 verarbeitet, siehe Bild 2.2.

Definition 2.4. Ein *nichtdeterministischer endlicher Automat* (*NEA*) ist gegeben durch

1. eine endliche Menge Q , deren Elemente *Zustände* heißen,
2. eine endliche Menge Σ , die *Eingabealphabet* heißt und deren Elemente *Eingabezeichen* genannt werden,
3. eine Abbildung
$$\delta : Q \times \Sigma \to \mathcal{P}(Q)$$
die *Zustandsübergangsfunktion* heißt und angibt, wie sich der Zustand des Automaten bei einer Eingabe ändern kann. Hier bezeichnet $\mathcal{P}(Q)$ die Potenzmenge von Q.
4. einen ausgezeichneten Zustand q_0, der *Startzustand* genannt wird,
5. eine Teilmenge $F \subseteq Q$, deren Elemente *akzeptierende Zustände* genannt werden.

Beispiel 2.3. Der im obigen Beispiel angegebene NEA N lässt sich nun formal als 5-Tupel darstellen
$$(\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_2\}) ,$$
wobei δ durch die folgende Zustandstafel definiert wird.

	0	1
$\to q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\varnothing$	$\{q_2\}$
$*q_2$	$\varnothing$	$\varnothing$

In der Folge definieren wir die *erweiterte Übergangsfunktion* für nichtdeterministische endliche Automaten.

Definition 2.5. Sei δ die Übergangsfunktion eines NEA. Dann definieren wir die *erweiterte Übergangsfunktion* $\hat{\delta}\colon Q \times \Sigma^* \to \mathcal{P}(Q)$ induktiv.

1. Basis:
$$\hat{\delta}(q, \epsilon) := \{q\} \ .$$

2. Schritt. Wir definieren $\hat{\delta}(q, x)$, $x \neq \epsilon$. Sei $x = ya$ und angenommen $\hat{\delta}(q, y) = \{p_1, \ldots, p_k\}$ und
$$\bigcup_{i=1}^{k} \delta(p_i, a) = \{r_1, \ldots, r_m\} \ .$$
Dann setzen wir
$$\hat{\delta}(q, ya) = \{r_1, \ldots, r_m\} \ .$$

Weniger formal ausgedrückt: Wir berechnen $\hat{\delta}(q, ya)$ indem wir zuerst $\hat{\delta}(q, y)$ berechnen und dann jedem Übergang von einem dieser Zustände folgen, der mit a markiert ist.

Beispiel 2.4. Wir verwenden die Definition der erweiterten Übergangsfunktion, um das Verhalten des obigen NEA auf der Eingabe 00101 zu beschreiben.

- $\hat{\delta}(q_0, \epsilon) = \{q_0\}$.
- $\hat{\delta}(q_0, 0) = \delta(q_0, 0) = \{q_0, q_1\}$.
- $\hat{\delta}(q_0, 00) = \delta(q_0, 0) \cup \delta(q_1, 0) = \{q_0, q_1\} \cup \varnothing = \{q_0, q_1\}$.
- $\hat{\delta}(q_0, 001) = \delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$.
- $\hat{\delta}(q_0, 0010) = \delta(q_0, 0) \cup \delta(q_2, 0) = \{q_0, q_1\} \cup \varnothing = \{q_0, q_1\}$.
- $\hat{\delta}(q_0, 00101) = \delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$.

Schließlich können wir die *Sprache* eines NEA $A = (Q, \Sigma, \delta, q_0, F)$ definieren. Ein NEA akzeptiert ein Wort x, wenn es möglich ist eine beliebige Sequenz von Auswahlen zu treffen, während die Buchstaben von x gelesen werden und der Automat vom Startzustand in den akzeptierenden Zustand gelangt. Natürlich müssen alle Zeichen des Wortes x betrachtet (man sagt auch *konsumiert*) werden. Diese Sprache wird als $\mathrm{L}(A)$ bezeichnet und wie folgt definiert.
$$\mathrm{L}(A) := \{x \mid \hat{\delta}(q_0, x) \cap F \neq \varnothing\} \ .$$

	0	1
$\varnothing$	$\varnothing$	$\varnothing$
$\rightarrow \{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_1\}$	$\varnothing$	$\{q_2\}$
$*\{q_2\}$	$\varnothing$	$\varnothing$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$*\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$
$*\{q_1, q_2\}$	$\varnothing$	$\{q_2\}$
$*\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

Abbildung 2.3: Übergangsfunktion für DEA D

2.2.1 Teilmengenkonstruktion

Wir wenden uns der Äquivalenz von deterministischen und nichtdeterministischen Automaten zu. Die dazu verwendete Konstruktion wird als *Teilmengenkonstruktion* bezeichnet.

Wir beginnen mit einem NEA, $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ und konstruieren einen DEA $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F)$, sodass $\mathrm{L}(D) = \mathrm{L}(N)$. Klarerweise besitzen die beiden Automaten dasselbe Alphabet Σ. Nun definieren wir die übrigen Komponenten für D.

1. Q_D ist die Menge der Teilmengen von Q_N. Wenn somit N über n Zustände verfügt, dann verfügt D über 2^n Zustände.

2. Zur Berechnung von δ_D betrachten wir jede Teilmenge $S \subseteq Q_N$ und jedes $a \in \Sigma$. Wir setzen:
$$\delta_D(S, a) = \bigcup_{p \in S} \delta_N(p, a) \ .$$
D.h. wir betrachten alle in S enthaltene Zustände, untersuchen zu welchem Zustand diese in N bei Eingabe a führen und nehmen die Vereinigung aller möglicherweise erreichten Zustände.

3. F_D ist definiert als die Menge $\{S \subseteq Q_N \mid S \cap F_N \neq \varnothing\}$.

Nun betrachten wir den NEA N aus Beispiel 2.2 und wenden die Teilmengenkonstruktion an. Wir erhalten die Übergangsfunktion, die in Abbildung 2.3 dargestellt ist. Diesen DEA bezeichnen wir mit D. Wir schreiben D um, um zu verdeutlichen, dass es sich hierbei um einen *deterministischen* Automaten handelt. Der erhaltene Automat D' ist in Abbildung 2.4 angegeben.

Es ist zu beachten, dass in diesem Fall die Teilmengenkonstruktion recht ineffizient ausgeführt wurde, da von den 8 angegebenen Zuständen in der oberen Tabelle, nur 3, nämlich B, E und F tatsächlich erreichbar sind. Die anderen Zustände können offensichtlich aus

	0	1
A	A	A
$\rightarrow B$	E	B
C	A	D
$*D$	A	A
E	E	F
$*F$	E	B
$*G$	A	D
$*H$	E	F

Abbildung 2.4: Übergangsfunktion für DEA D'

dem Automaten entfernt werden, ohne dass sich dabei die akzeptierte Sprache ändert. Um diesem Tatbestand Rechnung zu tragen, können wir die Teilmengenkonstruktion im Sinne einer *lazy evaluation* ändern.

Definition 2.6. Wir definieren induktiv jene Teilmengen von Q_N die tatsächlich *erreichbar* sind.

1. Basis: Wir wissen, dass die einelementige, den Startzustand von N enthaltende Menge erreichbar ist.

2. Schritt: Angenommen die Menge S ist erreichbar. Dann sind für jeden Eingabebuchstaben a die Teilmengen, die wir mit der Übergangsfunktion $\delta_D(S, a)$ berechnen können, erreichbar.

Es ist leicht einzusehen, dass die Zustände des DEA D auf die erreichbaren Teilmengen von Q_N eingeschränkt werden können. Folglich müssen wir nur für diese Teilmengen die Übergangsfunktion δ_D berechnen. Wir zeigen, dass die Teilmengenkonstruktion korrekt ist.

Satz 2.1. *Wenn $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ der DEA ist, der mit der Teilmengenkonstruktion aus dem NEA $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ konstruiert wurde, dann gilt* $\mathrm{L}(D) = \mathrm{L}(N)$.

Beweis. Wir beweisen für alle Wörter x mit Induktion über $\ell(x)$, dass

$$\hat{\delta}_D(\{q_0\}, x) = \hat{\delta}_N(q_0, x) \ . \tag{2.1}$$

Angenommen, der Beweis hierfür ist uns bereits gelungen. Dann erkennt man leicht aus

der Definition von $\mathrm{L}(D)$, $\mathrm{L}(N)$ die Korrektheit der Behauptung:

$$\begin{aligned}\mathrm{L}(N) &= \{x \mid \hat{\delta}_N(q_0, x) \cap F_N \neq \varnothing\} \\ &= \{x \mid \hat{\delta}_D(\{q_0\}, x) \cap F_N \neq \varnothing\} \\ &= \{x \mid \hat{\delta}_D(\{q_0\}, x) \in F_D\} = \mathrm{L}(D)\ .\end{aligned}$$

Nun zeigen wir (2.1).

1. Basis: Sei $\ell(x) = 0$, dh. $x = \epsilon$. Dann

$$\hat{\delta}_D(\{q_0\}, \epsilon) = \{q_0\} = \hat{\delta}_N(q_0, \epsilon)\ .$$

2. Schritt: Sei $x = ya$. Nach Induktionshypothese gilt $\hat{\delta}_D(\{q_0\}, y) = \hat{\delta}_N(q_0, y)$. Wir nehmen an, dass $\hat{\delta}_N(q_0, y) = \{p_1, \ldots, p_k\}$. Nach Definition von $\hat{\delta}_N$ gilt:

$$\hat{\delta}_N(q_0, ya) = \bigcup_{i=1}^{k} \delta_N(p_i, a)\ . \tag{2.2}$$

Andererseits gilt nach Definition von $\hat{\delta}_D$:

$$\delta_D(\{p_1, \ldots, p_k\}, a) = \bigcup_{i=1}^{k} \delta_N(p_i, a)\ . \tag{2.3}$$

Nun lösen wir die Definition der erweiterten Übergangsfunktion für δ_D auf:

$$\hat{\delta}_D(\{q_0\}, ya) = \delta_D(\hat{\delta}_D(\{q_0\}, y), a) = \delta_D(\{p_1, \ldots, p_k\}, a)\ . \tag{2.4}$$

Die Kombination der Gleichungen (2.2)–(2.4) liefert den Induktionschritt:

$$\hat{\delta}_D(\{q_0\}, ya) = \delta_D(\{p_1, \ldots, p_k\}, a) = \bigcup_{i=1}^{k} \delta_N(p_i, a) = \hat{\delta}_N(q_0, ya)\ .$$

Somit folgt die Gleichung (2.1) und der Beweis des Satzes ist vollständig.

□

Satz 2.2. *Eine Sprache L wird genau dann von einem DEA akzeptiert, wenn L von einem NEA akzeptiert wird.*

Beweis.

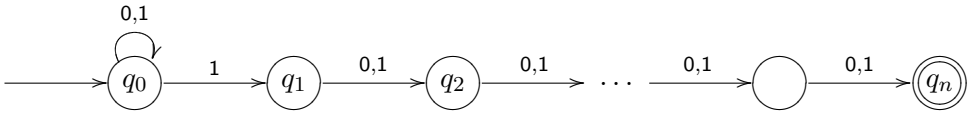
1. Wenn: Dieser Teil des Satzes folgt aus der Teilmengenkonstruktion.

2. Nur-dann-wenn: Dieser Teil ist einfach, da wir nur den gegebenen DEA in einen NEA umschreiben müssen. Formal ausgedrückt, sei $D = (Q_D, \Sigma, \delta_D, q_0, F)$. Dann definiert man einen NEA N indem man die Übergangsfunktion δ_N wie folgt definieren: wenn $\delta_D(p, a) = q$, dann $\delta_N(p, a) = \{q\}$.

□

Die Teilmengenkonstruktion führt potentiell zu einer exponentiellen Explosion der betrachteten Zustände. In der Praxis tritt diese Explosion jedoch recht selten auf. Das nächste Beispiel zeigt jedoch, dass wir einen (relativ) einfachen NEA angeben können, sodass der assoziierte DEA die potentielle obere Schranke fast erreicht.

Beispiel 2.5. Betrachte den folgenden NEA N:



Zunächst sehen wir, dass N genau die Wörter über dem Alphabet $\{0, 1\}$ akzeptiert, sodass das n-te Zeichen vor Ende des Strings eine 1 ist. Es ist einfach einzusehen, dass jeder DEA D, der dieselbe Sprache wie der NEA in Beispiel 2.5 akzeptieren soll, sich alle n Zeichen, die vor dem Wortende gelesen werden, merken muss. Angenommen D hätte nun weniger als 2^n Zustände. Dann gibt es aber einen Zustand q, der nicht zwischen zwei verschiedenen Zeichenreihen

$$a_1 a_2 \cdots a_n \qquad \text{und} \qquad b_1 b_2 \cdots b_n \ ,$$

unterscheiden kann. Das führt wie folgt zum Widerspruch. Da die beiden Wörter verschieden sind, muss es zumindest eine Position i geben, wo sie sich unterscheiden. Sei $i = 1$ und o.b.d.A.[1] können wir annehmen dass $a_1 = 1$ und $b_1 = 0$. Dann müsste q aber das Wort $a_1 a_2 \cdots a_n$ akzeptieren, wohingegen das Wort $b_1 b_2 \cdots b_n$ zu verwerfen wäre. Nach Voraussetzung kann q aber nicht zwischen den beiden Worten unterscheiden. Widerspruch. Sei nun $i > 1$, dann betrachten wir den Zustand p in den D nach dem Lesen von $a_1 a_2 \cdots a_n 0^{i-1}$ bzw. $b_1 b_2 \cdots b_n 0^{i-1}$ kommt. Nun kann das Argument für den Zustand p wiederholt werden und wir erhalten wiederum einen Widerspruch zur Annahme, dass D existiert.

Bemerkung. Sei N ein NEA mit $|\delta(q, a)| \leqslant 1$ für alle $q \in Q$ und alle $a \in \Sigma$. (Hier bezeichnet $|\delta(q, a)|$ die Kardinalität der Menge $\delta(q, a)$.) Durch Einführen eines sogenannten *Fangzustandes* f und Erweiterung der Zustandsübergangsfunktion

$$\delta'(q, a) := \begin{cases} \delta(q, a) & \text{falls } q \in Q \ \text{ und } \ |\delta(q, a)| = 1 \\ \{f\} & \text{falls } (\ q \in Q \ \text{ und } \ |\delta(q, a)| = 0\ ) \ \text{ oder } \ q = f \end{cases}$$

[1] Ohne Beschränkung der Allgemeinheit

erhält man einen NEA N', der dieselben Wörter wie N akzeptiert. Nun ist es leicht einzusehen, dass der NEA N' äquivalent zu einem DEA ist, der dieselbe Anzahl an Zuständen hat. Daher werden in der Folge auch Automaten mit höchstens einem Folgezustand als deterministisch bezeichnet.

2.2.2 Endliche Automaten mit Epsilon-Übergängen

Wir stellen die letzte Erweiterung von deterministischen endlichen Automaten vor. Diese Erweiterung erlaubt die Möglichkeit spontan (ohne das Lesen eines Zeichens) den Zustand zu wechseln. Dieser spontane Übergang in einen anderen Zustand wird mit *ϵ-Übergängen* modelliert.

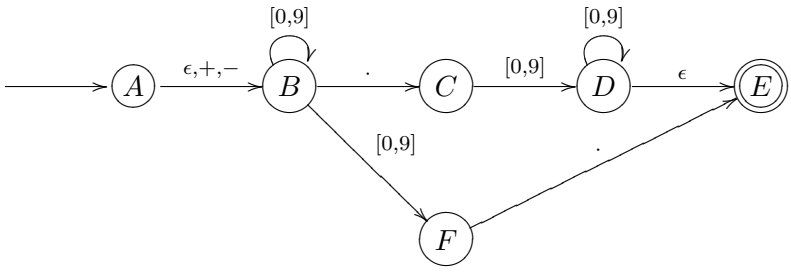
Wir betrachten Gleitkommazahlen und suchen einen Automat, der Zahlen akzeptiert, die sich wie folgt zusammensetzen.

1. ein optionales Vorzeichen: $+$, $-$,
2. einem String von Ziffern zwischen 0 und 9,
3. einem Dezimalpunkt und
4. einem weiteren String von Ziffern zwischen 0 und 9.

Die in den Punkten 2, 4 beschriebenen Ziffernstrings können leer sein, aber zumindest einer der beiden Strings muss vorhanden sein.

Der im nächsten Beispiel angeführte Automat akzeptiert nun genau die gesuchte Klasse von Gleitkommazahlen. Wir schreiben abkürzend $[0, 9]$ für $0, 1, \ldots, 9$.

Beispiel 2.6.



Im Unterschied zu früher betrachteten nichtdeterministischen Automaten, haben wir in diesem Automaten die Möglichkeit von A nach B zu gelangen ohne ein Zeichen zu lesen. Dies wird durch den mit ϵ markierten Übergang von A nach B ermöglicht.

Definition 2.7. Jeder ϵ-NEA kann auf ähnliche Art beschrieben werden wie ein NEA, der Unterschied liegt nur in der Übergangsfunktion δ. Hier ist

– das erste Argument ein Zustand in Q und

– das zweite ein Element von $\Sigma \cup \{\epsilon\}$.

Um Verwechslungen auszuschließen, fordern wir dass ϵ nicht in Σ vorkommt.

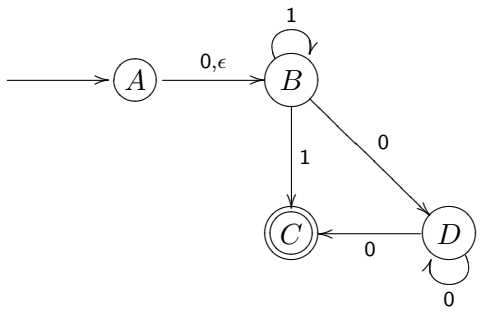
Beispiel 2.7. Ein ϵ-NEA sei gegeben durch

$$\Sigma = \{0, 1\} \quad , \quad Q = \{A, B, C, D\} \quad , \quad q_0 = A \quad , \quad F = \{C\}$$

und δ laut folgender Tabelle:

	0	1	ϵ
A	$\{B\}$	$\varnothing$	$\{B\}$
B	$\{D\}$	$\{B, C\}$	$\varnothing$
C	$\varnothing$	$\varnothing$	$\varnothing$
D	$\{C, D\}$	$\varnothing$	$\varnothing$

Den Zustandsgraphen von $\mathcal{A}$ kann man wie folgt zeichnen:



Um die erweiterte Übergangsfunktion für ϵ-NEA zu definieren, benötigen wir die Definition der ϵ-Hülle. Zum besseren Verständnis dieser Definition empfiehlt es sich, sich die Nachfolgersuche in der Graphentheorie (siehe [2, Satz 2.5]) zu vergegenwärtigen.

Definition 2.8. Wir definieren die ϵ-Hülle eines Zustands q induktiv.

1. Basis: Wir setzen $q \in \epsilon\text{-Hülle}(q)$.

2. Schritt: Sei $p \in \epsilon\text{-Hülle}(q)$ und δ die Übergangsfunktion des ϵ-NEA. Dann gilt $\delta(p, \epsilon) \subseteq \epsilon\text{-Hülle}(q)$.

Definition 2.9. Nun definieren wir die *erweiterte Übergangsfunktion* $\hat{\delta} \colon Q \times (\Sigma \cup \{\epsilon\})^* \to \mathcal{P}(Q)$ für einen ϵ-NEA induktiv.

1. Basis: $\hat{\delta}(q, \epsilon) = \epsilon\text{-Hülle}(q)$.

2. Schritt: Wir definieren $\hat{\delta}(q, x)$, sei $x = ya$. Angenommen $\hat{\delta}(q, y) = \{p_1, \ldots, p_k\}$ und

$$\bigcup_{i=1}^{k} \delta(p_i, a) = \{r_1, \ldots, r_m\} \ .$$

Dann setzen wir

$$\hat{\delta}(q, ya) = \bigcup_{j=1}^{m} \epsilon\text{-Hülle}(r_j) \ .$$

Definition 2.10. Sei $E = (Q, \Sigma, \delta, q_0, F)$ ein ϵ-NEA. Dann definieren wir die von E akzeptierte Sprache wie folgt: $\mathrm{L}(E) := \{x \mid \hat{\delta}(q_0, x) \cap F \neq \varnothing\}$.

Wie oben können wir für jeden ϵ-NEA E einen DEA D finden indem wir die Teilmengenkonstruktion anpassen. Wir beginnen mit einem ϵ-NEA, $E = (Q_E, \Sigma, \delta_E, q_0, F_E)$ und konstruieren einen DEA $D = (Q_D, \Sigma, \delta_D, q_D, F_D)$, sodass $\mathrm{L}(D) = \mathrm{L}(E)$.

- Q_D ist die Menge der Teilmengen von Q_E. Genauer wird sich zeigen, dass alle erreichbaren Zustände von D ϵ-abgeschlossen sind, das heißt die Zustände von D sind Mengen $S \subseteq Q_E$ sodass $S = \epsilon\text{-Hülle}(S)$.
- Zur Berechnung von δ_D betrachten wir eine Teilmenge $S \subseteq Q_E$ und jedes $a \in \Sigma$.

  Angenommen $S = \{p_1, \ldots, p_k\}$. Wir berechnen die Menge $\bigcup_{i=1}^{k} \delta_E(p_i, a)$. Sei diese Menge gleich $\{r_1, \ldots, r_m\}$. Dann setzen wir

$$\delta_D(S, a) = \bigcup_{j=1}^{m} \epsilon\text{-Hülle}(r_j) \ .$$

- $q_D = \epsilon\text{-Hülle}(q_0)$.
- F_D ist definiert als die Menge $\{S \subseteq Q_E \mid S \cap F_E \neq \varnothing\}$.

Mit Hilfe dieser Definition zeigen wir wie oben den folgenden Satz.

Satz 2.3. *Eine Sprache L wird genau dann von einem ϵ-NEA akzeptiert, wenn L von einem DEA akzeptiert wird.*

Beweis.

1. Wenn: Hier ist nur zu zeigen, dass ein DEA in einen äquivalenten ϵ-NEA transformiert werden kann. Das folgt analog zu oben.
2. Nur-dann-wenn: Wir wenden die erweiterte Fassung der Teilmengenkonstruktion an und erhalten einen DEA:

$$(Q_D, \Sigma, \delta_D, q_D, F_D) \ .$$

Die Behauptung folgt, wenn wir die folgende Gleichung zeigen können.

$$\hat{\delta}_D(q_D, x) = \hat{\delta}_E(q_0, x) \ ,$$

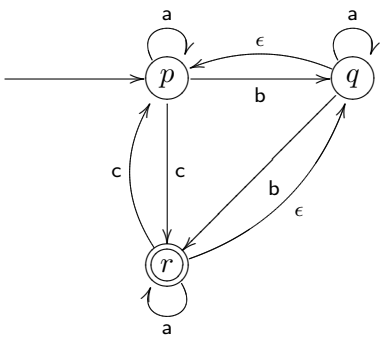
für jedes Wort x. Dies gelingt mit Induktion über $\ell(x)$; die Details überlassen wir der Leserin.

□

Beispiel 2.8. Wir betrachten den folgenden ϵ-NEA N:

	ϵ	a	b	c
$\rightarrow p$	$\varnothing$	$\{p\}$	$\{q\}$	$\{r\}$
q	$\{p\}$	$\{q\}$	$\{r\}$	$\varnothing$
$*r$	$\{q\}$	$\{r\}$	$\varnothing$	$\{p\}$

Der Zustandsgraph von N sieht folgendermaßen aus.



Die ϵ-Hüllen berechnen sich wie folgt:

$$\epsilon\text{-Hülle}(p) = \{p\} \qquad \epsilon\text{-Hülle}(q) = \{p, q\}$$
$$\epsilon\text{-Hülle}(r) = \{p, q, r\}$$

Anhand des Graphen ist es relativ leicht, die akzeptierten Zeichenreihen der Länge von 3 oder weniger Zeichen zu bestimmen.

- Wir beginnen mit den Zeichenreihen der Länge 1: Dies ist nur eine, nämlich c.
- Nun die Zeichenreihen der Länge 2: ac, bb, bc, ca, cb, cc.
- Und schließlich die Zeichenreihen der Länge 3: aac, abb, abc, aca, acb, acc, bab, bac, bba, bbb, bbc, bca, bcb, bcc, caa, cab, cac, cba, cbb, cbc, cca, ccb, ccc.

Um die von N akzeptierte Sprache exakt bestimmen zu können, wandeln wir den Automaten mit der Teilmengenkonstruktion in einen DEA D um. Wir benützen die "lazy evaluation" Methode. Und betrachten zunächst $\{p\}$. Dann ist δ_D für das erste Argument $\{p\}$ wie folgt definiert:

$$\delta_D(\{p\}, \mathsf{a}) = \epsilon\text{-Hülle}(p) = \{p\} \qquad \delta_D(\{p\}, \mathsf{c}) = \epsilon\text{-Hülle}(r) = \{p, q, r\}$$
$$\delta_D(\{p\}, \mathsf{b}) = \epsilon\text{-Hülle}(q) = \{p, q\}$$

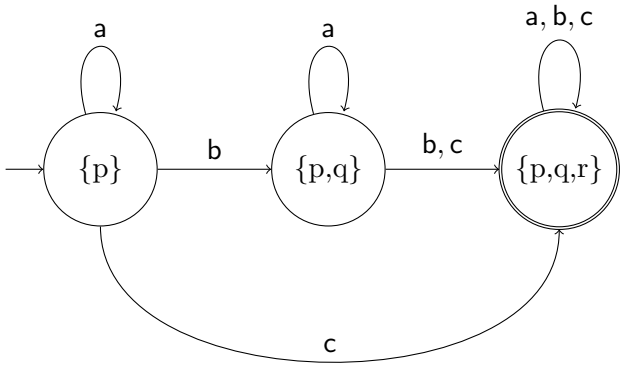
Wir fahren mit der Berechnung von δ_D mit dem ersten Argument $\{p, q\}$ fort:

$$\begin{aligned}
\delta_D(\{p, q\}, \mathsf{a}) &= \epsilon\text{-Hülle}(p) \cup \epsilon\text{-Hülle}(q) = \{p, q\} \\
\delta_D(\{p, q\}, \mathsf{b}) &= \epsilon\text{-Hülle}(q) \cup \epsilon\text{-Hülle}(r) = \{p, q, r\} \\
\delta_D(\{p, q\}, \mathsf{c}) &= \epsilon\text{-Hülle}(r) \cup \varnothing = \{p, q, r\}
\end{aligned}$$

Schließlich fehlt noch die Berechnung von δ_D mit dem ersten Argument $\{p, q, r\}$:

$$\begin{aligned}
\delta_D(\{p, q, r\}, \mathsf{a}) &= \epsilon\text{-Hülle}(p) \cup \epsilon\text{-Hülle}(q) \cup \epsilon\text{-Hülle}(r) = \{p, q, r\} \\
\delta_D(\{p, q, r\}, \mathsf{b}) &= \epsilon\text{-Hülle}(q) \cup \epsilon\text{-Hülle}(r) = \{p, q, r\} \\
\delta_D(\{p, q, r\}, \mathsf{c}) &= \epsilon\text{-Hülle}(r) \cup \epsilon\text{-Hülle}(p) = \{p, q, r\}\ .
\end{aligned}$$

Da der Zustand $\{p, q, r\}$ von D, als einziger betrachteter Zustand den akzeptierenden Zustand r von N enthält, ist der Zustand $\{p, q, r\}$ der einzige akzeptierende Zustand von D. Wir stellen D durch den folgenden Zustandsgraphen dar.



Nun können wir die von D akzeptierte Sprache leicht präzise fassen: Akzeptiert werden alle Zeichenketten (über a, b, c), die zumindest zwei bs oder ein c enthalten.

2.3 Endliche Automaten und reguläre Ausdrücke

Wir erinnern an die Definition von regulären Ausdrücken in Definition 1.1 und vervollständigen diese.

Dazu müssen wir zunächst die *Vereinigung*, *Verkettung* und den *Kleene Sterns* * auf Sprachen definieren. Seien L, M formale Sprachen über dem Alphabet Σ. Wie schon in Kapitel 6 des Skriptums "Diskrete Mathematik 1" wird hier die Verkettung der Wörter x und y durch xy ausgedrückt, siehe [2].

- $L \cup M := \{x \mid x \in L \text{ oder } x \in M\}$.
- $LM := \{xy \mid x \in L,\, y \in M\}$.
- $L^0 := \{\epsilon\}$, $L^1 := L$, $L^n := LL \dots L$, für $n-1$ Verkettungen der Sprache L.
- $L^* := \bigcup_{n \geqslant 0} L^n$.

Nun können wir Definition 1.1 vervollständigen:

Definition 2.11. Sei Σ ein endliches Alphabet.

1. Basis
   a) Die Konstanten $\varnothing$ und ϵ sind reguläre Ausdrücke. Wir definieren: $\mathrm{L}(\epsilon) := \{\epsilon\}$ und $\mathrm{L}(\varnothing) := \varnothing$.
   b) Für jedes $e \in \Sigma$ ist $\mathbf{e}$ ein regulärer Ausdruck (RA). Wir definieren: $\mathrm{L}(\mathbf{e}) := \{e\}$.
2. Schritt
   a) Für jeden RA E ist auch E^* ein RA. Wir definieren $\mathrm{L}(E^*) := (\mathrm{L}(E))^*$.
   b) Für Ausdrücke E und F ist auch EF ein RA, der die Verkettung von $\mathrm{L}(E)$ und $\mathrm{L}(F)$ angibt. Das heißt $\mathrm{L}(EF) := \mathrm{L}(E)\,\mathrm{L}(F)$.
   c) Für RA E und F ist auch $E+F$ ein RA. Wir definieren $\mathrm{L}(E+F) := \mathrm{L}(E) \cup \mathrm{L}(F)$.
   d) Wenn E ein RA ist, dann ist auch (E), der geklammerte Ausdruck E ein RA. Das heißt $\mathrm{L}((E)) := \mathrm{L}(E)$.

Die Menge der Wörter, die von einem RA E ausgedrückt werden, nennen wir die *Sprache* (geschrieben $\mathrm{L}(E)$) von E.

Beispiel 2.9. Wir wollen einen regulären Ausdruck formulieren, dessen Sprache L alle Strings mit abwechselnden 0en und 1en enthält. Wir gehen schrittweise vor. Zunächst beschreibt der reguläre Ausdruck **01** den String 01. Somit beschreibt $(\mathbf{01})^*$ alle Strings der Form

$$0101010101 \cdots$$

Die Sprache $\mathrm{L}((\mathbf{01})^*)$ ist noch nicht die gewünschte Sprache. Wir haben nur die Zeichenreihen beschrieben, die mit 0 anfangen und mit 1 enden, aber die Zeichenreihe 1010 soll auch in der Sprache aufgenommen werden, wie die Zeichenreihe 010. Schließlich haben wir noch den String 101 zu beachten. In jedem der Fälle sieht man leicht, wie der passende reguläre Ausdruck gestaltet sein muss. Wir können nun L beschreiben, indem wir diese 4 regulären Ausdrücke vereinigen:

$$(\mathbf{01})^* + (\mathbf{10})^* + \mathbf{0}(\mathbf{10})^* + \mathbf{1}(\mathbf{01})^* \ .$$

Nennen wir diesen regulären Ausdruck E_1. Dann gilt $L = \mathrm{L}(E_1)$.

Es ist jedoch eine kürzere Darstellung der Sprache aller Strings mit abwechselnden 0en und 1en möglich:

$$(\epsilon + \mathbf{1})(\mathbf{01})^*(\epsilon + \mathbf{0}) \ .$$

Nennen wir diesen RA E_2. Es ist leicht einzusehen, dass $\mathrm{L}(E_1) = \mathrm{L}(E_2) = L$.

Um RA konziser schreiben zu können, verwenden wir in der Folge die folgende Prioritäten für die Operatoren: * bindet stärker als Komposition und Vereinigung und Komposition bindet stärker als Vereinigung. Sowohl Komposition und Vereinigung sind assoziativ, also können auch hier die Klammern weggelassen werden. Ist eine Gruppierung jedoch erwünscht, wird Klammerung von links nach rechts angenommen.

In der Folge zeigen wir die Äquivalenz von endlichen Automaten und regulären Ausdrücken. Der Vollständigkeit halber sei erwähnt, dass diese Klasse von Sprachen auch durch sogenannte *reguläre Grammatiken* beschrieben werden kann. Wir verweisen die interessierte Leserin auf [3].

2.3.1 Endliche Automaten in regulären Ausdrücken umwandeln

Satz 2.4. *Sei A ein beliebiger DEA, dann gibt es einen regulären Ausdruck R, sodass* $\mathrm{L}(R) = \mathrm{L}(A)$.

Beweis. O.b.d.A. können wir annehmen, dass A die n Zustände $\{1, \ldots, n\}$ besitzt. Wir definieren reguläre Ausdrücke $R_{ij}^{(k)}$ induktiv. Informell beschreiben diese Ausdrücke die Symbole, die in A auf einem Weg vom Knoten i zum Knoten j gelesen werden. Als Zusatzbedingung für den Weg von i nach j fordern wir, dass zwischen i und j kein Knoten $> k$ besucht wird.

Wir definieren $R_{ij}^{(k)}$ induktiv:

Basis. $k = 0$. Wir betrachten nur die Wege der Länge 1, also Kanten, die von i nach j führen und Wege der Länge 0, also den Knoten i, wenn $i = j$. Wir unterscheiden die Fälle $i \neq j$ und $i = j$.

$i \neq j$ In diesem Fall kann es nur Wege der Länge 1 geben. Wir betrachten die markierten Kanten (i, a, j) zwischen i und j.

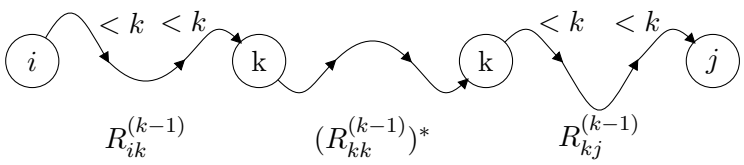
- Wenn es keine solche Kante gibt, dann setzen wir $R_{ij}^{(0)} = \varnothing$.
- Wenn es genau eine solche Kante (i, a, j) gibt, setzen wir $R_{ij}^{(0)} = \mathbf{a}$.
- Wenn es mehrere Kanten $(i, a_1, j), \ldots, (i, a_l, j)$ gibt setzen wir $R_{ij}^{(0)} = \mathbf{a_1} + \cdots + \mathbf{a_l}$.

$i = j$ In diesem Fall führen wir die obige Konstruktion durch, wenn es eine Kante von i nach i gibt. Zusätzlich repräsentieren wir jedoch auch noch die Wege der Länge 0 durch das Leerwort ϵ und fügen ϵ zu den nach der oberen Konstruktion erhaltenen regulären Ausdrücken hinzu.

Induktionsschritt. $k > 0$. Angenommen es gibt einen Weg von i nach j, der nur durch Zustände mit Nummer $\leqslant k$ führt. Wir betrachten zwei Fälle.

- Zustand k liegt nicht auf dem Weg. Dann können wir $R_{ij}^{(k-1)}$ verwenden.
- Zustand k liegt auf dem Weg. Wir spalten den Weg in mehrere Teilstücke. Der erste Teil führt von i nach k ohne k zu passieren, der letzte von k nach j, ebenfalls ohne k zu passieren. Die mittleren Stücke führen jeweils von k nach k.

  Graphisch lässt sich die Zerlegung wie folgt darstellen:



  Die Menge der gelesenen Zeichen können wir also durch

$$R_{ik}^{(k-1)}(R_{kk}^{(k-1)})^* R_{kj}^{(k-1)} \ ,$$

  beschreiben. Das beendet die induktive Definition.

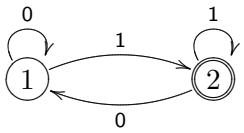
In Summe erhalten wir, die folgende rekursive Definition:

$$R_{ij}^{(k)} = R_{ij}^{(k-1)} + R_{ik}^{(k-1)}(R_{kk}^{(k-1)})^* R_{kj}^{(k-1)} \ .$$

Wir können annehmen, dass der Startknoten des Automaten A mit 1 bezeichnet ist. Der RA, der L beschreibt ist durch die Vereinigung aller $R_{1j}^{(n)}$, sodass j akzeptierend, gegeben. □

Beachten Sie die Ähnlichkeit des gegebenen Beweises zu den Korrektheitsbeweisen für die Algorithmen von Warshall und Floyd, siehe Kapitel 2 in [2].

Beispiel 2.10. Betrachte den Automaten A:



Mit Hilfe des Verfahrens folgt $R = (\mathbf{0}^*\mathbf{1})^+ = \mathrm{L}(A)$. Wir skizzieren den Rechengang. Die vollständige Berechnung von R überlassen wir der Leserin:

$$
\begin{aligned}
R_{12}^{(2)} &= R_{12}^{(1)} + R_{12}^{(1)}(R_{22}^{(1)})^* R_{22}^{(1)} = (\mathbf{0}^*\mathbf{1})^+ \\
R_{12}^{(1)} &= R_{12}^{(0)} + R_{11}^{(0)}(R_{11}^{(0)})^* R_{12}^{(0)} = \mathbf{1} + (\mathbf{0} + \epsilon)(\mathbf{0} + \epsilon)^*\mathbf{1} = \mathbf{0}^*\mathbf{1} \\
R_{22}^{(1)} &= R_{22}^{(0)} + R_{21}^{(0)}(R_{11}^{(0)})^* R_{12}^{(0)}
\end{aligned}
$$

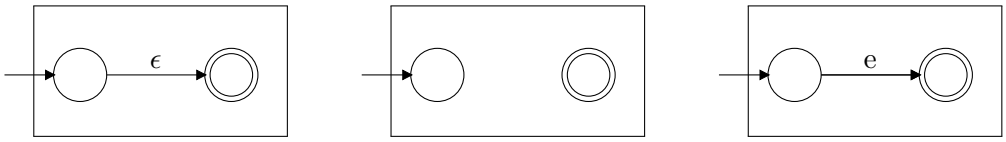
2.3.2 Reguläre Ausdrücke in Automaten umwandeln

Satz 2.5. *Sei R ein beliebiger regulärer Ausdruck. Dann existiert ein DEA A, sodass* $\mathrm{L}(A) = \mathrm{L}(R)$.

Beweis. Angenommen $L = \mathrm{L}(R)$, für einen RA R. Wegen Satz 2.3 genügt es zu zeigen, dass $L = \mathrm{L}(E)$ für einen $\epsilon - NEA$ E ist. Im Beweis zeigen wir darüber hinaus, dass der $\epsilon - NEA$ E:

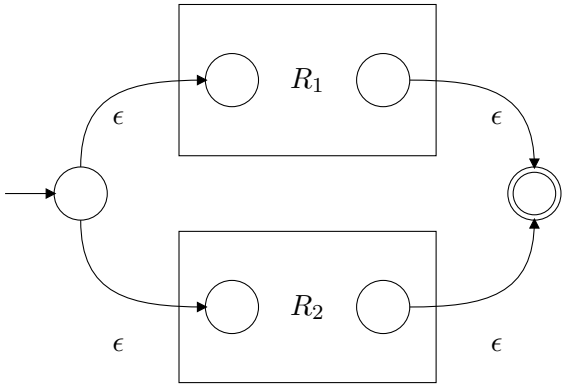
1. genau einem akzeptierenden Zustand,
2. keine Kanten zum Startzustand und
3. keine Kanten vom akzeptierenden Zustand hat.

Wir verwenden Induktion über reguläre Ausdrücke.
Basis. Für die Ausdrücke ϵ, $\varnothing$ und $\mathbf{e}$ betrachten wir die folgenden 3 Automaten.

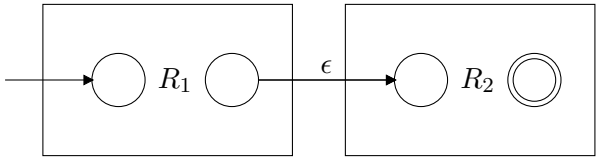


Induktionsschritt. Wir führen eine Fallunterscheidung nach dem RA R durch.

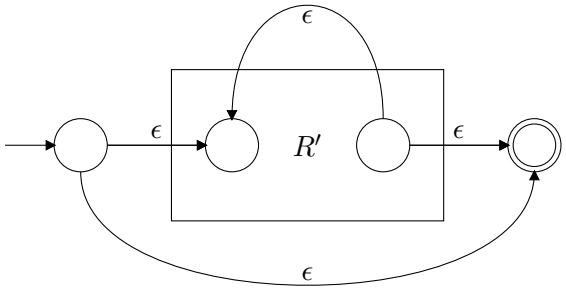
– $R \equiv R_1 + R_2$. Nach Induktionshypothese (IH) existieren Automaten für R_1 und R_2. Diese kombinieren wir wie folgt:



– $R \equiv R_1 R_2$. Nach IH existieren Automaten für R_1 und R_2. Diese kombinieren wir wie folgt:



– $R \equiv (R')^*$. Nach IH existiert ein Automat für R', wir transformieren diesen zu einem Automaten für R.

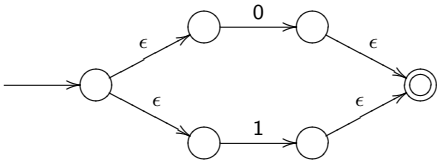


□

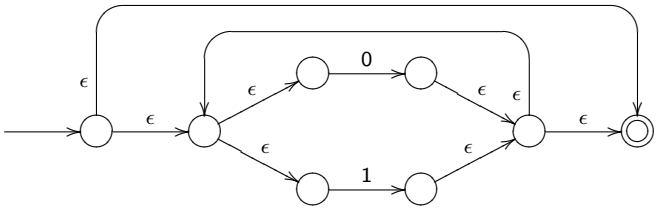
Beispiel 2.11. Wir wandeln den RA

$$(\mathbf{0}+\mathbf{1})^*\mathbf{1}(\mathbf{0}+\mathbf{1}) \ ,$$

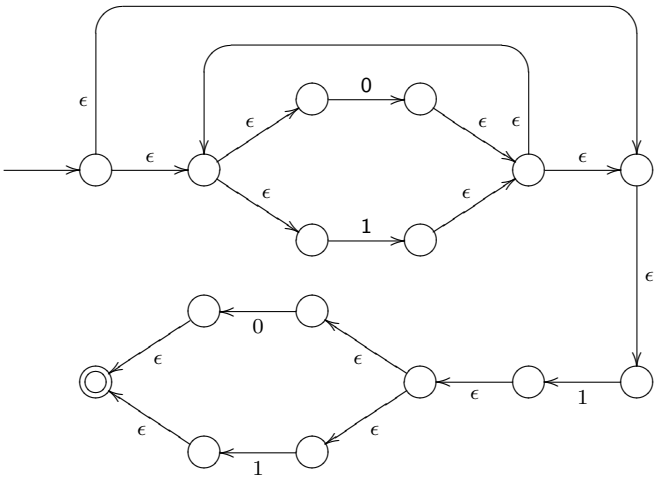
in einen ϵ-NEA um. Der erste Schritt besteht in der Konstruktion eines Automaten für $\mathbf{0}+\mathbf{1}$.



Nun wenden wir die Übersetzung des * Operators auf diesen Automaten an.



Schließlich muss nun der zweite Automat über eine mit **1** markierte Kante mit einer Kopie des ersten verkettet werden.



Satz 2.6. *Die folgenden Mengen sind gleich:*

- *die Menge der durch reguläre Ausdrücke beschriebenen Sprachen*
- *die Menge der von deterministischen endlichen Automaten akzeptierten Sprachen*
- *die Menge der von nichtdeterministischen endlichen Automaten akzeptierten Sprachen*
- *die Menge der von ϵ-NEA akzeptierten Sprachen.*

Diese Sprachen nennt man auch regulär.

Beweis. Der Satz ist eine direkte Konsequenz der Sätze 2.2, 2.3, 2.4, und 2.5. □

2.4 Algebraische Gesetze für reguläre Ausdrücke

Wir geben einige algebraische Gesetze für reguläre Ausdrücke auf. In den meisten Fällen folgen die Aussagen leicht aus der Definition der verwendeten regulären Ausdrücke. In diesen Fällen wurde auf die Beweise verzichtet.

Satz 2.7. *Seien D, E, F reguläre Ausdrücke, dann gilt:*

- $\mathrm{L}((E+F)) = \mathrm{L}((F+E))$, *das* Kommutativgesetz *der Vereinigung gilt.*
- $\mathrm{L}((D+E)+F) = \mathrm{L}(D+(E+F))$.
- $\mathrm{L}((DE)F) = \mathrm{L}(D(EF))$, *dh. das* Assoziativitätsgesetz *gilt für Komposition und Vereinigung.*

Satz 2.8. *Sei E ein regulärer Ausdruck, dann gilt:*

- $\mathrm{L}(\varnothing + E) = \mathrm{L}(E + \varnothing) = \mathrm{L}(E)$, *dh. $\varnothing$ ist das* neutrale Element *für die Vereinigung.*
- $\mathrm{L}(\epsilon E) = \mathrm{L}(E\epsilon) = \mathrm{L}(E)$, *dh. ϵ ist das* neutrale Element *der Verkettung.*
- $\mathrm{L}(\varnothing E) = \mathrm{L}(E\varnothing) = \mathrm{L}(\varnothing) = \varnothing$, *dh. $\varnothing$ ist ein* Annihilator *der Verkettung.*

Satz 2.9. *Seien D, E, F reguläre Ausdrücke, dann gilt:*

- $\mathrm{L}(D(E+F)) = \mathrm{L}(DE+DF)$, linkes Distributivgesetz *der Verkettung bezüglich der Vereinigung.*
- $\mathrm{L}((E+F)D) = \mathrm{L}(ED+FD)$, rechtes Distributivgesetz *der Verkettung bezüglich der Vereinigung.*

Satz 2.10. *Sei E ein regulärer Ausdruck, dann gilt:*

- $\mathrm{L}(E+E) = \mathrm{L}(E)$, Idempotenzgesetz.

Satz 2.11. *Sei E ein regulärer Ausdruck, dann gilt:*

- $\mathrm{L}(E^*) = \mathrm{L}(E^*E^*) = \mathrm{L}((E^*)^*) = \mathrm{L}(E+E^*)$.
- $\mathrm{L}(\varnothing^*) = \mathrm{L}(\epsilon^*) = \{\epsilon\}$.
- $\mathrm{L}(E^+) = \mathrm{L}(EE^*)$.
- $\mathrm{L}((E+F)^*) = \mathrm{L}((E^*+F^*)^*) = \mathrm{L}((E^*F^*)^*) = \mathrm{L}((E^*F)^*E^*) = \mathrm{L}(E^*(FE^*)^*)$.

Beweis. Wir zeigen zwei der vier Punkte. Die anderen Aussagen folgen ähnlich oder direkt aus der Definition der verwendeten regulären Ausdrücke.

Zunächst zeigen wir $\mathrm{L}(E^*) = \mathrm{L}(E^*E^*)$. Dazu reicht es zu zeigen, dass $\mathrm{L}(E)^* = \mathrm{L}(E)^*\,\mathrm{L}(E)^*$. Einerseits, wenn $x \in \mathrm{L}(E)^*$, dann $x\epsilon \in \mathrm{L}(E)^*\,\mathrm{L}(E)^*$, also $\mathrm{L}(E)^* \subseteq \mathrm{L}(E)^*\,\mathrm{L}(E)^*$. Andererseits wenn $x \in \mathrm{L}(E)^*\,\mathrm{L}(E)^*$, dann existieren Wörter y, z aus $\mathrm{L}(E)^*$, sodass $x = yz$. Also existieren $k, l \in \mathbb{N}$ sodass $y \in \mathrm{L}(E)^k$, $z \in \mathrm{L}(E)^l$ und somit $x \in \mathrm{L}(E)^{k+l}$, womit gezeigt wäre dass $x \in \mathrm{L}(E)^*$. Es folgt $\mathrm{L}(E)^*\,\mathrm{L}(E)^* \subseteq \mathrm{L}(E)^*$ und $\mathrm{L}(E)^* = \mathrm{L}(E)^*\,\mathrm{L}(E)^*$ ist gezeigt.

Nun zeigen wir $\mathrm{L}((E+F)^*) = \mathrm{L}((E^*+F^*)^*)$. Einerseits wenn $x \in \mathrm{L}((E+F)^*)$, dann existiert $n \in \mathbb{N}$, sodass $x = x_1 \cdots x_n$ und für alle $1 \leqslant i \leqslant n$: $x_i \in \mathrm{L}(E+F)$. Somit gilt $x_i \in \mathrm{L}(E^*+F^*)$ und in Summe: $x \in \mathrm{L}((E^*+F^*)^*)$, also $\mathrm{L}((E+F)^*) \subseteq \mathrm{L}((E^*+F^*)^*)$. Andererseits, sei $x \in \mathrm{L}((E^*+F^*)^*)$. Dann existiert $n \in \mathbb{N}$, sodass $x = x_1 \cdots x_n$ und für alle $1 \leqslant i \leqslant n$: $x_i \in \mathrm{L}(E^*+F^*)$. Zunächst nehmen wir an, dass $x_i \in \mathrm{L}(E^*)$, dann gilt aber auch $x_i \in \mathrm{L}((E+F)^*)$. In gleicher Weise folgt $x_i \in \mathrm{L}((E+F)^*)$, wenn $x_i \in \mathrm{L}(F^*)$. Somit gilt $x \in \mathrm{L}(((E+F)^*)^*) = \mathrm{L}((E+F)^*)$ und $\mathrm{L}((E^*+F^*)^*) \subseteq \mathrm{L}((E+F)^*)$, also auch $\mathrm{L}((E+F)^*) = \mathrm{L}((E^*+F^*)^*)$. □

Die, in den oben angegeben Sätzen verwendeten regulären Ausdrücke gelten wir alle Instanzen der Ausdrücken D, E, und F auf. Diese können wir also als Variablen ansehen. Der folgende Algorithmus erlaubt das allgemeine Überprüfen von Aussagen der Form $\mathrm{L}(G) = \mathrm{L}(H)$:

1. Alle Variablen in G und H werden durch konkrete Symbole ersetzt, sodass verschiedene Variablen durch verschiedene Symbole ersetzt werden.

2. Die Gleichung $\mathrm{L}(G) = \mathrm{L}(H)$ wird dadurch in eine Gleichung $\mathrm{L}(C) = \mathrm{L}(D)$ umgewandelt.

3. Teste ob $\mathrm{L}(C) = \mathrm{L}(D)$.
   - Wenn *Ja*, dann gilt auch $\mathrm{L}(G) = \mathrm{L}(H)$.
   - Wenn *Nein*, dann ist $\mathrm{L}(G) = \mathrm{L}(H)$ falsch und das Gegenbeispiel zur Gleichheit $\mathrm{L}(C) = \mathrm{L}(D)$ ist auch ein Gegenbeispiel zu $\mathrm{L}(G) = \mathrm{L}(H)$.

Satz 2.12. *Der angegebene Test ist korrekt: Für jedes so gefundene Gesetz gilt* $\mathrm{L}(G) = \mathrm{L}(H)$ *für alle beliebigen regulären Ausdrücke an der Stelle von* G *bzw.* H*.*

Beweis. Seien G, H reguläre Ausdrücke. O.b.d.A. nehmen wir an, dass G, H genau k verschiedene Variable für reguläre Ausdrücke enthalten. Natürlich muss $\mathrm{L}(G) = \mathrm{L}(H)$ für jede beliebige Substitution von Sprachen für diese Variable gelten. Im Weiteren seien C, D reguläre Ausdrücke, die wir erhalten, wenn für die k Variablen konkrete Symbole eingesetzt werden.

Wir beweisen den Satz mit einem Widerspruchsbeweis. Dazu nehmen wir an die Gleichung $\mathrm{L}(G) = \mathrm{L}(H)$ wäre falsch, dh., $\mathrm{L}(G) \neq \mathrm{L}(H)$, obwohl $\mathrm{L}(C) = \mathrm{L}(D)$ gilt.

Nach Voraussetzung existieren Sprachen L_1, L_2, $\ldots L_k$, sodass für die entsprechenden Instanzen G' und H' gilt: $\mathrm{L}(G') \neq \mathrm{L}(H')$. O.b.d.A. sei x eine Zeichenkette, sodass $x \in \mathrm{L}(G')$, aber $x \notin \mathrm{L}(H')$. Nun ersetzten wir in x alle Teilstrings, die aus L_1, L_2, $\ldots$, L_k stammen durch die konkreten Ausdrücke, die in den Ausdrücken C, D verwendet wurden. Wir bezeichnen das Ergebnis als x'. Beachte, dass durch die Substitution in x höchstens

verschiedene Teilstrings durch den gleichen Ausdruck ersetzt werden können. Wenn also $x \in \mathrm{L}(G')$, dann gilt auch $x' \in \mathrm{L}(C)$. Und aus dem selben Grund gilt $x' \notin \mathrm{L}(D)$. Somit haben wir einen Widerspruch zur Voraussetzung, dass $\mathrm{L}(C) = \mathrm{L}(D)$. □

Beispiel 2.12. Wir können den Test verwenden, um $\mathrm{L}((E+F)^*) = \mathrm{L}((E^*F^*)^*)$ zu zeigen, womit wir einen weiteren Teil der Regeln in Satz 2.11 beweisen werden.

Ersetze die regulären Ausdrücke E, F durch $\mathbf{a}, \mathbf{b}$, wir erhalten: $\mathrm{L}((\mathbf{a}+\mathbf{b})^*) = \mathrm{L}((\mathbf{a}^*\mathbf{b}^*)^*)$. Wir beschränken uns darauf zu zeigen, dass $\mathrm{L}((\mathbf{a}+\mathbf{b})^*) \subseteq \mathrm{L}((\mathbf{a}^*\mathbf{b}^*)^*)$. Sei $w \in \mathrm{L}((\mathbf{a}+\mathbf{b})^*)$. Dann bezeichnet w einen String aus as und bs. Triviallerweise gilt für jeden Buchstaben w_i von w: $w_i \in \mathrm{L}(\mathbf{a}^*\mathbf{b}^*)$. Somit liegt w im Abschluss von $\mathrm{L}(\mathbf{a}^*\mathbf{b}^*)$: $w \in \mathrm{L}(\mathbf{a}^*\mathbf{b}^*)^*$. Daher erhalten wir $\mathrm{L}((\mathbf{a}+\mathbf{b})^*) \subseteq \mathrm{L}((\mathbf{a}^*\mathbf{b}^*)^*)$. Die andere Richtung folgt in ähnlicher Weise.

2.5 Das "Pumping Lemma"

Satz 2.13. *Sei L eine reguläre Sprache, dann existiert eine Konstante n, sodass für jeden String $w \in L$, $\ell(w) \geqslant n$, Wörter x, y, und z existieren mit $w = xyz$ und*

- *$y \neq \epsilon$,*
- *$\ell(xy) \leqslant n$,*
- *Für alle $k \geqslant 0$ gilt: $x(y)^k z \in L$.*

Beweis. Angenommen L ist regulär, also existiert ein DEA A mit n Zuständen, sodass $L = \mathrm{L}(A)$. Sei

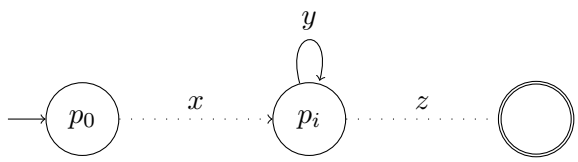
$$w = a_1 \cdots a_m \ ,$$

wobei $m \geqslant n$. Definiere $p_i = \hat{\delta}(q_0, a_1 \cdots a_i)$; beachte $p_0 = q_0$. Nach dem Schubfachprinzip muss gelten $p_i = p_j$ für $i, j \in \{0, \ldots, n\}$ und $i < j$. Zerlege w:

$$\underbrace{a_1 \cdots a_i}_{x} \qquad \underbrace{a_{i+1} \cdots a_j}_{y \neq \epsilon} \qquad \underbrace{a_{j+1} \cdots a_m}_{z} \ .$$

Um das Wort $x(y)^k z$ zu akzeptieren, läuft der Automat k-mal durch den Weg, der p_i mit p_j verbindet. Da $p_i = p_j$ ist dies beliebig oft möglich. □

Die Situation stellt sich graphisch wie folgt dar:



1. Spielerin 1 wählt eine Sprache L.
2. Spieler 2 wählt ein beliebiges n.
3. Spielerin 1 wählt ein Wort $w \in L$, $\ell(w) \geqslant n$.
4. Spieler 2 zerlegt w beliebig in 3 Teile x,y,z, sodass $\ell(xy) \leqslant n$, $y \neq \epsilon$. Spieler 2 muss die Zerlegung nicht mitteilen.
5. Spielerin 1 gewinnt, wenn sie k wählen kann, sodass $x(y)^k z \notin L$.

L ist genau dann *nicht regulär*, wenn Spielerin 1 immer gewinnt.

Abbildung 2.5: Das Pumping Lemma als Spiel

Wir formulieren die Kontraposition des Pumping Lemmas.

Satz 2.14. *Sei L eine beliebige formale Sprache und angenommen, die folgenden Bedingungen sind erfüllt:*

- *für alle $n \in \mathbb{N}$*
- *existiert ein String $w \in L$ mit $\ell(w) \geqslant n$, sodass*
- *für alle Zerlegungen von w in Teilwörter x, y und z ($w = xyz$) mit $y \neq \epsilon$ und $\ell(xy) \leqslant n$,*
- *existiert $k \in \mathbb{N}$ mit $x(y)^k z \notin L$*

Dann ist L nicht regular.

Beweis. Der Satz folgt unmittelbar aus Satz 2.13. □

Satz 2.14 können wir auch als Spiel formulieren, wie in Abbildung 2.5 zu sehen.

Beispiel 2.13. Wir wollen zeigen, dass die Sprache L über dem Alphabet $\Sigma = \{\mathsf{0}, \mathsf{1}\}$, deren Worte genauso viele 0en wie 1er enthalten, nicht regulär ist. Nennen wir die Spieler Anna und Otto.

1. Natürlich wählt Anna die Sprache L.
2. Otto wählt als Zahl n.
3. Anna wählt daraufhin das Wort $w = \mathsf{0}^n\mathsf{1}^n$, welches in L ist.
4. Otto zerlegt w beliebig in x,y und z. Er muss dabei darauf achten, dass $\ell(xy) \leqslant n$ und $y \neq \epsilon$.

5. Anna kennt die Zerlegung nicht, kann aber aus den Bedingungen und der Kenntnis von w darauf schließen, dass $xy = 0^i$ für $i \leqslant n$. Sie wählt für $k = 0$.

Nun hat Anna gewonnen: $y \neq \epsilon$, o.B.d.A. können wir annehmen $y = 0$. Dann fehlt aber in xz der String y, wohingegen x und z noch dieselbe Anzahl an 1en enthalten. Somit kann xz nicht in L liegen.

Beispiel 2.14. Wir betrachten die Sprache L', die aus allen Strings von 1er besteht, deren Länge eine Primzahl ist. Um zu zeigen, dass L' nicht regulär ist, wenden wir Satz 2.14 an, wobei wir dem Schema des Satzes genau folgen. Wir müssen zeigen, dass für L' gilt:

- für alle $n \in \mathbb{N}$
- existiert ein String $w \in L'$ mit $\ell(w) \geqslant n$, sodass
- für alle Teilwörter x, y und z von w, sodass $w = xyz$ mit $y \neq \epsilon$ und $\ell(xy) \leqslant n$,
- existiert $k \in \mathbb{N}$ mit $x(y)^k z \notin L'$.

Um den ersten Punkt zu erfüllen, wählen wir $n \in \mathbb{N}$ beliebig. Dann wählen wir eine Primzahl p sodass $p \geqslant n + 2$ und wir setzen $w = 1^p$. Damit ist der zweite Punkte erfüllt: $w \in L'$ und $\ell(w) = p \geqslant n$.

Seien nun x, y, und z beliebig, sodass $w = xyz$, $\ell(xy) \leqslant n$ und $y \neq \epsilon$. Somit ist die dritte Bedingung erfüllt. Setze $m := \ell(y)$; somit gilt $\ell(xz) = p - m$. Betrachte $v := x(y)^{(p-m)}z$. Wir zeigen, dass $v \notin L'$. Das gelingt nun relativ leicht, denn

$$\ell(v) = \ell(x(y)^{(p-m)}z) = (p - m) + m \cdot (p - m) = (p - m) \cdot (m + 1) \ .$$

Das heißt die Länge ℓ von v ist zusammengesetzt und somit keine Primzahl, wenn $(p-m) \neq 1$ und $(m + 1) \neq 1$. Diese zeigen wir durch direkte Rechnung.

1. $(m + 1) \neq 1$, da $m = \ell(y)$ und $y \neq \epsilon$.
2. $(p - m) \neq 1$, da $m = \ell(y) \leqslant \ell(xy) = n$ und $p \geqslant n + 2$; somit $(p - m) \geqslant n + 2 - m \geqslant n + 2 - n \geqslant 2$.

Somit haben wir ℓ als zusammengesetzt nachgewiesen. Aber dann gilt $v \notin L'$. Wir setzen nun $k := p - m$. Damit haben wir die vierte Bedingung erfüllt.

Mit Satz 2.14 schließen wir also: L' ist nicht regulär.

2.6 Abgeschlossenheit regulärer Sprachen

Satz 2.15. *Die Vereinigung $L \cup M$ zweier regulärer Sprachen L, M ist regulär.*

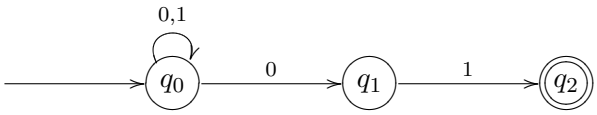
Beweis. Da L, M regulär, existieren reguläre Ausdrücke E, F, sodass $L = \mathrm{L}(E)$, $M = \mathrm{L}(F)$. Dann ist $(E + F)$ ein RA für die Sprache $\mathrm{L}(E) \cup \mathrm{L}(F) = L \cup M$. □

Satz 2.16. *Sei L regulär (über dem Alphabet Σ), dann ist das Komplement $\sim L = \Sigma^* - L$ ebenfalls regulär.*

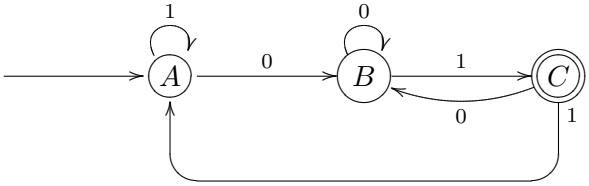
Beweis. Da L regulär, existiert ein DEA $A = (Q, \Sigma, \delta, q_0, F)$, sodass $L = \mathrm{L}(A)$. Dann gilt $\sim L = \mathrm{L}(B)$, wobei B der DEA $(Q, \Sigma, \delta, q_0, Q - F)$ ist. □

Beispiel 2.15. Wir betrachten die Sprache $L = \mathrm{L}((\mathbf{0} + \mathbf{1})^*\mathbf{01})$ und wollen $\sim L = \{\mathbf{0}, \mathbf{1}\}^* - \mathrm{L}((\mathbf{0} + \mathbf{1})^*\mathbf{01})$ bilden.

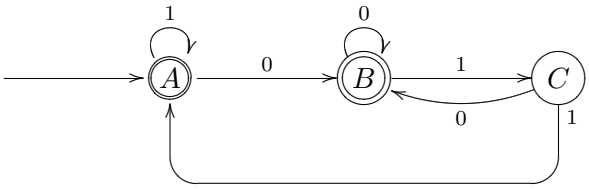
In Sektion 2.2 haben wir L studiert und den folgenden Automaten N dafür entworfen, siehe Beispiel 2.2. Es gilt: $L = \mathrm{L}((\mathbf{0} + \mathbf{1})^*\mathbf{01})$.



Mit Hilfe der Teilmengenkonstruktion konnten wir diesen Automaten in einen DEA A umwandeln.



Nun wenden wir die Konstruktion aus dem Beweis des Satzes 2.16 auf diesen Automaten an und erhalten für $\sim L = \{0, 1\}^* - \mathrm{L}((\mathbf{0} + \mathbf{1})^*\mathbf{01})$ den Automaten:



Beispiel 2.16. Betrachten Sie die Sprache M über $\{0, 1\}$, deren Worte eine ungleiche Anzahl von 0en und 1en enthalten. Wir zeigen, dass M *nicht regulär* ist.

Eine Anwendung des Pumping Lemmas erweist sich als schwierig. Angenommen wir haben ein Wort $w \in M$ gegeben und wollen nun für jede Aufteilung von w in x, y, und z, sodass $w = xyz$, ein k finden mit $x(y)^k z \notin M$. Sei nun y als Teilstring 01 angenommen, dann ändert sich für jedes k die Anzahl der 0en und 1en gleichermaßen, wenn wir y "aufpumpen", es gelingt uns also nicht zu zeigen, dass $x(y)^k z \notin M$.

Trotzdem ist es leicht einzusehen, dass M nicht regulär sein kann. Wir argumentieren mit Hilfe eines Widerspruchsbeweises. Angenommen M wäre regulär, dann wäre auch $\sim M$

regulär, unter Anwendung von Satz 2.16. Das Komplement von M ist jedoch gerade die Sprache L, wobei die Worte von L genauso viele 0en wie 1en enthalten. Die Nichtregularität von L haben wir in Beispiel 2.13 gezeigt. Widerspruch zu der Annahme, dass M regulär ist.

Satz 2.17. *Wenn L und M reguläre Sprachen sind, dann ist auch $L \cap M$ regulär.*

Beweis. Wir können das Gesetz von de Morgan anwenden.

$$L \cap M = \sim (\sim L \cup \sim M) \ .$$

Wir haben ja schon gezeigt dass reguläre Sprachen unter Vereinigung und Komplement abgeschlossen sind. □

Eine andere, direkte Beweismethode ist möglich, die wir kurz skizzieren. Wenn $L = \mathrm{L}(A)$, $M = \mathrm{L}(B)$, dann können wir den *Produktautomaten* $A \times B$ konstruieren.

Satz 2.18. *Wenn L und M reguläre Sprachen sind, dann ist auch $L - M$ regulär.*

Beweis. Verwende:

$$L - M = L \cap \sim M \ .$$

□

Die oben dargestellten Abschlusseigenschaften regulärer Sprachen, beziehen sich alle auf Boolsche Operationen. Nun wollen wir noch zwei weitere Abschlusseigenschaften untersuchen, nämlich *Spiegelung* und die *Homomorphismusbildung*.

Definition 2.12. Die *Spiegelung* einer Zeichenreihe $a_1 a_2 \cdots a_n$ besteht aus den in umgekehrter Reihenfolge angegebenen Symbolen: $a_n a_{n-1} \cdots a_1$.

Bei der Spiegelung L^R einer Sprache L, handelt es sich um die Menge der gespiegelten Worte in L: $L^R = \{w^R \mid w \in L\}$.

Satz 2.19. *Wenn L eine reguläre Sprache ist, dann ist auch L^R eine reguläre Sprache.*

Beweisskizze. Wir können annehmen es existiert ein RA E, sodass $L = \mathrm{L}(E)$. Dann wird der Satz mit struktureller Induktion nach E bewiesen. □

Aus dem (vollständig ausgeführten) Beweis lässt sich ein einfaches Verfahren zur Spiegelung von regulären Sprachen gewinnen. Sei L regulär und E ein RA über dem Alphabet Σ, sodass $L = \mathrm{L}(E)$. Dann erhalten wir die Sprache L^R indem wir die folgenden Operationen anwenden

Basis: Sei E der RA ϵ, $\varnothing$ oder e, für ein Eingabezeichen e. Dann ist $(E)^R = E$.

Schritt: Sei E ein zusammengesetzter RA:

- Angenommen $E \equiv (E_1)^*$. Dann ist $(E)^R = ((E_1)^R)^*$.
- Angenommen $E \equiv (DF)$. Dann ist $(E)^R = (F^R D^R)$.
- Angenommen $E \equiv (D + F)$. Dann ist $(E)^R = (D^R + F^R)$.

Beispiel 2.17. Sei $L = \mathrm{L}((\mathbf{0}+\mathbf{1})\mathbf{0}^*)$; dann ist L^R gegeben durch den RA

$$((\mathbf{0}+\mathbf{1})\mathbf{0}^*)^R = (\mathbf{0}^*)^R(\mathbf{0}+\mathbf{1})^R = (\mathbf{0}^R)^*(\mathbf{0}^R + \mathbf{1}^R) = \mathbf{0}^*(\mathbf{0}+\mathbf{1}) \ .$$

Definition 2.13. Seien Σ, z Alphabete, ein *Homomorphismus (über Σ)* ist eine Abbildung $h\colon \Sigma \to z^*$.

Sei h ein Homomorphismus, $w = a_1 \cdots a_n$ in Σ^*, dann ist $h(w)$ gleich

$$h(a_1)h(a_2)\cdots h(a_n) \ .$$

Beispiel 2.18. Setze $h(0) = ab$ und $h(1) = \epsilon$. Dann ist h ein Homomorphismus.

Für das Wort 1100 erhalten wir

$$h(1100) = h(1)h(1)h(0)h(0) = (\epsilon)(\epsilon)(ab)(ab) = abab \ .$$

Definition 2.14. Sei L eine Sprache über Σ und h ein Homomorphismus über Σ, dann ist $h(L) = \{h(w) \mid w \in L\}$

Den folgenden Satz geben wir ohne Beweis an.

Satz 2.20. *Wenn L eine reguläre Sprache über Σ, h ein Homomorphismus über Σ, dann ist $h(L)$ auch regulär.*

Wiederum lässt sich aus dem Beweis ein Verfahren zur Bestimmung von $h(L)$, für L eine reguläre Sprache gewinnen. Sei E ein RA über dem Alphabet Σ, sodass $L = \mathrm{L}(E)$. Wir erhalten die Sprache $h(L)$ indem wir in E, jedes Symbol $a \in \Sigma$ durch $h(a)$ ersetzen. (Um zu kennzeichnen, dass es sich hierbei um einen regulären Ausdruck handelt schreiben wir $\mathbf{h}(\mathbf{a})$ statt $h(a)$.)

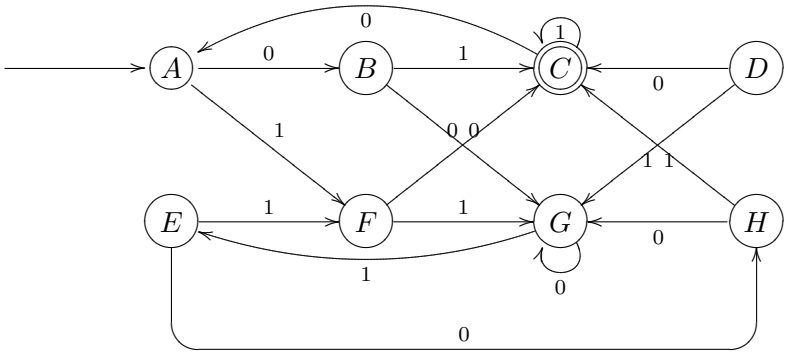
Beispiel 2.19. Sei $L = \mathrm{L}(\mathbf{10}^*\mathbf{1})$, dann gewinnen wir mit der beschriebenen Transformation den RA $(\mathbf{h}(\mathbf{1})\mathbf{h}(\mathbf{0})^*\mathbf{h}(\mathbf{1}))$. Es folgt

$$h(L) = \mathrm{L}((\mathbf{h}(\mathbf{1})\mathbf{h}(\mathbf{0})^*\mathbf{h}(\mathbf{1}))) = (\mathbf{ab})^* \ .$$

2.7 Minimierung von Automaten

2.7.1 Zustandsäquivalenz

Beispiel 2.20. Betrachte den folgenden DEA



Definition 2.15. Zwei Zustände p und q heißen *äquivalent* wenn

– für jeden String w, $\hat{\delta}(p, w)$ genau dann akzeptiert, wenn $\hat{\delta}(q, w)$ akzeptiert.

Sonst: Zustand p und q sind *unterscheidbar*.

Beispiel 2.21. Betrachte A und G in Beispiel 2.20: Die Wörter ϵ, 0, und 1 sind nicht geeignet die Zustände zu unterscheiden. Aber $\hat{\delta}(A, 01) = C$, $\hat{\delta}(G, 01) = E$; also sind A und G unterscheidbar.

Wir führen nun eine alternative Definition von "unterscheidbar" ein, welche direkt die Grundlage für ein algorithmischen Verfahren liefert. Im folgenden Satz wird gezeigt, dass sich die beiden Definitionen entsprechen.

Definition 2.16. Induktive Definition von *unterscheidbaren* Paaren in einem DEA $A = (Q, \Sigma, \delta, q_0, F)$.

1. Basis: Wenn p akzeptierend und q nicht, dann ist $\{p, q\}$ unterscheidbar.

2. Schritt: Seien p, q Zustände, sodass $\delta(p, a) = r$, $\delta(q, a) = s$ und $\{r, s\}$ ist unterscheidbar. Dann ist $\{p, q\}$ unterscheidbar.

Satz 2.21. *Wenn Zustände p, q nach der induktiven Definition nicht unterscheidbar sind, dann sind sie äquivalent.*

Aus der induktiven Definition erhalten wir unmittelbar einen Algorithmus zur Auffindung von unterscheidbaren Zustandspaaren. Diesen Algorithmus nennt man *Table-filling Algorithmus*. Bevor wir Satz 2.21 beweisen, wenden wir den Table-filling Algorithmus auf Beispiel 2.20 an.

Beispiel 2.22 (**Table-filling Algorithmus**).

B	$\times$						
C	$\times$	$\times$					
D	$\times$	$\times$	$\times$				
E		$\times$	$\times$	$\times$			
F	$\times$	$\times$	$\times$		$\times$		
G	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	
H	$\times$		$\times$	$\times$	$\times$	$\times$	$\times$
	A	B	C	D	E	F	G

Beweis (des Satzes). Betrachte $A = (Q, \Sigma, \delta, q_0, F)$. Angenommen die Aussage ist falsch, dh.

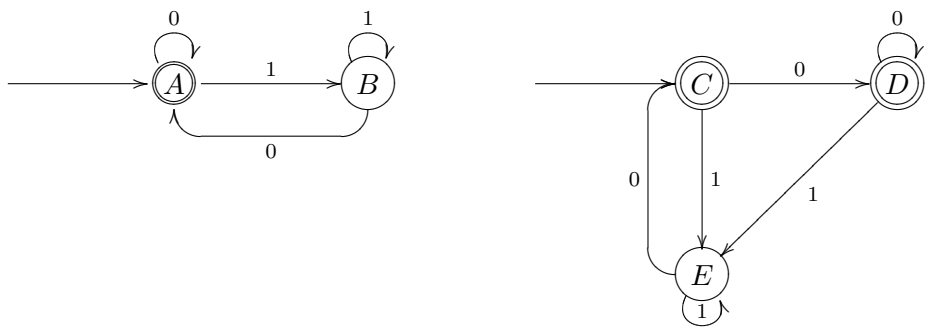
1. Es gibt zwei Zustände p, q, sodass ein String w existiert und genau einer der Zustände $\hat{\delta}(p, w)$ oder $\hat{\delta}(q, w)$ ist akzeptierend. Dh. p, q sind *nicht äquivalent.*

2. Aber das Paar $\{p, q\}$ ist nicht *unterscheidbar* nach dem Table-filling Algorithmus.

Solch ein Zustandspaar nennen wir ein *schlechtes Paar.* Wenn es schlechte Paare gibt, dann gibt es solche, deren unterscheidende Wörter minimale Länge haben. Wir wählen so ein Paar $\{p, q\}$ und können annehmen, dass w minimale Länge hat über alle Strings, die schlechte Paare unterscheiden. Beachte: $w \neq \epsilon$. Ansonsten hätte der Table-filling Algorithmus das Paar erkannt; also $w = a_1 a_2 \cdots a_n$. Betrachte $r = \delta(p, a_1)$, $s = \delta(q, a_1)$. Nach Voraussetzung sind r, s nicht äquivalent, da einer der Zustände $\hat{\delta}(r, a_2 \cdots a_n)$, $\hat{\delta}(s, a_2 \cdots a_n)$ nicht akzeptiert. Das Paar $\{r, s\}$ kann aber nicht schlecht sein, da $a_2 \cdots a_n$ echt kürzer als w. Also sind $\{r, s\}$ unterscheidbar nach dem Table-filling Algorithmus. Damit sind aber (IH) auch $\{p, q\}$ unterscheidbar nach dem Table-filling Algorithmus. Widerspruch. $\square$

Bemerkung. Beachten Sie, dass Satz 2.21—beziehungsweise der Beweis des Satzes—die Korrektheit des Table-filling Algorithmus zeigt.

Beispiel 2.23.



Beide DEA akzeptieren die Sprache $(\epsilon+(\mathbf{0}+\mathbf{1})^*\mathbf{0})$. Wir wenden den Table-filling Algorithmus an, wobei wir die beiden Automaten als einen einzigen mit zwei Startzuständen betrachten.

B	$\times$			
C		$\times$		
D		$\times$		
E	$\times$		$\times$	$\times$
	A	B	C	D

2.7.2 Minimierungsalgorithmus

Definition 2.17 (**Minimierungsalgorithmus** (informell))**.**

1. Alle Zustände eliminieren, die nicht vom Startzustand erreicht werden können.
2. Partitionierung der Zustände in Blöcke, sodass alle Zustände in einem Block äquivalent sind.

Beispiel 2.24. Wir betrachten den Automaten in Beispiel 2.20, wenn wir den Table-filling Algorithmus anwenden, dann bekommen wir die folgenden Blöcke von äquivalenten Zuständen: Die Blöcke sind $\{A, E\}$, $\{B, H\}$, $\{C\}$, $\{D, F\}$ und $\{G\}$.

Satz 2.22. *Die Äquivalenz von Zuständen ist transitiv. Wenn in $A = (Q, \Sigma, \delta, q_0, F)$ die Zustände p und q äquivalent sind und weiters q und r äquivalent sind, dann sind auch p und r äquivalent.*

Beweis. Angenommen die Paare $\{p, q\}$ und $\{q, r\}$ sind äquivalent, aber das Paar $\{p, r\}$ ist unterscheidbar. Dann existiert ein w, sodass genau einer der Zustände $\hat{\delta}(p, w)$ oder $\hat{\delta}(r, w)$ akzeptierend ist. O.b.d.A. nehmen wir an $\hat{\delta}(p, w)$ akzeptiert. Nun angenommen $\hat{\delta}(q, w)$ akzeptiert. Das bedeutet $\{q, r\}$ sind unterscheidbar. Widerspruch.

Also kann $\hat{\delta}(q, w)$ nicht akzeptieren. Das bedeutet aber $\{p, q\}$ ist unterscheidbar. Widerspruch zur Annahme, dass die Äquivalenz nicht transitiv ist. □

Satz 2.23. *Wir generieren für jeden Zustand eines DEA einen Block, der q und alle äquivalenten Zustände enthält. Dann formen die Blöcke eine* Partition *der Zustandsmenge.*

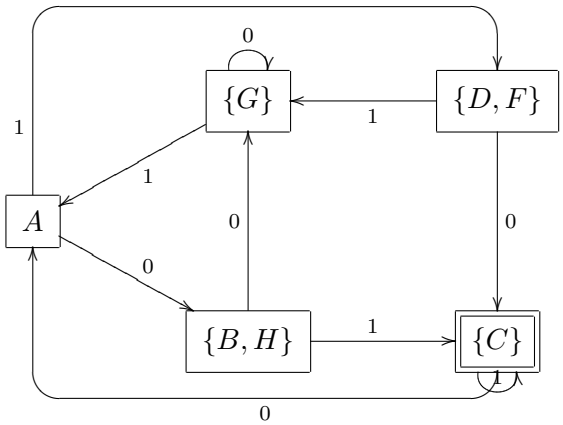
Definition 2.18 (**Minimierungsalgorithmus** (präzise))**.** Sei $A = (Q, \Sigma, \delta, q_0, F)$.

1. Alle Zustände eliminieren, die nicht vom Startzustand erreicht werden können.
2. Verwende den Table-filling Algorithmus um alle Paare äquivalenter Zustände zu finden.
3. Partitioniere die Zustände Q in Blöcke äquivalenter Zustände.

4. Konstruiere den minimalen DEA B wobei die Blöcke Zustände werden. Sei γ die Übergangsfunktion von B, S ein Block und a ein Eingabesymbol
   - Es existiert ein Block T, sodass für all $q \in S$ gilt $\delta(q, a) \in T$. Setze $\gamma(S, a) = T$.
   - Der Startzustand von B ist der Block, der q_0 enthält.
   - Die akzeptierenden Blöcke sind diejenigen, die Zustände aus F enthalten.

Beachte: Wenn ein Zustand in einem Block S akzeptiert, dann müssen alle Zustände in S akzeptieren.

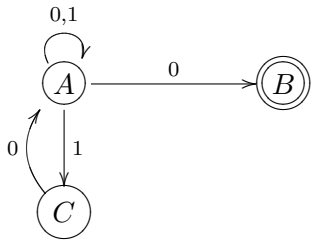
Beispiel 2.25. Wir wenden den Minimierungsalgorithmus auf den Automaten aus Beispiel 2.20 an und erhalten:



Satz 2.24. *Wenn A ein DEA ist und M ein DEA der durch Minimierung aus A erhalten wurde, dann hat M höchstens so viele Zustände, wie jeder zu A äquivalente DEA.*

Darüber hinaus ist der minimale DEA äquivalent zu A eindeutig bis auf Umbenennung.

Wenn man versucht den Minimierungsalgorithmus für nichtdeterministische Automaten anzuwenden, treten Probleme auf.



In diesem NEA sind die Zustände $\{A, B\}$ und $\{C, B\}$ unterscheidbar. Ebenso sind $\{A, C\}$ unterscheidbar bei Eingabe 0. Aber Zustand C kann einfach weggelassen werden, ohne die vom Automaten akzeptierte Sprache zu verändern.

3

Turing Maschinen und Berechenbarkeit

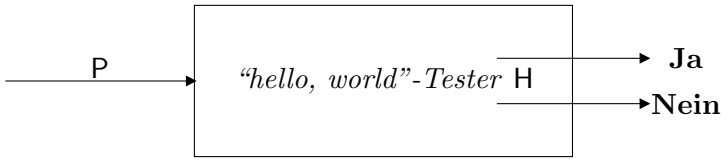
3.1 Einführung in die Berechenbarkeitstheorie

In dieser Sektion wollen wir uns einführend mit der Frage beschäftigen, ob alle Probleme algorithmisch lösbar sind. Hier bedeutet "algorithmisch lösbar", dass es einen *Algorithmus*, das heißt ein Programm gibt, welches das Problem löst. Leider müssen wir diese Frage mit "Nein" beantworten. In der Folge motivieren wir warum diese Antwort nicht wirklich überraschend ist. Zunächst betrachten wir ein sehr einfaches C-Programm P:

```
main()
{
      printf(‘‘hello, world\n’’);
}
```

Wie leicht einzusehen gibt P die Worte "hello, world" aus und terminiert. In der Folge nennen wir jedes Programm, das die Zeichenreihe "hello, world" als die ersten 12 Buchstaben seiner Ausgabe druckt ein *"hello, world"-Programm*. Beachte, dass wir dabei nicht voraussetzen, dass das Programm tatsächlich terminiert.

Nun untersuchen wir, ob es möglich ist ein Programm zu schreiben, das testet, ob ein bestimmtes Programm ein "hello, world"-Programm ist. Schematisch können wir einen hypothetisch angenommenen *"hello, world"-Tester* H wie folgt beschreiben:



Der Tester H erhält als Eingabe ein Programm P und antwortet entweder mit "Ja", wenn P ein "hello, world"-Programm ist und sonst mit "Nein". In Anbetracht der Einfachheit des Programms P erscheint diese Aufgabe recht einfach. Diese Einfachheit ist jedoch trügerisch. Dazu betrachten wir die folgende Variante P_1 des Programms P. Wir setzen die Funktion `exp` voraus, wobei $\texttt{exp}(x, y) = x^y$.

```
main()
{
      int n, summe, x, y, z;
      scanf(``%d'', &n);
      summe = 3;
      while (1) {
          for (x=1; x <= summe-2; x++)
              for (y=1; y<=summe-x-1; y++) {
                  z = summe - x - y;
                  if (exp(x,n) + exp(y,n) == exp(z,n))
                     printf(``hello, world\n'');
              }
          summe++;
      }
}
```

Es ist nicht schwer einzusehen, dass P_1 die Eingabe einer natürlichen Zahl n erwartet und dann prüft, ob es natürliche Zahlen x, y und z gibt ($x, y, z \geqslant 1$), sodass die Gleichung

$$x^n + y^n = z^n \ , \tag{3.1}$$

wahr wird. Wenn eine solche Lösung gefunden wird, wird der String "hello, world" gedruckt. Ist das Programm P_1 ein "hello, world"-Programm? Angenommen als Eingabe setzen wir $n = 2$. Dann ist leicht zu überprüfen, dass $x = 3$, $y = 4$ und $z = 5$ die Gleichung (3.1) löst. Das heißt für $n = 2$ ist P_1 ein "hello, world"-Programm. Was passiert nun für $n \geq 3$? Dann müssen wir feststellen, dass das Programm niemals den String "hello, world" schreibt, da die Gleichung (3.1) für $n \geq 3$ unlösbar ist. Dieser Tatbestand wurde von Pierre de Fermat bereits im 17. Jahrhundert vermutet, allerdings dauerte es über 300 Jahre bis der britische Mathematiker Andrew Wiles 1995 diese Vermutung auch tatsächlich beweisen konnte.

Es hat also die Anstrengungen von Generationen von Mathematikern bedurft, um festzustellen, dass das Programm P_1 in bestimmten Fällen *kein* "hello, world"-Programm ist. Wir betrachten eine weitere Variante P_2 von P. In P_2 setzten wir die Boolsche Funktion

`primes` voraus, wobei `primes`$(n) = 1$ genau dann wenn n eine Primzahl ist.[1]

```
main()
{
      int summe=4, x, y, test;
      while (1) {
          test = 0;
          for (x=2; x <= summe; x++) {
                  y = summe - x;
                  if (primes(x) && primes(y))
                     test = 1;
              }
          if !test printf(``hello, world\n'');
          summe=summe+2;
      }
}
```

Es ist nicht schwer einzusehen, dass das Programm P_2 ein "hello, world"-Programm ist, wenn eine gerade natürliche Zahl größer als 2 existiert, die nicht als Summe zweier Primzahlen geschrieben werden kann. Ob es überhaupt möglich ist, jede gerade natürliche Zahl größer als 2 als Summe zweier Primzahlen zu schreiben, ist zur Zeit nicht bekannt. Christian Goldbach hat diese Vermutung (die deshalb *Goldbachsche Vermutung* genannt wird) im 18. Jahrhundert aufgestellt. Ob die Vermutung korrekt ist oder nicht, ist bis heute offen.

Die beiden Varianten des einfachen anfänglichen "hello, world"-Programms zeigen uns, dass die Konstruktion eines "hello, world"-Testers keineswegs eine einfache Angelegenheit ist. In der Tat kann man beweisen, dass es keinen "hello, world"-Tester H geben kann. Wir formulieren allgemein das folgende Problem:

Problem. *Gegeben ein beliebiges Programm P. Ist P ein "hello, world"-Programm?*

Dieses Problem ist *algorithmisch nicht lösbar*, da wir keinen Algorithmus H wie oben angeben können. Probleme, die in diesem Sinn nicht gelöst werden können, nennen wir *unentscheidbar*. Wir schließen diese Sektion mit einer (sehr) unvollständigen Liste unentscheidbarer Probleme: Die folgenden Probleme sind *unentscheidbar*:

– Das Problem, ob ein beliebiges Programm auf seiner Eingabe terminiert. (*Halteproblem*)

[1] Eine solche Funktion wird auch *Primzahltest* genannt. Das Testen ob eine bestimmte natürliche Zahl eine Primzahl ist, ist ein lösbares Problem.

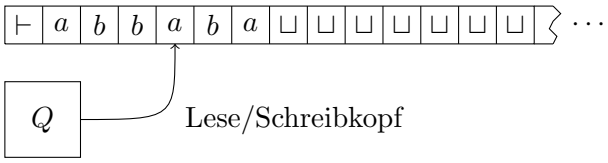


Abbildung 3.1: Schema einer Turingmaschine

– Postsches Korrespondenzproblem (PCP): Gegeben zwei Listen von Strings

$$w_1, w_2, \ldots, w_n \quad \text{und} \quad x_1, x_2, \ldots, x_n \ .$$

Gesucht sind Indizes $i_1, i_2, \ldots, i_m$, sodass

$$w_{i_1} w_{i_2} \ldots w_{i_m} = x_{i_1} x_{i_2} \ldots x_{i_m}$$

– Das Problem, ob eine beliebige gegeben formale Sprache regulär ist.

In den folgenden Sektionen werden wir die Theorie der Berechenbarkeit soweit entwickeln, dass wir die erste Behauptung beweisen können. Für die Beweise der restlichen Behauptungen wird auf [5] oder [4] verwiesen.

3.2 Turingmaschinen

Im Vergleich zu anderen Konzepten zur Beschreibung der Klasse der berechenbaren Funktionen, stellen Turingmaschinen eines der einfachsten abstrakten Berechnungsmodelle dar. Unter Annahme der Church-Turing These erhalten wir, dass jeder erdenkliche Algorithmus als Programm einer Turingmaschine dargestellt werden kann.

Zunächst beschreiben wir deterministische, 1-Band-Turingmaschinen. Eine *Turingmaschine* (abgekürzt TM) besteht aus einer endlichen Anzahl von Zuständen Q, einem einseitig unendlichen Band und einem Lese- und Schreibkopf, der eine Position nach links oder rechts wechseln kann und Symbole lesen, beziehungsweise schreiben kann. Das einseitig unendliche Band ist auf der linke Seite durch $\vdash$ begrenzt und unbeschränkt auf der rechten Seite. Skizze 3.1 liefert einen schematisierten Überblick.

Definition 3.1. Eine *deterministische, einbändige Turingmaschine* M ist ein 9-Tupel

$$M = (Q, \Sigma, \Gamma, \vdash, \sqcup, \delta, s, t, r) \ ,$$

sodass

1. Q eine endliche Menge von *Zuständen*,
2. Σ eine endliche Menge von *Eingabesymbolen*,
3. Γ eine endliche Menge von *Bandsymbolen*,
4. $\sqcup \in \Gamma \setminus \Sigma$, das *Blanksymbol*,
5. $\vdash \in \Gamma \setminus \Sigma$, der *linke Endmarker*,
6. $\delta \colon Q \times \Gamma \to Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}$ die *Übergangsfunktion*,
7. $s \in Q$, der *Startzustand*,
8. $t \in Q$, der *akzeptierende Zustand* und
9. $r \in Q$, der *verwerfende Zustand* mit $t \neq r$.

Informell bedeutet $\delta(p, a) = (q, b, d)$: "Wenn die TM M im Zustand p das Symbol a liest, dann ersetzt M a durch b und der Lese/Schreibkopf bewegt sich einen Schritt in die Richtung d." Wir verlangen, dass das Symbol $\vdash$ niemals überschrieben werden kann und die Maschine niemals über die linke Begrenzung hinaus fährt. Dies wird formal durch die folgende Bedingung festgelegt: Für alle $p \in Q$, existiert $q \in Q$ mit:

$$\delta(p, \vdash) = (q, \vdash, \mathrm{R}) \ . \tag{3.2}$$

Außerdem verlangen wir, dass die Maschine, sollte sie den akzeptierenden, beziehungsweise verwerfenden Zustand erreicht haben, diesen nicht mehr verlassen kann. Das heißt für alle $b \in \Gamma$ existieren $c, c' \in \Gamma$ und $d, d \in \{\mathrm{L}, \mathrm{R}\}$ sodass gilt:

$$\delta(t, b) = (t, c, d) \tag{3.3}$$

$$\delta(r, b) = (r, c', d') \tag{3.4}$$

Die Zustandsmenge Q und die Übergangsfunktion δ einer TM M werden wir auch als die *endliche Kontrolle* von M bezeichnen.

Beachten Sie, dass eine Turingmaschine M nach Definition 3.1 niemals zur Ruhe kommt. Zwar kann M in ihren akzeptierenden oder nicht-akzeptierenden Zustand wechseln und diesen dann auch nicht mehr verlassen, aber M bleibt trotzdem immer in Bewegung. Dennoch sprechen wir vom *Halten* der TM M, wenn M entweder den Zustand t oder r erreicht, andernfalls sagen wir: *M hält nicht* (oder auch *terminiert* nicht).

Beispiel 3.1. Die Turingmaschine $M = (\{s, p, t, r\}, \{\mathsf{0}, \mathsf{1}\}, \{\vdash, \sqcup, \mathsf{0}, \mathsf{1}\}, \delta, s, t, r)$ wechselt in den Zustand t, wenn der Bandinhalt den binären Nachfolger der Eingabe repräsentiert. In der Abbildung 3.2 geben die Übergangsfunktion δ tabellarisch an.

$p \in Q$	$a \in \Gamma$	$\delta(p, a)$
s	$\mathsf{0}$	$(s, \mathsf{0}, \mathrm{R})$
s	$\mathsf{1}$	$(s, \mathsf{1}, \mathrm{R})$
s	$\sqcup$	$(q, \sqcup, \mathrm{L})$
s	$\vdash$	$(s, \vdash, \mathrm{R})$
p	$\mathsf{0}$	$(t, \mathsf{1}, \mathrm{L})$
p	$\mathsf{1}$	$(p, \mathsf{0}, \mathrm{L})$
p	$\vdash$	$(t, \vdash, \mathrm{R})$

Abbildung 3.2: Eine Turingmaschine die den Nachfolger berechnet

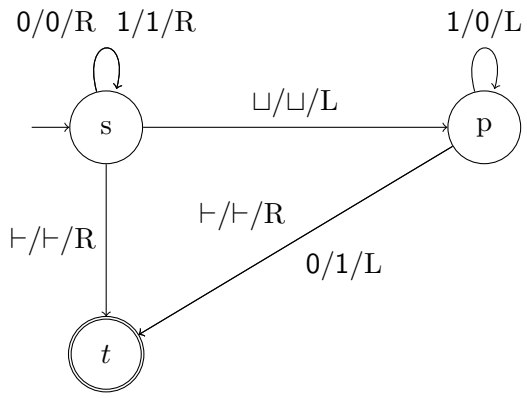


Abbildung 3.3: Übergangsdiagramm für die TM aus Beispiel 3.1

In der Darstellung wurde auf die Übergangsfunktionen für die Endzustände t und r verzichtet. Diese können beliebig eingetragen werden, solange die Bedingungen (3.3)–(3.4) erfüllt sind. Ebenso wurde auf den Eintrag für $\delta(p, \sqcup)$ verzichtet, da leicht nachzuprüfen ist, dass hier jeder beliebige Funktionswert stehen kann.

Wir können die Maschine M auch in der Form eines Übergangsdiagramms wie in Abbildung 3.2 darstellen. Wiederum verzichten wir auf die Darstellung der Übergangsfunktionen für die Endzustände t und r.

Beispiel 3.2. Die Turingmaschine $M = (\{s, q_0, q_1, q_0', q_1', t, r\}, \{\mathsf{0}, \mathsf{1}\}, \{\vdash, \sqcup, \mathsf{0}, \mathsf{1}\}, \delta, s, t, r)$ wechselt dann in den akzeptierenden Zustand t, wenn die Eingabe ein Palindrom über dem Alphabet $\{\mathsf{0}, \mathsf{1}\}$ mit gerader Länge darstellt, siehe Abbildung 3.4

Zu jedem Zeitpunkt enthält das Band einer TM M ein unendliches Wort der Form $y\sqcup^\infty$, wobei $\sqcup^\infty$ den unendlichen String

$$\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\sqcup\cdots,$$

$p \in Q$	$a \in \Gamma$	$\delta(p,a)$
s	$\mathsf{0}$	$(q_0, \vdash, \mathrm{R})$
s	1	$(q_1, \vdash, \mathrm{R})$
s	$\vdash$	$(s, \vdash, \mathrm{R})$
s	$\sqcup$	$(t, \sqcup, \mathrm{L})$
q_0	0	$(q_0, \mathsf{0}, \mathrm{R})$
q_0	1	$(q_0, 11, \mathrm{R})$
q_0	$\sqcup$	$(q_0', \sqcup, \mathrm{L})$
q_1	$\mathsf{0}$	$(q_1, \mathsf{0}, \mathrm{R})$
q_1	1	$(q_1, \mathsf{1}, \mathrm{R})$
q_1	$\sqcup$	$(q_1', \sqcup, \mathrm{L})$

$p \in Q$	$a \in \Gamma$	$\delta(p,a)$
q_0'	$\mathsf{0}$	$(q, \sqcup, \mathrm{L})$
q_0'	$\mathsf{1}$	$(r, 1, \mathrm{L})$
q_0'	$\vdash$	$(t, \vdash, \mathrm{R})$
q_1'	$\mathsf{0}$	$(r, \mathsf{1}, \mathrm{L})$
q_1'	$\mathsf{1}$	$(q, \sqcup, \mathrm{L})$
q_1'	$\vdash$	$(t, \vdash, \mathrm{R})$
q	$\mathsf{0}$	$(q, \mathsf{0}, \mathrm{L})$
q	1	$(q, \mathsf{1}, \mathrm{L})$
q	$\vdash$	$(s, \vdash, \mathrm{R})$

Abbildung 3.4: Turingmaschine die Palindrome akzeptiert

bezeichnet. Obwohl der String $y\sqcup^\infty$ unendlich ist, ist dieser endlich repräsentierbar.

Definition 3.2. Eine *Konfiguration* einer TM M ist ein Tripel (p, x, n), sodass

- $p \in Q$ der aktuelle Zustand,
- $x = y\sqcup^\infty$ der aktuellen Bandinhalt ($y \in \Gamma^*$) und
- $n \in \mathbb{N}$ die Position des Lese/Schreibkopfes am Band.

Wir verwenden griechische Buchstaben vom Anfang des Alphabets um Konfigurationen zu bezeichnen. Die *Startkonfiguration* bei Eingabe $x \in \Sigma^*$ ist die Konfiguration

$$(s, \vdash x\sqcup^\infty, 0)\ .$$

In der Folge definieren wir eine binäre Relation zwischen Konfigurationen, um einen Rechenschritt einer Turingmaschine konzise formalisieren zu können.

Definition 3.3. Sei $z \in \Gamma^*$, wir schreiben z_n für das n-te Symbol des Wortes z. Die Relation $\xrightarrow[M]{1}$ ist wie folgt definiert:

$$(p, z, n) \xrightarrow[M]{1} \begin{cases} (q, z', n-1) & \text{wenn } \delta(p, z_n) = (q, b, \mathrm{L}) \\ (q, z', n+1) & \text{wenn } \delta(p, z_n) = (q, b, \mathrm{R}) \end{cases}$$

Hier bezeichnet z' den String, den wir aus z erhalten, wenn $(z)_n$ (an der Position n) durch b ersetzt wird.

Definition 3.4. Wir definieren die reflexive, transitive Hülle $\xrightarrow[M]{*}$ von $\xrightarrow[M]{1}$ induktiv:

1. $\alpha \xrightarrow[M]{0} \alpha$

2. $\alpha \xrightarrow[M]{n+1} \beta$, wenn $\alpha \xrightarrow[M]{n} \gamma \xrightarrow[M]{1} \beta$ für eine Konfiguration γ und
3. $\alpha \xrightarrow[M]{*} \beta$, wenn $\alpha \xrightarrow[M]{n} \beta$ für ein $n \geqslant 0$.

Beispiel 3.3. Betrachte die Turingmaschine M aus Beispiel 3.1 mit Eingabe 0010. Dann gilt:

$$(s, \vdash \mathsf{0010}\sqcup^\infty, 0) \xrightarrow[M]{*} (q_0, \vdash\vdash \mathsf{010}\sqcup^\infty, 5) \xrightarrow[M]{*} (q_0', \vdash\vdash \mathsf{010}\sqcup^\infty, 4) \xrightarrow[M]{*} (q, \vdash\vdash \mathsf{01}\sqcup^\infty, 3)$$

Definition 3.5. Die Turingmaschine M *akzeptiert* die Eingabe $x \in \Sigma^*$, wenn gilt

$$(s, \vdash x\sqcup^\infty, 0) \xrightarrow[M]{*} (t, y, n) \ ,$$

für y und n beliebig und *verwirft* x, wenn

$$(s, \vdash x\sqcup^\infty, 0) \xrightarrow[M]{*} (r, y, n) \ ,$$

für y und n beliebig. Wir sagen M *hält* bei Eingabe x, wenn M entweder x akzeptiert oder verwirft. Andernfalls *terminiert* M auf x *nicht*. Eine TM M heißt *total*, wenn sie auf allen Eingaben hält. Die Menge $\mathrm{L}(M)$ bezeichnet die Menge aller von M akzeptierten Wörter.

Beispiel 3.4. Betrachte die TM M aus Beispiel 3.2. Es gilt dass M genau die Menge $A = \{ww^R \mid w \in \{\mathsf{0},\mathsf{1}\}^*\}$ akzeptiert, also gilt $A = \mathrm{L}(M)$. Die Menge A wird auch als die Menge der Palindrome gerader Länge bezeichnet.

Eine Sprache L (oder allgemein eine Menge) heißt *rekursiv aufzählbar* (abgekürzt *r.e.* für *recursive enumerable*), wenn eine Turingmaschine M existiert, sodass $L = \mathrm{L}(M)$. Eine Sprache L heißt *co-r.e.* wenn sie das Komplement einer r.e. Sprache und *rekursiv*, wenn es eine *totale* TM M gibt, sodass $L = \mathrm{L}(M)$.

Satz 3.1. *Rekursive Mengen sind unter Komplementbildung abgeschlossen.*

Beweis. Angenommen $A = \mathrm{L}(M)$, wobei die TM M total. Dann definieren wir eine TM M' indem der akzeptierende und der verwerfende Zustand von M vertauscht werden. Dann gilt offensichtlich $\sim A = \mathrm{L}(M')$ und M' ist total. □

Satz 3.2. *Jede rekursive Menge ist rekursiv aufzählbar. Andererseits ist nicht jede rekursiv aufzählbare Menge rekursiv.*

Beweis. Der erste Teil des Satzes ist eine triviale Konsequenz der Definitionen. Den Beweis des zweiten Teiles werden wir in Sektion 3.4 nachholen. □

Satz 3.3. *Wenn A und $\sim A$ rekursiv aufzählbar sind, dann ist A rekursiv.*

Beweis. Angenommen es existieren Turingmaschinen M und M', sodass $A = \mathrm{L}(M)$ und $\sim(A) = \mathrm{L}(M')$. Wir definieren eine neue TM N, die bei Eingabe x die Maschinen M und M' simultan auf x ausführt. Dazu teilen wir das Band von N in eine obere und untere Hälfte, sodass wir M auf der oberen und M' auf der unteren Hälfte ausführen können. Formal erreichen wir das, indem das Bandalphabet von N die folgenden Symbole enthält:

$$\begin{array}{|c|}\hline a \\ \hline c \\ \hline\end{array} \qquad \begin{array}{|c|}\hline \hat{a} \\ \hline c \\ \hline\end{array} \qquad \begin{array}{|c|}\hline a \\ \hline \hat{c} \\ \hline\end{array} \qquad \begin{array}{|c|}\hline \hat{a} \\ \hline \hat{c} \\ \hline\end{array}$$

Hier haben wir angenommen dass a ein Bandsymbol von M und c ein Bandsymbol von M' ist. Die Zusatzmarkierung $\hat{\cdot}$ wird verwendet, um die Position des Lese/Schreibkopfes von M, M' zu markieren. Das Band von N kann also folgende Gestalt haben:

$$\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|}\hline b & \hat{b} & a & b & a & a & a & a & b & a & a & a \\ \hline c & c & c & d & d & d & c & \hat{c} & d & c & d & c \\ \hline \end{array} \cdots$$

Die Maschine N führt abwechselnd einen Schritt von M oder einen Schritt von M' aus. Dazu werden die jeweiligen Zustände von M und M' durch zusätzliche Zustände in N gespeichert. Wenn M jemals akzeptiert, hält N und akzeptiert. Andererseits wenn M' akzeptiert, dann verwirft N. Nach Voraussetzung muss einer der Fälle für jede Eingabe auf jeden Fall eintreten. Somit hält N auf allen Eingaben und ist total. □

Eine Eigenschaft P von Wörtern nennen wir *entscheidbar*, wenn die Menge aller Strings mit der Eigenschaft P rekursiv ist. Eine Eigenschaft P heißt *semi-entscheidbar*, wenn die Menge $\{x \mid P(x)\}$ rekursiv aufzählbar ist. Somit folgt aus Beispiel 3.4, dass die Eigenschaft, ein Palindrome gerader Länge zu sein, entscheidbar ist.

3.3 Äquivalente Formulierungen

3.3.1 Mehrere Bänder

Eine offensichtliche Erweiterung von Definition 3.1 ist die Erweiterung um mehrere Bänder und Lese/Schreibköpfe, wie in Bild 3.5 dargestellt.

Die Übergangsfunktion dieser Turingmaschine hat den folgenden Typ:

$$\delta\colon Q \times \Gamma^3 \to Q \times \Gamma^3 \times \{\mathrm{L}, \mathrm{R}\}^3 \ .$$

Die Erweiterung diesen Beispieles zu einer formalen Definition einer *mehrbändigen*, *deterministischen Turingmaschine* überlassen wir der Leserin. Mehrbändige Turingmaschinen werden auch als *Mehrband-Turingmaschinen* bezeichnet.

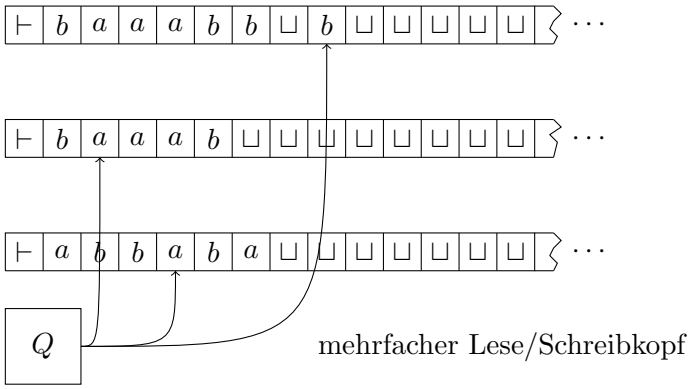


Abbildung 3.5: Mehrband-Turingmaschine

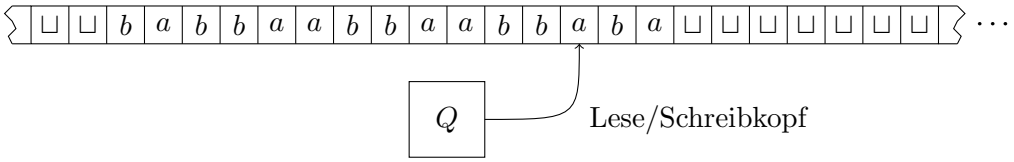


Abbildung 3.6: Beidseitig unendliches Lese/Schreibband

Satz 3.4. *Sei M eine deterministische Turingmaschine mit k Bändern. Dann existiert eine einbändige, deterministische Turingmaschine M', sodass* $\mathrm{L}(M) = \mathrm{L}(M')$.

Beweisskizze. In der Literatur finden sich zwei unterschiedliche Ansätze eine Mehrbandmaschine durch eine Einbandmaschine zu simulieren. Entweder werden die k Bänder von M hintereinander, durch Sonderzeichen getrennt, auf dem Band von M' simuliert. Oder das Band von M' wird in mehrere Bereiche geteilt, um die k Bänder von M simulieren zu können. Dabei folgt man der entsprechenden Konstruktion im Beweis von Satz 3.3. Die genaue Ausarbeitung dieser Beweisskizzen überlassen wir der Leserin. □

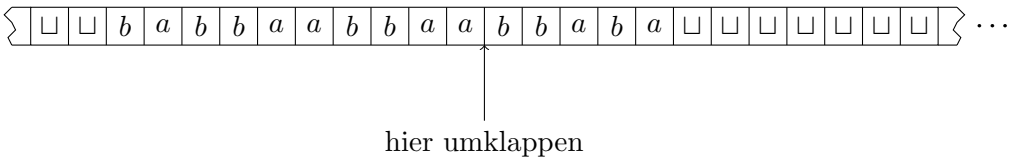
3.3.2 Zweiseitig Unbeschränkte Bänder

Eine andere einfache Möglichkeit Definition 3.1 zu erweitern ist es, das Lese/Schreibband sowohl nach links als auch nach rechts unbeschränkt zu definieren, wie in Abbildung 3.3.2 dargestellt.

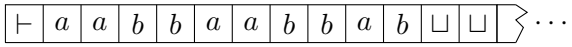
Auch diese Erweiterung erhöht die Ausdrucksfähigkeit des Turingmaschinenmodells nicht.

Satz 3.5. *Sei M eine einbändige, deterministische Turingmaschine, dessen Band in beide Richtungen unbeschränkt ist. Dann existiert eine einbändige, deterministische Turingmaschine M', sodass* $\mathrm{L}(M) = \mathrm{L}(M')$.

Beweisskizze. Wir zeigen den Satz, indem wir M durch eine 2-bändige Turingmaschine simulieren. In der Simulation wird das zweiseitig unendliche Band an einer beliebigen Stelle geknickt wie die folgende Grafik verdeutlicht:



Wir erhalten die folgende Repräsentation des ursprünglichen Bandes, wobei das obere Band verwendet wird, um den Bandinhalt links von der Knickstelle darzustellen und das untere Band für den Bandinhalt rechts von der Knickstelle.



□

3.3.3 Nichtdeterminismus

Eine nichtdeterministische, mehrbändige Turingmaschine N erweitert die einbändige Turingmaschine einerseits um mehrere Bänder und andererseits um die Möglichkeit, dass die Berechnungen der Maschine zu jedem Zeitpunkt anhand von mehreren Möglichkeiten voranschreiten kann.

Definition 3.6. Eine *nichtdeterministische, k-bändige Turingmaschine* (*NTM*) N ist ein 9-Tupel

$$N = (Q, \Sigma, \Gamma, \vdash, \sqcup, \delta, s, t, r) \ ,$$

sodass

1. Q eine endliche Menge von *Zuständen*,
2. Σ eine endliche Menge von *Eingabesymbolen*,
3. Γ eine endliche Menge von *Bandsymbolen*,
4. $\sqcup \in \Gamma \setminus \Sigma$, das *Blanksymbol*,
5. $\vdash \in \Gamma \setminus \Sigma$, der *linke Endmarker*,
6. $\delta \colon Q \times \Gamma^k \to \mathcal{P}(Q \times \Gamma^k \times \{\mathrm{L}, \mathrm{R}\}^k)$ die *Übergangsfunktion*,
7. $s \in Q$, der *Startzustand*,

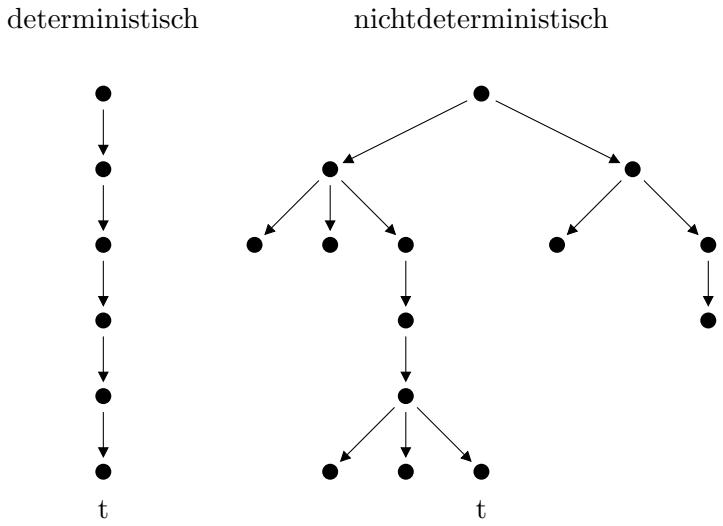


Abbildung 3.7: Darstellung des Berechnungsbaumes

8. $t \in Q$, der *akzeptierende Zustand* und

9. $r \in Q$, der *verwerfende Zustand* mit $t \neq r$.

Die Rechenschritte einer NTM können als Baum aufgefasst werden, dessen Wege die unterschiedlichen Möglichkeiten die Berechnung fortzusetzen repräsentiert. Schematisch lässt sich der Unterschied zwischen Determinismus und Nichtdeterminismus wie in Abbildung 3.7 Damit eine NTM N ihre Eingabe akzeptiert, muss nur eine Möglichkeit, also ein Pfad, existieren, sodass N auf einen akzeptierenden Zustand trifft.

Satz 3.6. *Sei N eine nichtdeterministische TM, dann existiert eine deterministische TM M, sodass $\mathrm{L}(M) = \mathrm{L}(N)$. Umgekehrt wird jede von einer deterministischen TM akzeptierte Sprache auch von einer nichtdeterministischen TM akzeptiert.*

Beweis. Es genügt den ersten Teil des Satzes zu beweisen, da der zweite Teil leicht folgt. Sei N eine nichtdeterministische TM. Wir konstruieren eine deterministische dreibändige TM M', welche nach Satz 3.4 in eine einbändige TM umgewandelt werden kann.

Sei x das Eingabewort für N. Das erste Band von M' wird immer nur diese Eingabe x enthalten. Auf dem zweite Band simulieren wir die Rechenvorgänge von N relativ zu einem fixen Weg durch den Baum. Schließlich dient das dritte Band dazu den aktuellen Pfad zu adressieren. Beachten Sie, dass der aktuell betrachtete Pfad nicht notwendigerweise in einem Blatt enden muss.

Zur Adressierung der Wege wird ein String über dem Alphabet $\Sigma_b = \{1, 2, \ldots, b\}$ verwendet, wobei b den maximalen Grad eines inneren Knoten im Berechnungsbaum darstellt. Anders ausgedrückt bezeichnet b den Grad des Nichtdeterminismus von N. Offensichtlich repräsentiert jeder String $w \in \Sigma_b^*$ entweder eine eindeutige Position in diesem Berechnungsbaum oder ist ungültig, wenn an dieser Stelle im Baum weniger als b Möglichkeiten bestehen. Das Leerwort ϵ bezeichnet die Wurzel des Berechnungsbaumes, das heißt den Beginn der Berechnung. Aufbauend auf diese Kodierung können wir das Verhalten von N wie folgt kodieren:

1. Anfangs enthält Band 1 von M' das Eingabewort x, die Bänder 2 und 3 sind leer.
2. Kopiere den Inhalt von Band 1 auf Band 2.
3. Verwende Band 2, um die Rechenschritte von N auf x zu simulieren. Bei jeder Stelle in der Berechnung in der die Übergangsfunktion δ mehrere Möglichkeiten zulässt, sieht M' auf Band 3 nach, welche Möglichkeit gewählt werden soll. Es wird also genau ein Pfad im Berechnungsbaum auf Band 2 simuliert. Wenn die Adresse auf Band 3 ungültig ist, oder keine weiteren Rechenschritte mehr betrachtet werden können, gehe zu Punkt (4). Andernfalls, wenn die Simulation von N akzeptiert, dann akzeptiere.
4. Ersetzte den String auf Band 3 durch den nächstgrößeren in der graduiert-lexikographische Ordnung $<_{\text{lex}}$. Gehe zu Schritt (2).

□

3.3.4 Zwei Keller

Ein *2-Kellerautomat* ist ein endlicher Automat, der zusätzlich zu einem Leseband noch zwei Keller (oder *Stapel*) zur Verfügung hat. Wir begnügen uns mit der genannten informellen Definition und überlassen es der Leserin 2-Kellerautomaten, sowie dem Begriff der *Sprache* $\mathrm{L}(K)$ eines Kellerautomaten formal einzuführen. Der folgende Satz zeigt dass 2-Kellerautomaten und Turingmaschinen äquivalent sind.

Satz 3.7. *Sei K ein Kellerautomat, dann existiert eine TM M, sodass $\mathrm{L}(M) = \mathrm{L}(K)$. Umgekehrt gibt es zu jeder TM M' einen Kellerautomaten K', sodass $\mathrm{L}(K') = \mathrm{L}(M')$.*

Beweisskizze. Die Simulation eines 2-Kellerautomaten durch eine Turingmaschine mit drei Bändern ist trivial: Das obere Band dient der Repräsentation des Lesebandes und die unteren beiden Bänder der Darstellung der Keller.

Andererseits kann jede TM durch eine 2-Kellermaschine wie folgt simuliert werden: Der Bandinhalt links vom Lese/Schreibkopf wird auf dem ersten Keller, der Bandinhalt rechts vom Lese/Schreibkopf auf dem zweiten Keller dargestellt. Dabei ist nur darauf zu achten,

dass die Symbole, die dem Lese/Schreibkopf am nächsten sind, auf den beiden Stapeln am höchsten liegen. Dann kann das Lesen und Schreiben durch Kopieren von einem Stapel auf den anderen simuliert werden. □

3.3.5 Registermaschinen

Eine *k-Registermaschine* ist eine Maschine, die mit einem Leseband und zusätzlich k Registern, die natürlichen Zahlen aufnehmen können ausgestattet ist. Jedes Register kann eine beliebige Zahl in $\mathbb{N}$ speichern. In jedem Schritt kann der Automat unabhängig jeden Registerinhalt um eins erhöhen und verkleinern. Außerdem erlauben die Instruktionen den Test der Register auf 0 und das Verschieben des Lesekopfs (in beide Richtungen) um eine Zelle.

Satz 3.8. *Sei R eine k-Registermaschine, dann existiert eine Turingmaschine M, sodass $\mathrm{L}(M) = \mathrm{L}(R)$. Umgekehrt gibt es zu jeder TM M' eine 2-Registermaschine R', sodass $\mathrm{L}(R') = \mathrm{L}(M')$.*

Beweis. Sei R eine k-Registermaschine. Diese kann am einfachsten durch eine $k+1$-Band Turingmaschine M'' simuliert werden indem jedem Register ein eigenes Arbeitsband entspricht. Als Bandalphabet für die zusätzlichen Bänder können wir $\{0, 1\}$ verwenden und den Inhalt der Register binär kodieren. Dann ist es nicht schwer entsprechende Turingmaschinenprogramme zu schreiben, die das Inkrementieren und Dekrementieren von Registern kodieren. Um abschließend den ersten Teil des Satzes zu beweisen, muss nur noch die Maschine M'' durch eine einbändige TM M' simuliert werden.

Andererseits sei M' die Turingmaschine, die wir durch eine 2-Registermaschine simulieren wollen. Dazu wandeln wir zunächst M' in einen 2-Kellerautomaten um und zeigen wie wir einen Stapel durch zwei Register simulieren können. Anschließend skizzieren wir wie die erhaltene 4-Registermaschine durch eine 2-Registermaschine simuliert werden kann.

Wir zeigen die Simulierung eines Kellers durch zwei Register. Dazu können wir o.b.d.A annehmen, dass das Kelleralphabet nur die Symbole 0 und 1 enthält: Sollte das Alphabet mehr Symbole enthalten können wir diese leicht durch verschiedene Binärstrings kodieren. Wir gehen also davon aus, dass der Kellerinhalt als Binärzahl dargestellt werden kann, dessen niedrigwertigste Ziffer ganz oben am Stapel liegt. Die Simulation speichert die Zahl im ersten Register und verwendet das zweite Register um Stapeloperationen zu simulieren. Etwa um eine 0 oben auf den Stapel zu legen muss der Stapelinhalt (als Zahl gelesen) verdoppelt werden. Dazu wird eine Schleife aufgerufen, die iterativ das erste Register dekrementiert und in jedem Schritt 2 zum Inhalt des zweiten Registers dazu zählt. In einer ähnlichen Weise wird simuliert, dass eine 1 auf den Stapel gelegt werden soll. Um das Entfernen des obersten Elements zu simuliert müssen wir nur den Registerinhalt durch 2 dividieren und durch einen $\mod 2$ Test feststellen, ob der Stapel von einer 0 oder einer 1 gekrönt war.

Damit haben wir die Simulation eines 2-Kellerautomaten durch eine 4-Registermaschine vollständig beschrieben.

Wir skizzieren die Simulation einer 4-Registermaschine R'' durch eine 2-Registermaschine R'. Angenommen die Register von R'' enthalten die Werte i, j, k und l. Dann schreiben wir die Zahl $2^i \cdot 3^j \cdot 5^k \cdot 7^l$ in das ersten Register von R'. Das zweite Register dient zur Simulation der Rechenoperationen von R''. □

3.3.6 Aufzählmaschinen

Wir haben die rekursiv aufzählbaren Mengen als Mengen definiert, die durch eine Turingmaschine akzeptiert werden. Der Ausdruck *rekursiv aufzählbar* kommt allerdings von einem alternativen Maschinenmodell, den *Aufzählmaschinen*.

Wir verstehen unter einer Aufzählmaschine, einen Automaten mit einer endlichen Kontrolle und zwei Bändern. Das erste Band ist eine Lese/Schreibband, das auch *Arbeitsband* genannt wird. Das zweite Band ist das *Ausgabeband* auf das nur geschrieben werden kann. Es gibt kein Eingabeband und keinen akzeptierenden oder verwerfenden Zustand. Wie in einer Turingmaschine werden die Rechenschritte durch eine Übergangsfunktion kodiert. Von Zeit zu Zeit gelangt die Aufzählmaschine in einen speziellen Zustand, den *Aufzählungszustand*. Wenn das passiert, wird das aktuell am Ausgabeband stehende Wort als *aufgezählt* bezeichnet. Das Wort wird gelöscht und die Maschine generiert das nächste, aufzuzählende Wort. Eine Aufzählmaschine terminiert nicht. Es ist möglich, dass eine Aufzählmaschine E niemals in den Aufzählungszustand gelangt, dann gilt $\mathrm{L}(E) = \varnothing$, oder die Sprache die von E aufgezählt wird ist unendlich. Andererseits kann das selbe Wort mehrmals aufgezählt werden.

Satz 3.9. *Die Familie der Mengen, die von einer Aufzählungsmaschine aufgezählt werden können entspricht genau der Familie von r.e. Mengen. In anderen Worten, eine Menge ist* $\mathrm{L}(E)$ *für eine Aufzählungsmaschine* E *genau dann, wenn diese Menge* $\mathrm{L}(M)$ *einer Turingmaschine* M *ist.*

Beweis. Der vollständige und nicht schwere Beweis kann in [5] nachgelesen werden. □

3.4 Universelle Maschinen und Diagonalisierung

In Sektion 3.1 haben wir behauptet, dass es nicht algorithmisch feststellbar ist, dass ein beliebiges Programm auf seiner Eingabe hält. Dazu werden wir zeigen, dass die Menge der Turingmaschinen, die auf ihrer Eingabe halten nicht rekursiv ist. Gleichzeitig ist diese Menge aber rekursiv aufzählbar, also werden wir damit auch ein Beispiel einer Menge, die r.e. aber nicht rekursiv ist, angeben. Eine wichtige Grundlage für dieses Resultat ist die Tatsache, dass Turingmaschinen ihren eigenen Code interpretieren können.

Die Eigenschaft der Selbstinterpretierbarkeit der Programmen von Turingmaschinen ist ebenfalls in modernen Programmiersprachen höherer Stufe (wie etwa `Lisp`, `Scheme`, `Ocaml`, `Prolog`) gegeben. Dies erlaubt ein Programmierparadigma dass man als *Metaprogrammierung* bezeichnet.

3.4.1 Universelle Turingmaschinen

Um das Programm einer TM für eine TM lesbar zu machen, müssen wir uns auf eine sinnvolle Kodierung von Turingmaschinen einigen. Diese Kodierung sollte einfach genug sein, sodass alle notwendige Informationen wie etwa Anzahl der Zustände, Übergangsfunktion, Eingabe- und Bandalphabet leicht und problemlos aus der Kodierung gewonnen werden können.

Als Beispiel könnte die Kodierung mit dem folgenden String beginnen:

$$0^n 1 0^m 1 0^k 1 0^s 1 0^t 1 0^r 1 0^u 1 0^v 1 \ .$$

Wobei dieses Wort bedeutet, dass die dargestellte TM M, n Zustände $\{0, \ldots, n-1\}$ wobei s der Startzustand, t der akzeptierende Zustand und r der verwerfende Zustand ist. Außerdem enthält das Band die m Symbole $\{0, \ldots, \mathsf{m}-1\}$ wobei $\{0, \ldots, \mathsf{k}-1\}$, Symbole des Eingabealphabets sind. Schließlich bezeichnet das Symbol u den linken Endmarker und v das Blanksymbol. Hier steht als letztes Zeichen 1, um die Kodierung der TM von der Kodierung der Übergangfunktion abzugrenzen. Der Rest des Codesegments kann nun aus einer Sequenz von Wörtern bestehen, welche die Übergangsfunktion kodieren, etwa könnte $\delta(p, a) = (q, b, d)$ wie folgt dargestellt werden:

$$0^p 1 0^a 1 0^q 1 0^b 1 1 \ .$$

Hier steht als letztes Zeichen die“0” für die Richtung L und die “1” für R.

Definition 3.7. Aufbauend auf eine geeignete Kodierung von Turingmaschinen, konstruieren wir eine *universelle Turingmaschine* U sodass

$$\mathrm{L}(U) = \{\ulcorner M \urcorner \sharp \ulcorner x \urcorner \mid x \in \mathrm{L}(M)\} \ .$$

Das Symbol $\sharp$ ist ein Hilfssymbol im Alphabet von U, um den Code $\ulcorner M \urcorner$ der TM M vom Code $\ulcorner x \urcorner$ der Eingabe für M abzugrenzen. Die Maschine U operiert wie folgt:

1. Zunächst prüft U, ob $\ulcorner M \urcorner$ tatsächlich der Code einer TM M ist und $\ulcorner x \urcorner$ die Kodierung eines Eingabewortes über dem Alphabet von M darstellt. Falls nicht, wird U sofort verwerfen.

2. Wenn die Kodierung gültig ist, simuliert U die Maschine M auf der Eingabe x. Dazu werden am besten drei Bänder verwendet. Auf Band 1 steht während der ganzen

Simulation die Beschreibung von M. Auf Band 2 wird das (dekodierte) Eingabewort x geschrieben und zu jedem Zeitpunkt repräsentiert das dritte Band den (simulierten) Bandinhalt des Bandes von M. (O.b.d.A. können wir annehmen dass M nur ein Band besitzt.)

3. Wenn M jemals auf der Eingabe x hält, hält U ebenfalls und akzeptiert, beziehungsweise verwirft entsprechend.

Wenn keine Unklarheiten auftauchen können lassen wir die expliziten Hinweise auf die Kodierung weg und schreiben $\mathrm{L}(U) = \{M\sharp x \mid x \in \mathrm{L}(M)\}$.

3.4.2 Unentscheidbarkeit des Halteproblems

Wir zeigen nun wie die Existenz von universellen Turingmaschinen zusammen mit der *Diagonalisierungmethode* verwendet wird um das *Halteproblem* und *Zugehörigkeitsproblem* von Turingmaschinen als unentscheidbar nachzuweisen. Eine Anwendung der Diagonalisierungsmethode wurde bereits in "Diskrete Mathematik 1" behandelt, siehe [2, Satz 3.10].

Definition 3.8. Wir definieren, die dem *Halteproblem* und *Zugehörigkeitsproblem* von Turingmaschinen entsprechenden Mengen:

$$\begin{aligned} \mathsf{HP} &:= \{M\sharp x \mid M \text{ hält bei Eingabe } x\} \\ \mathsf{MP} &:= \{M\sharp x \mid x \in \mathrm{L}(M)\}\ . \end{aligned}$$

Wir machen uns die Kodierungsmöglichkeit von Turingmaschinen zunutze, um eine Aufzählung aller Turingmaschinen anzugeben.

Definition 3.9. Sei M_x eine TM mit *Eingabealphabet* $\{\mathsf{0},\mathsf{1}\}$, deren Code x ist. Als *Kodierungsalphabet* verwenden wir wiederum $\{\mathsf{0},\mathsf{1}\}$. Wenn x keine gültige Beschreibung einer Turingmaschine darstellt, definieren wir M_x als eine beliebige aber fixe TM über dem Alphabet $\{\mathsf{0},\mathsf{1}\}$.

Als Konsequenz von Definition 3.9 erhalten wir eine unendliche Liste von Turingmaschinen, deren Programmcodes in aufsteigender Reihenfolge geordnet sind.

$$M_\epsilon, M_{\mathsf{0}}, M_{\mathsf{1}}, M_{\mathsf{00}}, M_{\mathsf{01}}, M_{\mathsf{10}}, M_{\mathsf{000}}, M_{\mathsf{001}}, \ldots \tag{3.5}$$

Diese Liste enthält alle möglichen Turingmaschinen über dem Alphabet $\{\mathsf{0},\mathsf{1}\}$. Wir betrachten eine zweidimensionale Matrix, einerseits indiziert mit Wörtern $w \in \{0,1\}^*$ und

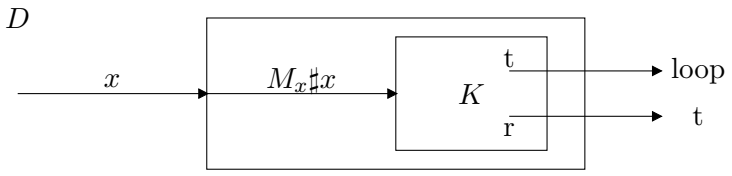


Abbildung 3.8: Diagonialisierungsmaschine

andererseits mit den Turingmaschinen aus der Liste (3.5).

	ϵ	0	1	00	01	10	11	000	001	010	...
M_ϵ	$\times$	$\circlearrowright$	$\circlearrowright$	$\times$	$\times$	$\circlearrowright$	$\times$	$\circlearrowright$	$\times$	$\times$	
M_0	$\circlearrowright$	$\circlearrowright$	$\times$	$\times$	$\circlearrowright$	$\times$	$\times$	$\circlearrowright$	$\circlearrowright$	$\times$	
M_1	$\circlearrowright$	$\times$	$\circlearrowright$	$\times$	$\circlearrowright$	$\times$	$\times$	$\circlearrowright$	$\circlearrowright$	$\times$	
M_{00}	$\times$	$\circlearrowright$	$\circlearrowright$	$\times$	$\times$	$\times$	$\times$	$\circlearrowright$	$\circlearrowright$	$\times$	
M_{01}	$\times$	$\times$	$\times$	$\times$	$\circlearrowright$	$\circlearrowright$	$\circlearrowright$	$\times$	$\times$	$\circlearrowright$	$\cdots$
M_{10}	$\times$	$\times$	$\circlearrowright$	$\times$	$\times$	$\circlearrowright$	$\times$	$\times$	$\circlearrowright$	$\times$	
M_{11}	$\times$	$\times$	$\circlearrowright$	$\circlearrowright$	$\times$	$\circlearrowright$	$\times$	$\circlearrowright$	$\times$	$\circlearrowright$	
M_{000}	$\times$	$\times$	$\times$	$\times$	$\circlearrowright$	$\times$	$\times$	$\circlearrowright$	$\times$	$\circlearrowright$	
M_{001}	$\circlearrowright$	$\times$	$\times$	$\times$	$\times$	$\circlearrowright$	$\times$	$\times$	$\times$	$\times$	
$\vdots$						$\vdots$					$\ddots$

Die $x$te Zeile dieser Matrix beschreibt für jedes Eingabewort y ob M_x auf y hält oder nicht. Hier wird das Halten von M_x durch $\times$ und das Nichtterminieren durch $\circlearrowright$ ausgedrückt.

Satz 3.10. *Die Menge* HP *ist nicht rekursiv, aber rekursiv aufzählbar.*

Beweis. Wir verfahren indirekt: Angenommen es existiert eine totale TM K, sodass $\mathsf{HP} = \mathrm{L}(K)$. Basierend auf K definieren wir eine *Diagonaliserungsmaschine* D:

1. Die TM D erhält als Eingabe einen String $x \in \{0,1\}^*$. Mit Hilfe von x konstruiert D die Maschine M_x und schreibt $M_x\sharp x$ auf ihr Eingabeband.

2. Dann simuliert D die TM K auf der Eingabe $M_x\sharp x$, allerdings startet D eine nicht-terminierende Schleife, wenn K akzeptieren würde und akzeptiert, wenn K verwirft.

Zeichnung 3.8 stellt die Maschine D graphisch dar.

Es ist leicht einzusehen, dass die Maschine D genau das Komplement der Diagonale der oben dargestellten Matrix berechnet. Im Summe gilt:

$$\begin{aligned} D \text{ hält auf } x &\Leftrightarrow K \text{ verwirft } M_x\sharp x && \text{Definition von } D \\ &\Leftrightarrow M_x \text{ terminiert nicht auf } x && \text{Definition von } K \end{aligned}$$

Damit ist das Verhalten von D verschieden von jeder der TM M_x auf zumindest einem Eingabewort. Aber die Liste (3.5) sollte alle TM enthalten. Widerspruch zur Annahme, dass HP rekursiv ist. Somit ist HP nicht rekursiv.

Andererseits ist es nicht schwer zu sehen, dass die Menge HP durch eine Aufzählungsmaschine aufgezählt werden kann. Nach Satz 3.9 ist HP also rekursiv aufzählbar. □

Satz 3.11. *Die Menge* MP *ist rekursiv aufzählbar, aber nicht rekursiv.*

Beweis. Wiederum ist es einfach eine Aufzählungsmaschine anzugeben, welche Elemente von MP aufzählt, somit ist MP r.e. Um zu zeigen, dass MP nicht rekursiv ist, argumentieren wir wieder indirekt.

Angenommen MP wäre rekursiv, dann existiert eine total TM K, sodass $\mathsf{MP} = \mathrm{L}(K)$. Angenommen wir wollten nun für eine beliebige TM M feststellen, ob M auf x hält. Dann transformieren wir M wie folgt: Die neue TM N, ist genau wie M definiert, aber mit der Ausnahme, dass N einen neuen akzeptierenden Zustand t besitzt und wir von den alten akzeptierenden und verwerfenden Zuständen von M nach t wechseln, wenn diese von M erreicht werden. Also akzeptiert N das Wort x genau dann, wenn M auf x hält. Nun können wir die Maschine K verwenden um das Halteproblem zu lösen. Nach Satz 3.9 ist dieses jedoch unentscheidbar. Widerspruch zur Annahme der Existenz von K. □

Im Beweis von Satz 3.11 haben wir eine Standardtechnik angewandt, um die Unentscheidbarkeit bestimmter Eigenschaften zu zeigen: die *Reduktion* eines Problems (in diesem Fall das Halteproblem) auf ein anderes (in diesem Fall das Zugehörigkeitsproblem). Im Rahmen der Komplexitätstheorie werden wir solche Reduktionen intensiv studieren.

Eine kritische Analyse des Beweises von Satz 3.10 zeigt, dass wir zwar bewiesen haben, dass das Halteproblem für TM unentscheidbar ist, dass wir damit aber eigentlich nicht gezeigt haben dass das Halteproblem für *alle* Programme unentscheidbar ist. Anderseits ist es ein Leichtes ein Programm geschrieben etwa in C, OCaml, oder Prolog in ein Turingmaschinenprogramm umzuwandeln. Darüber hinaus gilt sogar, dass jedes im Laufe der letzten Dekaden vorgeschlagene abstrakte Rechenmodell äquivalent zu Turingmaschinen ist.

Diese Beobachtung hat schon in den 1930er Jahren die Grundlage für die sogenannte *Church-Turing These* geliefert:

These. *Jedes algorithmisch lösbare Problem ist auch mit Hilfe einer Turingmaschine lösbar.*

Die folgenden Sätze geben wir ohne Beweis an und verweisen die interessierte Leserin auf [5] oder [4].

Definition 3.10. Wir definieren das Postsches Korrespondenzproblem (*PCP*). Gegeben zwei Listen von Strings

$$w_1, w_2, \ldots, w_n \quad \text{und} \quad x_1, x_2, \ldots, x_n \,.$$

Gesucht sind Indizes $i_1, i_2, \ldots, i_m$, sodass

$$w_{i_1} w_{i_2} \ldots w_{i_m} = x_{i_1} x_{i_2} \ldots x_{i_m}$$

Satz 3.12. *Das Postsches Korrespondenzproblem ist semi-entscheidbar, aber nicht entscheidbar.*

Satz 3.13. *Das Problem, ob eine beliebige gegeben formale Sprache regulär ist weder entscheidbar, noch semi-entscheidbar.*

4

Komplexitätstheorie

4.1 Einführung in die Komplexitätstheorie

Komplexitätsabschätzungen haben Sie schon im Kapitel 2 des Skriptums "Diskrete Mathematik 1" kennen gelernt (siehe [2]). Etwa wurde in Satz 2.5. gezeigt, dass die Nachfolgesuche in polynomiell vielen Schritte (in der Anzahl der Knoten und Kanten) alle von einer Startmenge S aus erreichbaren Knoten in einem gerichteten Multigraphen G markiert. Eine weitere Art der Komplexitätsabschätzung ist durch das Master-Theorem (Satz 3.14) gegeben.

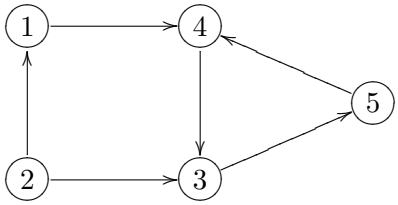
Die Abschätzungen, die wir jetzt anstellen werden, sind etwas verschieden. Hat uns vorher die Komplexität eines bestimmten Algorithmus interessiert, wird nun ein bestimmtes *Problem* in den Vordergrund rücken. Die verwendeten Algorithmen rücken dabei in den Hintergrund. Diese Art von Komplexitätsabschätzungen, die sich hauptsächlich mit der Komplexität der Lösung von Problemen beschäftigt wird in der *Komplexitätstheorie* studiert.

Die Komplexitätstheorie analysiert Algorithmen und Probleme hinsichtlich der von ihnen benötigten Ressourcen wie etwa Speicherplatz oder Rechenzeit. Hierbei bezeichnet der Ausdruck *Problem* eine allgemeine, zu beantwortende Frage; ein Verfahren zur Beantwortung dieser Frage heißt *Algorithmus*. Beide Begrifflichkeiten haben wir oben bereits verwendet. Zu jedem Problem gibt es nun viele verschiedene Algorithmen unterschiedlichster Komplexität. Unter der Komplexität eines Algorithmus verstehen wir die notwendigerweise aufgewandten Ressourcen (in Relation zur Eingabe), um das Problem zu lösen. Im Besonderen sind wir daran interessiert *wie lange* ein Algorithmus braucht, um ein Problem zu lösen (*Laufzeitkomplexität*) oder *wie viel Platz* benötigt wird (*Speicherplatzkomplexität*). Die Komplexität eines Problems ist nun die Komplexität des effizientesten Algorithmus, um das Problem zu lösen. Wir betrachten beispielhaft drei (nur scheinbar artifizielle) Probleme.

Problem. *Gegeben einen gerichteten Graph G und Knoten $s, t \in E$, wobei E die Eckenmenge von E bezeichnet. Gibt es einen (gerichteten) Pfad zwischen s und t. Dieses Problem nennt man* MAZE.

Zur Verdeutlichung betrachten wir ein einfaches Beispiel.

Beispiel 4.1.



Wie schon angedeutet, kann das Problem MAZE mittels der Nachfolgersuche (Satz 2.5 in [2]) gelöst werden:

1. Setze $S = \{s\}$ und markiere s.
2. Solange S nicht leer ist, wiederhole:
   - Wähle eine Ecke $e \in S$ und entferne sie aus S.
   - Bestimme alle unmarkierten unmittelbaren Nachfolger von e, markiere sie und gebe sie zu S hinzu.
3. Antwort "ja" wenn t markiert, sonst "nein".

Aus Satz 2.5 folgt, dass die *(Zeit)-Komplexität* des Suchalgorithmus $\mathrm{O}(n^2)$ ist, wenn n die Summe der Knoten und Kanten darstellt. Je nach Wahl der Auswahlfunktion kann die Suche *depth-first* oder *breadth-first* sein. Der *Platzbedarf* des Suchalgorithmus ist $\mathrm{O}(n)$.

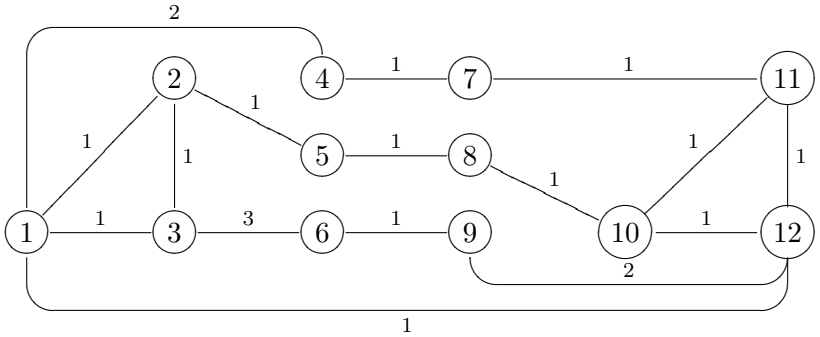
Problem. *Gegeben n Städte, mit Distanz $d_{ij} > 0$, sodass $d_{ij} = d_{ji}$. Was ist die* billigste Tour *durch die Städte? Gesucht ist das Minimum der Kostensumme:*

$$\sum_{i=1}^{n} d_{\pi(i),\pi(i+1)} \ ,$$

wobei $\pi\colon \{1, \ldots, n\} \to \{1, \ldots, n\}$ eine Permutation ist und wir definieren $\pi(n+1) := \pi(1)$. Dieses Problem nennt man Problem des Handlungsreisenden *(*traveling salesperson problem*, kurz* TSP*).*

Beachten Sie, dass die Tour in jeder beliebigen Stadt beginnen kann, aber wieder zum Ausgangspunkt zurückführen muss.

Beispiel 4.2.



Um die Darstellung zu vereinfachen, haben wir einige Kanten weggelassen, wir nehmen an dass die Distanz zwischen den Städten, wo Kanten fehlen, unendlich ist. Die billigste Tour ist gegeben durch

$$\langle 1, 4, 7, 11, 10, 8, 5, 2, 3, 6, 9, 12, 1 \rangle \ .$$

Ein (naiver) Algorithmus würde etwa wie folgt verfahren:

1. Generieren aller möglichen Lösungen
2. Berechnen der Kosten
3. Die Tour mit den geringsten Kosten wählen.

Die Zeitkomplexität ist $\mathrm{O}(n!)$ und die Platzkomplexität $\mathrm{O}(n)$. Die Zeitkomplexitätsschranke lässt sich leicht verbessern, bleibt aber *exponentiell*! Das heißt das Problem ist nur für recht kleine Probleminstanzen mit verträglichen Zeitaufwand lösbar.

Als letztes Problem, betrachten wir das Problem festzustellen, ob ein Spieler das Spiel *verallgemeinerte Länderkunde* immer gewinnen kann. Hierbei ist das Spiel *verallgemeinerte Länderkunde* wie folgt definiert. Gegeben seien ein gerichteter Graph G und ein ausgewählter Startknoten s. Das Spiel *verallgemeinerte Länderkunde* wird von zwei Spielern gespielt. Spielerin I beginnt in der Ecke (dem "Land") s und wählt eine in G erreichbare Ecke ("Land"), das markiert wird. Dann zieht Spieler II und markiert das erreichte Land wieder und so weiter. Derjenige Spieler, der kein (unmarkiertes) Land mehr erreichen kann, verliert.

Problem. *Seien ein gerichteter Graph und ein ausgewählter Startknoten s gegeben. Das* verallgemeinerte Länderkundeproblem *(*generalised geography problem*, kurz* GG*) ist das Problem, ob Spielerin I das Spiel immer gewinnen kann, wenn sie im Land s beginnt. Es wird also nach einer Gewinnstrategie für Spielerin I gesucht.*

Die folgende rekursive Entscheidungsprozedur A stellt fest, ob Spielerin I eine Gewinnstrategie hat. Dabei bezeichnet G den Graphen, E, K Eckenmenge und Kantenmenge und b das aktuell besuchte Land.

1. Wenn der Ausgangsgrad von b, 0 ist, dann verliert Spielerin I immer, deshalb verwerfen wir die Eingabe.

2. Eliminiere Knoten b und alle wegführenden Kanten. Der erhaltene Graph wird mit G' bezeichnet.

3. Für jede der Ecken b' aus der Menge der unmittelbaren Nachfolger von b (in G), rufe A rekursiv mit (G', b') auf.

4. Wenn alle Aufrufe akzeptieren, dann hat Spieler II eine Gewinnstrategie, also verwerfe. Andererseits hat Spieler II keine Gewinnstrategie, dafür Spielerin I, also akzeptiere.

Wir schätzen nur den Platzbedarf des Algorithmus ab. Der einzige Platz den A benötigt, ist der Platz für die rekursiven Aufrufe. In jedem rekursiven Aufruf wird eine Ecke gespeichert. Die Anzahl der Ecken ist durch $|E|$ beschränkt, also ist der Platzbedarf linear in Größe des Graphen G.

4.2 Laufzeitkomplexität

Wenn wir nun die drei eingeführten Probleme näher betrachten stellt sich natürlich die Frage, ob die angegebenen Algorithmen wirklich die bestmöglichen sind. Und wie schwierig die jeweiligen Probleme im Vergleich tatsächlich sind. Um diese beantworten zu können führen wir *Komplexitätsklassen* ein.

Definition 4.1. Sei M eine deterministische und totale TM. Die *Laufzeit* oder *Laufzeitkomplexität* von M ist die Funktion $T\colon \mathbb{N} \to \mathbb{N}$, wobei $T(n)$ die maximale Anzahl von Schritten von M auf allen Eingaben der Länge n bezeichnet. Wir sagen auch dass $T(n)$ die Laufzeit von M ist, oder dass M eine $T(n)$-Zeit Turingmaschine ist.

Definition 4.2. Sei $T\colon \mathbb{N} \to \mathbb{N}$ eine numerische Funktion. Die deterministische Zeitkomplexitätsklasse $\mathsf{DTIME}(T(n))$ ist wie folgt definiert:

$$\mathsf{DTIME}(T(n)) := \{\mathrm{L}(M) \mid M \text{ ist eine deterministische Mehrband-TM mit Laufzeit } \mathrm{O}(T(n))\ \}$$

die Klasse $\mathsf{DTIME}(T(n))$ enthält also alle Sprachen, deren Zugehörigkeitstest von einer TM in Zeit $\mathrm{O}(T(n))$ entschieden werden kann.

Wir können das Problem MAZE auch als Sprache betrachten, wenn wir eine geeignetes Alphabet zur Kodierung von Graphen voraussetzen.[1]

[1] In diesem Kapitel verwenden wir die Begriffe *Sprache* und *Problem* synonym, da man jedes algorithmisches Problem geeignet als formale Sprache darstellen kann und umgekehrt.

$$\mathsf{MAZE} = \{(G, s, t) \mid G \text{ ist ein gerichteter Graph mit einem Weg von } s \text{ nach } t\}$$

In ähnlicher Weise können wir die Sprachen TSP und GG definieren. Für das "Traveling Sales Person"-Problem ist dabei zu beachten, dass wir dieses Problem als *Optimierungsproblem* definiert haben, das heißt die kostengünstigste Tour wird gesucht. Um TSP als Sprache verstehen zu können, müssen wir es erst als Entscheidungsproblem definieren. Wir geben zunächst eine äquivalente graphentheoretische Formulierung des Problems TSP.

Problem. *Ein* Hamiltonscher Kreis *in einem Graphen ist ein Kreis, der jede Ecke (genau einmal) besucht. Gegeben ein bewerteter ungerichteter Graph G, dann ist der Hamiltonsche Kreis mit der minmalen Bewertung gesucht. Dieses Problem wird* Traveling Salesperson Problem (TSP) *genannt.*

Der Einfachheit halber bezeichnen wir das Optimierungsproblem und das Entscheidungsproblem gleich.

$$\mathsf{TSP} = \{(G, b, B) \mid G \text{ ist ein Graph mit Bewertung } b \text{ der einen Hamiltonschen Kreis mit Bewertung kleiner gleich } B \text{ besitzt } \}$$

Beispiel 4.3. Sei $(G, s, t) \in \mathsf{MAZE}$, das heißt in G gibt es einen gerichteten Weg von s nach t. Wir haben oben gesehen, dass diese Tatsache mit Hilfe einer deterministischen Turingmaschine in Laufzeit $\mathrm{O}(n^2)$ entschieden werden kann. Also gilt $\mathsf{MAZE} \in \mathsf{DTIME}(n^2)$.

Definition 4.3. Sei M eine nichtdeterministische und totale TM. Die *Laufzeit* oder *Laufzeitkomplexität* von M ist die Funktion $T\colon \mathbb{N} \to \mathbb{N}$, wobei $T(n)$ die maximale Anzahl von Schritten von M auf jedem möglichen Pfad im Berechnungsbaum auf allen Eingaben der Länge n ist.

Definition 4.4. Sei $T\colon \mathbb{N} \to \mathbb{N}$ eine numerische Funktion. Die nichtdeterministische Zeitkomplexitätsklasse $\mathsf{NTIME}(T(n))$ ist wie folgt definiert:

$$\mathsf{NTIME}(T(n)) := \{\mathrm{L}(M) \mid M \text{ ist eine nichtdeterministische Mehrband-TM mit Laufzeit } \mathrm{O}(T(n)) \}$$

4.3 Die Klassen P und NP

Wir interessieren uns für das asymptotische Wachstum von Komplexitätsschranken, also sind Laufzeitkomplexitätsfunktionen, die polynomiell sind, solchen die exponentiell wachsen deutlich vorzuziehen.

Im Besonderen interessieren wir uns für diejenigen Probleme die durch einen Algorithmus, der in polynomieller Zeit läuft, berechnet werden kann. Diese Probleme werden in der Komplexitätsklasse P zusammengefasst. Hier steht das P für *P*olynomielle Zeit. Die Möglichkeit von nicht-deterministischen Algorithmen motiviert das Interesse an einer zweiten Komplexitätsklasse, der Klasse NP. Hier steht NP für *N*ichtdeterministische *P*olynomielle Zeit. Diese Klasse kann am einfachsten als die Menge derjenigen Probleme beschrieben werden, deren *Lösung* einfach, d.h. in polynomineller Zeit, verifiziert werden kann. Die Komplexität des Problems liegt in der *Suche* einer geeigneten Lösung. Wir werden sehen, dass etwa $\mathsf{TSP} \in \mathsf{NP}$.

In der Folge definieren wir die Klassen P und NP formal:

Definition 4.5. Die Komplexitätsklasse P fasst alle Sprachen zusammen, deren Zugehörigkeitstest in polynomieller Zeit durch eine deterministische Turingmaschine entschieden werden kann, also

$$\mathsf{P} = \bigcup_{k \geqslant 1} \mathsf{DTIME}(n^k) \ .$$

Beispiel 4.4. Es gilt $\mathsf{MAZE} \in \mathsf{P}$.

Wir geben zwei unterschiedliche, aber äquivalente, Definitionen der Klasse NP. Zunächst formalisieren wir die eingangs erwähnte Intuition. Dies Definition erleichtert auch das Feststellen, ob eine bestimmte Sprache in NP ist oder nicht.

Definition 4.6. Ein *Verifikator* einer Sprache L ist ein Algorithmus V, sodass

$$L = \{x \mid \text{es existiert ein String } c\text{, sodass } V \text{ akzeptiert } (x, c)\} \ .$$

Ein *polytime Verifikator* ist ein Verifikator mit Laufzeit $\mathrm{O}(n^k)$ (k beliebig), wobei $n = \ell(x)$. Das Wort c wird auch als *Zertifikat* bezeichnet.

Die Klasse NP ist definiert als die Klasse der Sprachen L, die einen polytime Verifikator haben.

Beispiel 4.5. Es gilt $\mathsf{TSP} \in \mathsf{NP}$. Das ist am leichtesten dadurch einzusehen, indem die optimale Tour als Zertifikat c betrachtet wird. Gegeben eine Tour c ist es leicht möglich mit Hilfe eines deterministischen Algorithmus nachzuprüfen, dass diese Tour wirklich eine Bewertung kleiner gleich B hat.

Analog können wir NP definieren als die Klasse von Sprachen, deren Zugehörigkeitstest in polynomieller Zeit durch eine *nichtdeterministische* Turingmaschine entschieden werden kann. Solange wir noch nicht gezeigt haben, dass diese Definition äquivalent zur obigen ist, verwenden wir eine andere Notation.

Definition 4.7.

$$\mathsf{NP}' = \bigcup_{k \geqslant 1} \mathsf{NTIME}(n^k) \ .$$

Beispiel 4.6. Es gilt $\mathsf{TSP} \in \mathsf{NP}'$. Das ist am leichtesten dadurch einzusehen, dass mit einer nichtdeterministischen Turingmaschine zuerst eine bestimmte Tour geraten werden kann, die danach auf in polynomieller Zeit auf die Einhaltung des Budgets getestet werden kann.

Satz 4.1. *Es gilt* $\mathsf{NP} = \mathsf{NP}'$*, das heißt die Klasse der Sprachen, die einen polytime Verifikator haben ist gleich der Klasse von Sprachen, die durch eine NTM entschieden werden.*

Beweis. Sei $L \in \mathsf{NP}'$. Dann existiert eine nichtdeterministische TM, N, sodass $L = \mathrm{L}(N)$. Wir definieren eine polytime Verifikator V mit Eingabe (x, c), wobei x und c Wörter sind.

1. Simuliere auf der Eingabe x die NTM N, wobei die Symbole in c als Auswahlsequenz verwendet werden, um die Nichtdeterminismuspunkte aufzulösen. (Vergleiche dazu mit dem Beweis von Satz 3.6.)

2. Wenn N akzeptiert, akzeptiert V, ansonsten verwerfe.

Andererseits sei $L \in \mathsf{NP}$. Das heißt es existiert ein polytime Verifikator V für L. Dann konstruieren wir eine nichtdeterministische Maschine N aus V mit Eingabe x. Wir nehmen an V läuft in Zeit n^k.

1. Wähle (nichtdeterministisch) einen String c, der Länge n^k.

2. Simuliere V auf (x, c).

3. Wenn V akzeptiert, dann akzeptiere, sonst verwerfe.

□

4.4 Many-One Reduktionen

Am Ende von Kapitel 3.4.2 haben wir den Begriff derh *Reduktion* informell eingeführt. Nun werden wir diesen Begriff, genauer den Begriff der polynomiellen many-one Reduktion, formal einführen.

Definition 4.8. Eine Funktion $f\colon \Sigma^* \to \Sigma^*$ heißt in *polynomieller Zeit* berechenbar, wenn es eine deterministische totale, k-bändige TM T mit Eingabealphabet Σ gibt, sodass die folgenden Bedingungen erfüllt sind:

1. T läuft in polynomieller Zeit, das heißt es existiert ein $k \in \mathbb{N}$ sodass T eine $\mathrm{O}(n^k)$-Zeit Turingmaschine ist.

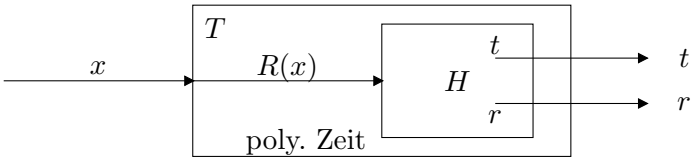


Abbildung 4.1: Polynomielle Reduktion: $x \in A \iff R(x) \in B$

2. Bei Eingabe $x \in \Sigma^*$, schreibt T $f(x)$ auf das k-te Band.

Definition 4.9. Seien L, M Sprachen über dem Alphabet Σ. Wir sagen, dass L auf M in polynomieller Zeit reduzierbar ist ($L \leqslant_m^{\mathrm{p}} M$), wenn eine in polynomieller Zeit berechenbare Funktion $R\colon \Sigma^* \to \Sigma^*$ existiert, sodass gilt:

$$x \in L \iff R(x) \in M\ .$$

Wir verwenden Reduktionen, wenn wir zeigen wollen dass ein bestimmtes Problem A nicht schwieriger zu lösen ist als ein gegeben Problem B. Sei etwa für das Problem B schon ein Algoithmus bekannt, der B entscheidet. Wir nennen diesen Algorithmus H. Gelte nun $A \leqslant_m^{\mathrm{p}} B$, wobei wir die Reduktion R verwendet haben. Dann können wir mit Hilfe von R das Problem A entscheiden.

In Abbildung 4.1 wird der Zusammenhang anschaulich beschrieben, der Entscheidungsalgorithmus für A wird mit T bezeichnet. Da es sich bei R um eine in polynomieller Zeit berechenbare Reduktion handelt, wissen wir auch, dass das Laufzeitverhalten von H und T sich nur um einen polynomialen Faktor unterscheidet.

Das folgende Lemma folgt unmittelbar aus der Definition.

Lemma 4.1. *Seien A und B beliebige Probleme. Wenn $A \leqslant_m^p B$ und $B \in \mathsf{P}$ ($B \in \mathsf{NP}$), dann $A \in \mathsf{P}$ ($A \in \mathsf{NP}$).*

Definition 4.10. Sei $\mathcal{C}$ eine beliebige Komplexitätsklasse (also etwa P oder NP), dann ist eine Sprache A *$\leqslant_m^p$-hart* für $\mathcal{C}$, wenn für alle Sprachen $B \in \mathcal{C}$ gilt: $B \leqslant_m^{\mathrm{p}} A$. Eine Sprache A heißt *vollständig für $\mathcal{C}$ in Bezug auf $\leqslant_m^p$*, wenn gilt

1. A ist $\leqslant_m^{\mathrm{p}}$-hart für $\mathcal{C}$ und
2. $A \in \mathcal{C}$.

Satz 4.2. *Das Problem TSP ist vollständig für NP in Bezug auf $\leqslant_m^p$.*

Als einfache Konsequenz von Satz 4.2 sehen wir, dass es extrem unwahrscheinlich ist, dass das Problem TSP in polynomieller (deterministischer) Zeit gelöst werden kann. Denn wäre das der Fall, dann könnte *jedes* Problem in der Komplexitätsklasse NP in polynomieller (deterministischer) Zeit gelöst werden.

Bemerkung. Wenn Sie einen deterministischen Algorithmus für TSP finden, der in polynomieller Zeit läuft, dann lösen Sie ein 1 Millionen Dollar Problem!

> Die Direktoren von CMI [Clay Mathematics Institute] haben 7 Millionen Dollar Preisgeld für die sogenannten *Millenium Probleme* ausgeschrieben. Eines der Millenium Probleme ist die Frage nach $\mathsf{P} = \mathsf{NP}$.

4.5 Logarithmisch Platzbeschränkte Reduktionen

In der vorangehenden Sektion haben wir Reduktionen betrachtet, die in polynomieller Zeit berechnet werden können. In manchen Situation, etwa wenn wir Platzkomplexitätsklassen betrachten, ist diese Definition zu grob. Deshalb definieren wir in dieser Sektion eine feinere Reduktionsrelation.

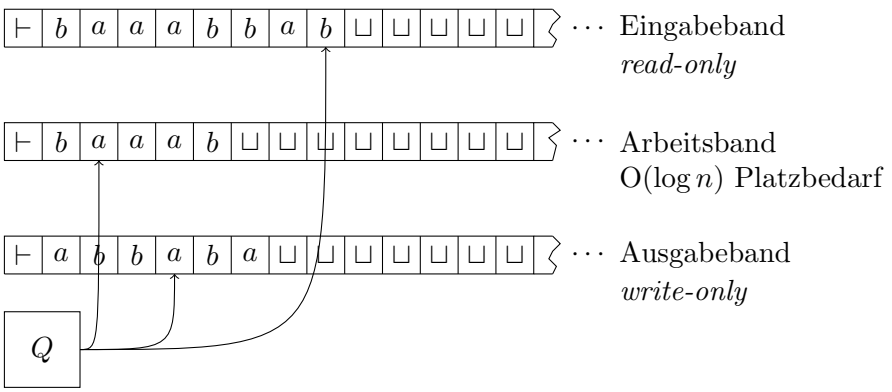
Definition 4.11. Ein *logarithmischer Umwandler* T ist ein 9-Tupel

$$T = (Q, \Sigma, \Gamma, \Delta, \vdash, \sqcup, \delta, s, t) \ ,$$

sodass Δ eine endliche Menge von *Ausgabesymbolen* ist und Q, Σ, Γ, $\vdash$, $\sqcup$, δ, s, t wie bei einer deterministischen und totale dreibändigen Turingmaschine definiert ist.

Zusätzlich gilt, dass das erste Band nur gelesen werden, das dritte Band nur beschrieben werden und am zweite Band bei jeder Eingabe der Länge n maximal $\log n$ Zeichen gelesen werden dürfen.

Schematisch kann ein logarithmischer Umwandler wie folgt dargestellt werden.



Definition 4.12. Eine Funktion $f\colon \Sigma^* \to \Delta^*$ heißt *berechenbar mit logarithmischem Platz*, wenn ein logarithmischer Umwandler mit Eingabealphabet Σ und Ausgabealphabet Δ existiert, der bei Eingabe von $x \in \Sigma^*$, $f(x)$ auf seinem Ausgabeband stehen hat, wenn er hält.

Lemma 4.2. *Sei T ein logarithmischer Umwandler, dann ist die Länge der Ausgabe von T, bei allen Eingaben der Länge n, polynomiell in n beschränkt.*

Beweisskizze. Wenn die Länge des Eingabebandes durch n und die Länge des Arbeitsbandes durch $\log n$ beschränkt, dann gibt es nur polynomiell viele Konfigurationen, also nur polynomiell viele mögliche Schritte. Die genaue Ausarbeitung des Beweises überlassen wir der Leserin. □

Definition 4.13. Eine Sprache $L \subseteq \Sigma^*$ ist *reduzierbar mit logarithmischem Platz* auf eine Sprache $M \subseteq \Delta^*$, wenn eine Funktion $R\colon \Sigma^* \to \Delta^*$ existiert, die berechenbar mit logarithmischem Platz ist, sodass gilt

$$x \in L \iff R(x) \in M \ ,$$

für alle Wörter x (über Σ). Wir schreiben $L \leqslant_m^{\log} M$, wenn L auf M reduzierbar ist, mit logarithmischem Platzbedarf.

Der nächste Satz zeigt, dass die Relation $\leqslant_m^{\log}$, die vorher definierte Relation $\leqslant_m^{\mathrm{p}}$ verfeinert.

Satz 4.3. *Seien A und B beliebige Probleme.*

1. *Wenn $A \leqslant_m^{log} B$, dann gilt auch $A \leqslant_m^{p} B$.*
2. *Wenn $A \leqslant_m^{log} B$ und $B \in \mathsf{P}$ ($B \in \mathsf{NP}$), dann $A \in \mathsf{P}$ ($A \in \mathsf{NP}$).*

Beweis. Der erste Teil des Satzes folgt direkt aus Lemma 4.2. Der zweite Teil folgt aus dem ersten Teil und Lemma 4.1. □

Für den Beweis des folgenden Lemmas verweisen wir auf [7, 6].

Lemma 4.3. *Die Relationen $\leqslant_m^{p}$ und $\leqslant_m^{log}$ sind transitiv.*

Nun geben wir eine alternative Definition 4.10 an, die auf die auf die Relation $\leqslant_m^{\log}$, statt auf die Relation $\leqslant_m^{\mathrm{p}}$ bezogen ist.

Definition 4.14. Sei $\mathcal{C}$ eine beliebige Komplexitätsklasse dann ist eine Sprache A *$\leqslant_m^{log}$-hart* für $\mathcal{C}$, wenn für alle Sprachen $B \in \mathcal{C}$ gilt: $B \leqslant_m^{\log} A$. Eine Sprache A heißt *vollständig für $\mathcal{C}$ in Bezug auf $\leqslant_m^{log}$*, wenn gilt

1. A ist $\leqslant_m^{\log}$-hart für $\mathcal{C}$ und
2. $A \in \mathcal{C}$

Oft sagt man auch abkürzend A is $\mathcal{C}$-vollständig, wenn die zugrundeliegende Art der Reduktionsrelation aus dem Kontext klar wird.

Die folgenden Verstärkung von Satz 4.2 geben wir ohne Beweis an, siehe zum Beispiel [8] oder [6].

Satz 4.4. *Das Problem* TSP *ist* NP*-vollstandig, das heißt vollständig für* NP *in Bezug auf* $\leqslant_m^{log}$.

4.6 Speicherplatzkomplexität

Analog zur Laufzeitkomplexität führen wir nun die *Speicherplatzkomplexität* formal ein.

Definition 4.15. Sei M eine deterministische und totale TM. Der *Speicherplatz* oder die *Speicherplatzkomplexität* von M ist die Funktion $S\colon \mathbb{N} \to \mathbb{N}$, wobei $S(n)$ die maximale Anzahl von Bandfeldern bezeichnet, die M auf allen Eingaben der Länge n liest, wobei aber nur die Zeichen auf den Arbeitsbändern betrachtet werden. Wir sagen auch dass $S(n)$ der Speicherplatz von M ist, oder dass M eine $S(n)$-Platz Turingmaschine ist.

Für eine nichtdeterministische, totale TM N ist der Speicherverbrauch $S(n)$ als die maximale Anzahl von Arbeitsbandfeldern, die auf einem beliebigen Pfad des Berechnungsbaumes gelesen wird, definiert.

Definition 4.16. Sei $S\colon \mathbb{N} \to \mathbb{N}$ eine numerische Funktion. Die deterministischen und nichtdeterministischen Platzkomplexitätsklassen sind wie folgt definiert:

$$\begin{aligned}
\mathsf{DSPACE}(S(n)) &:= \{\mathrm{L}(M) \mid M \text{ ist eine deterministische Mehrband-TM mit Speicherplatz } \mathrm{O}(S(n))\ \} \\
\mathsf{NSPACE}(S(n)) &:= \{\mathrm{L}(M) \mid M \text{ ist eine nichtdeterministische Mehrband-TM mit Speicherplatz } \mathrm{O}(S(n))\ \}
\end{aligned}$$

Wir erweitern die Klasse von Komplexitätsklassen, die wir betrachten und definieren gängige Platzkomplexitätsklassen.

Definition 4.17.

$$\begin{aligned}
\mathsf{LOGSPACE} &:= \mathsf{DSPACE}(\log n) & \mathsf{NLOGSPACE} &:= \mathsf{NSPACE}(\log n) \\
\mathsf{PSPACE} &:= \bigcup_{k \geqslant 1} \mathsf{DSPACE}(n^k) & \mathsf{NPSPACE} &:= \bigcup_{k \geqslant 1} \mathsf{NSPACE}(n^k)
\end{aligned}$$

Ohne Beweis geben wir den folgenden Satz an; der Beweis kann etwa in [8] oder [6] nachgelesen werden.

Satz 4.5.

1. *Das Problem* MAZE *ist* NLOGSPACE*-vollständig, das heißt vollständig für* NLOGSPACE *in Bezug auf* $\leqslant_m^{log}$.

2. *Das Problem* GG *ist* PSPACE*-vollständig, das heißt vollständig für* PSPACE *in Bezug auf* $\leqslant_m^{log}$.

Abschließend wollen wir noch einen Überblick über die Zusammenhänge der betrachteten Komplexitätsklassen geben:

$$\mathsf{LOGSPACE} \subseteq \mathsf{NLOGSPACE} \subseteq \mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE} = \mathsf{NPSPACE}\ .$$

Wir sehen also unter anderem, dass die polynomielle Platzkomplexitätsklasse unter Nichtdeterminismus abgeschlossen ist (d.h. $\mathsf{PSPACE} = \mathsf{NPSPACE}$). Wie schon berichtet, ist die selbe Frage für die polynomielle Zeitkomplexitätsklasse P ein offenes Problem.

Literaturverzeichnis

[1] M. Brownlow. *Goldene Regeln der Spieleprogrammierung.* Car Hanser Verlag, 2004.

[2] A. Dür. *Diskrete Mathematik 1.* Institut für Mathematik, 3. auflage edition, 2010. Skriptum zur Vorlesung.

[3] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979. 1st edition.

[4] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 2001. 2nd edition.

[5] D. Kozen. *Automata and Computability.* Springer Verlag, 1997.

[6] D. Kozen. *Theory of Computation.* Springer Verlag, 2006.

[7] C.H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1995.

[8] M. Sipser. *Introduction to the Theory of Computation.* PWS Publishing Company, 1997.